First Order Optimization Methods in Training Deep Neural Networks

Rui Zhang Yujia Liu

April 2015

Outline

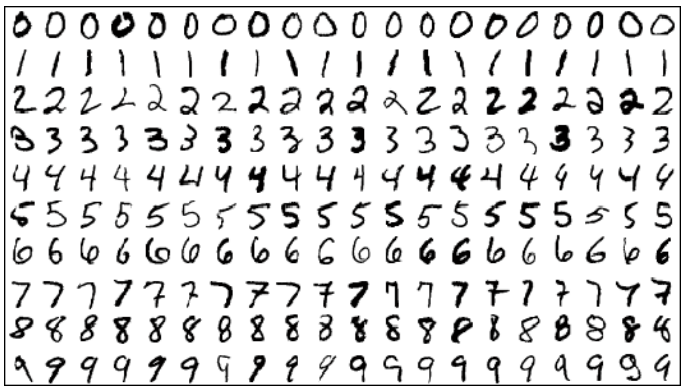
1. Objective
2. Datasets
   - MNIST
   - CIFAR
3. Framework
   - Denoising Autoencoder
   - Softmax
   - Stacked Denoising Autoencoders
4. Algorithms
   - First Order Algorithm
   - Methods Details
5. Experiment & Result
6. Conclusion

Objective

- Framework
  Build a Stacked Denoising Autoencoder to classify datasets such as MNIST and CIFAR.
- Algorithms
  Implement different First Order Optimization algorithms and compare them in terms of accuracy, converge rate, etc.

MNIST

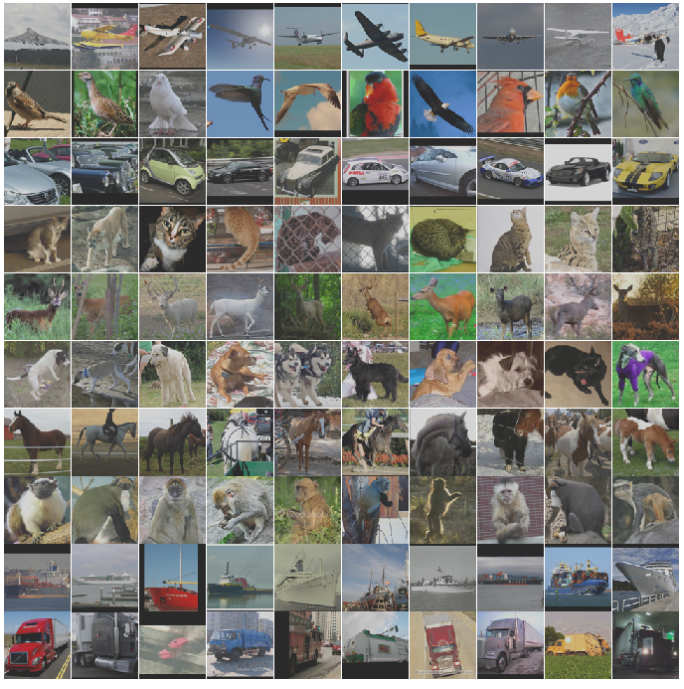
The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a 28x28 fixed-size image.



LeCun Y, Bottou L, Bengio Y, et al.
Gradient-based learning applied to document recognition.

CIFAR

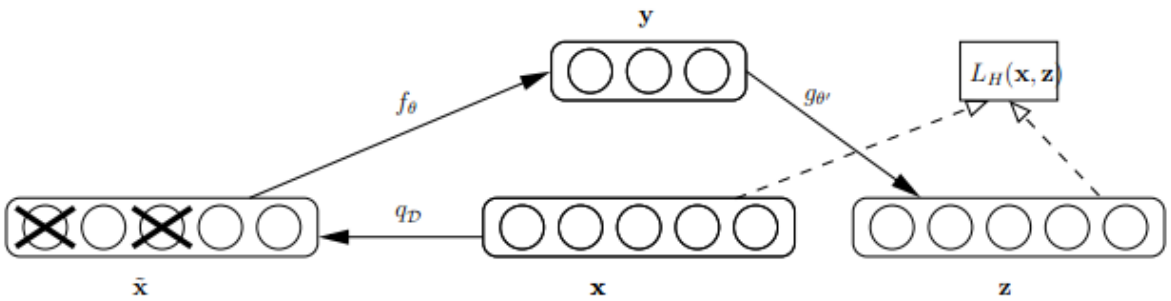
The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.



