Validating the Theoretical Foundations of Residual Networks through Experimental Testing

Alex Ho Pranay Mittal

Abstract

Convolutional neural networks traditionally initialize their learning parameters randomly and symmetrically about zero. This initialization makes it difficult for convolutional layers to "learn" the identity mapping, which in theory forces layers to alter intermediate mappings which may offer better performance than the overall network. The residual network architecture aims to address this problem, however the foundational theory behind their performance has yet to be established. (Hardt & Ma, 2016) explores the theoretical foundations of residual networks while also providing experimental results. We aim to validate and replicate their findings through our own practical experiments.

1. Literature Overview

1.1. Convolutional Neural Networks for Computer Vision

With the popularization of the backpropogation algorithm in (Rumelhart et al., 1985), an era of machine learning using artificial neural networks emerged. Over time, multilayer perceptrons were improved upon and different varieties of artificial neural networks emerged. One of those specialized varieties was the convolutional neural network. Convolutional networks perform conceptually in same way using the backpropogation algorithm and nonlinear activation functions, however instead of perceptrons, hidden layers consisted of a kernel matrix which was used to convolve with the input by multiplication or other dot product. This architecture allowed for spatial structure in the inputs to be used.

One of the first successful uses of convolutional networks was by (LeCun et al., 1989) in 1989. Here, convolutional neural networks were used to recognize handwritten digits. This paper marked the start of computer vision related tasks using convolutional neural networks, however it did not gain significant ground until graphics processing units were shown to be useful in artificial neural networks (Goodfellow et al., 2016). The convolutional network used in this paper only had three hidden layers, and it wasn't until 2012 when "deep" convolutional networks were practically viable.

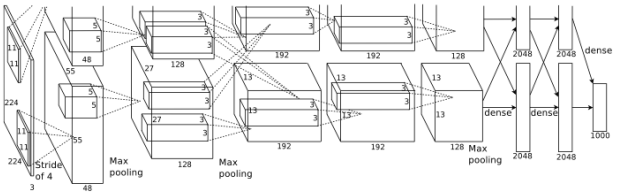
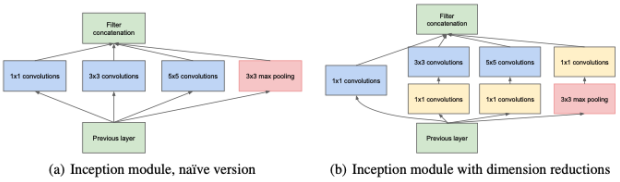


Figure 1. Illustration of AlexNet architecture from (Krizhevsky et al., 2017)

1.2. Advances through ImageNet

The ImageNet LSVRC-2010 contest was the first contest of its kind to have a large and high quality data set for computer vision tasks, containing 1.2 million high-resolution images coming from 1000 different classes. Learning this data set proved to be highly nontrivial for the first few years. Then in 2012, (Krizhevsky et al., 2017) made a breakthrough in convolutional network architecture. Achieving an error rate of 15.3%, the AlexNet architecture won the 2012 ImageNet Challenge by more than 10 percentage points compared to the runner up. It was the first of its kind to use rectified linear unit (ReLU) proposed in (Nair & Hinton, 2010) as the activation function. It also boasted a massive 60 million trainable parameters from its five convolutional and three fully-connected layers. Training this model was only possible with GPU processing. This architecture is considered the start of the computer vision revolution which rapidly evolved in the following years. An illustration of the architecture can be seen in Figure 1.

Just two years later in the 2014 ImageNet challenge, both (Szegedy et al., 2014) and (Simonyan & Zisserman, 2014) made large strides with a top-5 error rate less than half of the error rate of the AlexNet architecture. The network by (Szegedy et al., 2014), now called the GoogLeNet, introduced the inception module (see Figure 2) and architecture which showed that convolutional layers do not need to be stacked sequentially for good performance. This network also had 22 layers, which made it one of the first to be what



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2. Illustration of an Inception module from (Szegedy et al., 2014)

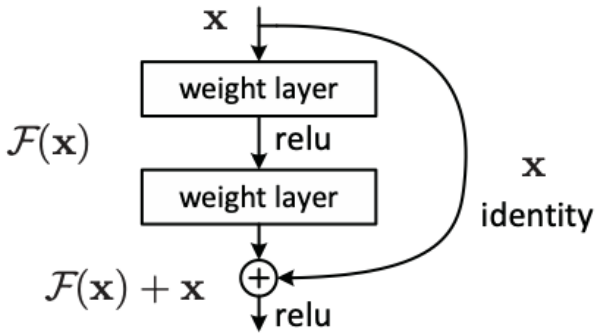
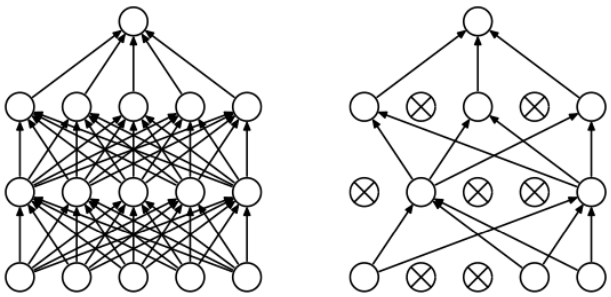


Figure 3. Illustration of a residual block from (He et al., 2015b)



(a) Standard Neural Net (b) After applying dropout.

Figure 4. Comparison of a standard neural network (left) to a thinned neural network resulting from dropout (right) from (Srivastava et al., 2014)

is now considered a "deep" network and achieved an error rate of 6.7%. The authors of (Simonyan & Zisserman, 2014) also created a 16 and 19 layer network called VGG-16 and VGG-19 after the name of their group, Visual Geometry Group. This network built upon the AlexNet architecture and used smaller filter sizes and achieved an error rate of 7.1%.

The next year, Kaiming He, et al. from (He et al., 2015b) reduced the error rate by 47% relative to the previous year down to 3.57% top-5 error when they developed the ResNet architecture. This architecture introduced a novel idea of using residual connections between convolutional blocks (two convolutional filters with ReLU activation functions and batch normalization applied) to allow neural networks to be even deeper than before. Their winning model was considered "ultra-deep" with 152 layers, but they also managed to train a network with over a thousand layers. This contribution was revolutionary because it seemingly fixed the problem of the vanishing and exploding gradient (Pascanu et al., 2013) with the residual connections. If each convolutional layer is thought of as multiplying the weights with an input image x, then the output y of a convolutional block with a residual connection, a residual block, can be seen as $y = W_2\sigma(W_1x)+x$ where $\sigma(x)$ is the ReLU activation function (also see Figure 3). The way the weights of a convolutional layer are initialized, such as He initialization and Xavier initialization (He et al., 2015a; Glorot & Bengio, 2010), mean they are centered about and near zero. In order for a standard convolutional block to learn the identity function, all of the weights would have to tend towards a value of 1. However in the residual connection architecture, the weights would only have to converge towards zero which is much easier given standard initialization. The residual connections also allow for easier backpropogation. With very deep networks, the gradient flow either vanishes or explodes, but with the skipping residual connections, there is a direct path when backpropogating from every layer to all layers before it. The ResNeXt architecture builds upon the standard ResNet architecture by adding a "cardinality" dimension which implements a similar structure to the Inception module.

1.3. Other Methods to Improve CNNs

In addition to novel convolutional network architectures, there have also been several methods which aim to improve their performance. One interesting and useful method is called dropout (Srivastava et al., 2014). Because these deep networks have so many parameters in comparison to the size of the training set, it can be easy for models to overfit. Dropout aims to mitigate the possibility of overfitting by randomly dropping units and their connections to the following layer. The goal of this approach is to reduce reliance on any particular input or set of inputs when training the weights which should improve robustness and generalizability to out of training sample data. The dropped connections only occur during training, so at test time all the weights are balanced and available to predict a test input. When backpropogating on a training mini-batch, only the weights which were in use get adjusted so that unused weights are not penalized. Their experimental results supported their hypothesis, showing minor improvements to state of the art models on all standard benchmark data sets.

Another method which has proved to be useful is batch nor-

malization (Ioffe & Szegedy, 2015). By using mini-batches for stochastic gradient descent, the inputs may have slightly different distributions for each mini-batch. This difference in distributions, or internal covariate shift as they call it, forces users to lower learning rates which causes training time to be unnecessarily long. The goal of batch normalization is to address the issue of internal covariate shift by normalizing the inputs by each training mini-batch. This approach allows the distributions of the inputs to be more similar across batches while also providing a degree of regularization. They claimed that with batch normalization, other methods like dropout are not necessary. Their experimental results validated their hypothesis by both increasing training time and increasing performance. They also noted that batch normalization allowed them to use higher learning rates along with the model being less sensitive to the random weight initialization. Several state of the art convolutional network architectures implement batch normalization, usually over dropout. Notable examples include the ResNet architecture and its variants.

1.4. Theoretical Foundations of ResNets

(Hardt & Ma, 2016) was one of the first to establish the theoretical foundation for identity parameterization. In this work, they prove that arbitrarily deep linear residual networks do not have spurious local optima, and that residual networks with ReLu activation functions can express any function given that the sample size is sufficiently large compared to the number of parameters. Hardt and Ma also showed that a large fully convolutional neural network with only residual blocks can match top-1 classification errors which are comparable to convolutional networks that used dropout, batch normalization, and intensive preprocessing in CIFAR-100 and ImageNet data sets.

(Zou et al., 2020) provided analysis for global minimum convergence of gradient descent and stochastic gradient descent for training arbitrarily deep linear residual networks. Their results showed that global convergence is sharper by a factor of $O(\kappa L)$, where κ is the condition number of the covariance matrix of the training data and L is the depth of the linear residual network, when compared to gradient descent on a standard linear network with zero initialization.

(Arora et al., 2018) showed that increasing depth not only improves expressiveness but also improves optimization. They decoupled the the effects of increasing depth on expressiveness from its effects on optimization by focusing on linear neural networks, in which additional layers caused overparametrization. Their results show that on convex problems (like linear regression) with l_p loss ($p > 2$), overparametrization via depth significantly sped up training. The authors also validate (Hardt & Ma, 2016)'s choice of identity (or near identity) initialization as a way to reap the benefits of accelerated convergence (by improving depth) while avoiding the vanishing gradient problem.

2. Experiments and Results

The goal of our experiments is to replicate the results shown in (Hardt & Ma, 2016). Namely, we want to match the accuracy of their residual network on CIFAR-10 (Krizhevsky, 2009), confirm their claims of no overfitting using large models, and validate their claim that the addition of batch normalization is unnecessary.

2.1. Dataset

The data set which will be used to compare the different models will be the CIFAR-10 data set (Krizhevsky, 2009) which is a well-known and standard baseline. The CIFAR-10 data set consists of 10 image classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck) where each class has exactly 5000 examples in the training set and 1000 examples in the test set. Each image is 32 by 32 pixels and in red, green, blue channels giving each image dimensions of $32 \times 32 \times 3$. We will not be replicating the experiments on the CIFAR-100 and ImageNet data sets due to time and resource constraints.

The CIFAR-10 data set is ideal for our use case for several reasons. First, it is sufficiently complex to be nontrivial, unlike MNIST (LeCun et al., 2010) which can easily reach less than 10 percent error rate with a single hidden layer fully connected neural network, and the dimensions of the images are small enough that progress and results can be made even with limited computational resources. Additionally, the data set is is large enough to train deeper models which will help validate the effects of our tested methods. State of the art error rates for CIFAR-10 are less than one percent (Foret et al., 2020; Dosovitskiy et al., 2020), however the goal of this experiment is not to achieve the highest accuracy, but rather to validate the findings of (Hardt & Ma, 2016).

2.2. Implementation

All models were created in TensorFlow, and the code can be found here. We attempted to match their setup as closely as possible to the original experiment: momentum 0.9, batch size 128, initial learning rate 0.05 (dropping by a factor of 10 at 30000 and 50000 steps). They did not provide access to their original code, and noted that it can be replicated by altering open source ResNet implementations from TensorFlow. We believe we adapted the architecture in the same way, but it is not possible to confirm.

We made a few changes to the setup which we thought were not worth implementing due to difficulty and lack of impact on the final result. First, they used a fixed random projection for the last layers, however we did not implement

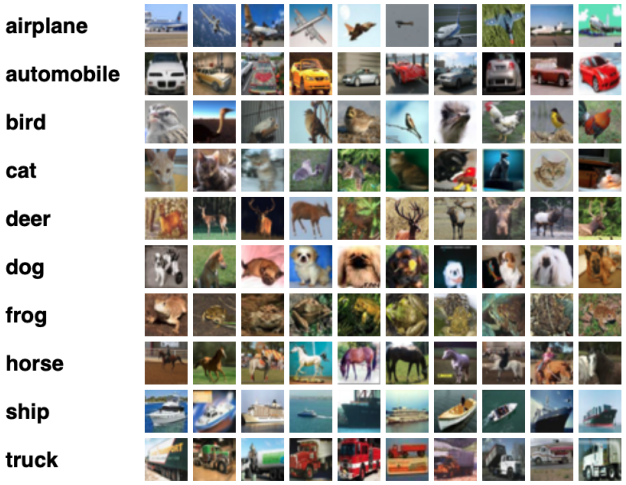


Figure 5. Sample images from the CIFAR-10 data set (Krizhevsky, 2009)

# Resblocks/size	Top-1 Accuracy (%)
2	89.11
3	90.03
8	88.05
9	88.19

Table 1. Summary of accuracy data for each model tested.

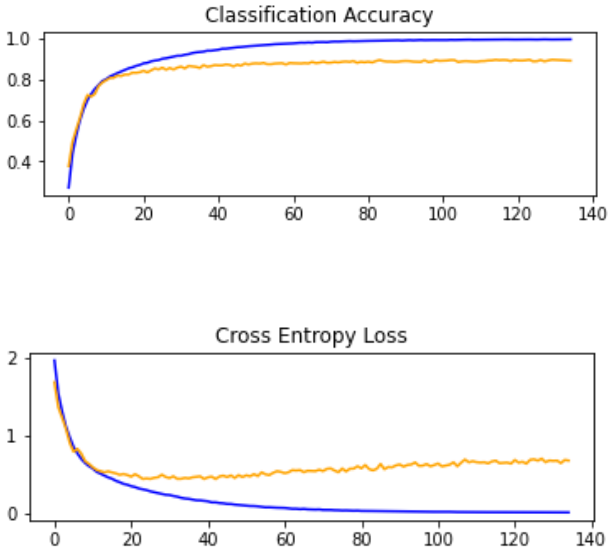


Figure 6. Top-1 classification accuracy and cross entropy loss of the residual network with 2 residual blocks per filter size over 135 epochs.

it, and second they used a smaller variance for their weight initialization, however we used standard Xavier uniform initializer. Third, we used an exponential decay learning rate which resulting in learning rate reduction at approximately the same number of training steps as their step function decay.

2.3. Results

A summary of the results can be found in Table 1. Our accuracy has hovered around 88-90% instead of the expected 93%. We suspect Hardt and Ma's improved accuracy is the result of hyperparameter fine tuning, which we were unable to do due to time constraints. This is unexected since they publish their hyperparameters. Another possible reason for the discrepancy between the accuracy is image preprocessig techniques. They only specifically say that they do not use ZCA whitening, and instead use standard data preprocessing, but without any further details.

While Hardt and Ma used nine residual blocks per filter size, we changed the number of residual blocks per filter size. We used 2, 3, and 9 residual blocks per filter for 13, 19, and 55 convolutional layers respectively. Our results showed that there is not a significant difference in performance between the networks of varying sizes despite large differences in the number of trainable weights.

2.4. Discussion

Overall, we were unable to validate many of the results found in (Hardt & Ma, 2016), but we do not have reason to believe that they are not achievable. While our own results did get fairly close to their reported accuracy, the difference is large enough that we hypothesize that there is another missing element of their implementation that would have helped incrase performance. For example, more details regarding their image preprocessing methods may have helped. Additionally, their initial learning rate did not appear to be useful. Using an initial learning rate of 0.05 on the 9 residual blocks per filter size model made convergence difficult. After 50 epochs, this model would still have performance equivalent to random guessing (about 10% validation and training accuracy). It was only until we changed the inital learning rate to 0.01 did we see convergence in the model. However, their loss and accuracy curves showed that their model was converging and learning almost immediately which does not reflect our findings.

Regarding overfitting in large models, we also differed in results than what (Hardt & Ma, 2016) reported. While there was not significant overfitting, there was evidence of a nontrivial amount of overfitting as seen in the loss curves seen in Figures 6 and 7. Both the smallest model with about 2.7 million trainable parameters and the largest (base) model with 13.6 million trainable parameters had evidence

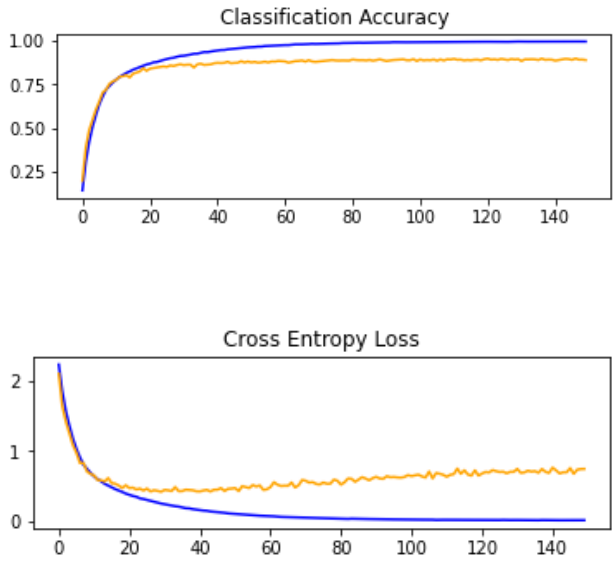


Figure 7. Top-1 classification accuracy and cross entropy loss of the residual network with 9 residual blocks per filter size over 150 epochs. Here we can see evidence of overfitting and marginal differences from the model with 2 residual blocks per filter size.

of overfitting, but it was more pronounced in the larger models. This evidence is seen by the diverging training and validation loss curves as the number of epochs increases.

(Ioffe & Szegedy, 2015) suggested the use of batch normalization in order to speed up convergence of models. However (Hardt & Ma, 2016) claimed that using batch normalization with their architecture led to overfitting. We hypothesize that their overfitting was a result of training the model for too long, and they could have mitigated it by early stopping. Several of the state of the art models take advantage of batch normalization and have shown that it is beneficial to increasing performance, unlike what was reported by (Hardt & Ma, 2016). Moreover, we saw additional increases in performance in preliminary experiments compared to the same architectures without batch normalization reported in this paper. We do not report them here because they were not repeated enough times to be shown to be reliable, however initial results showed that the addition of batch normalization added one to two points across the board in top-1 accuracy on validation sets, about the same as the difference between the accuracy of the best and worst performing model sizes.

Overall, we cannot attribute the discrepancies between our results and those reported in (Hardt & Ma, 2016) to any one specific reason. From our experiments, we can see an example of the reproducibility problem in machine learning research. Without providing an open source implementation or detailing the hyperparameters and methods used, it is difficult to replicate any results. Additionally, many papers do not report on how many attempts they had before achieving their reported error rate. Due to the inherently stochastic nature of weight initialization, results can be truly random each time and impossible to reproduce. Researchers can get away with cherry picking their data in order to report "state of the art" results without showing that it can consistently do so.

3. Future Work and Open Questions

For (Hardt & Ma, 2016), the open problems resulting from this work are extending the proof of spurious local optima to non-linear cases as well as finding conditions that alter the performance of practical residual networks. They give big-O notation for the number of training examples needed to properly train a residual network, however this is not practically useful due to how big-O is inherently a flexible upper bound. Fixing a specific residual network architecture and then finding an experimental bound on how many samples are needed to properly converge the model would be very useful for practical use cases as opposed to their current bound.

For (Zou et al., 2020), again, one of the open questions resulting from this paper is extending the convergence of deep residual networks to the non-linear case, but also to the case with adaptive descent methods or methods with momentum and acceleration. As it has been shown in recent work, it is highly nontrivial to accomplish this task regarding (Hardt & Ma, 2016) and (Zou et al., 2020).

Another interesting application which could lead to future work is the use of residuals for deep matrix factorization. (Arora et al., 2018), as discussed before showed that increasing depth in matrix factorization may help optimization of convergence contrary to conventional wisdom. Instead of using standard matrices to decompose some given matrix, we could hypothetically also use residual units consisting of matrices summed with the identity matrix. The addition of residuals may increase performance by allowing unnecessary matrices which do not contribute to the optimization to become simply the identity matrix while only the useful factorization units are non-identity transformations.

References

Arora, S., Cohen, N., and Hazan, E. On the optimization of deep networks: Implicit acceleration by overparameterization. *arXiv preprint arXiv:1802.06509*, 2018.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16

words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Foret, P., Kleiner, A., Mobahi, H., and Neyshabur, B. Sharpness-aware minimization for efficiently improving generalization. *arXiv preprint arXiv:2010.01412*, 2020.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Hardt, M. and Ma, T. Identity matters in deep learning. *arXiv preprint arXiv:1611.04231*, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015a.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015b.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, 2009.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

LeCun, Y., Cortes, C., and Burges, C. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pp. 1310–1318, 2013.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

Zou, D., Long, P. M., and Gu, Q. On the global convergence of training deep linear resnets. *arXiv preprint arXiv:2003.01094*, 2020.

THE USE OF MOMENTUM IN STOCHASTIC METHODS

Benito Geordie
bg31@rice.edu

Wenqing (Arthur) Wu
ww31@rice.edu

December 14, 2020

ABSTRACT

Researchers approach problems faced by stochastic gradient descent (SGD) through momentum-based variants of SGD and stochastic adaptive optimizer algorithms. In this literature review, we aim to compare popular stochastic adaptive optimizers and analyze their use of momentum. To facilitate the comparison of these algorithms, we devised a *uniform framework* – a generalization of SGD, its variants, and stochastic adaptive optimizers. For the interest of the length of this review, we will touch on AdaGrad, AdaDelta & RMSProp, Adam, and NAdam.

***K*eywords** Stochastic gradient descent · Momentum · Nesterov accelerated gradients · AdaGrad · AdaDelta · RMSProp · Adam · NAdam

1 Introduction

Following increased interest in deep learning, specifically neural network (NN) algorithms, many adaptive optimizers have been proposed, analyzed, and incorporated into new ones. The choice of optimizer significantly impacts the speed and outcome of a neural networks's training. Given the variety in dataset properties and loss function configurations, there isn't a one-size-fits-all optimization algorithm that can solve any training problem, as explained in [1] and demonstrated by [2]. While gradient descent (GD) is the poster child of NN training, in practice, researchers opt for SGD and SGDM. This makes them the most notable, most widely applied, and most studied optimization technique used in NN training, with numerous papers like [3, 4, 5] and more recently [6] written to analyze its convergence guarantees. Despite its effectiveness, challenges arose under more varied scenarios: plateau of saddle points is hard for SGD to escape [7]; hardcoded learning rate scheduling cannot gain information about the dataset [8]; it is difficult to choose the proper learning rate as the learning rate cannot be too big or too small for it to converge; fine tuning on learning rate on different weights can be meaningful, but is unavailable in SGD and SGDM[1]. In response, researchers developed adaptive optimizers such as AdaGrad, AdaDelta/RMSProp, Adam, all the way to the more recent AdaMax and Nadam, that are inspired by or developed on top of SGD with momentum (SGDM).

