First Order Optimization Methods in Training Deep Neural Networks

Rui Zhang, and Yujia Liu

1 Problem Statement

In this project, we have built a stacked denoising autoencoder to classify two datasets, namely MNIST[3] and CIFAR[2]. We also implement this deep neural network architecture in C++. All matrix manipulations are done with Eigen3 library. We then implement three variants of gradient descent method, namely stochastic gradient descent (SGD)[1], adaptive gradient descent (AdaGrad)[6] and Nesterov's accelerated gradient (NAG)[5].We compare them using both datasets in terms of accuracy, converge rate, etc.

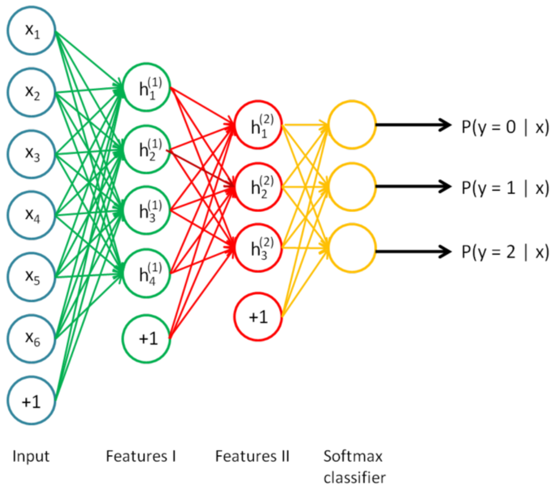

Figure 1: Stacked denoising autoencoder

2 Framework

Given the labeled data as training set, the stacked denoising autoencoder (SDA) consisting of multiple layers of autoencoder is able to make prediction on new data from the test set. In our project, the SDA has two layers of autoencoders connected with a softmax classifier.

Softmax Classifier

The classification is done by projecting an input vector onto a set of hyperplanes, each of which corresponds to a class. The distance from the input to a hyperplane reflects the probability that the input is a member of the corresponding class.

$$\begin{aligned} P(Y=i|x,W,b) &= \text{softmax}_i(Wx+b) \\ &= \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}} \end{aligned}$$

where the input vector x is a member of a class i, a value of a stochastic variable Y, W and b are the parameter matrix and vector in the classifier. In other words, we want the label $\text{argmax}_i P(Y=i|x,W,b)$ to match the true label as much as possible for all data. This can be done by minimizing a cost function.

Denoising Version

Since sometimes the input data suffers from noise and might not explicitly contain the information that we are interested, a denoising version of stacked autoencoder is applied here. Before we feed the data into the model, we first stochastically corrupted it with some noise. This allows the representation to be more robust from the noisy input and is useful for recovering the corresponding clean input.

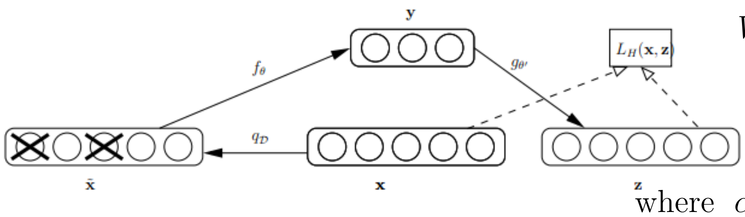

Figure 2: Denoising autoencoder

Cost Function

In our project, we define the following function as our loss function.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2$$

$$= \left[\frac{1}{m}\sum_{i=1}^{m}(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2$$

where $h_{W,b}$ is the output of the last layer of the autoencoder. The first term is an average sum of squares error term and the second term is a regularization term that tends to decrease the magnitude of the weights and therefore avoid overfitting. In the cost function, n_l is the number of layers and s_l is the length of the input vector.

Gradient Descent

To minimize the cost function, the basic idea is applying gradient descent to find the optimal parameters of the model. For each iteration, the parameters w and b are updated as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(W,b)$$

where α is the learning rate. The difficulty here is computing the partial derivatives above, and this is done by the backpropagation algorithm. In this project, we implemented three different versions of the variants of this basic gradient algorithm. Details will be included in the next section.

Backpropogation Algorithm

The idea of backpropogation algorithm is that we first run forward the model to compute all the output value $a_i^{(l)}$ of each layer, including the $h_{W,b}(x)$. Then we compute an "error term" $\sigma_i^{(l)}$ that measures how much the node effects the output value. And the gradient w.r.t each parameter is computed as follows:

For layer n_l

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}}\frac{1}{2}\|y-h_{W,b}(x)\|^2 = -(y_i-a_i^{(n_l)})f'(z_i^{(n_l)})$$

For layers $l = 2,3,\cdots,n_l-1$

$$\delta_i^{(l)} = (\sum_{j=1}^{s_l+1} W_{ji}^{(l)}\delta_j^{(l+1)})f'(z_i^{(l)})$$

Then the gradients are

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$

3 First Order Optimization Methods

In this project, we implemented three first order optimization methods to train this neural networks, i.e stochastic gradient descent (SGD), adaptive gradient descent (AdaGrad) and Nesterov's accelerated gradient (NAG). Suppose Dataset is D and the loss function is defined as

$$L(w) = \frac{1}{D} \sum_{i}^{D} F_w(x_i) + \lambda R(w)$$

Instead of using all the dataset(batch) or randomly picked one(sequential), we use a small set of the training data to find the gradients and update.

$$L(w) \approx \frac{1}{N} \sum_{i}^{N} F_w(x_i) + \lambda R(w), \text{ where } N \ll D$$

In the following three methods, the gradients are calculated using a small batch of the whole dataset. This saves time than looping through all the dataset and is more stable than randomly pick one sample.