Krizhevsky A, Hinton G
Learning multiple layers of features from tiny images.

Denoising Autoencoder

By stochastically corrupting the signal, a denoising autoencoder neural network attempts to capture the representation of the signal that is invariant to noise.



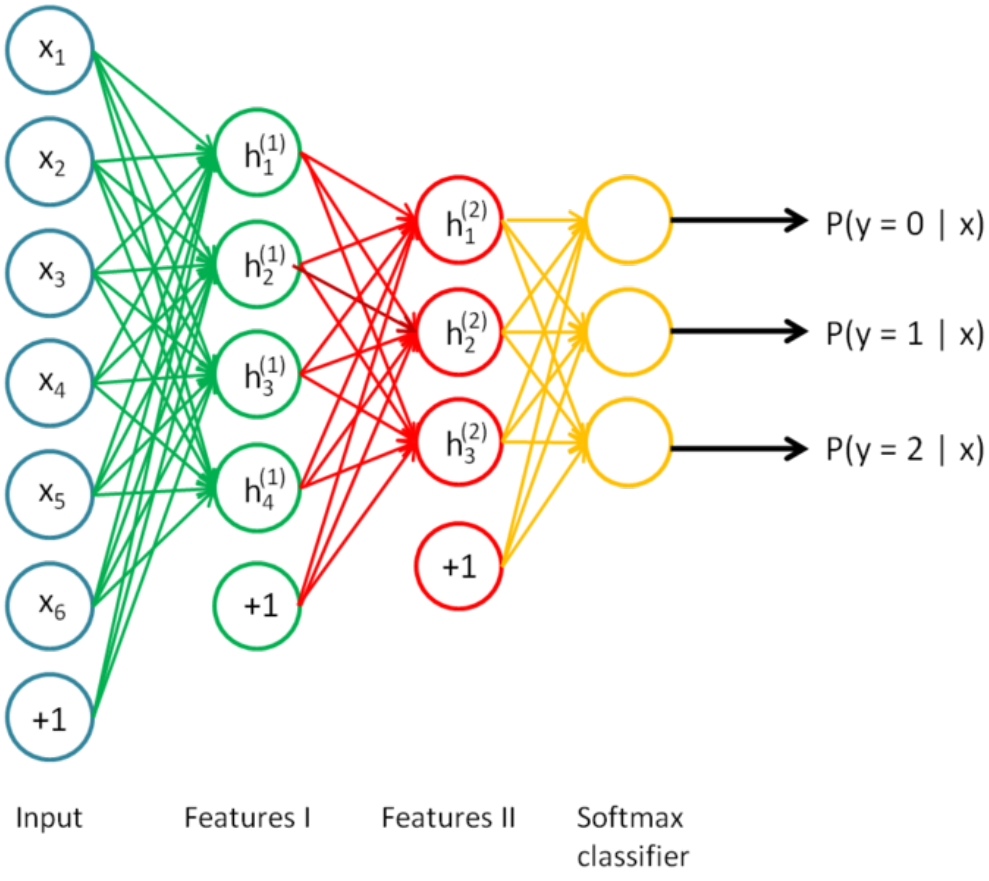
Softmax

Softmax model generalizes logistic regression to classification problems where the class label can take on more than two possible values.

$$h_\theta(x^{(i)}) = \begin{bmatrix} P(y^{(i)} = 1|x^{(i)};\theta) \\ P(y^{(i)} = 2|x^{(i)};\theta) \\ \vdots \\ P(y^{(i)} = k|x^{(i)};\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$

Stacked Denoising Autoencoders

A stacked denoting autoencoder is a neural network consisting of multiple layers of denoising autoencoders in which the outputs of each layer is wired to the inputs of the successive layer.



Georgia Tech

Methods we implemented

- Stochastic Gradient Descent
- Adaptive Gradient Descent
- Nesterov's Accelerated Gradient

John Duchi
Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.

Ilya Sutskever
On the importance of initialization and momentum in deep learning.