While *momentum* typically refers to the algorithm proposed by Rumelhart, et al. in [9], many popular modern optimizers employ the same ideas, rendering momentum's impact on the field to be vastly larger than just its vanilla implementation. Thus, with regards to our goal of evaluating the use of momentum in stochastic methods, we find it too restricting to exclusively analyze SGDM – the bare-bones adoption of Rumelhart et al.'s algorithm into SGD. Some of the algorithms we analyze don't explicitly use momentum, but reuse ideas from momentum in one way or another. Unfortunately, the interconnectedness of these algorithms is often lost in the nuances of notation. In response to this, we devised the *uniform framework* – a generalization of all the algorithms analyzed in this paper – presented in section 1.2.

1.1 Notation

Before introducing our uniform framework, we introduce the following notation:

- $\mathbf{I}$ is the identity matrix.
- The subscript of x_t denotes that a variable x is calculated in iteration t.

- $w \in \mathbb{R}^d$ represents the given weight vector of the model that needs to be trained.
- $\ell(x)$ is the objective/loss function evaluated at x.
- α denotes the initial learning rate.
- ζ, ϕ and ψ each represent arbitrary functions.
- The $\sqrt{x}$ and x^2 operations output the element-wise square root and square of a vector x respectively.
- $\frac{x}{y}$ and $x \cdot y$ are element-wise multiplication and division respectively. When either x or y is a scalar, they are treated like $x\mathbf{I}$ or $y\mathbf{I}$ respectively.

1.2 The Uniform Framework

Based on the algorithms we analyze, we define the *uniform framework* as follows:

For each training step t:

1. Calculate the gradient of the loss function with current weight vector:
$$g_t \leftarrow \nabla \ell(\zeta(w_t, m_{t-1}, V_{t-1}))$$
2. Calculate the first- and second-order moments from history of gradients:
$$\textit{First-order moment: } m_t \leftarrow \phi(g_1, g_2, g_3, ..., g_t, \beta_1)$$
$$\textit{Second-order moment: } V_t \leftarrow \psi(g_1^2, g_2^2, g_3^2, ..., g_t^2, \beta_2)$$
where β_1 and β_2 are constants used in calculating the first- and second-order moments respectively.
3. Calculate the the descent step:
$$\eta_t \leftarrow \frac{\alpha}{\sqrt{V_t}} \cdot m_t$$
4. Update the weight vector:
$$w_{t+1} \leftarrow w_t - \eta_t$$

1.3 Review Structure

In section 2, we will cover variants of SGD, showing that they are special cases of the *uniform framework*. In section 3, we will then briefly summarize the aforementioned popular adaptive optimization algorithms by representing them with the *uniform framework*. Subsequently, in section 4, we analyze how these algorithms employ ideas from momentum. Afterwards, in section 5, we show the results of a simple NN experiment optimized with the algorithms discussed in this paper. Finally, in section 6, we will summarize the role of momentum described in the previous section and take a look at cases where momentum is viewed almost adversarially by schemes that curb its impact on each training step.

2 Variants of SGD

2.1 Stochastic Gradient Descent (SGD)

In vanilla SGD, there is no momentum involved. Thus, we can represent it with the *uniform framework* by setting $\zeta(w_t, m_{t-1}, V_{t-1}) = w_t$ such that $g_t \leftarrow \nabla \ell(w_t)$ in step 1, $\phi(g_1, ..., g_t, \beta_1) = g_t$ and $\psi(g_1^2, ...g_t^2, \beta_2) = \mathbf{I}$ such that $m_t \leftarrow g_t$ and $V_t \leftarrow \mathbf{I}^2$ in step 2 of the *uniform framework*. It follows that step 3 of the *uniform framework* is

$$\eta_t \leftarrow \frac{\alpha}{\sqrt{\mathbf{I}^2}} \cdot g_t = \alpha \cdot g_t$$

SGD is known to have the property of performing frequent updates with a high variance, which causes a fluctuation of the objective function. This results in jumping out of the local minimum and simultaneously complicate or slow down the convergence rate [1].

2.2 SGD with Momentum (SGDM)

In order to solve the overshooting and fluctuation of the objective function, and in the interest of accelerating SGD's convergence process, we add momentum to SGD [10]. While V_t stays the same as the SGD case, m_t is now:

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

To be consistent with the *uniform framework*, this is facilitated by changing phi as follows:

$$\phi(g_1, ..., g_t, \beta_1) = \sum_{i=1}^{t} \beta_1^{t-1}(1 - \beta_1)g_i$$

The first-order momentum is the exponential moving average of the gradient direction at each iteration, which is approximately equal to the average of the sum of the gradient vectors at the most recent $1/(1 - \beta_1)$ training steps.

If $\beta_1 = 0.9$, we can intuitively think that the descent direction at iteration t is not only decided by gradient evaluated at the current weight vector, and is instead 90% based on the weighted average from previous gradients. When current gradients are in the same direction as previous, we will be more aggressive in this direction.

2.3 SGD with Nesterov Accelerated Gradients (SGD-NAG)

Another problem with SGD is that it oscillates in the gully of local optimum. Imagine walking into a basin, surrounded by slightly higher hills. Thinking that there is no downhill direction, you can only stay here. But if you climb up higher ground, you will find that the outside world is still very vast. Therefore, we cannot stay in the current position to observe the future direction, but must take a step forward, look one step ahead, and look farther.

SGD-NAG is another variant of SGDM. The difference is in step 1 of the *uniform framework*:

$$\zeta(w_t, m_{t-1}, V_{t_1}) = w_t - \alpha \cdot m_{t-1}$$

$$\textit{Thus, } g_t \leftarrow \nabla\ell(w_t - \alpha \cdot m_{t-1})$$

It does not calculate the gradient at the current position, but calculates the gradient direction at the position that the model would be in if it took a step in the direction of the accumulated momentum [11]. The ensuing steps in the *uniform framework* are the same as SGDM.

3 Stochastic Adaptive Optimizers

The algorithms above did not utilize second-order moment. The emergence of second-order moment indicates the arrival of the era of "adaptive learning rate" optimization algorithms. SGD and its variants update each parameter at the same learning rate, but deep neural networks often contain a large number of parameters, which are not always available (thinking about large-scale embedding). For the frequently updated parameters, we have accumulated a lot of knowledge about it. We don't want to be affected too much by a single sample, and therefore want the learning rate for these parameters to be slower. For the occasionally updated parameters, we know too little information, and we want to learn more from each occasion by having a higher learning rate for these parameters.

With this intuition, AdaGrad introduces the second-order moment and it is calculated at step 2 of the uniform framework:

$$V_t \leftarrow \psi(g_1^2, ..., g_t^2, \beta_2) = \sum_{i=1}^{t} g_i^2$$

Note that Adagrad does not necessarily use momentum, so $g_t \leftarrow \nabla\ell(w_t)$ *and* $m_t \leftarrow g_t$.

If we take another look at step 3 in the *uniform framework*:

$$\eta_t = \frac{\alpha}{\sqrt{V_t}} \cdot m_t$$

We can notice that the learning rate changes from α to $\alpha/\sqrt{V_t}$. The more frequently a parameter is updated, the greater its second-order moment, and therefore the smaller its learning rate.

This method performs very well in sparse data scenarios [12], but it also creates a new problem: because $\sqrt{V_t}$ is monotonically increasing, the learning rate will monotonously decrease to 0, which may end the training process prematurely. Even if there is useful data in the ensuing iterations, it is impossible to learn from this new information.

3.1 AdaDelta & RMSProp

Since AdaGrad's monotonically decreasing learning rate changes are aggressive, we consider a strategy to change the second-order moment calculation method: do not accumulate all historical gradients, but focus on the most recent fixed-length time window. To avoid the inefficient storage of past gradients, this accumulation is implemented with an exponential moving average [13]. AdaDelta modifies step 3 in AdaGrad as follows:

$$V_t \leftarrow \beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot g_t^2$$

To be consistent with the *uniform framework*, this is facilitated by changing ψ as follows:

$$\psi(g_1^2, ..., g_t^2, \beta_2) = \sum_{i=1}^{t} \beta_2^{t-1}(1 - \beta_2)g_i^2$$

This avoids the problem of continuous accumulation of second-order moment, which led to the premature end of the training process.

The *uniform framework* expression of RMSProp would be identical. RMSProp and Adadelta have both been developed independently around the same time, stemming from the need to resolve Adagrad's radically diminishing learning rate [1].

3.2 Adam

At this point, the emergence of Adam and NAdam are very natural; they are the intuitive momentum-based follow-up to the aforementioned methods. We see that SGDM adds first-order moment to SGD, and AdaGrad and AdaDelta add second-order moment to SGD. Incorporating both the first- and second-order momentum gives us Adam [14], which is a combination of adaptive and momentum-based optimizers, where

$$m_t \leftarrow \frac{\beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t}{1 - \beta_1^t}$$

$$V_t \leftarrow \frac{\beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot g_t^2}{1 - \beta_2^t}$$

To put it in the *uniform framework*, we modify ϕ and ψ as follows:

$$\phi(g_1, ..., g_t, \beta_1) = \frac{\sum_{i=1}^{t} \beta_1^{t-1}(1 - \beta_1)g_i}{1 - \beta_1^t}$$

$$\psi(g_1^2, ..., g_t^2, \beta_2) = \frac{\sum_{i=1}^{t} \beta_2^{t-1}(1 - \beta_2)g_i^2}{1 - \beta_2^t}$$

3.3 NAdam

NAdam can be intuitively understood as Adam + Nesterov accelerated gradients [15]. Analogous to how SGD-NAG was introduced in section 2 as SGDM with a modification in step 1 where

$$\zeta(w_t, m_{t-1}, V_{t_1}) = w_t - \alpha \cdot m_{t-1}$$

NAdam can be presented as Adam with a modification in step 1 where

$$\zeta(w_t, m_{t-1}, V_{t_1}) = w_t - \alpha \cdot \frac{m_{t-1}}{V_{t-1}}$$

$$\textit{Thus, } g_t \leftarrow \nabla \ell \left(w_t - \alpha \cdot \frac{m_{t-1}}{V_{t-1}} \right)$$

4 The Role of Momentum

In this section, we analyze how the adaptive algorithms introduced above utilize momentum. While not all of the methods below explicitly use a momentum term, they draw insights from the momentum term's construction.

4.1 AdaGrad

Adagrad is one such algorithm that, while it does not incorporate a momentum term, it employs a familiar idea of collecting information from previous gradients. Specifically, as mentioned above, Adagrad collects the second moment of the gradients through the V_t term. We would like to highlight the similarities between the function ψ used to calculate V_t in AdaGrad:

$$\psi(g_1^2, ..., g_t^2, \beta_2) = \sum_{i=1}^{t} g_i^2$$

and the function ϕ used to calculate m_t in SGDM:

$$\phi(g_1, ..., g_t, \beta_1) = \sum_{i=1}^{t} \beta_1^{t-1}(1-\beta_1)g_i$$

Unlike SGDM and Nesterov, however, AdaGrad does not use moment to accelerate descent in the historically dominant direction, and instead uses moment to curb learning in those directions, allowing the optimizer can focus on parameters with sparser, lower magnitude features.

While AdaGrad uses moments in a different way, it is apparent that momentum contributes to it through the idea of calculating moments using past gradients.

4.2 AdaDelta & RMSProp

AdaDelta and RMSProp expand on AdaGrad with yet another idea that is reminiscent of momentum: decaying weights. Just as momentum is a moving average of past gradients that emphasizes more recent gradients, the second moment calculated by these algorithms is also a moving average of past second moments that prioritize more recent values. The resemblance is even more jarring when we compare AdaDelta & RMSProp with SGDM using the *uniform framework*. ψ used to calculate V_t in AdaDelta & RMSProp:

$$\psi(g_1^2, ..., g_t^2, \beta_2) = \sum_{i=1}^{t} \beta_2^{t-1}(1-\beta_2)g_i^2$$

and the function ϕ used to calculate m_t in SGDM:

$$\phi(g_1, ..., g_t, \beta_1) = \sum_{i=1}^{t} \beta_1^{t-1}(1-\beta_1)g_i$$

The only difference is ϕ uses past gradients while ψ uses the squares of those gradients. Here, we see that momentum contributes the idea of decaying weights and the resulting moving average of past moments to AdaDelta & RMSProp.

4.3 Adam & NAdam

Adam takes RMSProp and goes a step closer to SGDM by incorporating an explicit momentum term, making momentum's contribution to Adam the most obvious among the adaptive optimizers discussed in this paper. ϕ used to calculate m_t in Adam:

$$\phi(g_1, ..., g_t, \beta_1) = \frac{\sum_{i=1}^{t} \beta_1^{t-1}(1-\beta_1)g_i^2}{1-\beta_1^t}$$

and the function ϕ used to calculate m_t in SGDM:

$$\phi(g_1, ..., g_t, \beta_1) = \sum_{i=1}^{t} \beta_1^{t-1}(1-\beta_1)g_i$$

The only difference between the two first-order moment terms is Adam's contribution of the bias-correcting denominator. The explicit momentum term allows Adam to focus sparse features like RMSProp while enjoying acceleration and stability like SGDM. This ability utilize the best of both worlds makes Adam one of the most popular optimizers in recent years [16]. NAdam, which incorporates Nesterov acceleration instead of vanilla momentum, also enjoys the benefits of looking ahead at the gradient at the destination.

5 Experiment

To see the above algorithms in action, we devised a simple experiment involving a neural network with two hidden layers containing 200 nodes and 80 nodes respectively, each with its own bias. As our independent variable, we used 7 optimizers under a mini-batch setting: SGD, SGDM, SGD-NAG, AdaGrad, AdaDelta, RMSProp and Adam, thus covering every algorithm discussed here except NAdam. To ensure a fair experiment, we kept the following constant:

- The initial values of weights and biases in the model
- The shuffled sequence of samples fed into each optimizer

Other details:

- Mini-batch size = 10
- Loss function value sampled every 20 steps
- We used the Iris dataset, which is a classifying training data with 150 samples and 4 attributes each, categorizable into 3 classes. To preprocess the data, we shuffled it then took the first 100 samples for training.

The results are as follows:

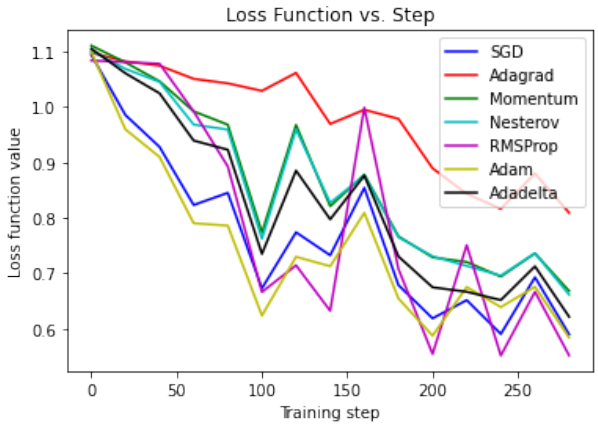


Figure 1: Loss function values vs. training steps

Figure 2: Loss function values vs. training time

From the graphs above we can see that, given the stochastic nature of these algorithms, they all go experience instability and fluctuating loss functions. Digging deeper, we notice the following:

1. SGD performs very well, and better than SGDM, SGD-NAG as well as Adagrad, supporting the claim that there is no one-size-fits-all solution when it comes to choosing optimizers. The small size and lack of sparsity of the Iris dataset potentially gave vanilla SGD an edge as learning it does not require the nuances of the other optimizers.
2. SGDM (Momentum) and SGD-NAG (Nesterov) delivered what they promised: a more stable descent, as supported by the smaller number of zigs and zags in the graphs.
3. Adagrad delivered the worst performance, both in terms of descent rate and final loss value. This is expected as AdaGrad delivers the best performance in large, sparse datasets.
4. Adam and RMSProp seem to deliver the best performance among the adaptive algorithms, with Adam enjoying more stability, as expected due to its incorporation of momentum.

6 Discussion

From section 4, we can see that as adaptive algorithms progress, they seem to adopt more ideas from momentum, from calculating moments from past gradients, to the use of decaying weights to produce a moving average, to blatantly using a momentum term. Thanks to these improvements, adaptive optimizers get more comprehensive and effective [1], therefore showing the magnitude of momentum's impact on stochastic methods.

Despite its apparent effectiveness, copying momentum is clearly far from being the solution to every gradient descent problem. In fact, many optimizer schemes work by curbing the contribution of the momentum term. One of these schemes is quasi-hyperbolic momentum (QHM), which could be expressed in terms of the *uniform framework* the same way as SGDM, except step 3 is modified to

$$\eta \leftarrow \alpha \cdot (\nu m_t + (1-\nu) g_t)$$

where ν is a newly introduced constant that limits the contribution of the momentum term to only ν of the descent step. This allows the momentum term to enjoy lower variance by using a larger β_1 without making the descent step "unusably biased" to older descent directions [17]. Another scheme that also curbs the contribution of the momentum term is DEMON, which, in the interest of brevity, can be expressed using the *uniform framework* the same way as SGDM, but with an evolving β_1:

$$\beta_{1_t} = \beta_{1_{init}} \cdot \frac{(1 - t/T)}{(1 - \beta_{1_{init}}) + \beta_{1_{init}}(1 - t/T)}$$

where T is the total number of training steps. Here, the contribution of the momentum term decreases the closer that it gets to the last training step [18]. While there is no official theoretical explanation for why this helps, one can imagine that DEMON works by reducing the chance of momentum-caused overshooting when it approaches the global minimum.

This shows that while momentum is undeniably valuable in stochastic optimization, there is still plenty to discover regarding how to use it optimally.

7 Conclusion

In this review, we have initially looked at the three variants of gradient descent: SGD, SGDM, SGD Nesterov, where the last two are most common in practice. Using a uniform framework, we have then investigated algorithms that are most commonly used for optimizing SGD. Specifically, we looked at popular adaptive optimization algorithms Adagrad, Adadelta & RMSprop, and Adam & Nadam to show the chain of development, their differences, their strengths, and how they adopted ideas from Rumelhart et al.'s momentum algorithm. Finally, we discussed how optimizers may want to reduce the contribution of the momentum term in their training step, showing that there is still room for progress in the use of momentum in stochastic methods.

References

[1] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[2] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning, 2018.

[1] In the average case. As previously mentioned, no optimizer provides a one-size-fits-all solution.

[3] Igor Gitman, Hunter Lang, Pengchuan Zhang, and Lin Xiao. Understanding the role of momentum in stochastic gradient methods. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 9633–9643. Curran Associates, Inc., 2019.

[4] Rahul Kidambi, Praneeth Netrapalli, Prateek Jain, and Sham M. Kakade. On the insufficiency of existing momentum schemes for stochastic optimization. *CoRR*, abs/1803.05591, 2018.

[5] Nicolas Loizou and Peter Richtárik. Linearly convergent stochastic heavy ball method for minimizing generalization error, 2017.

[6] Yanli Liu, Yuan Gao, and Wotao Yin. An improved analysis of stochastic gradient descent with momentum, 2020.

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25, pages 1223–1231. Curran Associates, Inc., 2012.

[8] Christian Darken, Joseph Chang, Joseph Chang Z, and John Moody. Learning rate schedules for faster stochastic gradient search. IEEE Press, 1992.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[10] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[11] Y. E. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Dokl. Akad. Nauk SSSR*, 269:543–547, 1983.

[12] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 07 2011.

[13] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[15] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[16] S. Bock and M. Weiß. A proof of local convergence for the adam optimizer. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[17] Jerry Ma and Denis Yarats. Quasi-hyperbolic momentum and adam for deep learning. *CoRR*, abs/1810.06801, 2018.

[18] John Chen and Anastasios Kyrillidis. Decaying momentum helps neural network training. *CoRR*, abs/1910.04952, 2019.

Deep double descent

Tianyang Pan [*1] **Rui Zhang** [*2]

Abstract

Machine learning has been developing rapidly in the recent years and changed to landscape of science and society. As the learning models evolving fast, some conventional concept in statistical learning are facing new challenges. Among them is the diminishing of 'overfitting' effect for large-scale models in the conventional bias-variance trade-off risk curve. Here we investigated the "double descent" risks curve for contemporary machine learning models. We showed that the such behavior is robust for architectures ranging from random Fourier features to convolutional neural networks. Further investigation and experiments show that model architecture, training input and training process will all have impact on the double descent phenomenon.

1. Introduction

Bias-variance trade-off is a concept first introduced in statistics text books and later adapted in the study of supervised learning: a balance between bias and variance is ideal while training a model: bias error is originated from inaccurate or simplified assumptions in the parameters and causes underfitting, while the variance error is caused by models that are too complicated or followed the training data too closely, which will result in overfitting. It is widely accepted that by controlling the capacity of the model, a sweetspot between the two should be achieved in the U-shape bias-variance curve (Fig. 1(a)). However, in the modern machine learning practice, predictors that utilize large scale of parameters (e.g. deep neural network architectures) that is certain to overfit predicted under the conventional bias-variance mind set usually have very accurate prediction results, despite of reaching zero training error.

Such contradiction is reconciled by the double descent risk curve (Fig. 1(b)) proposed by researchers. For high- complexity modern models with given training set of size n, the risk of a learner as a function of (or some approximation of) its complexity N have double descent behavior: the risk initially decreases and reaches a minimum as N increases, then increases due to overfitting until N equals n, forming a peak at $N = n$. As N increases even further, the risk curve decreases a second and final time. The double descent curve is consistent with the high performance of predictors with near-perfect fit of the training data, but its correlation to the different variables in the learning procedure as well as a rigorous analysis is still wanting.

Early works on artificial data has exposed double descent of classifiers trained with minimum norm linear regression (MNLR) (F. Vallet, 1989), theoretical work (M. Opper, 1990) proved that for MNLR, the solution improves as soon as N increase beyond n. Further investigations(Duin, 2000) on real-world data also showed the double descent behavior of classifier with similar solutions (T. L. H. Watkin, 1993). However, in these studies, double descent behavior is still considered within the bias-variance trade-off frame and hold true only for particular models.

This exposition analyzes literatures and conducts experiments on risk curves of various contemporary machine learning models. Risk as a function of model complexity exposes a universal double descent behavior. We also test the curve under different experiment settings, especially on how the sample size, training time, noise level of the training data affects the overall risk curve. In the introduction part, some basic concept such as effective model complexity is introduced. In the second part, we conducted a detailed analysis of double descent in a few model prototypes, such as fully connected neural network and random fourier features. In the third part, we focused on the performance of the model in double descent critical regime. we experiment and summarize the works of double descent in training architectures such as residual neural network (ResNet) and convolutional neural networks (CNN), where the impact of different training input and training process are discussed. In the last part, we focused on sample-wise double decent behavior and showed some analytical results to explain why for networks of certain complexity, adding more training samples

[*]Equal contribution [1]Department of Computer Science, Rice University, Houston, Texas, USA [2]Department of Physics and Astronomy, Rice University, Houston, Texas, USA. Correspondence to: Tianyang Pan <tp36@rice.edu>, Rui Zhang <rz15@rice.edu>.