3.1 Non-convexity of neural networks

The loss function that we hope to minimize in neural networks is highly non-convex and the curvature of such function can be long and narrow, which calls for optimization algorithms that could rescale the learning rate to jump out of the valley. As can be seen in

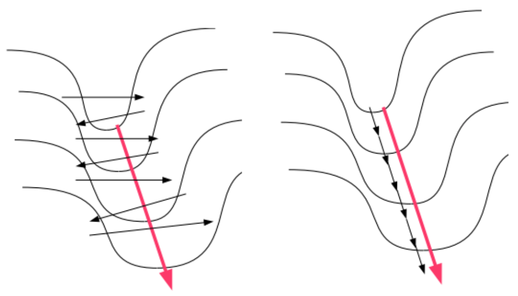

Figure 3: Narrow valley of the loss function

Figure 3, vanilla gradient descent without any modification would lead to the first scenario in Figure 3. The three methods we implemented are aimed to avoid frequent instability and achieve better convergence rate. v

3.2 Stochastic Gradient Descent (SGD)

The update rule for this algorithm is that

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t)$$
$$W_{t+1} = W_t + V_{t+1}$$

The intuition behind this update rule is that by combing the descent direction of the last update, the current direction would be more stable. As mentioned above, the gradients are calculated using a small batch of the dataset, instability would occur if we only update the weights based on the current gradients. By incorporating the last update, hopefully the loss function would drop in a more a smooth and stable fashion.
Theoretical analysis reveal that the convergence rate of stochastic gradient descent vary between $r^t = t^{-1}$ to $r^t = t^{-\frac{1}{2}}$

- When the Hessian matrix of the loss function at the global optimal is strictly positive definite, the best convergence speed obtained using SGD is $r^t = t^{-1}$[4]. This is on par with batch gradient descent, but is far more time-efficient.

- When this assumption does not hold, numerical study [7] shows that the learning rate is similar to $r^t = t^{-\frac{1}{2}}$.

3.3 Adaptive Gradient Descent(AdaGrad)

The update rule for AdaGrad is as follows:

- $\alpha_{t,i} = \frac{\alpha}{\sqrt{G_{ti}}} = \frac{\alpha}{\sqrt{\sum_1^t W_{t',i}^2}}$

- $W_{t+1,i} = W_{t,i} - \alpha_{t,i} \nabla L(W_t)_i$

The intuition behind this update rule is that the sparsity of each feature is not the same. Therefore it may not be a good idea to set the learning rate the same for all the features. By setting unique learning rate for each feature, one can hope that at certain frequent axises, the learning rate is small and stable but at sparse axises, the weights drops more steeper and thus the loss function could drop faster. This makes sense because for sparse features, one can set the learning rate higher for the info about those features are not sufficient enough and it is ok to drop faster. But for more frequent features, higher chance is that the it is already stuck in the long narrow valley, higher learning rate is more likely to bring about oscillations and thus is unstable.

Unlike SGD, the lower bound of convergence rate for AdaGrad is guaranteed to be $r^t = t^{-\frac{1}{2}}$.[7]. However the convergence rate generally depends on the dataset and each algorithm may perform differently under different datasets.

3.4 Nesterov's Accelerated Gradient(NAG)

Compared to SGD, weights are also updated when computing the loss function.

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t + \mu V_t)$$

$$W_{t+1} = W_t + V_{t+1}$$

The intuition behind this is shown in Figure 4. Consider a scenario where the addition of

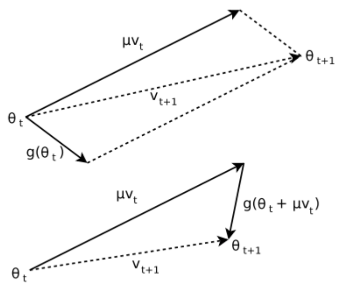

Figure 4: NAG algorithm, upper is NAG and lower is SGD [5]

μv_t results in an immediate undesirable increase in the objective f. The gradient correction to the velocity v_t is computed at position $\theta t + \mu v_t$ and if μv_t is indeed a poor update, then $\nabla f(\theta t + \mu v_t)$ will point back towards θt more strongly than $\nabla f(\theta t)$ does, thus providing a larger and more timely correction to v_t than SGD.[5]