Why First Order

Reasons not to choose second order optimization methods like BFGS

- Constructing a Hessian Matrix is memory consuming
  - $28 * 28$ image, 100 hidden units a layer, single layer, double precision: $(28 * 28 * 100 * 1)^2 * 8 = 49e9$ byte $= 49GB$, Unacceptable
- Computational expensive and no clear improvements for highly non convex neural networks compared to First Order Methods

Minibatch Optimization

Suppose Dataset is D and the loss function is defined as

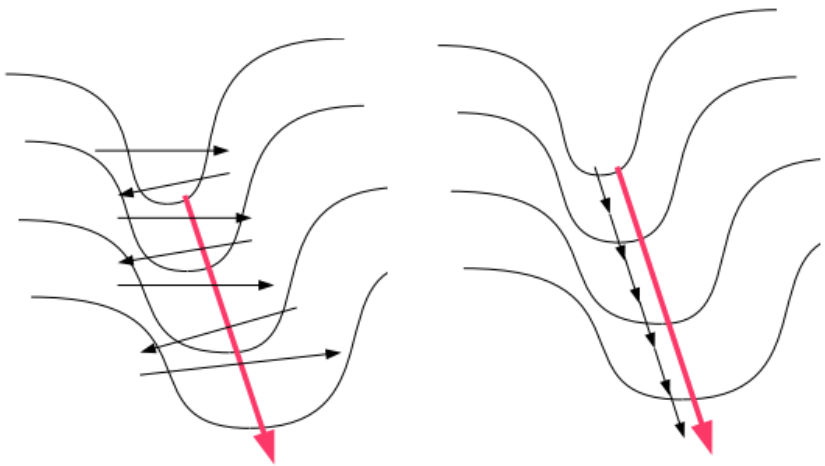
$$L(w) = \frac{1}{D}\sum_{i}^{D} F_w(x_i) + \lambda R(w)$$

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

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

Optimization in a long narrow Valley

- Highly non-convex loss function
- Loss function is the Long narrow valley and the arrow is the gradient descent direction



SGD

Suppose momentum, the weight of previous update is μ, the learning is α, the Weights W are updated in the following fashion

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

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

Rules of setting α and μ

- $\mu = 0.5$ in the beginning and set to 0.9 later
- α=0.01 is a good start and drop by 10 when loss reaches a plateau

AdaGrad

- Per-feature learning rate $\alpha_{t,i} = \frac{\alpha}{\sqrt{G_{ti}}} = \frac{\alpha}{\sqrt{\sum_1^t W_{t',i}^2}}$
- Update Rule $W_{t+1,i} = W_{t,i} - \frac{\alpha}{\sqrt{\sum_1^t W_{t',i}^2}} \nabla L(W_t)$

Implementations details: Only O(f) extra storage and one extra element wise vector multiplication is needed.
Advantage: Sparse features are weighted more and thus could help higher recognition rate.

NAG

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

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

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

Advantage: If μV_t is a poor update, then $\nabla L(W_t + \mu V_t)$ will point backwards to W_t more than $\nabla L(W_t)$