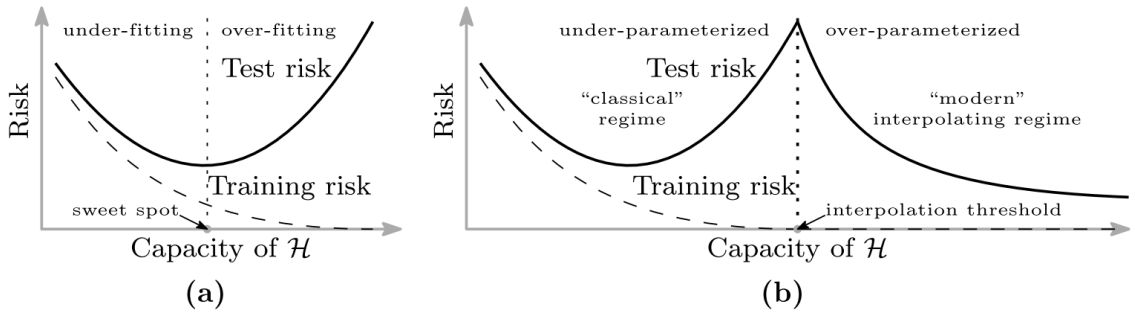


Figure 1. Our result showing similar sample-wise double descent phenomenon in the same linear regression problem discussed in (Nakkiran, 2019). The sample dimension is set to 1000. The x-axis is the number of samples, and the y-axis is the test error.

can adversely affect the performance of the model. We believe understanding double descent can help us improve performance of a model especially when under limited capacity. Under such circumstances, one can not train the model with infinite time, training parameters or training data, thus it is crucial to achieve an optimal solution with the presence of double descent curve.

2. Double Descent

This section presents the analysis and experiments on the double descent phenomenon developed in (Belkin et al., 2019). The paper investigated double descent in a set of different popular machine learning models. We will review the discussion and support with our own experimental results.

2.1. Background

We first define the Effective Model Complexity (EMC) of a training procedure T as our primary variable in risk curves introduced below, with respect to train samples S that has distribution D and parameter $\epsilon > 0$, is defined as:

$$EMC_{D,\epsilon}(T) := max\{n | E_{S\sim D^n}[Error_S(T(S))] \leq \epsilon\}$$

where $Error_S(M)$ is the mean error of model M.

And the key hypothesis is that for any natural data distribution D, neural-network-based training procedure T , and small $\epsilon > 0$, if we consider the task of predicting labels based on n samples from S, then

Under-paremeterized regime. Where $EMC_{D,\epsilon}(T)$ is sufficiently smaller than n, so that increasing EMC will lead to $decreasing$ of the test error.

Over-parameterized regime. Where $EMC_{D,\epsilon}(T)$ is sufficiently larger than n, so that increasing EMC will lead to $decreasing$ of the test error.

Critically parameterized regime. Where $EMC_{D,\epsilon}(T) \approx n$, so that increasing EMC might lead to $decreasing$ or $increasing$ of the test error.

Intuitively, when $EMC = n$, we arrive at interpolation threshold, where the model achieve perfect fits of the training data. There is also a critical interval in the vicinity of the interpolation threshold, and we will analyze the performance of the model in the critical interval below.

2.2. Random Fourier Features

Random Fourier Features (RFF) is a class of non-linear parametric models. As discussed in the paper, RFF can be seen as a class of two-layer neural networks. The RFF model family $\mathcal{H}_N$ with N parameters consists of functions $h : \mathbb{R}^d \to \mathbb{C}$:

$$h(x) = \Sigma_{k=1}^{N} a_k \phi(x; v_k) \text{ where } \phi(x; v) = e^{\sqrt{-1}\langle v, x\rangle}$$

v_i should be sampled from normal distribution in $\mathbb{R}^d$. The learning process is to find a predictor $h_{n,N} \in \mathcal{H}_N$ that minimizes squared loss given n data points $x_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$, i.e. the following objective is minimized:

$$\frac{1}{n}\Sigma_{i=1}^{n}(h(x_i) - y_i)^2$$

Belkin's paper trained the RFF model on a subset of MNIST datset. The result from the paper is shown in figure 1. The number of training samples is $n = 10^4$. From the figure, we can see that the error reaches a peak (i.e., the interpolation threshold) at $N = 10^4$, which equals to n. On the left side of the threshold is the underparameterized regime, where the test risk first decreases then increases. The bias-variance tradeoff should be considered if the problem setting falls in this regime. On the right side is the overparameterized

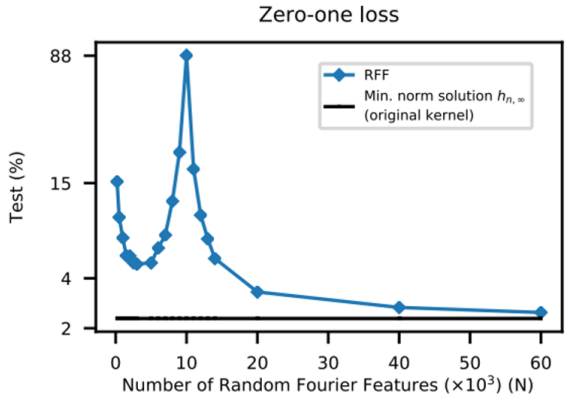


Figure 2. The result of RFF from (Belkin et al., 2019). The RFF model is trained using a subset ($n = 10^4$) of MNIST dataset.

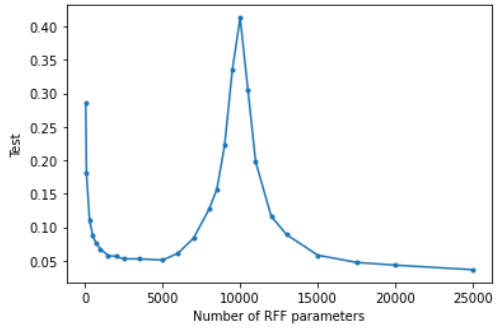


Figure 3. Our result of RFF trained using a subset ($n = 10^4$) of MNIST dataset. It shows a similar trend as the original result.

regime, where richer models have lower test risk. Figure 3 shows our result. We adapted code from an RFF repo[1] and modified to work with this problem setting. As shown in the figure, we can observe a similar trend. The double descent behavior exists and the interpolation threshold is achieved when $n = N$.

2.3. Fully Connected Neural Networks

Neural networks uses back propagation to update weights in the layers. The paper investigates the double descent behavior on a fully connected two-layer neural network. For such a neural network, the model capacity can be modeled using the number of weights in the layers. Assume we have a layer of H hidden units, learning a subset of MNIST dataset (n training samples, d samples dimension, and K classes), the number of parameters is:

[1]https://github.com/tiskw/Random-Fourier-Features

$$n_p = (d + 1)H + (H + 1)K$$

The neural network is trained on a subset of MNIST dataset. The original results from the paper are shown in figure 4 (with weight reuse techinque) and figure 5 (without weight reuse). For multi-class classification problems, the interpolation threshold should be achieved at nK parameters. As shown in figure 4, the threshold is achieved at $4 * 10^4$ parameters, separating the two regimes as expected.

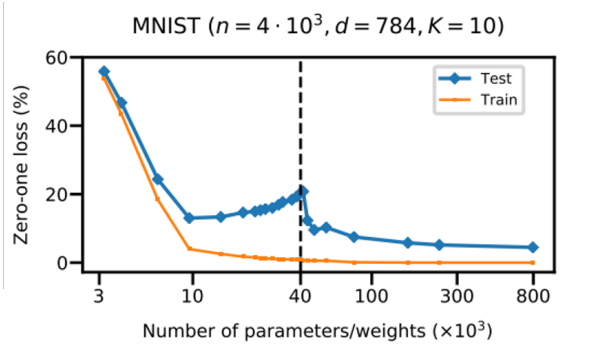


Figure 4. The result of a fully connected two-layer neural network from (Belkin et al., 2019). The network is trained using a subset ($n = 4 * 10^3, d = 784, K = 10$) of MNIST dataset with weight resuse before interpolation threshold and random initialization after it.

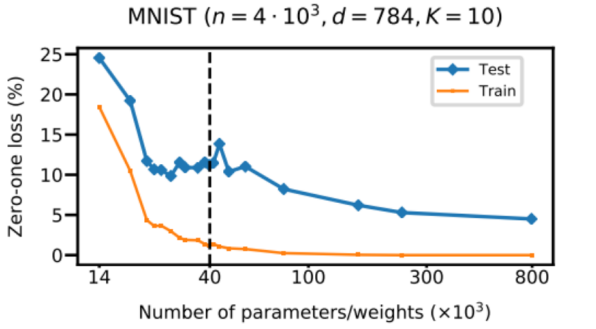


Figure 5. The result of the fully connected two-layer neural network from (Belkin et al., 2019). The network is trained using a subset ($n = 4 * 10^3, d = 784, K = 10$) of MNIST dataset without weight resuse.

We trained a similar neural network on a subset of MNIST without using weight reuse techinque. Our result is shown in figure 6. Similar to the original result in 5, our result shows roughly the same trend as the one using the weight reuse techinque, but it is more blurry around the interpolation threshold. This is because stochastic gradient descent using in the neural network is sensitive to the initialization point

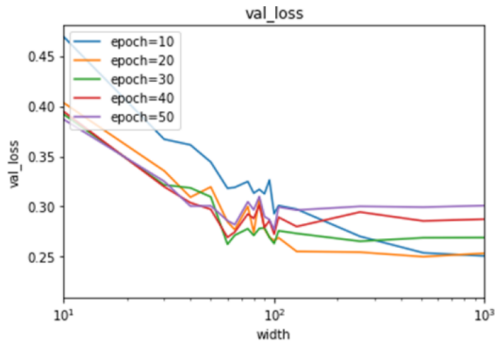


Figure 6. Our result of RFF trained using a subset ($n = 10^4$) of MNIST dataset. It shows a similar trend as the original result.

in nature. The weight reuse technique helps to mitigate the sensitivity.

3. Deep Double Descent

This section we present empirical evidence from literature as well as our experiment to show that double descent is a robust phenomenon occurs under variety of deep learning settings. From these experiment results, effective model complexity (EMC) as a generalized variable is investigated. Results show that the EMC depends not only on architecture of the model as the second part concluded, but also on training data distribution and the training procedure itself. For example, adding more parameter and increasing training time will both increase EMC. And double descent can be observed only when EMC is larger than the number of samples. Besides model-wise double descent introduced in the previous section, epoch-wise and sample-wise double descent phenomenon are presented.

3.1. Experiment Setup

In our experiments, two families of architectures are investigated: ResNets and standard CNNs, we also include the Transformer results from literature. For ResNet, we developed our Keras (Chollet et al., 2015) routine based on a github repo[2]. We have 4 ResNet blocks, each one consists of two BatchNorm-ReLU-Convolution layers. The widths of each blocks are [k; 2k; 4k; 8k] respectively. The maximum width we tested is standard ResNet18 that corresponds to $k = 64$. For strander CNNs, we set up our a 5-layer CNNs, with 4 convolutional layers of widths [k; 2k; 4k; 8k] for varying k and a fully-connected output layer. Each convolutional layer consists of Conv-BatchNorm-ReLU-MaxPool layers. The Maxpool is [1, 2, 2, 8], kernel size = 3, stride = 1 and padding = 0. We trained both ResNets and standard CNNs with Adam optimizer with learning-rate 0.0001.

We also added label noise to the training data, p in label noise means training on samples which have the correct label with probability $(1-p)$, and uniformly random incorrect label with probability p.

[2]https://github.com/raghakot/keras-resnet

3.2. Model-wise double descent

We first study the risk curve of models of increasing size. We demonstrate model-wise double descent across different architectures, datasets, and training procedures. Figure 7 shows training results with varying model size using ResNet18 and CNN for 4000 epochs, using optimizer Adam. We can see for both architecture, there is a peak at $k = 12$ for ResNet and $k = 10$ for CNN. We also conducted experiment with similar training setup (figure 10), in our cases, peak in both architectures occurs at width ($k = 12$ and $k = 15$, respectively) corresponds to model size 60000. This is inline with the hypothesis that the double descent peak occurs at $EMC = n$ and proves that such feature is robust across different classifiers. Moreover, we examined the risk curve in CNN with different noise level present in the training data (Figure 8) and shows similar trend. This proves that besides model size, EMC can also be affected by the structure of the input training data.

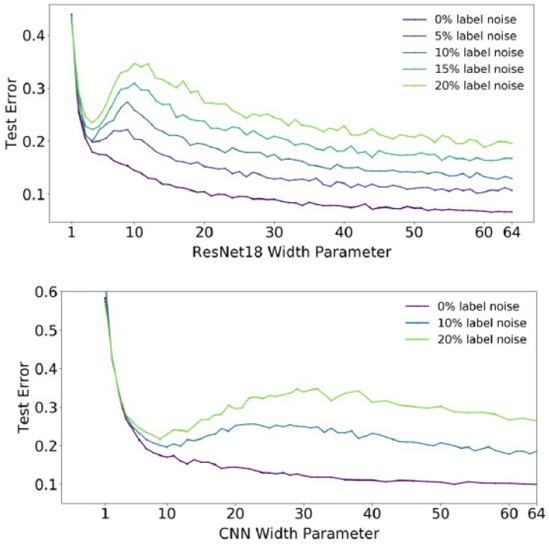


Figure 7. Result of ResNet18 and CNN on cifar10 in (Nakkiran et al., 2019). Different colors represent labelled noise level in each training data input.

3.3. Epoch-wise double descent

This section we discussed a novel form of double-descent in terms of training epochs. The primary conclusion is that the

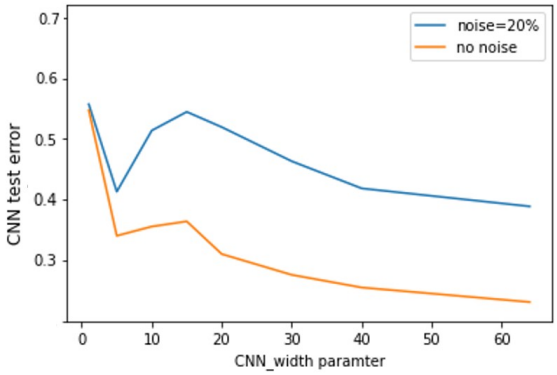


Figure 8. Our result in CNN network with different noise level by varying network size, there is similar behavior but wee do need finer steps in width for a conclusive trend.

EMC is also correlated with how long the model is trained. As shown in figure 9, for a standard ResNet18 model, when width k is sufficiently large (k=64 corresponds to 1 million parameters), the test error of this model has a peak at $\sim$100 epochs, then decreases over longer training time and have best performance at 2000 epochs, showing a double descent behavior. However, for a smaller model shown as the green and yellow line, the best result does not correspond to the longest training time, these models will achieve their best result at lower epochs, after that the test error will only go up as the training time increases, which means the yellow and green line are following the conventional bias-variance trade-off curve. Our own experiment on both ResNet18 and CNN shown in figure 10 has the same result: for smaller size models, the best result is achieved at lower training epochs, as the blue solid lines shown in both plots, when the model becomes bigger, the test error will also have epoch-wise double descent.

3.4. Sample-wise double descent

Here we illustrate that with double descent present in the critical interval, how does changing the input sample can adversely impact the performance of the model. Intuitively, we come to understand that adding more training sample will have two effects: 1) preventing overfitting by reducing variance, essentially shrink the area below the test error curve. 2) as the input sample size n increases, since $EMC = n$, the interpolation threshold, which is the peak of the double descent curve will also be shifting rightward. Outside of critical interval, the first effect usually prevails and lower the test error overall, but in the critical interval, these effect can cancel out with each other and sometimes the second effect will be dominant. As Figure 11 shows, for the dark blue curve, quadrupling samples will increase loss around the interpolation threshold. We will analyze the sample-wise

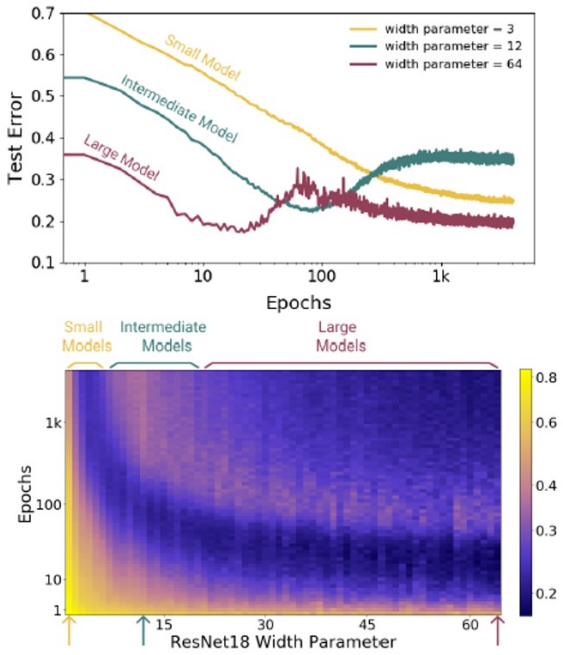


Figure 9. Top: Training dynamics from (Nakkiran et al., 2019) for models with varying width. Bottom: Test error over epochs. Different slices of this plot are shown on the top.

double descent in detail in the next section.

4. Sample-wise Double Descent

The sample-wise double descent phenomenon is also observed in linear regression problems (Nakkiran, 2019), which provided a case study in a simple linear regression setting that demonstrates more data can actually hurt the performance of the estimator. The double descent behavior in linear regression problems has also been analyzed by some other work such as (Hastie et al., 2019) and (Mei & Montanari, 2020), but Nakkiran's paper provides a well-specified problem setting and focus on the sample-wise behavior. This section discusses the phenomenon and theoretical analysis from Nakkiran's paper, as well as our own experiment results.

4.1. Problem Setup

The problem setting from the Nakkiran's paper is as follows. We have a distribution $\mathcal{D}$ that is $(x, y) \sim \mathbb{R}^d \times \mathbb{R}$, where $x \sim \mathcal{N}(0, I_d)$ and $y = \langle x, \beta \rangle + \mathcal{N}(0, \sigma^2)$. Note that β is unknown satisfying $\|\beta\|_2 \leq 1$. Given n samples of (x_i, y_i), the goal is to learn an estimator $\hat{\beta}$ with small test MSE $\mathcal{R}(\hat{\beta}) = \mathbb{E}[(\langle x, \hat{\beta} \rangle - y)^2]$. Assume that we do gradient descent from 0 on the objective $\min_{\hat{\beta}} \|X\hat{\beta} - y\|^2$,

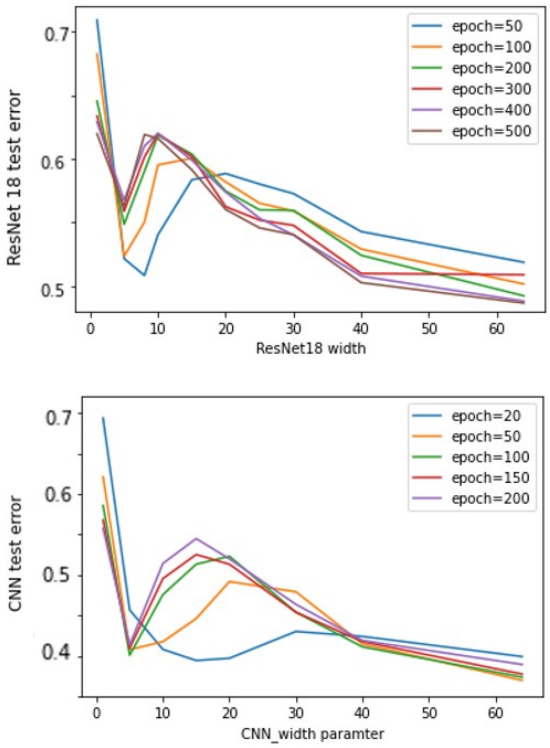


Figure 10. Our result with varying model size trained with ResNet18 and CNN.

where $X \in \mathbb{R}^{n \times d}$ is the samples x_i stacked together, and y are the corresponding observations stacked, the solution at convergence would be $\hat{\beta} = X^{\dagger} y$.

4.2. Experiment and Analysis

The solution $\hat{\beta} = X^{\dagger} y$ actually has different forms based on the ratio between the number of samples n and the sample dimension d:

$$\hat{\beta} = X^{\dagger} y = \begin{cases} \operatorname{argmin}_{\beta : X\beta = y} \|\beta\|^2, & \text{if } n \leq d \\ \operatorname{argmin}_{\beta} \|X\beta - y\|^2, & \text{otherwise} \end{cases} \quad (1)$$

When $n \leq d$, the model is overparameterized, and there are more than one β that minimizes the objective. In this regime, the gradient descent actually finds the one with smallest l_2 norm. But when $n \geq d$, the model becomes underparameterized, and there exists a unique minimizer. Write the minimizer as $\hat{\beta} = X^{\dagger} y = X^{\dagger}(X\beta + \eta) = X^{\dagger} X\beta + X^{\dagger}\eta$. We see that it consists of two terms, signal and noise. When $n = d$, there is exactly one $\hat{\beta}$ that minimizes the objective, which may have high norm that fits the noise term $X^{\dagger}\eta$.

The following analysis from Nakkiran's paper provides insight of how the bias-variance tradeoff happens in this case study. Consider the excess risk of $\hat{\beta}$ which helps to omit the additive error:

$$\bar{\mathcal{R}}(\hat{\beta}) = \|\hat{\beta} - \beta\|^2$$

For and estimator derived from samples $(X, y) \sim \mathcal{D}^n$, the expected excess risk is:

$$\mathbb{E}_{X,y}[\bar{\mathcal{R}}(\hat{\beta}_{X,y})] = \|\beta - \mathbb{E}[\hat{\beta}]\|^2 + \mathbb{E}[\|\hat{\beta} - \mathbb{E}[\hat{\beta}]\|^2]$$

Denote the first term as B_n, and the second term as V_n. They are actually bias and variance of the estimator on n samples. These terms can be approximately computed, and the value aligns with the experiment result from the paper, as shown in figure 12. The theoretical result and the experimental result both shows double descent behavior occurs with increasing number of samples.

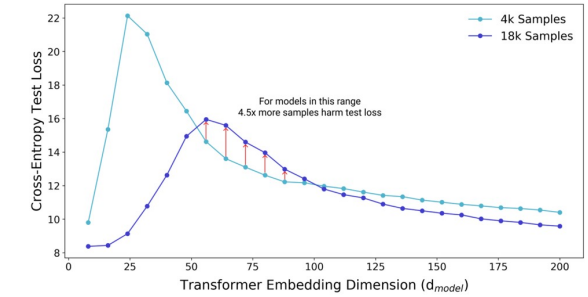


Figure 11. Test loss from (Nakkiran et al., 2019) (per-token perplexity) as a function of Transformer model size (embedding dimension d) on language translation (IWSLT'14 German-to-English).