4 Experiements

4.1 Convergence Plot of MNIST

First, we did an experiment on MNIST comparing the convergence rate of three algorithms. Also to emphasis the importance of momentum, we also compared the performance of vanilla gradient descent without considering the previous update. As can be

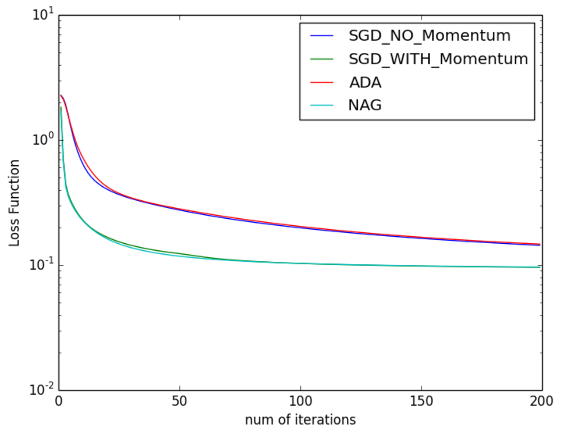

Figure 5: Convergence Rate of different algorithms

seen in the plot, NAG and SGD(Momentum) drop faster than the other two.

4.2 Test Error of MNIST

After 200 iterations, the corresponding test error is

	NaiveSGD	SGD	ADA	NAG
Error	3.22%	**1.82%**	3.28%	2.16%

Table 1: Error

	NaiveSGD	SGD	ADA	NAG
Time	9.1973	**8.9759**	9.94	9.15

Table 2: Running time in minutes

SGD with momentum outperforms other methods in both recognition rate and running time.

4.3 Convergence Plot of Cifar10

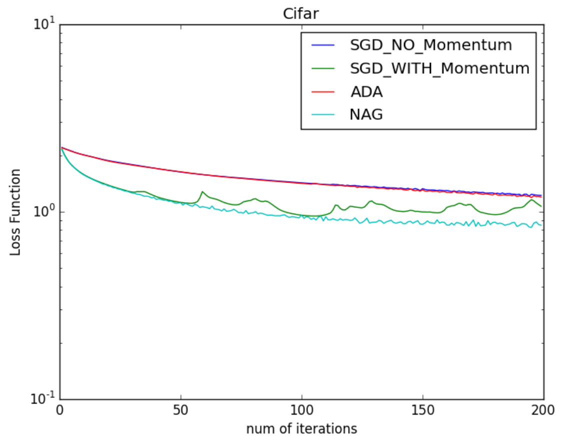

Figure 6: Convergence Rate of different algorithms

NAG and SGD(Momentum) still outperforms the other two. But SGD is more volatile.

4.4 Test Error of Cifar

With 100 hidden units and 200 iterations, the corresponding test error is

	NaiveSGD	SGD	ADA	NAG
Error	53.22%	51.82%	**47.28%**	52.16%

Table 3: Error

	NaiveSGD	SGD	ADA	NAG
Time	26.79	**27.09**	29.78	28.12

Table 4: Running time(in minutes)

This time ADA performs best but more number of hidden units are called for to increase the capacity of the model.

4.5 Sensitivity to Learning Rate

The learning rate in training neural network needs to be tuned by hand and thus a study of the sensitivity of each algorithm to the learning rate is called for. In this project, we only did the sensitivity test on MNIST due to the time pressure. As can be seen, the

	Naive	SGD	Ada	NAG
α=1	2.26%	70.02%	45.09%	43.12%
α=0.2	2.12%	1.98%	2.12%	2.23%
α=0.02	7.57%	2.78%	2.34%	3.12%
α=0.002	18.09%	7.12%	6.78%	7.45%
α=0.0002	59.23%	19.12%	13.63%	21.12%

Table 5: Learning Rate VS Different Methods

SGD with/without momentum is more sensitive to the learning rate, which makes sense because there is no learning rate scaling compared to the other two methods. But SGD with momentum is more stable than the one without momentum because the last update is take account of during update and thus is more stable.

5 Conclusion

In this project, we did a through study of three algorithms and compared their performance on two datasets in terms of accuracy, convergence rate and running time. Some conclusions we have right now are:

- NAG and SGD with momentum drops faster than other two methods
- SGD with/without momentum are more sensitive to parameters tuning
- A well-tuned SGD with momentum seems to perform the best

All in all, training neural networks is difficult and it calls for a lot of parameter tuning and choices of optimization algorithms. The non-convexity of the networks also make the optimization more challenging.

References

[1] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[2] Alex Krizhevsky and G Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 2010.

[3] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits, 1998.

[4] Noboru Murata. A statistical study of on-line learning. *Online Learning and Neural Networks. Cambridge University Press, Cambridge, UK*, pages 63–92, 1998.

[5] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

[6] MA Vorontsov, GW Carhart, and JC Ricklin. Adaptive phase-distortion correction

based on parallel gradient-descent optimization. *Optics letters*, 22(12):907–909, 1997.

[7] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. 2003.