Convergence Plot of MNIST

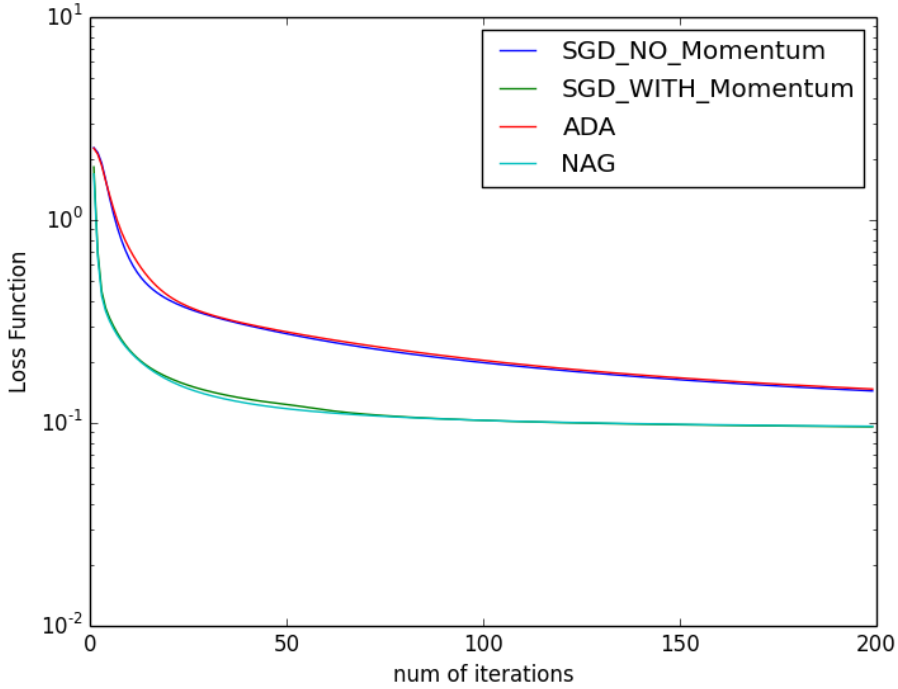


Figure: Convergence Rate of different algorithms

NAG and SGD(Momentum) seems to drop faster than the other two.

Georgia Tech

Test Error of MNIST

After 200 iterations, the corresponding test error is

	SGD(No Momentum)	SGD	ADA	NAG
Error	3.22%	**1.82%**	3.28%	2.16%

Table: Error

	SGD(No Momentum)	SGD	ADA	NAG
Time(minutes)	9.1973	**8.9759**	9.94	9.15

Table: Running time

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

Georgia Tech

Convergence Plot of Cifar10

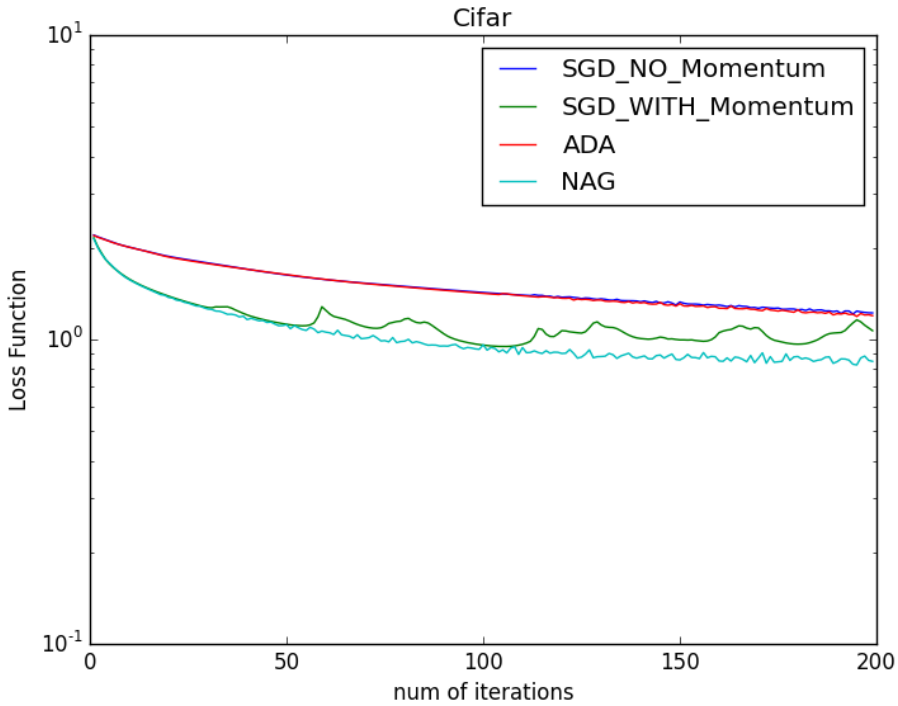


Figure: Convergence Rate of different algorithms

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

Georgia Tech

Test Error of Cifar

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

	SGD(No Momentum)	SGD	ADA	NAG
Error	53.22%	51.82%	**47.28%**	52.16%

Table: Error

	SGD(No Momentum)	SGD	ADA	NAG
Time(minutes)	26.79	**27.09**	29.78	28.12

Table: Running time

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

Georgia Tech

Sensitivity to Learning Rate

	SGD(no Momentum)	SGD	Ada	NAG
α=1	2.26%	70.02%	45.09%	43.12%
α=2e-1	2.12%	1.98%	2.12%	2.23%
α=2e-2	7.57%	2.78%	2.34%	3.12%
α=2e-3	18.09%	7.12%	6.78%	7.45%
α=2e-4	59.23%	19.12%	13.63%	21.12%

Table: Learning Rate VS Different Methods

Conclusion

- 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

Implementation Platforms

- C++ with Eigen
- Code only tested on my own Mac and Jinx