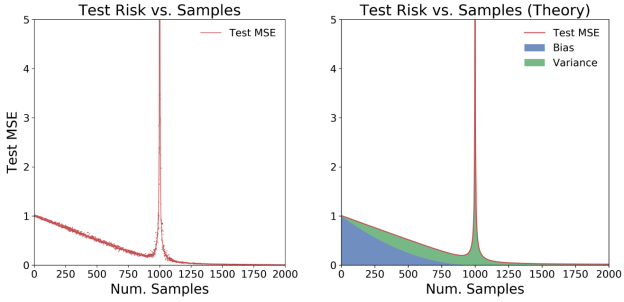


Figure 12. Sample-wise double descent phenomenon in a linear regression problem (Nakkiran, 2019). The parameters are $d = 1000, \sigma = 0.1$. *Left*: The test MSE. *Right*: The test MSE in theory.

The paper presents detailed computation of B_n and V_n that helps understanding the behavior. Let $\gamma = \frac{n}{d}$ be the ratio

of number of samples over the sample dimension. The overparameterized case corresponds to $\gamma < 1$. The bias and the variance can be computed as follows:

$$B_n = (1-\gamma)^2 \|\beta\|^2 \tag{2}$$

$$V_n \approx \gamma(1-\gamma)\|\beta\|^2 + \sigma^2 \frac{\gamma}{1-\gamma} \tag{3}$$

Obviously, in the overparameterized regime, when γ increases towards 1, the bias monotonically decreases to 0, but the variance first decreases and then increases monotonically, until it diverges at $\gamma = 1$. This corresponds to the peak of the error.

When $\gamma > 1$, it corresponds to the underparamterized regime, where we have more samples than the sample dimension. The bias and the variance can be computed as follows:

$$B_n = 0 \tag{4}$$

$$V_n \approx \frac{\sigma^2}{\gamma - 1} \tag{5}$$

As shown in figure 12, the theoretical result still matches the experimental result. In the underparameterized regime, the bias is always zero, while the variance monotonically decreases and converges to zero as γ approaches infinity.

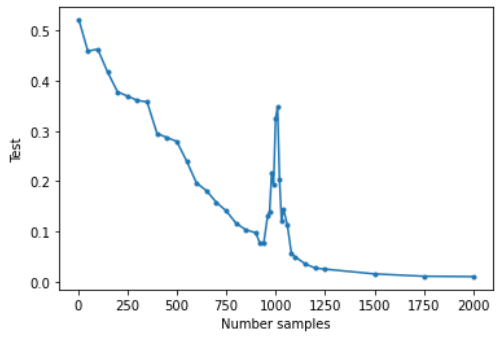


Figure 13. Our result showing similar sample-wise double descent phenomenon in the same linear regression problem discussed in (Nakkiran, 2019). The sample dimension is set to 1000. The x-axis is the number of samples, and the y-axis is the test error.

Figure 13 shows our result on the same problem setting. The β used for generating the random samples should be different from the paper, as the paper does not provide the value. As shown in the figure, the threshold is achieved at $n = d$ as expected. Note that the peak value is not as high as the original result in the paper. From the theoretical analysis above, we know that it diverges at $n = d$. Since it does not converge, we set the timeout to be 1 hour and obtained the result showing in the graph. If we let it continue to run, it should approach infinity as figure 12.

References

Belkin, M., Hsu, D., Ma, S., and Mandal, S. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019. ISSN 0027-8424. doi: 10.1073/pnas.1903070116. URL https://www.pnas.org/content/116/32/15849.

Chollet, F. et al. Keras, 2015. URL https://github.com/fchollet/keras.

Duin, R. P. W. Classifiers in almost empty spaces". *Proceedings of the 15th International Conference on Pattern Recognition (IEEE, 2000), vol. 2, pp. 1–7*, 2000.

F. Vallet, J.-G. Cailton, P. R. Linear and nonlinear extension of the pseudo-inverse solution for learning boolean functions. *Europhys. Lett. 9, 315–320*, 1989.

Hastie, T., Montanari, A., Rosset, S., and Tibshirani, R. J. Surprises in high-dimensional ridgeless least squares interpolation. *arXiv preprint arXiv:1903.08560*, 2019.

M. Opper, W. Kinzel, J. K. R. N. On the ability of the optimal perceptron to generalise. *J. Phys. A Math. Gen. 23, L581–L586*, 1990.

Mei, S. and Montanari, A. The generalization error of random features regression: Precise asymptotics and double descent curve, 2020.

Nakkiran, P. More data can hurt for linear regression: Sample-wise double descent. *arXiv preprint arXiv:1912.07242*, 2019.

Nakkiran, P., Kaplun, G., Bansal, Y., Yang, T., Barak, B., and Sutskever, I. Deep double descent: Where bigger models and more data hurt. *arXiv preprint arXiv:1912.02292*, 2019.

T. L. H. Watkin, A. Rau, M. B. The statistical mechanics of learning a rule. *Rev. Mod. Phys. 65, 499.*, 1993.

Different theory techniques for shallow neural networks

Han Guo [*1] **Yikai Liu** [*1]

Abstract

We give an literature review on different theory techniques for shallow neural networks. We specifically focus on limited number of fully connected layer and convolutional layer with standard activation function and pooling. Our literature review will cover three topics: 1) how different optimization techniques affect training performance on shallow neural network, 2) how different architectures of shallow neural network differ in optimizations, 3) what assumptions/conditions are held in papers of interest. While we do not propose new methodology and analysis, our review provide some insights on different neural network settings and convergence analysis by quantitatively and qualitatively cross examining state-of-the-art work in relevant area.

1. Introduction

The stunning performance of deep learning which is backboned by neural network has made itself a promising topic in machine learning area. Deep learning performs well on variety kinds of problems including object detection and natural language processing. Neural network even outperforms human in image classification competition on ImageNet dataset, with human scored 94.9% (Russakovsky et al., 2015) and a well trained deep neural network scored 95.06% (He et al., 2015b). Despite its extraordinary empirical success, however, theoretical reasoning on why deep neural network works so well has remain relatively less well understood(Soltani & Hegde, 2019), though tremendous efforts are made trying to unveil its mystery. The significant difficulty that hinders from developing thorough and generalized theoretical aspect of deep neural network comes from highly non-convex nature of optimization posed by neural networks.(Goel et al., 2018) Empirically, it demonstrated that neural networks with more layers ("deep" learning) are essential for better performance. However, due to its non-convex optimization, it is hard to tackle with deep neural network directly; instead, focusing on shallower neural network might provide some insightful discoveries that serve as stepping stones to understand deeper and more complex models. Nevertheless, (Blum & Rivest, 1989) has proved that, without any constraints, training on shallow neural network can be NP-Complete. Thus, many works have provided convergence analysis with certain constraints to reduce the workload in reasonable sense (Brutzkus & Globerson, 2017; Jagatap & Hegde, 2018; Soltanolkotabi, 2017; Blum & Rivest, 1989; Zhang et al., 2018; Du et al., 2019; 2018a; Hardt & Ma, 2018; Du et al., 2018b; Goel et al., 2018).

The main motivation of our paper is to introduce some state-of-the-art theoretical analysis of different optimization techniques for shallow neural network. In particular, we will cover three aspects:

- How do different optimization techniques (e.g. Gradient Descent and Alternating minimization) affect the performance of shallow neural network.
- How do different neural network architectures (e.g. Conv block and fully connected layer) affect corresponding convergence analysis .
- How do conditions and assumptions (e.g. input distribution and weight initialization) differ from different theoretical works, and how do those conditions affect the convergence analysis.

The rest of this paper is structured as follow: Section 2 will provide necessary textual definitions and mathematical notations and other background information, Section 3 will focus on individual aspect by analytically cross examining variety of SOTA manuscripts, and Section 4 will proceed to discussion on current research works and future directions.

2. Background

In this section, we will briefly introduce key concepts in shallow neural network and optimization. We will then proceed to the problem setup in the context of optimization in next section. In the architecture subsection, we will provide mathematical expressions for fully connected layers, convolutional layers, residual block, and activation layers. In

[*]Equal contribution [1]Department of Computer Science, Rice University. Correspondence to: Han Guo <hg31@rice.edu>, Yikai Liu <yl163@rice.edu>.

Follow ICML 2020 format Copyright 2020 by the author(s).

optimization technique subsection, we will provide definitions of gradient descent and alternating minimization; more proposed techniques will be introduced in next section.

2.1. Architecture

The concept of neural network feed-forward pass is simple enough; here we provide the definitions for some variations of shallow neural networks. For the sake of simplicity, we do not include structures other than convolutional layer, fully connected layer, activation layer, input layer, and output layer.

2.1.1. Fully connected layer

Definition 2.1. (Zhang et al., 2018) A typical one-hidden-layer neural network (one input layer, one hidden layer, one output layer) has the following form:

$$y_i = \sum_{j=1}^{K} \sigma((\mathbf{w}_j^*)^\top \mathbf{z}_i) + \epsilon_i \tag{1}$$

Here, $\mathbf{w}_j^* \in \mathbb{R}^d$ is the weight parameter with respect to the j-th neuron, $\sigma(x)$ denotes activation function, $\{x_i\}_i^N \subseteq \mathbb{R}^d$ denotes input, $\{y_i\}_i^N \subseteq \mathbb{R}^d$ denotes output, and $\{\epsilon_i\}_i^N \subseteq \mathbb{R}^d$ denotes noise.

The above expression provides a general structure of shallow neural network though, variation exists. One structural variation based on equation 1 that is adopted in (Soltani & Hegde, 2019; Du et al., 2018b;c).

Definition 2.2. (Soltani & Hegde, 2019)

$$\hat{y} = \sum_{j=1}^{r} a_j \sigma(w_j^\top x) = \sum_{j=1}^{r} a_j \langle w_j, x \rangle^2 \tag{2}$$

here, the network comprises p input nodes, a single hidden layer with r neurons with activation function $\sigma(x)$, weights $\{w_j\}_{j=1}^r \subset \mathbb{R}^p$, and the single node output layer with weights $\{a_j\}_{j=1}^r \subset \mathbb{R}$.

An multilayer fully connected neural network can be generalized as a variation from section 3.3 of (Du et al., 2019) without normalization factor.

Definition 2.3. (Du et al., 2019) $x^{(h)} = \sigma\left(\mathbf{W}^{(h)}\mathbf{x}^{(h-1)}\right)$, $1 \le h \le H$

$$f(\mathbf{x}, \theta) = \mathbf{a}^\top \mathbf{x}^{(H)} \tag{3}$$

here, $\mathbf{x} \in \mathbb{R}^d$ denotes input, $\mathbf{W}^{(1)} \in \mathbb{R}^{m \times d}$ denotes the first weight matrix, $\mathbf{W}^{(h)} \in \mathbb{R}^{m \times m}$ denotes the h-th layer's weights, $\mathbf{a} \in \mathbb{R}^m$ denotes the output layer, and $\sigma(x)$ denotes the activation function.

2.1.2. Convolutional layer

The convolutional layer creates "patches", which complicates the mathematical notation.

Definition 2.4. (Goel et al., 2018) Convolutional layer with overlapping patches is computed as follows (we exclude average pooling which appeared in original equation):

$$f_w(x) = \sum_{i=1}^{k} \sigma(w^\top P_i x) \tag{4}$$

here, $x \in \mathbb{R}^n$ denotes the input, the neural network computes k patches of size r where the location of each patch is indicated by matrix $P_1, \cdots, P_k \in 0, 1^{r \times n}$ and each P_i has exactly one 1 in each row and at most one 1 in every column, $\sigma(x)$ denotes the activation function, and $w \in \mathbb{R}^r$ denotes the weight vector of convolution filter.A special case of convolutional layer arises when the patches do not overlap. This results an easier analysis which can be found in Section 3. (Du et al., 2018b) has a slightly different definition of convolutional layer, though the key concepts align with equation 4.

2.1.3. Activation layer

Activation layer provides non-linear features to the model, which increase the expressiveness (Raghu et al., 2017). One of the most widely used activation function is Rectified Linear Unit (ReLU), and it's defined as follows.

Definition 2.5. (Soltanolkotabi, 2017) ReLU preserves the positive values and set all negative values to 0s

$$\sigma(x) = \max(0, \langle \mathbf{w}, \mathbf{x} \rangle) \tag{5}$$

here, $\mathbf{x} \in \mathbb{R}^{d \times n}$ denotes input, and $\mathbf{w} \in \mathbb{R}^d$ denotes weights.
An derivation from ReLU is called Leaky ReLU, and its definition follows.

Definition 2.6. (Goel et al., 2018) Instead turning all negative values to 0s, Leaky ReLU scales those values by some factors α

$$\sigma(x) = \begin{cases} x & \text{if } x \ge 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{6}$$

here, $\alpha \in [0, 1]$.

Another activation function that does not belong to ReLU family is quadratic activation.

Definition 2.7. (Soltani & Hegde, 2019) Quadratic activation is not as commonly used as ReLU,

$$\sigma(x) = x^2 \tag{7}$$

but it has shown competitive expressive power as well(Livni et al., 2014).

2.1.4. RESIDUAL CONNECTION

ResNet utilizes this structure achieves outstanding image classification scores. Residual connection enables neural networks to go deeper without worrying about vanishing gradient problem (He et al., 2015a). The key observation here is that the neural network learns the identity mapping.

Definition 2.8. (Li & Yuan, 2017) The following function contains identity mapping

$$f(x, \mathbf{W}) = \left\|\sigma((\mathbf{I} + \mathbf{W})^\top x)\right\|_1 \tag{8}$$

here, $x \in \mathbb{R}^d$ denotes input vector, $\mathbf{W} \in \mathbb{R}^{d\times d}$ denotes weights, and $\mathbf{I}$ is the identity matrix. This representation is equivalent to $\mathcal{H}(x) = \mathcal{F}(x) + x$(He et al., 2015a).

A generalized multilayer residual connection takes the following expression

Definition 2.9. (Hardt & Ma, 2018)

$$\hat{y} = (\mathbf{I} + \mathbf{A}_l)\cdots(\mathbf{I} + \mathbf{A}_1)x \tag{9}$$

where $\mathbf{A}_1 \cdots \mathbf{A}_l \in \mathbb{R}^{d\times d}$ denotes weights, and $\mathbf{I}$ is the identity matrix.

2.2. Optimization

2.2.1. GRADIENT DESCENT

Gradient descent perhaps is the most commonly used optimization technique in deep learning. The following describes gradient decent optimization.

Definition 2.10. (Soltanolkotabi, 2017) The following expression is a variation from original projected gradient descent

$$\mathbf{w}_{\tau+1} = \mathbf{w}_\tau - \eta\nabla\mathcal{L}(\mathbf{w}_\tau) \tag{10}$$

here, $\mathbf{w}_{\tau+1}$ denotes $\tau+1$-th weights, $\mathbf{w}_\tau$ denotes τ-th weights, η denotes stepsize, and $\nabla\mathcal{L}(w_\tau)$ denotes loss function which will be introduced in Subsection 2.3. Other problem-specific modification on gradient descent concept will be specified in Section 3.

2.2.2. ALTERNATING MINIMIZATION

Another less common optimization technique is alternating minimization. The core concept behind this technique is to always treat one unknown as variable and alternating the process of choosing unknown and minimizes it. This technique is problem-specific which will be introduced in detail in Section 3.

2.3. Loss function

In deep learning optimization, a function used to evaluate a candidate solution is referred as loss function or objective solution. Canonically, minimization operation is performed on a loss function, meaning that we are searching for a candidate solution that has the highest score (the score is calculated by comparing the predicted value and ground-truth label). Section 2.3.1 introduces empirical risk minimization, and Section 2.3.2 introduces population loss minimization.

2.3.1. EMPIRICAL RISK MINIMIZATION

In practice, we do not have access to the true distribution of data that we are working on; however, we have access to some amount of data, and we are trying to approximate the loss of entire population with the limited access of sampling data. The following defines a typical empirical loss of a simple feedforward function with only weights and activation.

Definition 2.11. (Soltanolkotabi, 2017) The least-squares empirical loss is a variation from the original expression in (Soltanolkotabi, 2017)

$$\min_{\mathbf{w}\in\mathbb{R}^d} \mathcal{L}(\mathbf{w}) := \frac{1}{2}(f(\mathbf{w}, \mathbf{x}) - y)^2 \tag{11}$$

here, $f(\mathbf{w}, \mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}\sigma(\mathbf{w}^\top\mathbf{x}_i)$ denotes a feedforward function with weights $\mathbf{w}$, input data $\mathbf{x}_i$, activation function $\sigma(x)$, and y is ground truth label.

2.3.2. POPULATION LOSS

In an ideal case, if we have access to true distribution of data that we are working on, we can assume a ground-truth global minimum weight $\mathbf{w}^*$. This assumption simplifies some of the convergence analysis. The following defines a typical population loss of a simple feedforward function with only weights and activation.

Definition 2.12. (Du et al., 2018a) A least-squares population loss is a variation from the original expression in (Du et al., 2018a)

$$\min_{\mathbf{w}\in\mathbb{R}^d} \mathcal{L}(\mathbf{w}, \mathbf{x}) := \frac{1}{2}(f(\mathbf{w}, \mathbf{x}) - f(\mathbf{w}^*, \mathbf{x}))^2 \tag{12}$$

here, $f(\mathbf{w}, \mathbf{x})$ follows the same definition as in Definition 2.11, and $f(\mathbf{w}^*, \mathbf{x})$ in equation 12 has ground-truth weights w^*.

Section 2.3.1 and 2.3.2 denotes empirical loss and population loss in general form; however, depending on the problem setup, some adaptations might be necessary. Detailed modification will be noted in Section 3 as needed.

So far, we have provided some basic notations that will be utilized in next section. We will then proceed to our analysis on three main aspects.

3. Methodology

In this section, we are trying to answer the questions that we have brought up in Section 1 by quantitatively and qualitatively reviewing relevant literature works on optimization with different settings. In Section 3.1, we will discuss how Gradient Descent and other optimization techniques. In Section 3.2, we will analyze how different neural network architectures result in different convergence analysis. In Section 3.3, we will discuss on some common assumptions and conditions that either loose or constrain the applicability and generalizability of convergence analysis, and how does those conditions help to develop theoretical bounds/constraints.

3.1. Different optimization techniques

3.1.1. Gradient-based optimization

In Section 2.2.1, we have briefly introduced the basic concept of gradient descent in which weights are updated in the direction where the loss is maximally minimized. A line of research focusing on the behavior of gradient-based algorithm has shown its power. (Tian, 2017b) use a variant of gradient descent algorithm – population gradient descent where instead of empirical loss, population loss is considered. The population gradient takes the following form.

Definition 3.1. (Tian, 2017b)If we assume population loss, then population gradient $\mathbb{E}_X[\nabla J_w(\mathbf{w})]$ with population loss $J(\mathbf{w}) = \frac{1}{2}\|g(X;\mathbf{w}^* - g(X;\mathbf{w}))\|^2$ where $g(X;\mathbf{w}) = \sum_{j=1}^{K}\sigma(\mathbf{w}_j^\top x)$ with respect to weight $\mathbf{w}_j$ has the expression

$$\mathbb{E}_X[\nabla_{\mathbf{w}_j} J] = \sum_{j'=1}^{K}\mathbb{E}[F(e_j,\mathbf{w}_{j'})] - \sum_{j'=1}^{K}\mathbb{E}[F(e_j,\mathbf{w}^*_{j'})] \tag{13}$$

here, $e_j = \mathbf{w}_j/\|\mathbf{w}_j\|$. Then simply substituting $\nabla\mathcal{L}(\mathbf{w}_\tau)$ in equation 10 with equation 13, we will have the final population gradient descent

$$\mathbf{w}_{\tau+\mathbf{1}} = \mathbf{w}_\tau - \eta\nabla\mathbb{E}_X[\nabla_{\mathbf{w}} J] \tag{14}$$

Under the conditions of spherical Gaussian input and randomized weight initialization, (Tian, 2017b) has proved that one-layer one neuron model is able to recover true weight vector.

(Soltanolkotabi, 2017) improves this result by using empirical loss that looses the constraint. Particularly,

$$n_0 = \mathcal{M}((R),\mathbf{w}^*) = \omega^2(\mathcal{C}_R(\mathbf{w}^*)\cap\mathcal{B}^d) \tag{15}$$

defines n_0 to be the exact minimum number of samples required, where $\mathcal{C}_R(\mathbf{w}^*)$ is a cone descent of a regularizer function $\mathcal{R}$ at $\mathbf{w}^*$ and $\mathcal{B}^d$ denotes unit ball of $\mathbb{R}^d$, then the empirical projected gradient descent can be defined as follow.

Definition 3.2. (Soltanolkotabi, 2017) Let $\nabla\mathcal{L}(\mathbf{w}_\tau)$ be empirical loss function, then empirical projected gradient descent is

$$\mathbf{w}_{\tau+\mathbf{1}} = \mathcal{P}_k(\mathbf{w}_\tau - \eta\nabla\mathcal{L}(\mathbf{w}_\tau)) \tag{16}$$

here, η denotes the step size and $\mathcal{K} = \{\mathbf{w}\in\mathbb{R}^d : \mathcal{R}(\mathbf{w}\le\mathcal{R})\}$ is the constraint set with $\mathcal{P}_\mathcal{K}$ denoting the Euclidean projection onto this set. The key theorem here is that if equation 15 is satisfied, then equation 16 obey $\|\mathbf{w}_\tau - \mathbf{w}^*\|_F \le \left(\frac{1}{2}\right)^\tau\|\mathbf{w}^*\|_F$. This theorem shows that with near minimal number of data sample n_0, projected gradient descent learns ground-truth weight with linear convergence rate. This result also applies to both convex and nonconvex regularization functions, and it shows that with near minimal number of data samples, project gradient descent converges without getting trapped in bad local optima.

3.1.2. Alternating minimization optimization

Alternating minimization is another optimization technique proposed by (Jagatap & Hegde, 2018). On the high level, the idea of alternating minimization is to estimate the activation patterns of each ReLU for all given samples and interleave with weight updates via a least-squares loss.

Specifically, they linearize all samples by defining *state* of the neural network as the collection of binary variables that indicates whether ReLU is active or not and fixing that state. This idea is inspired by the feature of ReLU that positive values of ReLU will remain their weights and negative values will be clipped to 0, so we can separate the value of weights to an indicator matrix and values of weights. This process can be linearized. Let

$$B = [\mathrm{diag}(p_1)X...\mathrm{diag}(p_k)X]_{n\times dk} \tag{17}$$

be linearized state of ReLU where $p_i = \mathbb{1}_{\{Xw_i>0\}}$ denotes the indicator function for sign of weights. Then

Definition 3.3. (Jagatap & Hegde, 2018)Feedforward function can be expressed as

$$f(X) = \sum_{i=1}^{k}\mathrm{ReLU}(Xw_i) = B\cdot\mathrm{vec}(W) \tag{18}$$

here, $\mathrm{vec}(W)$ vectorize weight W. And the minimization update can be thus described as

$$\mathrm{vec}(W)^{t+1} = \underset{\mathrm{vec}(W)}{\arg\min}\left\|B^t\cdot\mathrm{vec}(W) - y\right\|_2^2 \tag{19}$$

Thus, by alternating equation 17 and equation 19 , alternating minimization converges to global minimum with linear convergence rate if the initial weight W^0 satisfying $\mathrm{dist}(W^0,W^*) \le \delta_1\|W^*\|_F$ for $0<\delta_1<1$, where $\mathrm{dist}(W,W')$ is defined as

$$\mathrm{dist}(W,W') = \min_{\text{all possible of column perturbations}}\|W - W'\|_F$$

Algorithm 1 Alternating Minimization

Require: X, y, T, k
Initialize W^0 s.t. $\text{dist}(W^0, W^*) \leq \delta_1 \|W^*\|_F$
for $t = 0, \cdots, T-1$ **do**
$p_q^t = \mathbb{1}_{\{Xw_q^t > 0\}}, \forall q \in \{1...k\}$
$B^t = [\text{diag}(p_1^t)X ... \text{diag}(p_k^t)X]_{n \times dk}$
$\text{vec}(W)^{t+1} = \arg\min_{\text{vec}(W)} \left\| B^t \cdot \text{vec}(W) - y \right\|_2^2$
$W^{t+1} \leftarrow \text{reshape}(\text{vec}(W)^{t+1}, [d, k])$
end for
Return $W^T \leftarrow W^t$

. This weight initialization can be obtained by set $W^0 \leftarrow \mathbf{I}$. The detailed implementation is shown in Algorithm 1.

3.1.3. Low rank matrix estimation

Though theoretical aspect of neural network is not well understood, there are areas that we do have enough theoretical findings. One natural idea is to bridge the problem of learning (shallow) neural network with a well understood problem of low-rank matrix estimation. Specifically, the problem of learning a shallow neural network can be treated as a low-rank matrix estimation problem where the rank of the resulting matrix equals to the number of hidden neurons. The following definition provides a problem setup the network of our interest.

Definition 3.4. (Soltani & Hegde, 2019) The network of our interest consists p input nodes, a single hidden layer with r neurons with quadratic activation function $\sigma(z) = z^2$, first layer weights $\{w_j\}_{j=1}^r \subset \mathbb{R}^p$, and an output layer comprising of a single node and weights $\{a_j\}_{j=1}^r \subset \mathbb{R}$, then the input-output relation can be expressed as the following

$$\hat{y} = \sum_{j=1}^{r} a_j \sigma(w_j^T x) = \sum_{j=1}^{r} a_j \langle w_j^T x \rangle^2 \tag{20}$$

Consider set of training input-output pairs $\{(x_i, y_i)\}_{i=1}^m$ and set of weights $\{(a_j, w_j)\}_{j=1}^r$. We define matrix variable $L_* = \sum_{j=1}^r a_j w_j w^T$, then input-output relation becomes

$$\hat{y}_i = x_i^T L_* x_i = \langle x_i x_i^T, L_* \rangle \tag{21}$$

here $x_i \in \mathbb{R}^p$ denotes the i_{th} training sample, L_* is a rank-r matrix of size $p \times p$, so empirical loss function

$$\min_{W \in \mathbb{R}^{r \times p}, a \in \mathbb{R}^r} F(W, a) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \tag{22}$$

can be viewed as an instance of learning a fixed rank-r symmetric matrix $L_* \in \mathbb{R}^{p \times p}$ where $r \ll p$ from small number of *rank-one* linear observation given by $A_i = x_i x_i^T$.

A few algorithms are proposed to estimate L_*,given $\{x_i, y_i\}_{i=1}^m$. The first method, called Exact Projections

Algorithm 2 EP-ROM

Inputs: y, number of iterations K, independent data samples $\{x_1^T ..., x_m^T\}$ for t = 1,....,K,rank r
Outputs: Estimates $\hat{L}$
Initialization: $L_0 \leftarrow 0, t \leftarrow 0$
Calculate: $\hat{y} = \frac{1}{m} \sum_{i=1}^m y_i$
while $t \leq K$ **do**
$L_{t+1} = P_r(L_t - \frac{1}{2m} \sum_{i=1}^m ((x_i^t)^T L_t x_i^t - y_i) x_i^t (x_i^t)^T - (\frac{1}{2m} 1^T A(L_t) - \frac{1}{2}\hat{y}) I)$,
$t \leftarrow t + 1$
end while
Return $\hat{L} = L_k$

Algorithm 3 AP-ROM

Inputs: y, number of iterations K, independent data samples $\{x_1^T ..., x_m^T\}$ for t = 1,....,K,rank r
Outputs: Estimates $\hat{L}$
Initialization: $L_0 \leftarrow 0, t \leftarrow 0$
Calculate: $\hat{y} = \frac{1}{m} \sum_{i=1}^m y_i$
while $t \leq K$ **do**
$L_{t+1} = \tau(L_t - H(\frac{1}{2m} \sum_{i=1}^m (((x_i^t)^T l_t x_i^t - y_i) x_i^t (x_i^t)^T - (\frac{1}{2n} 1^T A(L_t) - \frac{1}{2}\hat{y}) I))$,
$t \leftarrow t + 1$
end while
Return $\hat{L} = L_k$

for Rank-One Matrix, or EP-ROM, which solves the non-convex, constrained risk minimization problem:

$$\min_{L \in \mathbb{R}^{p \times p}} F(L) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - x_i^T L x_i)^2 \tag{23}$$

is demonstrated in Algorithm 2.

While EP-ROM exhibits linear convergence, the per-iteration complexity is still high since it requires projection onto the space of rank-r matrices, which necessitates the application of SVD. The total running time of EP-ROM is $O(mp^2 log(\frac{1}{\epsilon}))$Thus, a second algorithm, called Approximate Projection for Rank One Matrix estimation, or AP-ROM, is proposed, as shown in Algorithm 3.
The specific choice of approximate SVD algorithm that simulates the operators $\tau(.)$ and H(.) is flexible. AP-ROM also demonstrates linear convergence as EP-ROM. However, AP-ROM demonstrates a better running time of $O(mprlog(p)log(\frac{1}{\epsilon}))$

Note that without any assumptions on spectral norm, estimating L_* takes $\mathcal{O}(p^3 r^2)$ running time complexity, due to the calculation of SVD. However, this can be improved by replacing standard SVD with approximate heuristics such as randomized Block Krylov SVD to $\mathcal{O}(p^2 r^4 \log^2(\frac{1}{\epsilon}) \text{polylog}(p))$.

3.2. Different neural network architectures

Mathematical formulation of the problem setup depends strictly on the architectural design of neural network. For shallow neural network, most common design patterns are input layer, hidden layer, and output layer. While input and output layer are canonically invariant across different network designs, hidden layers take much versatile forms. Modern neural network architecture design mainly focus on better hidden layer construction. Here we convey three common architecture practices.

3.2.1. Fully Connected Architecture

Fully connected layer (FC) forms fundamental connection between neurons in adjacent layers. While we have briefly provide some definitions in Section 2.1.1, we have yet provide a interpretation of FC in general. The following definition defines fully connected layer in linear algebraic aspect.

Definition 3.5. Assuming $\sigma(\cdot)$ is activation operator (if assume $\sigma(x)$ to be ReLU, then $\sigma(x) = \max(0, x)$). Let $x \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$ be the i-th output, then

$$y_i = \sigma(w_1 x_1 + ...w_m x_m) \tag{24}$$

holds. Full output y is then

$$y = \begin{bmatrix} \sigma(w_{1,1}x_1 + ...w_{1,m}x_m) \\ \vdots \\ \sigma(w_{n,1}x_1 + ...w_{n,m}x_m) \end{bmatrix} \tag{25}$$

Note that since the concept of FC involves summation over all weights, in the convergence analysis, we can easily rearrange summation operator in our favor. In (Zhang et al., 2018), where the original population loss function

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\mathbb{E}_{\mathbf{X}\sim\mathcal{D}_\mathbf{X}}\left(\sum_{j=1}^{K}\sigma(\mathbf{w}_j^\top \mathbf{X}) - \sum_{j=1}^{K}\sigma(\mathbf{w}_j^{*\top}\mathbf{W})\right)^2$$

its partial derivative takes form

$$\left[\nabla\hat{\mathcal{L}}_N(\mathbf{W})\right] = \sum_{j=1}^{K}(\hat{\mathbf{\Sigma}}(\mathbf{w}_j, \mathbf{w}_k)\mathbf{w}_j - \hat{\mathbf{\Sigma}}(\mathbf{w}_j^*, \mathbf{w}_k)\mathbf{w}_j^*) - \frac{1}{N}\sum_{i=1}^{N}\epsilon_i \mathbf{x}_i \cdot \mathbb{1}\{\mathbf{w}_k^\top \mathbf{x}_i \geq 0\}$$

Where we can see that the summation is taken out as a stand alone factor. (Zhang et al., 2018) also shows that with this partial derivative as gradient update and initial weight $\mathbf{W}^0$ satisfies $\left\|\mathbf{W}^0 - \mathbf{W}^*\right\|_F \leq c\sigma_K/(\lambda\kappa^3 K^2)$, gradient descent converges to ground truth $\mathbf{W}^*$ in linear time. The weight initialization requirement can be achieved by tensor initialization that is discussed in Section 3.3.2.

3.2.2. Convolutional Filter Architecture

The defining characteristic of Convolutional Neural Network (CNN) is its convolutional layer. Unlike fully connected layer, convolutional layer relies on convolution filter, or kernal, to extract features from original input. The mathematical expression for convolutional shallow neural network is described in Section 2.4. Based on the dimension of input, we consider two general types of convolution – 1D Convolution and 2D Convolution.

Definition 3.6. (Goel et al., 2018) Consider a 1D image of dimension n. Let the patch size be r and stride be d. Let the patches be indexed from 1 and let patch i start at position $(i-1)d+1$ and be contiguous through position $(i-1)d+r$. The matrix P_i of dimension $r \times n$ of patch i looks as follows,

$$P_i = (0_{r\times((i-1)d+1)} I_r 0_{r\times(n-r-(i-1)d)}) \tag{26}$$

here $0_{a\times b}$ indicates all zero matrix of size $a \times b$, and I_r indicates identity matrix of size r. Let $k = \lfloor \frac{n-r}{d} \rfloor + 1$ The structure of P is summarized as below.

$$P_{i,j} = \begin{cases} k-a & \text{if } |i-j| = ad \\ 0 & \text{otherwise} \end{cases} \tag{27}$$

We bound extremal eigenvalue $P = \sum_{i,j=1}^{k} P_i P_j^\top$.

Definition 3.7. (Goel et al., 2018) Consider a 2D image of dimension $n_1 \times n_2$. Let the patch size be $r_1 \times r_2$ and the stride in both directions be d_1, d_2 respectively. Enumerate patches such that patch (i, j) starts at position $((i-1)d_1 + 1, (i-1)d_2 + 1)$ and is a rectangle with diagonally opposite point $((i-1)d_2+r_1, (j-1)d_2+r_2)$. Let $k_1 = \lfloor \frac{n_1-r_1}{d_1} \rfloor + 1$ and $k_2 = \lfloor \frac{n_2-r_2}{d_2} \rfloor + 1$. Let $Q_{(i,j)}$ be indicator matrix of dimension $r_1 r_2 \times n_1 n_2$ with 1 at (a, b) if ath location of patch (i, j) is b. Formally,

$$(Q_{(i,j)})_{a,b} = 1 \tag{28}$$

for all $a = pr_2 + q + 1$ for $0 \leq p < r_1$, $0 \leq q < r_2$, and $b = ((i-1)d_1 + p)n_2 + jd_2 + q + 1$ else 0. The extremal eigenvalue is bounded by $Q = \sum_{i,p=1}^{k_1}\sum_{j,q=1}^{k_2} Q_{(i,j)} Q_{(p,q)}^\top$

Notice that the above 1D Convolution and 2D Convolution make no assumption on whether patches overlap or not. In fact, if there is one patch P_1 that does not overlap with any other patches, the convergence analysis simplifies significantly because the term $P_q P_j^\top = P_j^\top P_q = 0$ for all $P_j \neq 1$; in particular, the resulting expectation of loss in non-overlapping case eliminates the bounding eigenvalue terms comparing to overlapping case due to the orthogonality exhibited by $P_q P_j^\top = P_j^\top P_q = 0$.

Another work by (Du et al., 2018a) considers patches with "close relations". Specifically, they show if the input patches

are highly correlated $\theta(Z_i, Z_j) \leq \rho$ for some small $\rho > 0$, then gradient descent with random initialization recovers the filter in polynomial time, and the stronger the correlation, the faster the convergence rate. The high level approach is to first divide input patches into 4 events, find average patch in each event, and find max and min eigenvalues respectively. Assume

$$\max_{w:\theta(w,w_*)\leq\phi} \lambda_{max}(\mathbb{E}[Z_{S_{w,w_*}} Z_{S_{w,-w_*}}^\top]) + \lambda_{max}(\mathbb{E}[Z_{S_{w,w_*}} Z_{S_{-w,w_*}}^\top]) + \lambda_{max}(\mathbb{E}[Z_{S_{w,-w_*}} Z_{S_{-w,w_*}}^\top]) \leq L_{cross}$$

here $Z_{S_{w,w_*}}, Z_{S_{w,-w_*}}, Z_{S_{-w,w_*}}$ are patch average of four events mentioned previously. The, if patch Z_i and Z_j are very similar, then joint probability density of $Z_i \in S(w, w_*)_i$ and $Z_j \in S(w, -w_*)_j$ is small and implies L_{cross} is small. If L_{cross} is small, then by

$$\|w_{t+1} - w_*\|_2^2 \leq \left(1 - \frac{\eta(\gamma(\phi_t) - 6L_{cross})}{2}\right) \|w_t - w_*\|_2^2 \tag{29}$$

we have faster convergence rate in polynomial time.

3.2.3. Residual Connection Architecture

In Section 2.1.4, we have briefly introduce the mathematical notations for residual connection network. In this section we will convey some relevant analysis that leverages this structure.

Definition 3.8. (Li & Yuan, 2017)If the loss function takes form

$$\mathcal{L}(\mathbf{W}) = \mathbb{E}_x[(\sum_i \mathrm{ReLU}(\langle w_i + w_i, x\rangle) - \sum_i \mathrm{ReLU}(\langle e_i + w_i^*, x\rangle))^2] \tag{30}$$

there exists $\gamma > \gamma_0 > 0$ such that if $x \sim \mathcal{N}(0, \mathrm{I})$, $\|\mathbf{W}_0\|_2, \|\mathbf{W}^*\|_2 \leq \gamma_0, d \geq 100, \epsilon \leq \gamma^2$, then stochastic gradient descent on $\mathcal{L}(\mathrm{W})$ will find W^* by two phases:

- Phase I, setting step size $\eta \leq \frac{\gamma^2}{G_2^2}$, potential function $g = \sum_{i=1}^d (\|e_i + w_i^*\|_2 - \|e_i + w_i\|_2)$ takes at most $\frac{1}{16\eta}$ steps to decrease to smaller than $197\gamma^2$
- Phase II, for $a > 0$ and $\forall T$ s.t. $T^a \log T \geq \frac{36d}{100^4(1+a)G_F^2}$, if $\eta = \frac{(1+a)\log T}{\delta T}$, then $\mathbb{E}\|\mathbf{W}_T - \mathbf{W}^*\|_F^2 \leq \frac{(1+a)\log T G^2}{\delta^2 T}$

The key observation here is that the residual network converges to global minimum in two phases. In Phase I, the potential function g is decreasing to a small value, and in Phase II, g remains small, so $\mathcal{L}$ is one point convex and $\mathbf{W}$ starts to converge to $\mathbf{W}^*$. The proof of Phase I is fairly simple, by introducing an auxiliary variable $s = (\mathbf{W}^* - \mathbf{W})u$; the proof of Phase II leverages Taylor expansion an controls higher order terms.

Another work done by (Hardt & Ma, 2018) gives simple proof that arbitrarily deep linear residual networks have no spurious local optima. Specifically, they suggest that it is sufficient for the optimizer to converge to critical points of the population risk since all critical points are global minima. If $\mathcal{B}_\tau = \{A \in \mathbb{R}^{l\times d\times d} : |||A||| \leq \tau\}$ where $|||A||| := \max_{1\leq i\leq j} \|A_i\|$, then

$$\|\nabla f(A)\|_F^2 \geq 4l(1-\tau)^{2l-2}\sigma_{min}(\Sigma)(f(A) - C_{opt}) \tag{31}$$

equation 31 says the gradient has fairly large norm compared to the error, which guarantees convergence if the gradient descent to a global minimum if the iterates stay inside $\mathcal{B}_\tau$. Here C_{opt} is global minimum of f, and $\|\nabla f(A)\|_F^2$ denotes Euclidean norm of $\nabla f(A)$

3.3. Conditions and assumptions

It is known that without any assumptions on the problem, the learning in neural network is NP-hard. To reduce the difficulty of convergence analysis to a more reasonable workload, most of the works we focus on have made certain assumptions on the problem setup. There are two particular conditions that prevail in current works – input distribution and weight initialization. We will discuss about different input distribution in Section 3.3.1 and different weight initialization techniques in Section 3.3.2.

3.3.1. Input distribution

Neural network deals with enormous amount of data. Since most optimization algorithms are not shift invariant, a good input data distribution is critical to obtain good results and to reduce significantly calculation time(Sola & Sevilla, 1997). Through an extensive survey, though not exhaustive, we discover that one of the most commonly used input assumptions is i.i.d Gaussian distribution.

Definition 3.9. (Li & Yuan, 2017) We say input data is in standard i.i.d Gaussian distribution, if the input vector $x \in \mathbb{R}^d$ is sampled from normal distribution $\mathcal{N}(0, I)$.

In (Li & Yuan, 2017), here the original derivative of loss function is defined as

$$\nabla L(\mathbf{W})_j = 2\mathbb{E}_x[(\sum_i \mathrm{ReLU}(\langle e_i + w_i, x\rangle) - \sum_i \mathrm{ReLU}(\langle e_i + w_i^*, x\rangle))x\mathbb{1}_{\langle e_i+w_j, x\rangle\geq 0}]$$

This indicates that the original original loss function is not well defined everywhere, and analysis is only valid for each case. However, with the assumption of input is from Gaussian distribution and some modifications of equation 13 from (Tian, 2017a), the derivative of loss function can be rewritten as

$$-\nabla L(\mathbf{W})_j = \sum_{i=1}^{d}(\frac{\pi}{2}(w_i^* - w_i) + (\frac{\pi}{2} - \theta_{i^*,j})(e_i + w_i^*) - (\frac{\pi}{2} - \theta_{i,j})(e_i + w_i) + (\|e_i + w_i^*\|_2 \sin\theta_{i^*,j} - \|e_i + w_i\|_2 \sin\theta_{i,j})\overline{e_j + w_j})$$

One key observation here is that, if we assume the condition that the input x is from the standard Gaussian distribution, the loss function is smooth and the gradient is well defined every where. This idea conforms with (Tian, 2017a) in a sense that the gradient is treated as a random variable that can be expressed in terms of the expectation.

The idea that having input data distribution simplifies and improves convergence analysis is further supported by (Brutzkus & Globerson, 2017) in *No-Overlap Networks*. In particular, they derive the problem *No-Overlap Networks* from set splitting problem and argues that its complexity is NP-complete; however, if input $x \sim \mathcal{N}(0, 1)$, *No-Overlap Networks* is upper bounded by polynomial factors. This claim comes from the observation that the if input condition is satisfied, gradient descent will stay away from the degenerate saddle point. This claim highlights the importance of input distribution being Gaussian distribution in a sense of asymptotic bounds in convergence analysis. Empirical experiments conducted by (Brutzkus & Globerson, 2017) confirms that Gaussian input trial converges to global minimum while non-Gaussian input trial gets trapped in bad local minimum.

3.3.2. Weight Initialization

Weight initialization is crucial in practice, yet we have primitive understanding on this subject(Goodfellow et al., 2016). A natural idea is to have all weights initialized to be 0s. However, this is very unfavorable because zero weights initialization causes **symmetry problem** – all hidden units are symmetric and different layers don't learn different features. Therefore, to break this symmetry, weight initialization technique is necessary. There are several popular choices of weight initialization. One common practice is random weight initialization. This technique is fairly simple – randomly initialize weight parameters with standard Gaussian distribution. (Du et al., 2018b) utilizes random weight initialization and proves that gradient descent learns one-hidden-layer convolutional neural network with non-overlapping patches in polynomial time.

Another weight initialization technique is to randomly initialize weights with $O(1/\sqrt{d})$. This technique is commonly known as "Xavier initialization"(Glorot & Bengio, 2010) or "He initialization"(He et al., 2015a). The difference between "He initialization" and "Xavier initialization" is trivial, and we will not expand more on this topic due to the scope of our paper. (Li & Yuan, 2017) utilizes this initialization technique and leverages the fact that spectral norm of random matrix is $O(1)$. This result justifies $\|W^*\|_2 = O(1)$.

Algorithm 4 Tensor Initialization

procedure INITIALIZATION(set S)
$S_2, S_3, S_4 \leftarrow$ PARTITION$(S, 3)$
$\hat{P}_2 \leftarrow \mathbb{E}_{S_2}[P_2]$
$V \leftarrow$ POWERMETHOD$(\hat{P}_2, k)$
$\hat{R}_3 \leftarrow \mathbb{E}_{S_3}[P_3(V, V, V)]$
$\{\hat{u}\}_i i \in [k] \leftarrow$ KCL$(\hat{R}_3)$
$\{\hat{u}_i\}_{i\in[k]} \leftarrow$ RECMAGSIGN$(V, \{\hat{u}_i\}_{i\in[k]}, S_4)$
Return $\{w_i^{(T)}, v_i^{(0)}\}_{i\in[k]}$
end procedure

Tensor Initialization proposed by (Zhong et al., 2017) is fairly uncommon yet intriguing. Tensor initialization is a derivation from tensor problem. Although in general tensor problems are NP-hard(Hillar & Lim, 2013), by assuming noiseless and Gaussian input conditions, the authors are able to develop an efficient tensor method of weight initialization algorithm described in Algorithm 4. The core idea of tensor initialization is to leverage tensor decomposition and tensor estimation after dimension reduction. By applying tensor initialization, (Zhang et al., 2018) are able to prove that initial weight W^0 falls into small neighborhood of ground-truth weight W^*, thus leading to the proof of linear convergence rate.

4. Discussion

We have provided rather brief introduction of different theory techniques for shallow neural networks. We have touched upon several commonly used architecture, e.g. residual connection and covolutional filters, and optimization techniques including gradient-based algorithm and alternating minimization. Though not with too much proof detail, we have deliver problem setup and analysis conceptually. Since our purpose is to inform and convey a bigger scope of optimization in shallow neural networks in general, our review does cover essential topic that are crucial to understand.

However, we do notice that our literature review does not cover all topics in shallow neural networks. For example, we only consider simplified network where only input, hidden layer, and output present – we do not consider BN and pooling. We also noticed that most papers assume this sim-

plified version of network architecture as we do. So one possible future direction would be a systematic and holistic review on how BN/pooling/Dropout affect optimization and convergence analysis in shallow neural network. Also, we do not go into detail on theoretical definition of $O/\sqrt{d}$ weight initialization which is largely embraced by the community. Specifically how "Xavier initialization" and "He initialization" differ in different network setting (for example, empirically, He initialization works better on residual connection). This input assumption based analysis is also interesting for further investigation.

References

Blum, A. and Rivest, R. L. Training a 3-node neural network is np-complete, 1989. URL https://papers.nips.cc/paper/125-training-a-3-node-neural-network-is-np-complete.pdf.

Brutzkus, A. and Globerson, A. Globally optimal gradient descent for a convnet with gaussian inputs, 2017.

Du, S. S., Lee, J. D., and Tian, Y. When is a convolutional filter easy to learn?, 2018a.

Du, S. S., Lee, J. D., Tian, Y., Poczos, B., and Singh, A. Gradient descent learns one-hidden-layer cnn: Don't be afraid of spurious local minima, 2018b.

Du, S. S., Zhai, X., Póczos, B., and Singh, A. Gradient descent provably optimizes over-parameterized neural networks. *CoRR*, abs/1810.02054, 2018c. URL http://arxiv.org/abs/1810.02054.

Du, S. S., Lee, J. D., Li, H., Wang, L., and Zhai, X. Gradient descent finds global minima of deep neural networks, 2019.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. volume 9 of *Proceedings of Machine Learning Research*, pp. 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. JMLR Workshop and Conference Proceedings. URL http://proceedings.mlr.press/v9/glorot10a.html.

Goel, S., Klivans, A., and Meka, R. Learning one convolutional layer with overlapping patches, 2018.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Hardt, M. and Ma, T. Identity matters in deep learning, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015a.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015b.

Hillar, C. J. and Lim, L.-H. Most tensor problems are np-hard. 60(6), 2013. ISSN 0004-5411. doi: 10.1145/2512329. URL https://doi.org/10.1145/2512329.

Jagatap, G. and Hegde, C. Learning relu networks via alternating minimization, 2018.

Li, Y. and Yuan, Y. Convergence analysis of two-layer neural networks with relu activation, 2017.

Livni, R., Shalev-Shwartz, S., and Shamir, O. On the computational efficiency of training neural networks, 2014.

Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., and Sohl-Dickstein, J. On the expressive power of deep neural networks, 2017.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. Imagenet large scale visual recognition challenge, 2015.

Sola, J. and Sevilla, J. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468, 1997. doi: 10.1109/23.589532.

Soltani, M. and Hegde, C. Fast and provable algorithms for learning two-layer polynomial neural networks. *IEEE Transactions on Signal Processing*, 67(13):3361–3371, 2019. doi: 10.1109/TSP.2019.2916743.

Soltanolkotabi, M. Learning relus via gradient descent, 2017.

Tian, Y. Symmetry-breaking convergence analysis of certain two-layered neural networks with relu nonlinearity. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017a. URL https://openreview.net/forum?id=r1lVgRNtx.

Tian, Y. An analytical formula of population gradient for two-layered relu network and its applications in convergence and critical point analysis, 2017b.

Zhang, X., Yu, Y., Wang, L., and Gu, Q. Learning one-hidden-layer relu networks via gradient descent, 2018.

Zhong, K., Song, Z., Jain, P., Bartlett, P. L., and Dhillon, I. S. Recovery guarantees for one-hidden-layer neural networks, 2017.

A Review of Some Second Order Optimization Methods for Machine Learning
Final project for COMP 514 (Fall 2020)

Anusha Sekar [1]

Abstract

Underlying almost all machine learning (ML) algorithms is an optimization problem, where we minimize an objective function or loss function. There is usually a black box for computing the value of this loss function given some input. We are also usually given a method to compute the gradient of the loss function for given input. Given these black boxes, and data, we iteratively solve for input parameters that will best fit the given data.

For most modern ML methods, these optimization problems are non-linear and the parameter space dimension is high. One typical method to solve these optimization problems is using gradient descent, which involves following a sequence of iterates which minimize the loss function locally. The gradient of the loss function provides first order information about the shape of the loss surface, that is the slope of the surface at the current iteration point in parameter space. In this review, we will consider how gradient descent methods can be improved using second order information about the shape of the loss surface, the curvature.

We will review ideas from three main papers, [Agarwal et al., 2017], [Yao et al., 2020] and [Thiele et al., 2020], to illustrate some possible ways of using second order information to improve the optimization.

1. Introduction

Let us begin this review by setting up some notation. For this review, we will restrict our attention to ML methods that have an optimization problem of the form

$$\text{Find } x^* = \arg\min_x f(x) = \arg\min_x \sum_{i=1}^{n} f_i(x). \quad (1)$$

[1]Chevron, Houston, Texas, USA. Correspondence to: Anusha Sekar <Anusha.Sekar@chevron.com, as244@rice.edu>.

Submitted as final project report for COMP 514, Fall 2020

Gradient descent methods to solve (1) can be generally written as

$$x_{t+1} = x_t - \eta_t \nabla f(x_t) \quad (2)$$

where x_t is the current iteration, η_t is an appropriate step size taken in the descent direction. The descent direction is computed using the gradient of the objective (loss) function, $\nabla f(.)$. For the most part, optimization algorithms using gradient descent perform quite well. They are simple to understand and implement and are usually the first choice of algorithm that we would try. As the complexity of the optimization problem grows, however, we need to find ways to make gradient descent more efficient. For a certain class of function, vanilla steepest descent methods do converge to the solution, but may take a meandering path to get there.

For ML problems, we have an added complexity. Typically the data size is quite big and it is impractical to compute the full gradient. Instead, in practice, we take advantage of the separable form that the loss function has ($\sum_i f_i$) and use only some of the data components ($\mathcal{I}_t \subset \{1, \cdots n\}$) to compute the gradient. This method is called Stochastic Gradient Descent (SGD) if we use only one piece of the data and Mini-Batch SGD when we use several data components simumtanenouly. That is, in each iteration, we update x_t as follows

$$x_{t+1} = x_t - \eta_t \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \quad (3)$$

In fact, it can be shown that the total work required to obtain a desired ϵ-optimality (error less than a desired ϵ) is $\mathcal{O}(1/\epsilon)$, which can be much faster than regular gradient descent for large n. See section 4 of [Bottou et al., 2018] for example. Any improvements we make to the basic steepest descent algorithm must also work well in this stochastic framework!

There are several ways to improve upon gradient descent methods. One way, which is quite popular in the optimization community is to condition the descent direction (gradient) before taking a step in that direction. Mathematically, we can write this down as

$$x_{t+1} = x_t - \eta_t B(x_t) \nabla f(x_t) \quad (4)$$

This is sometimes referred to as *preconditioning*. If we knew that some parameters were more important than others,

we could choose a preconditioner, $B(x_t)$, which encodes this information and forcibly change the path taken by the iterates. "Adaptive" methods (e.g. Adagrad, Adam) do precisely this using first order information. Recently there has been some interest in the ML optimization community to see if we can do better by using second order information and there are very good reasons for this.

Using second order information in optimization is not a new concept. In fact, using $B(x_t)$ as the inverse of the Hessian operator for f at x_t is the classical Newton's method ([Nocedal & Wright, 2006], [Nesterov, 2004]). It is well known that Newton's method leads to local quadratic convergence. The Hessian operator can be written as a matrix H with entries $H_{i,j}(x_t) = \frac{\partial^2 f}{\partial x_i \partial x_j}$. The Hessian encodes information about the curvature of the loss function f. In gradient descent each iteration steps in the direction the negative of the gradient points to. If the loss surface is such that it has a narrow valley, steepest descent step can overshoot the valley. When we we precondition by the inverse of the Hessian, the update direction points more towards the center of the valley. See figure 1 for an illustration.

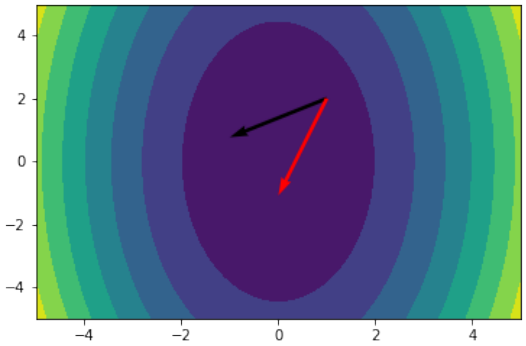


Figure 1. Contours for function $x^2 + y^2/5$. Black arrow is the original negative gradient direction. Red arrow is the preconditioned gradient direction

Given that using the Hessian can get us to the solution faster, there are reasons why this method has not gained immediate traction within the ML community. Computing the Hessian is usually quite expensive. One can come up with pathalogical examples where data is noisy and the local Hessian information is actually detrimental to the optimization. It is also not immediately clear how the Hessian will interact with the stochastic framework. In order to introduce second order information into a stochastic framework, ideally, we would like to have a method that looks like

$$x_{t+1} = x_t - \eta_t \sum_{i \in \mathcal{I}_t} B_i(x_t) \nabla f_i(x_t) \tag{5}$$

The main question is what should we use for $B_i(x_t)$. Naively, we might want to use the inverse of the Hessian of each component, but this does not produce good results. In the following sections, we will look at three different methods that use different approaches to overcome these hurdles and successfully combine second order information with stochasticity. The first method directly approximates the inverse of the full Hessian matrix to obtain an "approximate Newton" method, (section 2). The second method further simplifies by approximating only the diagonal of the Hessian matrix, (section 3). The third method splits the Hessian into two parts and ignores higher order terms to obtain a Gauss-Newton approximation to the Hessian matrix, (section 4).

2. Approximate Hessian: LiSSA

Key Paper: Second-Order Stochastic Optimization for Machine Learning in Linear Time, [Agarwal et al., 2017]

LiSSA stands for Linear (time) Stochastic Second-order Algorithm. LiSSA and some of its variations were proposed by Agarwal and others in [Agarwal et al., 2017]. For self-concordant functions (see [Nesterov, 2004] for definition), LiSSA has $\mathcal{O}(d)$ runtime, where d is the dimension of the space of parameters we are inverting for, which is on par with steepest descent! In this section, we will look at the key ideas behind this algorithm.

One of the main reasons SGD has favorable convergence properties is due to the fact that $\nabla f_i(x)$ is an unbiased estimator of $\nabla f(x)$. That is, the gradient of a single data sample is an unbiased estimator of the full gradient. As a result, if we have enough samples, we can get a statistically "good" estimate of the full gradient. The first key observation of the authors of LiSSA is that while $\nabla^2 f_i(x)$ is an unbiased estimator of $\nabla^2 f(x)$, $\left[\nabla^2 f_i(x)\right]^{-1}$ is not an unbiased estimator for $\left[\nabla^2 f(x)\right]^{-1}$. The authors derive an unbiased estimator as follows.

Any linear operator (matrix) A with $||A|| < 1$ is invertible and has the infinite series expansion $A^{-1} = \sum_{j=0}^{\infty} (I - A)^j$ By normalizing, we can use this series expansion for $B = \left[\nabla^2 f_i(x)\right]^{-1}$ as well. Let

$$B^{(k)} = \sum_{j=0}^{k} (I - \nabla f_i(x))^j \tag{6}$$

for some fixed k. We can show that $B^{(k)}$ is an unbiased estimator of the Hessian inverse and that $\mathbb{E}\left[B^{(k)}\right] \to \left[\nabla^2 f(x)\right]^{-1}$ as $k \to \infty$. We can now use $B^{(k)}$ in (5) after

we pick a suitable k. In fact, we can use a recursive formula,

$$B^{(0)} = I$$
$$B^{(j)}\nabla f_i(x_t) = \nabla f_i(x_t) + (I - H(x_t))B^{(j-1)}\nabla f_i(x_t) \quad (7)$$

Notice that the recursive formula only uses products of the form $H(x_t)z$ given a z. As long as we have a "black-box" or "oracle" which can compute Hessian vector products, we can form the inversion using the recursion. The full matrix of the Hessian is never formed.

Recall that Newton method only promises convergence if the starting point is close enough to the true solution. LiSSA also includes within it a "warm start" – A suitable first order algorithm is run until x_t is within the region where it is possible to get linear convergence for its second order method. We will paraphrase and state the main convergence theorem (Theorem 7) from [Agarwal et al., 2017] for this method here without proof. We need the following definitions. Let

$$\beta_{max} = \max_i \lambda_{max}\left(\nabla^2 f_i(x)\right);$$
$$\alpha_{min} = \min_i \lambda_{min}\left(\nabla^2 f_i(x)\right)$$
$$\hat{\kappa}_l = \max_x \frac{\beta_{max}}{\lambda_{min}\left(\nabla^2 f(x)\right)}; \quad \hat{\kappa}_l^{max} = \max_x \frac{\beta_{max}}{\alpha_m in}$$

Theorem 2.1 *Let f be α-strongly convex and β-smooth and a generalized linear model (GLM). Let $\hat{\kappa}_l$ and $\hat{\kappa}_l^{max}$ be the local condition numbers of f as defined above. Let T_1 be the time taken for the first order method to achieve an error of $\frac{1}{4\hat{\kappa}_l M}$. Let $S_1 = \mathcal{O}\left((\hat{\kappa}_l^{max})^2 \ln\left(\frac{d}{\delta}\right)\right)$, and $S_2 \geq 2\hat{\kappa}_l \ln\left(4\hat{\kappa}_l\right)$, then for every $t \geq T_1$,*

$$P\left(\|x_{t+1} - x^*\| \leq \frac{1}{2}\|x_t - x^*\|\right) = 1 - \delta$$

Moreover, each step of the algorithm takes at most $\mathcal{O}(md + (\hat{\kappa}_l^{max})^2 \hat{\kappa}_l d^2)$, where d is dimension of model parameter space. As a result, LiSSA returns a point x_t such that

$$f(x_t) \leq \min_{x^*} f(x^*) + \epsilon$$

in total time, $\mathcal{O}\left((m + S_1\kappa_l)d\log(\frac{1}{\epsilon})\right)$

Notice that S_1, the number of samples in a mini-batch and S_2, the number of terms of the recursive series Hessian approximation, both depend on the condition number of the problem. For a well conditioned problem, we can get away with smaller numbers. For an ill-conditioned problem, with small α and large β, we have to use more terms of the series, which will increase the cost of each iteration.

The authors also include several additions to this basic algorithm. Consider the quadratic

$$Q_t(y) = f(x_{t-1}) + \nabla f(x_{t-1})^T y + \frac{1}{2} y^T \nabla^2 f(x_{t-1}) y$$

If we were to run any first order gradient descent algorithm to minimize this quadratic form for a fixed x_t, we will get the update

$$y_t^{j+1} = y_t^j - \nabla Q_t(y_t^j) = (I - \nabla^2 f(x_t))y_t^j + \nabla f(x_t) \quad (8)$$

The authors notice that (8) is the same as the recursion formula (7) with $y^j = B^{(j)}\nabla f(x_t)$, so we can replace the recursion formula with any first order gradient descent method of our choice. In particular, the authors choose to use a method that reduces variance. In the regime where $m >> d$, that is we have more data samples than we have model parameters, the authors are able to choose a fast quadratic solver to accelerate the inner loop that computes the Hessian approximation. For each sub-problem, leveraging the fact that the Hessian approximations are positive semi definite, they improve upon existing techniques of matrix-sampling/sketching and combine with accelerated SVRG (variance reduced SGD) to accelerate convergence. Although these are necessary to arrive at the $\mathcal{O}(d)$ convergence rate they derive, we will omit details from this report since it is not directly related to second order ML methods.

3. Diagonal of the Hessian: AdaHessian

Key Paper: AdaHessian: An Adaptive Second Order Optimizer for Machine Learning [Yao et al., 2020]

As we mentioned in the introduction, we can think of $B\nabla f$ as scaling and rotating the gradient vector, that is "conditioning" the gradient vector, before taking a step in the scaled and rotated direction. In many cases, we may be able to get almost all the advantages of the Hessian by using only the diagonal of the Hessian matrix for this. The diagonal of the Hessian ignores any cross correlations between different model parameters, but focuses on the scaling of each model parameter itself. If the loss surface had some "steep" directions and some "flat" directions, gradient descent steps tend to "ping pong" in alternating directions. This curvature information is encoded in the diagonal of the Hessian matrix. Scaling by it's inverse lets us take longer steps in directions which are flatter, avoiding the "ping pong effect", which helps us get to the optimal solution faster.

Authors of the paper [Yao et al., 2020] take advantage of this fact and present an algorithm AdaHessian, [Gholami]. Two key ideas of the algorithm are the use of approximate iterative methods to compute the diagonal of the Hessian and the use of exponential averages of the diagonal similar to Adam. Let us take a look at both of these ideas now.

AdaHessian does not directly compute or store the diagonal of the Hessian matrix. Instead it uses Hutchinson's diagonal algorithm [Bekas et al., 2007], which can compute the diagonal of a matrix given a black box that can perform matrix vector multiplies. Hutchinson's method picks a random vec-

tor z_k whose entries are 1 or -1 with equal probability. The Hessian black box is used to compute $w_k = Hz_k$. After a few iterations, taking the average of w_k over k gives us an approximation of the diagonal of the Hessian. Since this is a diagonal matrix, computing it's inverse is simply taking the reciprocal of each element. In practice, AdaHessian seems to only compute one iteration of Hutchinson's algorithm, and also suggest only changing the Hessian every few iterations of the gradient descent to reduce cost.

AdaHessian is also adaptive similar to the Adam method. Recall the Adam method [Kingma & Ba, 2015] with updates of the form

$$x_{t+1} = x_t - \eta_t m_t / v_t \tag{9}$$

where m_t is a bias corrected version of the gradient and v_t is a bias corrected preconditioner to the gradient. In Adam, updates for m_t and v_t can be written as

$$m_t = \frac{(1-\beta_1)\sum_{j=1}^{t}\beta_1^{t-j}\nabla f_j(x_t)}{1-\beta_1^t}$$

$$v_t = \sqrt{\frac{(1-\beta_2)\sum_{j=1}^{t}\beta_2^{t-j}\nabla f_j(x_t)\odot\nabla f_j(x_t)}{1-\beta_2^t}}$$

where j in $\nabla f_j(x_t)$ is the index picked up by the stochastic gradient descent in the previous time step. We can think of the $1/v_t$ as a preconditioning term for the gradient descent. AdaHessian replaces this preconditioning term with an unbiased version of the approximate Hessian.

$$v_t = \sqrt{\frac{(1-\beta_2)\sum_{j=1}^{t}\beta_2^{t-j}D_j(x_t)\odot D_j(x_t)}{1-\beta_2^t}} \tag{10}$$

where D_j is the diagonal of the Hessian approximated using Hutchinson's method. The authors consider this step crucial since it is computing a moving average of the Hessian approximation over many iterations and this smoothes out noisy local curvature information, focussing on the global curvature information, much like what Adam's preconditioner does with gradients.

A third piece of the algorithm is that the authors also smooth spatially in the parameter space, which helps reduce variations between different convolutional layers for instance. Several results are shown where AdaHessian performs slightly better or on par with Adam. However the main advantage of AdaHessian seems to be in it's robustness to learning rate variability. As with Newton's method, approximate Newton's methods are also usually robust to changes in learning rate. That is, we do not need extensive tuning to figure out the correct learning rate. Reproducing here table 5 from [Yao et al., 2020] as figure 5 that shows this robustness. While AdaHessian can be twice as expensive per epoch compared to Adam, we can save actual time

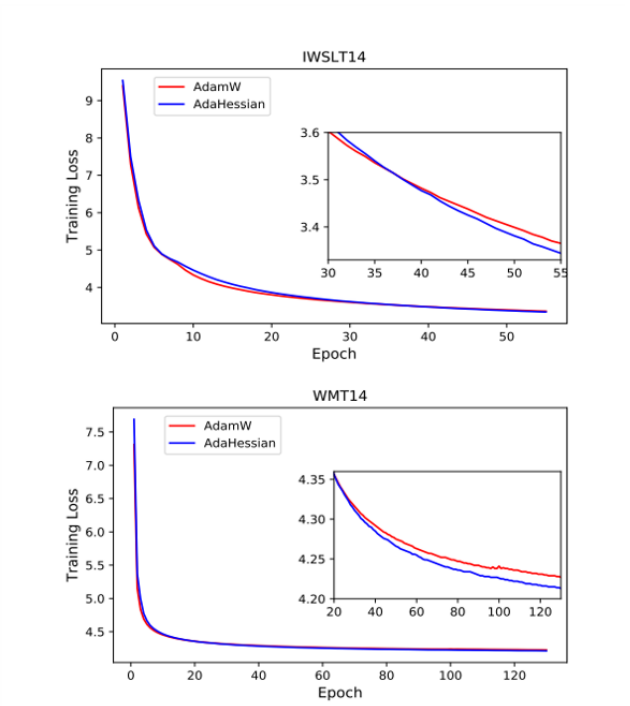


Figure 2. Training loss curves for Adam and AdaHessian for machine translation benchmark datasets, IWSLT14 and WMT14

by not having to run several rounds of tuning to find the best learning rate.

Finally, the authors also show a proof of convergence in the appendix. Using almost the same arguments as one would use for proving convergence of the Newton method, they first show that for α-strongly convex and β-smooth functions with $\alpha I \preceq \nabla^2 f(x) \preceq \beta I$, if we define updates of the form $\Delta x_t = H_t^{-k} g_t$, then with the proper learning rate, we get

$$f(x_{t+1}) - f(x_t) \leq -\frac{\alpha^k}{2\beta^{1+k}}\|\nabla f(x_t)\|^2 \tag{11}$$

Here H^{-k} is defined for any $k : 0 \leq k \leq 1$ as $H^{-k} = U^T\Lambda^{-k}U$, using the eigen value decomposition of the H. This result is more general than the result in many textbooks (e.g.[Nesterov, 2004]) where it is shown for $k = 1$, which is Newton's method. Using the fact that the diagonal elements of H can be written as $D_{i,i} = e_i^T D e_i = e_i^T H e_i$ where e_i are the unit vectors for the standard basis on the parameter space, they get that $\alpha \leq D_{i,i} \leq \beta$. As a result, they can show (11) when H is replaced by it's diagonal.

LR Scaling	0.5	1	2	3	4	5
AdamW	35.48	35.60	35.28	34.78	13.75	0.50
ADAHESSIAN	35.36	35.87	35.12	34.95	34.11	33.32

Figure 3. The numbers reported for each method are BLEU scores on the IWSLT14 machine translation dataset.Top row is the learning rate and we can see that while the scores change drastically for Adam with learning rate, they change very little with AdaHessian

4. Stochastic Quasi-Gauss-Newton: SQGN

Key Paper: Deep Learning with a Stochastic Quasi-Gauss–Newton Method, [Thiele et al., 2020]

When our loss function is of the form of a sum of squares (usually arising from a least squares minimization problem), there is another approximation that one can use for the Hessian. If $f(x) = \frac{1}{2}(h(x) - d)^T(h(x) - d)$, where d is some given data, then the gradient is $\nabla f(x) = \nabla h(x)^T(h(x) - d)$ and the Hessian is of the form $\nabla^2 f(x) = \nabla h(x)^T \nabla h(x) + \nabla^2 h(x)^T(h(x) - d)$. We can ignore the second term and take only the first term as an approximation to the Hessian,

$$\nabla^2 f(x) \approx \nabla h(x)^T \nabla h(x) \tag{12}$$

which gives us the classic Gauss Newton method. That is

$$x_{t+1} = x_t - \eta_t \left[\nabla h(x)^T \nabla h(x)\right]^{-1} \nabla h(x)^T (h(x) - d) \tag{13}$$

This idea can be easily generalized to distance functions that are not least squares as well. If $f = l \circ h$, then $\nabla^2 f(x) \approx \nabla h(x)^T \nabla_l^2(h) \nabla h(x)$. Note that in this case, $\nabla l^2(h)$ is the Hessian of l as a function of h. This is usually easy to compute and has closed form solution for many distance functions. Computing $h(x)$ and it's gradient and Hessian are the expensive operations in this process. The Gauss Newton update is of the form

$$x_{t+1} = x_t - \eta_t \left[\nabla h(x)^T \nabla l^2(h) \nabla h(x)\right]^{-1} \nabla h(x)^T \nabla l(h) \tag{14}$$

Neither the matrix nor its inverse is formed in this approach. Usually there is an inner loop optimization which iteratively solves

$$\left[\nabla h(x)^T \nabla l^2(h) \nabla h(x)\right] \delta x = \nabla h(x)^T \nabla l(h) \tag{15}$$

The iterative approach only requires a black-box implementation of how a gradient and it's transpose act on vectors. We can also take advantage of the fact that we do not need to solve this inner problem perfectly and can sub-sample and use a much smaller sample of the data to compute this approximate Hessian inverse. this This method has been used successfully for ML in [Martens & Stuskever, 2011].

Also see section 6.1 of [Bottou et al., 2018] for a summary and some discussion.

Let us switch gears for a minute and talk about Limited memory BFGS (L-BFGS), which has been the workhorse of optimization for several decades. L-BFGS forms an approximation of the Hessian by building it up over several iterations by taking differences between gradients at current and past iterates. L-BFGS comes from a class of methods referred to as quasi-Newton methods. Without going into the actual iteration details (see [Nocedal & Wright, 2006]), we recall that L-BFGS computes terms of the form

$$s_t = x_{t+1} - x_t; \quad v_t = \nabla f(x_{t+1}) - \nabla f(x_t) \tag{16}$$

It is immediately obvious that it is not easy to extend this to a stochastic framework. When we only use some of the data samples in computing the gradient at an iteration, LBFGS has the form

$$s_t = x_{t+1} - x_t; \quad v_t = \sum_{i \in \mathcal{I}_{t+1}} \nabla f_i(x_{t+1}) - \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t) \tag{17}$$

v_t is the difference between two terms which could have come from possibly disjoint samples of the data and tends to be "noisy". Taking differences between two "noisy" objects does not help the optimization. One of the methods proposed is to take an average of the gradients over several iterations, but this does not seem to improve the situation either. A method proposed by [Byrd et al., 2016] is to define

$$s_t = x_{t+1} - x_t; \quad v_t = \sum_{i \in \mathcal{I}_t} \nabla^2 f_i(x_t) s_t \tag{18}$$

Using (18) in an L-BFGS method does seem to converge faster and improve the optimization. The cost of the Hessian is amortized over many iterations. This can be confusing since we were trying to avoid computing the Hessian's in the first place! However, once we know that we can use (18), we can replace the Hessian computation in it with any of the approximations we have been discussing! For instance [Byrd et al., 2016] use a sub-sampling technique.

In [Thiele et al., 2020], the paper we are considering now, authors combine the ideas of Gauss Newton and stochastic L-BFGS and use the Gauss Newton approximation (12) in (18). The idea was first proposed by [Liu et al., 2019], and the authors in this paper have augmented it by using a SVRG-like variance reduction. They call this method Stochastic Quasi Gauss Newton (SQGN).

While the mathematics in this paper [Thiele et al., 2020] is not completely new, what makes this paper interesting is that the authors are trying to solve a problem which would traditionally be solved using a partial differential equations (PDE) constrained optimization. The authors are treating this as a ML problem and the comparison to first order

methods is interesting. Seismic tomography or seismic full waveform inversion (FWI) are classic optimization problems that arise in geophysics for oil exploration or studies of the earth's mantle [Fichtner, 2011]. The problem is to reconstruct the earth's subsurface structure (velocities) from seismic measurements on the surface. This is a non-linear optimization problem and since we only have data at the surface, there are known issues like non-uniqueness ("cycle-skipping") and limited depth penetration of seismic signals, especially in the presence of salt in the subsurface. Sometimes the PDE used as a constraint does not model all the physics of the problem and as a result there is now some interest in the geophysics community to use ML to solve this problem and leave the physical modeling out.

Figure 4 shows the training loss and testing accuracy of Adam and SQGN. We immediately notice that Adam outperforms SQGN in both metrics. The figure shows results from five different runs. SQGN is also reported to be 2.5 times as expensive as Adam. Once again, however, the advantage is in hyper-parameter tuning. SQGN was applied to this new problem using the same hyper-parameters that the author used for a different ML problem. They did not need to tune their algorithm for this new setting or new data. They also point out that there is little variation between the results of five runs shown in the figure.

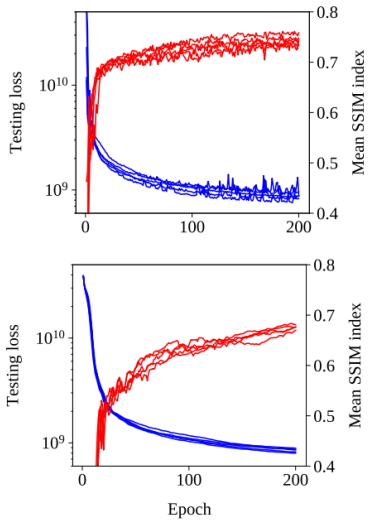


Figure 2. Convergence behavior of Adam ($\alpha = 10^{-2}$, top) and SQGN (bottom) for the tomography problem. Loss (blue) and SSIM index (red) are plotted for five repetitions of the experiment.

Figure 4. Figure 2 from [Thiele et al., 2020]. Convergence rate and accuracy of Adam are better, but there is less variation in the result from SQGN

One can still argue that Adam has performed quite well for the seismic tomography problem. However, the image produced by SQGN is more realistic. The metrics we normally use to compare different algorithms like the structural similarity index do not seem to highlight that the SQGN image is more realistic. This suggests that for this type of problem, we may need newer metrics to evaluate the performance of ML methods. We should point out that the experiments in this paper have been performed using synthetic seismic data and it has not been proven that this technique will perform well with real seismic data.

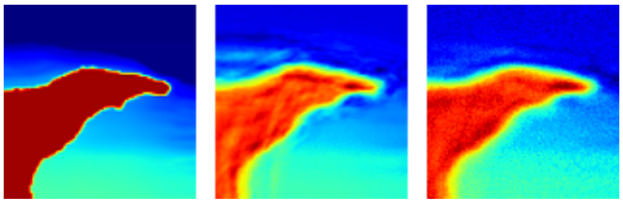


Figure 5. Figure 4 from [Thiele et al., 2020]. Left panel: Reference (True) solution. Middle panel: Result from Adam. Right panel: Result using SQGN. Although metrics seem to show the Adam performs better, for this problem there are fewer physical artifacts when using SQGN as compared to Adam

5. Discussion

We have seen three different approaches to using second order information for ML. There are some common features among all three

- None of them form the actual Hessian matrix or use it's functional representation
- All of the approximate the Hessian using iterative methods and some of the speed up is related to how well this inner problem can be solved
- All of them have to include some variance reduction techniques to reduce the effect of noisy local Hessians from corrupting the optimization
- All methods show a robustness to hyper-parameters, which might be the key advantage of second order methods.

ML algorithms differ from classical optimization problems in that we have to worry about generalization of recovered models as well. In classical optimization, we have achieved success once we are able to fit a model to the given data within some error tolerance. However, for ML algorithms, it is not enough to determine a model within error tolerance on the given training data, the model needs to work well enough to predict on new data as well. That is, our optimization algorithm must reduce the loss function, but must do it in

such a way that we do not lose accuracy on test data (and future unknown data). Since second order methods do a very good job of reducing the loss function, there is always the danger of over-fitting the training data and not achieving required accuracy. There is still a lot of research that can be done!

While it is not immediately clear whether second order methods are good at generalization, there do seem to be several advantages in using them. One key advantage is their robustness to learning rate, which can avoid expensive tuning runs for new problems. Another advantage of second order methods is their ability to resolve cross talk [Pan & Innanen, 2016] which is evident in non ML applications (where first order methods struggle). When we are simulataneously solving for multiple parameters, using limited data, we could have two solutions that minimize the data misfit in the similar ways. Second order methods have the ability to mitigate some of these non-uniqueness issues. It is not clear yet whether this is an issue for ML or not, since methods like Adam can mitigate cross talk to some extent. Since all nonlinear problems are different, this may be a future research possibility.

This report only touched on some of the research on this topic. There are several other flavors of second order methods. [Dong et al., 2019] use the Hessian in a different way to reduce the memory footprint of neural networks. Several other authors are pursing stochastic L-BFGS variations as well, see [Bollapragada et al., 2018], and [Meng et al., 2020]. Second order information has also been used to reduce dimensionality of problems via matrix-sketching in [Berahas et al., 2019], [Berahas et al., 2020]. We also recommend the review paper, [Bottou et al., 2018] for a holistic view of second order methods in machine learning.

5.1. Citations and References

References

Agarwal, N., Bullins, B., and Hazan, E. Second-order stochastic optimization for machine learning in linear time. *Journal of Machine Learning Research, VOL 18, p 1-40*, 2017.

Bekas, C., Kokiopoulou, E., and Saad, Y. An estimator for the diagonal of a matrix. *Applied Numerical Mathematics, VOL 57, p 1214-1229*, 2007.

Berahas, A. S. et al. An investigation of newton-sketch and subsampled newton methods. *ArXiv preprint*, 2019.

Berahas, A. S. et al. Quasi-newton methods for deep learning: Forget the past, just sample. *ArXiv preprint*, 2020.

Bollapragada, R. et al. A progressive batching l-bfgs method for machine learning. *Proceedings of the 35th Int. Conf. on Machine Learning*, 2018.

Bottou, L., Curtis, F., and Nocedal, J. Optimization methods for large-scale machine learning. *SIAM Review*, 2018.

Byrd, R. et al. A stochastic quasi-newton method for large-scale optimization. *SIAM Jou. on Optimization*, 26(2), 2016.

Dong, Z. et al. Hessian aware trace-weighted quantization of neural networks. *ArXiv preprint*, 2019.

Fichtner, A. *Full Seismic Waveform Modeling and Inversion*. Springer, 2011.

Gholami, A. Adahessian. URL `https://github.com/amirgholami/adahessian`.

Kingma, D. and Ba, J. A method for stochastic optimization. *International Conference on Learning Representations*, 2015.

Liu, J. et al. Accelerating distributed stochastic l-bfgs by sampled 2nd-rder information. *Proc. 33rd conf. on Neural Inf. Proc. Sys.*, 2019.

Martens, J. and Stuskever, I. Learning recurrent neural networks with hessian-free optimization. *Proc. of 28th Int. Conf. on Machine Learning*, 2011.

Meng, S. Y. et al. Fast and furious convergence: Stochastic second order methods under interpolation. *Proc. of the 23rd Int. Conf. on AI and Stats.*, 2020.

Nesterov, Y. *Introductory Lectures on Convex Optimization, A Basic Course*. Kluwer, 2004.

Nocedal, J. and Wright, S. J. *Numerical Optimization, 2nd edition*. Springer, 2006.

Pan, W. and Innanen, K. A summary of several challenges facing multi-parameter elastic full waveform inversion. *CSEG Recorder, VOL. 41 NO. 08*, 2016.

Thiele, C., Araya-Polo, M., and Hohl, D. Deep learning with a stochastic quasi-gauss–newton method. *Proceedings of 37th International Conference on Machine Learning*, 2020.

Yao, Z., Gholami, A., Shen, S., Keutzer, K., and Mahoney, M. Adahessian: An adaptive second order optimizer for machine learning. *Arxiv preprint*, 2020.

Exploring the Effects of Discrete Optimizers as Layers of Continuous Methods

Byungjun Kim [1]

Abstract

Traditionally, there has been a gap of knowledge between discrete solvers and continuous methods. As machine learning grows in importance to problems such as image classification and segmentation, the incompatibility between continuous methods and discrete problems becomes apparent, and the discreteness of problems severely inhibits their utility to machine learning advancement. We will explore some real examples of discrete optimizer relaxations to use as layers in continuous methods, and their effects on algorithm functionality or optimality, as well as a comparison with current leading algorithms. In particular, we discuss a relaxed SAT solver to train a neural network to solve sudoku puzzles (as well as image identification and classification) and a differential mask matching network to address image segmentation and identify distinct moving objects within multiple frames of videos.

1. Introduction

What exactly are discrete solvers and continuous methods? What makes them irreconcilable? In order to understand the relationship between discrete optimization and continuous methods, we first define the two fields, and elaborate on their key differences that cause tension between them.

1.1. Discrete Solvers/Optimizers

Discrete solvers, which involve algorithms that solve problems on discrete variables, typically require integer constraints, and are good for solving purely discrete models. That is, they compute a model's next simulation time step, and thus do not compute continuous states. A classic example of a discrete optimizer, and one that we will discuss further today, is the sudoku puzzle. The puzzle consists of a structure of numbers defined by "rules", i.e. constraints, that apply digits 1-9 into various boxes in the 9x9 sudoku board. Some other noteworthy discrete solvers include ILP, the knapsack problem, and graph algorithms, like traveling salesman. Typically, discrete solvers require knowledge of the various constraints necessary to solve the problem, and thus require an extensive and comprehensive infrastructure for the algorithm to properly execute. This can be costly to implement, and does not fare well with low-info/unsupervised learning, such as image classification.

1.2. Continuous Methods

Continuous methods represent models that use continuous, differentiable data as input. The traditional example of a continuous method is a neural network, or any ML model dealing with continuous data, like using gradient descent. Continuous methods are highly advantageous in solving problems where the problem is not completely defined, or when one wants to look for patterns or trends in the data. However, continuous methods are not applicable to discrete data, and have long been unable to solve ILP or sudoku-like problems with integer constraints. Consequently, continuous methods like neural networks are not able to solve discrete problems, thus creating this tension between discrete solvers and continuous methods.

1.3. Resolving Differences

To explore possible solutions to apply discrete solvers to continuous methods, we will discuss two experimental implementations of discrete solver relaxation to obtain a continuous, differentiable stream that can be used in continuous methods. In particular, we will explore the following two papers, with supplemental insight from many others:

- **SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver** by Wang et al. (2019).
- **DMM-Net: Differentiable Mask-Matching Network for Video Object Segmentation** by Zeng et al. (2019).

In analyzing these papers, we will discuss key factors the authors used to apply relaxed discrete solvers to their respective continuous methods, and seek to define those characteristics that might further bridge discrete solvers to continuous method, hopefully clarifying a solution for resolving their tension.

2. MAXSAT Relaxation

SAT, or boolean satisfiability, is a well known problem in computer science and logic, in which a solution is sought (if there exists one) using the logical operator constraints provided. A simple example is the N Queens problem, which solves the maximum number of queens allowed on a standard 8x8 chess board such that the queens cannot attack each other. This problem can be represented as a set of integer constraints of each queen's "range" and position encoding. This problem can then be fed as input as an algorithm to determine solutions that satisfy all of the boolean clauses.

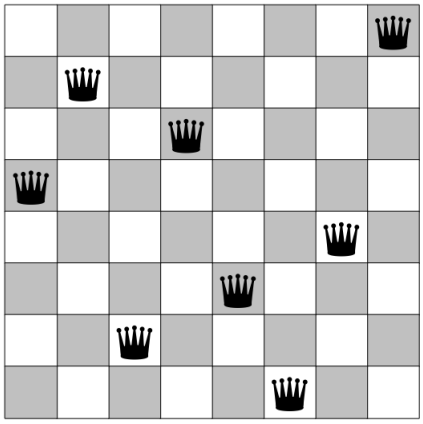


Figure 1. A valid solution to the N Queens problem

The discrete solver used to solve the sudoku problem is the MAXSAT solver, or "Maximum Satisfiability". Similar to SAT, MAXSAT aims to solve a given set of boolean operators, but instead of finding unique solutions that satisfy ALL boolean clauses, MAXSAT determines those solutions that satisfy the greatest number of clauses. In a typical SAT sudoku solver implementation, the 9x9 sudoku board is encoded in bit representation, with its rules and relationships defined in the encoding. However, for relaxed MAXSAT, a looser approach is taken, without all of the rules to sudoku explicitly represented in the data.

2.1. SDP Relaxation

In order to obtain a continuous, differentiable MAXSAT variation, the authors apply a fast coordinate descent approach to solving a Semidefinite Programming (SDP) relaxation to MAXSAT (Wang et al., 2017). This SDP relaxation produces strong approximation guarantees for MAXSAT, while producing a differentiable optimization-based MAXSAT solver that can be used as a layer in a machine learning algorithm.

Consider a MAXSAT instance with n variables and m clauses. If we let $\tilde{v} \in \{-1, 1\}^n$ denote binary, integer assignments to the problem variables, where $\tilde{v}_i$ is assigned the truth value of variable $i \in \{1, ..., n\}$, and $\tilde{s} \in \{-1, 0, 1\}$ for $i \in \{1, ..., n\}$ where s_{ij} denotes the sign of $\tilde{v}_i$ in clause $j \in \{1, ..., m\}$, then we can present the MAXSAT problem as

$$\underset{\tilde{v}\in\{-1,1\}^n}{\text{maximize}} \sum_{j=1}^{m} \bigvee_{i=1}^{n} 1\{\tilde{s}_{ij}\tilde{v}_i > 0\}.$$

To form an SDP relaxation of the above MAXSAT equation, we relax the discrete variables $\tilde{v}_i$ into continuous variables $v_i \in \mathbb{R}^k, \|v_i\| = 1$, with respect to some "truth direction" $v_\top \in \mathbb{R}^k, \|v_\top\| = 1$, with coefficient vector $\tilde{s}_\top = \{-1\}^m$ associated with $v_\top$. The continuous variable v_i relates to the discrete $\tilde{v}_i$ by $P(\tilde{v}_i = 1) = \cos^{-1}(-v_t^T v_\top)/\pi$, an approach determined by randomized rounding (Goemans & Williamson, 1995). The SDP relaxation yields

$$\underset{V\in\mathbb{R}^{k\times(n+1)}}{\text{minimize}} \left\langle S^T S, V^T V \right\rangle, \text{ subject to}$$
$$\|v_i\| = 1, i = \top, 1, ..., n$$

where $V \equiv [v_\top \quad v_1 \quad ... \quad v_n] \in \mathbb{R}^{k\times(n+1)}$ and $S \equiv [\tilde{s}_\top \quad \tilde{s}_1 \quad ... \quad \tilde{s}_n]\, \text{diag}(\frac{1}{\sqrt{4|\tilde{s}_j|}}) \in \mathbb{R}^{m\times(n+1)}$.

Which represents a continuous, differentiable variation of MAXSAT that can be fed as a layer in a deep network, as we describe in the following section.

2.2. SATNet Layer

Using the relaxed SDP MAXSAT solver, we can create a deep network layer to aid in training. We start by defining $I \subset \{1, ..., n\}$ as the indices of MAXSAT variables with known assignments and $O \equiv \{1, ..., n\}\ I$ the indices of variables with unknown assignments. Then, our layer takes our relaxed probabilistic inputs $z_i \in [0, 1]$ and produces the assignments of unknown probabilistic variables $z_0 \in [0.1]$. Using these inputs Z_i, a forward pass is computed using our coordinate descent algorithm and the weights of the layer are determined from SDP.

From the relaxed outputs from our forward pass, we convert these outputs to discrete/probabilistic variable assignments Z_0, or assignments to our unknown variables, by randomized rounding. For every $v_o, o \in O$, we take some random hyperplane r from the unit sphere and assign

$$\tilde{v}_o = \begin{cases} 1 & \text{if sign}(v_o^T r) = \text{sign}(v_\top^T r) \\ -1 & \text{otherwise} \end{cases}, \ o \in \mathcal{O},$$

This randomized rounding procedure, with the correct weights S, assures an optimal expected approximation ratio for our MAXSAT solver (Goemans & Williamson, 1995;

Model	Train	Test
ConvNet	72.6%	0.04%
ConvNetMask	91.4%	15.1%
SATNet (ours)	99.8%	**98.3%**

(a) Original Sudoku.

Model	Train	Test
ConvNet	0%	0%
ConvNetMask	0.01%	0%
SATNet (ours)	99.7%	**98.3%**

(b) Permuted Sudoku.

Model	Train	Test
ConvNet	0.31%	0%
ConvNetMask	89%	0.1%
SATNet (ours)	93.6%	**63.2%**

(c) Visual Sudoku. (Note: the theoretical "best" test accuracy for our architecture is 74.7%.)

Figure 2. Results for 9x9 Sudoku experiments.

Wang & Kolter, 2019). In essence, this aids in obtaining the correct z_o boolean solution that maximizes our MAXSAT objective, or solve the sudoku puzzle.

Now that we have our forward pass computed, the authors derive the backward pass updates to integrate our SATNet layer with the deep network. The backward pass, which I will not go into but leave in the appendix to explore, takes a coordinate descent approach that applies gradients with respect to output relaxations to a modified coordinate descent algorithm. The algorithm uses rank-one updates to maintain and modify Φ, a variable key in making the algorithm's runtime $O(nmk)$.

2.3. Experimental Results

The relaxed MAXSAT-deep network integration was tested in three categories for sudoku, each with 9K training examples and 1K test examples:

- Bit representation, with locality structure implicitly built into the data stream
- Bit representation, but permuted in order to remove any and all locality that may have been used
- Visual representation of a sudoku board with filled/"empty" squares.

To compare results of SATNet, two alternate, deep learning options display results alongside SATNet:

- ConvNet, a standard implementation of a convolutional neural network
- ConvNetMask, a modified CNN that receives a binary mask indicating those bits that must be learned.

The goal of each sudoku puzzle representation is to demonstrate SATNet's ability to learn the rules of the game and to complete each board accurately without being explicitly told the relationships between variables, and to demonstrate SATNet's flexibility in learning from various streams and methods of data, as well as its efficiency and competitiveness against other neural networks.

2.3.1. Bit Representation (Original Sudoku)

The bit representation of a sudoku puzzle tests SATNet's ability to learn to solve sudoku puzzles without hard-coded relationships between the bits. In SATNet, the input is vectorized, and does not rely on locality structures or other to learn to solve puzzles. Referring to the table displayed above, SATNet outperforms both ConvNet and ConvNetMask by a great margin, as expected. These results demonstrate SATNet's massive advantage in learning rules to previously discrete problems that continuous methods struggled to learn and solve. In fact, the standard CNN performed dismally, as it cannot learn the rules of the discrete problem without guidance. The masked CNN performs better, given its reliance on the bit mask to determine bits to learn, but still nowhere near SATNet.

2.3.2. Permuted Bit Repr. (Permuted Sudoku)

In order to remove any locality structure that may be taken advantage of in the bit representation, a permutation is applied to the representation such that the relationship between bits is maintained, but order is scrambled and locality effectively erased. CNN, predictably, performs just as poor as previously. In fact, it performs even worse now that any inkling of additional information it had used before has been removed. ConvNetMask previously had been provided a bit mask marking those bits to be learned, but now that the bits have been scrambled, the binary mask is not particularly useful without the locality structure, and it's clear that CNN and ConvNetMask are not able to learn the underlying rules of the game. SATNet, on the other hand, does not rely on the structure of the bit representation, but rather the relationship between the bits themselves, and is thus able to perform exactly the same as before permutation, showing strong evidence of superiority in learning using discrete solvers.

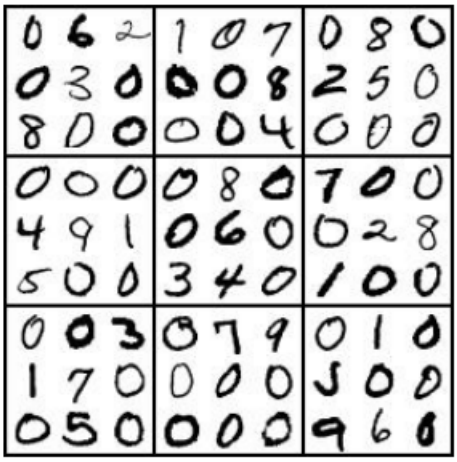


Figure 3. An example of a MNIST digit-filled sudoku board, with 0s representing spaces to be filled.

2.3.3. VISUAL SUDOKU

Visual sudoku, as displayed above, is a visual representation of a sudoku board, where cells are filled with given numbers 1-9, and spaces to be filled are represented with zeroes. This requires additional layers and combining multiple network layers to process the MNIST digits and convert the visual representation to some readable, logical stream, so that the sudoku algorithm can compute and solve the puzzle. Traditionally, neural network architectures are not able to represent this complexity well, and the results show exactly that, with CNN performing poorly per usual. ConvNetMask performs well during the training phase, but ultimately overfits and cannot convert its results to the test phase. SATNet, however, performs well in both training and testing, and demonstrates its clear ability to learn the "rules of the game" directly from visual input that previously had been unheard of with both discrete solvers and neural network architectures.

2.4. Conclusion

SATNet has a clear advantage in processing and accurately solving a variety of input types compared to traditional continuous methods. Additionally, it has a runtime that outperforms most discrete solvers, and does not require computational "hand-holding" with a hefty infrastructure and the need for many constraints and rules. Instead, SATNet utilizes the discrete, optimization aspects of discrete solvers and applies them to continuous methods so that they can learn the rules and constraints, and produce accurate and efficient results, taking the best of both sides. We will see in the following section that a similar process ensues for image segmentation.

3. DMM-Net Image Segmentation

Differentiable Mask-Matching Network, or DMM-Net, is a novel solution to video object segmentation that applies relaxation to a discrete optimizer and integrates it as a layer in a continuous method. Specifically, using a Mask-R-CNN backbone, the Mask-R-CNN extracts mask proposals per video frame and create matches between the object templates and proposals as a linear assignment problem (LAP)(He et al., 2017). DMM-Net resolves its LAP by unrolling a projected gradient descent algorithm, whose projection exploits Dykstra's projection algorithm (not to be confused with *Dijkstra's Algorithm*). This algorithm relaxes the LAP to a continuous and differentiable model, which is applied as a layer in the Mask-R-CNN to achieve competitive video segmentation mask-matching results.

3.1. Introduction

Video segmentation, exampled below, is the process of partitioning a video frame into background and objects in the foreground, typically that draw significant attention. This is an incredibly important task in computer vision, especially with its use in security/surveillance, autonomous driving, video editing, and more.

Figure 4. An example of DMM-Net's video object segmentation: The runner, cart, and rider are all determined "objects" while the background is not.

DMM-Net applies to semi-supervised video object segmentation (VOS), which means instance masks are provided for the first few frames of the test videos. With just this information, the algorithm must process the information frame by frame and maintain a tight representation of the object(s) in focus throughout the video (i.e. we want the same mask to apply to the same object over multiple frames, whether the object moves in frame or not). Currently existing VOS algorithms, which leverage pretrained deep neural network for object masks, cannot integrate DMM-Net's optimal matching algorithm that provides more accurate masks due to its inherent non-differentiable nature. This is a key advantage with DMM-Net that we will see when comparing results with competitive algorithms.

DMM-Net first extracts mask proposals with a pre-trained Mask-R-CNN (similar to other leading deep networks), and then matches those proposals against the mask templates provided (i.e. the template for the first frames, then every subsequent computed template), assigning matching costs of each proposal. While similar to a standard deep network VOS model, the key difference is DMM-Net's leverage of the linear cost assignment problem, and its relaxation that makes the cost assignment problem differentiable and thus includable as a layer in the network.

3.2. Mask Matching

There are two main classes of image matching: Pixel-level matching and Mask-level matching. While both are viable options for video object segmentation and determining instance masks, pixel-level matching is quite memory and computation intensive and does not utilize nor address the optimal matching algorithm used in DMM-Net. Because of this, DMM-Net uses mask-level matching instead.

DMM-Net's mask-level matching involves tracking the object parts in the video by computing similarity scores between the proposal and templates in the reference frame, solving the matching problem with an iterative solver, a verifiably better solution opposed to many other models that rely on a greedy algorithm that takes the greatest score.

3.3. Model

DMM-Net's novelty arises from its differentiable mask matching (DMM) and its method for mask refinement, involving an iterative discrete-relaxed matching algorithm. First, as input for the DMM stage, the model extracts mask proposals per frame with a COCO-pretrained Mask R-CNN, selecting the top 50 proposals. Then, using a CNN f_θ, where θ denotes learnable parameters for the mask matching cost, we extract features for the given mask proposals P^t and templates R (proposals from time t, templates taken from the first frame).

Once the proposals and templates have been identified, we can determine the features for both; that is, we can compute $f_\theta(r_i)$ and $f_\theta(p_j^t)$, where i and j denote the i^{th} and j^{th} template and proposal, respectively. DMM-Net's matching cost matrix, C^t, utilizes cosine similarity and Intersection-over-Union (IoU) to compute the difference cost between masks:

$$C_{i,j}^t = (\lambda - 1)\cos\left(f_\theta(p_j^t), f_\theta(r_i)\right) - \lambda \text{IoU}(p_j^t, r_i)$$

Where $0 < \lambda < 1$ is a hyperparameter determined by the user to control the learning process. The result of this applied to our mask proposals and templates is the cost matrix C^t, of size $n \times m^t$, where each row and column reference a template and mask proposal, respectively.

3.4. Cost Matching Problem

The discrete solver DMM-Net uses is a well known integer linear programming problem, the bipartite matching problem. To summarize, bipartite matching is a typically graphical problem that entails partitioning a vertex set into two smaller sets such that no edge exists between two vertices in the same set. For DMM-Net's purposes, we focus on minimum-cost bipartite matching, or determining the maximum cardinality matching that has minimum cost.

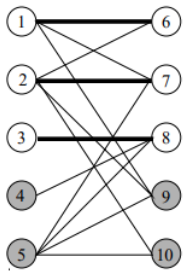


Figure 5. An example of a bipartite graph, where no two vertices on either the left nor right have edges between themselves

Algorithmically, minimum cost bipartite matching can be expressed as the following ILP problem:

$$\begin{aligned} \min_X \quad & \text{Tr}\left(CX^\top\right) \\ s.t. \quad & X\mathbf{1}_m = \mathbf{1}_n \\ & X^\top \mathbf{1}_n \leq \mathbf{1}_m \\ & X \geq 0 \\ & X_{i,j} \in \{0,1\} \qquad \forall (i,j) \end{aligned}$$

Where $X \in \mathbb{R}^{n \times m}$ is a boolean assignment matrix (hence the last line), and $1_n, 1_m$ are one vectors of size n and m, respectively. Using this discrete solver directly for VOS would give us a time complexity of $O(m^3)$, which is far too large, and would also not back propagate easily. Additionally, a pure discrete solver would supply exact matchings, which are not necessary in VOS - a real, closely approximated assignment matrix would suffice for mask matching. Thus, DMM-Net proposes the following linear programming relaxation:

$$\begin{aligned} \min_X \quad & \text{Tr}\left(CX^\top\right) \\ s.t. \quad & X\mathbf{1}_m = \mathbf{1}_n \\ & X^\top \mathbf{1}_n \leq \mathbf{1}_m \\ & X \geq 0. \end{aligned}$$

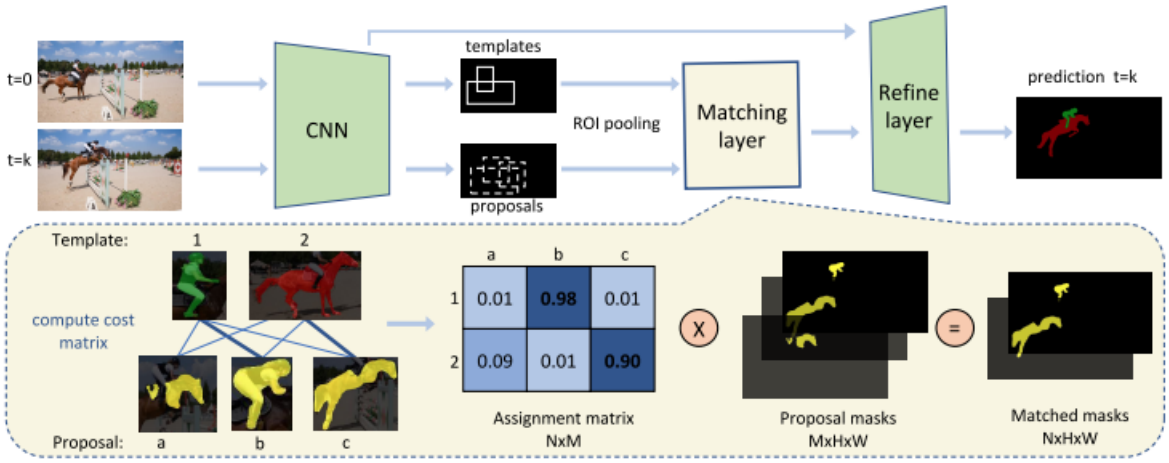


Figure 6. DMM-Net process from template to predictions; Proposals and templates are generated, assignment matrix is computed, masks are matched, refined, and predictions are outputted.

Namely, the differences being that instead of $X_{i,j} \in \{0, 1\}$, i.e. binary, we drop that clause entirely, such that $X_{i,j} \in \mathbb{R}$.

With this relaxed problem, DMM-Net introduces the projected gradient descent algorithm below, where N_grad and N_proj are the number of gradient descent steps and projection steps, respectively. At each iteration, X is updated following the negative gradient direction, as is typical, but to compute the projection of the updated X onto the constraint set, DMM-Net exploits a cyclic constraint projection method, Dykstra's projection algorithm, which is guaranteed to converge, and as a bonus, is straightforward to implement (Dykstra, 1983).

Algorithm 1 : Projected Gradient Descent for Matching

Input: $N_{\text{grad}}, N_{\text{proj}}, X, \alpha, C$
Initialization: $X^0 = X$
For $i = 1, 2, \ldots, N_{\text{grad}}$:
  $X^i = X^{i-1} - \alpha C$
  $Y_0 = X^i, q_1 = 0, q_2 = 0, q_3 = 0$
  For $j = 1, 2, \ldots, N_{\text{proj}}$:
    $Y_1 = \mathcal{P}_1(Y_0 + q_1), \quad q_1 = Y_0 + q_1 - Y_1$
    $Y_2 = \mathcal{P}_2(Y_1 + q_2), \quad q_2 = Y_1 + q_2 - Y_2$
    $Y_3 = \mathcal{P}_3(Y_2 + q_3), \quad q_3 = Y_2 + q_3 - Y_3$

  $X^i = Y_3$
Return $\hat{X} = \frac{1}{N_{\text{grad}}} \sum_{i=1}^{N_{\text{grad}}} X^i$

3.5. Mask Refinement

After matching according to the projected gradient descent algorithm, for each mask template, DMM-Net outputs one mask that is fed to the final refinement stage before prediction. After obtaining the optimal assignment $\hat{X}$, we compute a weighted combination of the proposal masks P to refine our mask.

First, we resize the mask proposals such that the proposals' resolutions match that of the image. Then, we paste the proposals onto empty images such that the proposal, $\tilde{P}$, is now the same size as the input image. Then, to obtain the cost-matched mask $\hat{P}$, we take the tensor product of the cost matrix and the corrected proposal:

$$\hat{P} = \hat{X} \otimes \tilde{P}$$

where $\hat{X} \in \mathbb{R}^{n \times m}$, $\tilde{P} \in \mathbb{R}^{m \times H \times W}$, $\hat{P} \in \mathbb{R}^{n \times H \times W}$, and H and W denote the height and width of the input image. $\hat{P}$ is a matrix containing the matched masks, where each $H \times W$ slice in $\hat{P}$ represents the matched mask corresponding to a particular template. With the matched masks given in $\hat{P}$, we refine them against the corresponding templates, and run them through the rest of the deep network's four ConvLSTM refinement layers to generate our final VOS predictions(Shi et al., 2015).

3.6. DMM-Net Results

The dataset used for training and validation are YouTube-VOS(Xu et al., 2017), DAVIS 2017(Perazzi et al., 2016), and SegTrack v2. Some primary metrics the authors used to score different models are

- mIoU*, the mean Intersection-over-Union over all frames, used for the SegTrack dataset
- $\mathcal{G}_{\mathcal{M}}$, the average match score between a model's prediction and actual object instance, used for YouTube-VOS and DAVIS 2017

	$\mathcal{J}_S$	$\mathcal{J}_U$	$\mathcal{F}_S$	$\mathcal{F}_U$	$\mathcal{G}_M$
OSMN [46]	60.0	40.6	60.1	44.0	51.2
SiamMask [41]	60.2	45.1	58.2	47.7	52.8
RGMP [43]	59.5	-	45.2	-	53.8
OnAVOS [39]	60.1	46.6	62.7	51.4	55.2
RVOS [37]	63.6	45.5	67.2	51.0	56.8
S2S [44]	66.7	48.2	65.5	50.3	57.7
OSVOS [6]	59.8	54.2	60.5	60.7	58.8
S2S [44]	**71.0**	**55.5**	**70.0**	**61.2**	**64.4**
DMM-Net	60.3	50.6	63.5	57.4	58.0

Methods	$\mathcal{J}_M$	$\mathcal{F}_M$	$\mathcal{G}_M$
MaskRNN [14]	45.5	-	-
OSMN [46]	52.5	57.1	54.8
FAVOS [9]	54.6	61.8	58.2
VideoMatch [15]	56.5	-	-
MSK [28]	63.3	67.2	65.3
RGMP [43]	64.8	68.6	66.7
FEELVOS [38]*	65.9	72.3	69.1
DyeNet [21]	67.3	71.0	69.1
DMM-Net	**68.1**	**73.3**	**70.7**

Methods	Online Learning	mIoU*	mIoU†
OSVOS [6]	✔	61.9	65.4
OFL [36]	✔	67.5	-
MSK [28]	✔	67.4	70.3
RGMP [43]		-	71.1
MaskRNN [14]		72.1	-
LucidTracker [17]	✔	-	77.6
DyeNet [21]		-	78.3
DyeNet [21]	✔	-	**78.7**
DMM-Net		**76.8**	76.7

Figure 7. Results of running various models on YouTube-VOS, DAVIS 2017, and SegTrack v2, respectively. Subscripts S and U stand for "Seen" and "Unseen", referring to object categories in the validation set that have previously been seen in the training set or not. Subscript M refers to mean. Measurement is taken on each table by the Jaccard Index, or IoU.

In addition, some supplemental metrics offer more depth to each model's efficacy:

- mIoU†, the mean IoU over all instances of appearance
- $\mathcal{J}$, the region score of the prediction, computed as a Jaccard Index (IoU) of the pixels within the mask (Pont-Tuset et al., 2017)
- $\mathcal{F}$, the boundary (contour) score of the prediction, computed as IoU of pixels at the spatial extent of the mask.

Referring to the table above, each score reported tracks the IoU score for each outputted prediction against the actual instance, meaning a higher score reflects a more accurate prediction. For the YouTube-VOS dataset(left), while DMM-Net underperforms S2S, a sequence-to-sequence RNN driven deep network (Xu et al., 2018), it ranks as one of the highest scores in the table. For DAVIS 2017, DMM-Net outperforms other models significantly, and for SegTrack v2, DMM-Net tends toward the higher scores. Outside of accuracy, we compare runtime and computational efficiency with other models. DMM-Net achieves competitive results in half of S2S's time. Additionally, for the backbone used for feature extraction, using a different backbone (i.e. ResNet-50, ResNet-101, ResNet-152, or other) could significantly improve DMM-Net's accuracy.

3.7. Conclusion

DMM-Net achieves competitive results for video object segmentation with significant advantages in runtime. These results are obtained by use of a discrete solver relaxation, i.e. modified minimum cost bipartite matching. The relaxed discrete solver applied to a deep network as a layer allows improved speeds without sacrificing much accuracy, if any. Additionally, DMM-Net's novelty offers room for growth, perhaps by using different backbones to obtain accuracy improvements, or selecting a multi-partite matching problem rather than bipartite. Overall, DMM-Net establishes itself as a solid competitor in VOS, and introduces the capabilities of a differentiable relaxed discrete solver in a computer vision setting, opening the door for novel solutions in the discrete solver-continuous method field.

4. Discussion

SATNet and DMM-Net share the core concept of relaxing a discrete solver for use as a layer in neural networks, a continuous method. Though the two algorithms deal with separate problem sets — Sudoku puzzle-solving for SATNet and video object segmentation for DMM-Net — both algorithms successfully and competitively accomplish their tasks as novel solutions to a difficult problem. In both instances, relaxing a discrete problem to a continuous, differentiable one allowed significant runtime improvements, and were more applicable to machine learning's acceptance of fewer rules and correctness.

For SATNet, that meant modifying a MAXSAT problem, which traditionally runs until exhaustion to find the maximum satisfying set, to accept approximations that, though not 100% accurate, provide reasonably close results for a fraction of the time. For DMM-Net, that meant relaxing a traditionally ILP problem, the minimum cost bipartite matching problem, to an LP problem by allowing real numbers in the cost matrix, rather than exhaustively computing the single unique cost matrix binary solution.

Utilizing this relaxed discrete solver as a layer in a continuous method allows the model to tackle traditionally unresolvable differences between discrete solvers and continuous methods. SATNet blazes a path for solving previously high-infrastructure demanding problems with low guidance neural networks, whereas DMM-Net introduces using relaxed discrete solvers to solve problems that may not need

100% accuracy. Overall, discrete solvers for continuous methods is a burgeoning field in optimization and machine learning, and is one we expect to make large advancements to ways we approach problems in the future.

References

Amos, B. and Kolter, J. Z. Optnet: Differentiable optimization as a layer in neural networks. arXiv preprint arXiv:1703.00443, 2017.

Dykstra, R.L. An algorithm for restricted least squares regression. Journal of the American Statistical Association, 78(384):837–842, 1983

Fernandez-Moral, E., Martins, R., Wolf, D., and Rives, P. A new metric for evaluating semantic segmentation: leveraging global and contour accuracy. Workshop on Planning, Perception and Navigation for Intelligent Vehicles, PPNIV17, Sep 2017, Vancouver, Canada.

Goemans, M. X. and Williamson, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of the ACM (JACM), 42(6):1115–1145, 1995.

Gomes, C. P., van Hoeve, W.-J., and Leahu, L. The power of semidefinite programming relaxations for max-sat. In International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, pp. 104–118. Springer, 2006.

He, K., Gkioxari, G., Dollar, P., and Girshick, R. Mask R-CNN. In ICCV, 2017.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollar, P., and Zitnick, C.L. Microsoft coco: Common objects in context. In ECCV, 2014.

Palm, R. B., Paquet, U., and Winther, O. Recurrent relational networks. arXiv preprint arXiv:1711.08028, 2017.

Park, K. Can neural networks crack sudoku?, 2016. URL https://github.com/Kyubyong/sudoku.

Perazzi, F., Pont-Tuset, J., McWilliams, B., Van Gool, L., Gross, M., and Sorkine-Hornung, A. "A benchmark dataset and evaluation methodology for video object segmentation," in CVPR, 2016.

Pont-Tuset, J., Perazzi, F., Caelles, S., Arbelaez, P., Sorkine-Hornung, A., and Van Gool, L. The 2017 DAVIS challenge on video object segmentation. arXiv preprint arXiv:1704.00675, 2017.

Selsam, D., Lamm, M., Bunz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a sat solver from single-bit supervision. arXiv preprint arXiv:1802.03685, 2018.

Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and Woo, W.-C. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In NIPS, 2015.

Vlastelica, M., Paulus, A., Musil, V., Martius, G., and Rolínek, M. "Differentiation of Blackbox Combinatorial Solvers," in ICLR, 2020.

Wang, P.-W. and Kolter, J. Z. Low-rank semidefinite programming for the max2sat problem. In AAAI Conference on Artificial Intelligence, 2019.

Wang, P.-W., Donti, P.L., Wilder, B., and Kolter, J. Z. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. arXiv preprint arXiv:1905.12149, 2019.

Wang, P.-W., Chang, W.-C., and Kolter, J. Z. The mixing method: coordinate descent for low-rank semidefinite programming. arXiv preprint arXiv:1706.00476, 2017.

Xu, N., Yang, L., Fan, Y., Yue, D., Liang, Y., Yang,J., and Huang, T. Youtube-vos: A large-scale video object segmentation benchmark. arXiv preprint arXiv:1809.03327, 2018.

Xu, N., Yang, L., Fan, Y., Yang, J., Yue, D., Liang, Y., Price, B.L., Cohen, S., and Huang, T.S. Youtube-vos: Sequence-to-sequence video object segmentation. In ECCV, 2018.

Zeng, X., Liao, R., Gu, L., Xiong, Y., Fidler, S., and Urtasun, R. DMM-Net: Differentiable Mask-Matching Network for Video Object Segmentation. arXiv preprint arXiv:1909.12471, 2019.