ECE598: Representation Learning Fall 2017

Lecture 10: Training of Neural Networks

Lecturer: Prof. Pramod Viswanath *Scribe: Tao Sun, Sep 28, 2017*

10.1 Neural Networks: Recap

Artificial Neural Networks were developed stimulated by the behavior of brains as networks of neurons[RHW85, MP43]. Each neuron receives signals through synapses controlling the strength of the signal on the neuron. The network is activated by input nodes providing signals, and this activation spreads out the network along the weighted connections. Figure 10.1(a) presents a schematic view of an artificial neuron, showing the 3 basic components of the neuron: (1) The synapses or connections with weights w_i, associated with the input value x_i, for $i = 1, 2, ...m$. (2) An adder sums the weighted input values to compute the input to the activation function, $v = w_0 + \sum_{i=1}^{m} w_i x_i$, where w_0 is called the bias. (3) An activation function g that maps v to $g(v)$, giving the output of the neuron. The two most common choices of g are the hyperbolic tangent $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$, and the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. Sometimes in order for simplicity, people use one circle to represent both the input summation and the output activation function evaluation.

Among different neural netowrk architectures, feedforward neural networks (FNNs) are of extreme importance because they form the basis of many commercial applications. Figure 10.1(b) shows a schematic view of a FNN, where there is no feedback connections (the network is acyclic). The first layer is the input layer, consisting of neurons that simply generate the input signals. Successive layers are formed with different number of neurons. The outputs of neurons in one layer are the inputs to the neurons in the next layer. The last layer is called output layer. The layers between the input and output layers are called hidden layers. When the network is used for predict a numerical quantity, there is normally only one neuron in the output layer. When the network is used for classification, the output layer may have many neurons showing the probability of the corresponding class. There are connections between successive layers, each connection transfers the output of a neuron i in the current layer to the input of a neuron j in the next layer, with a weight w_{ij}. The input $p_j(t)$ to the neuron j in layer t is calculated from the outputs $o_i(t-1)$ of predecessive neurons in layer $(t-1)$ as $p_j(t) = w_0 + \sum_i o_i(t-1)w_{ij}$, where w_0 is the bias. The output $o_j(t)$ is calculated with the activation function of neuron j as $o_j(t) = g(p_j(t))$.

The outputs of all the neurons in the network are computed with a forward pass fashion. The algorithm starts with the first hidden layer, with the input layer providing all the input signals. The neuron outputs are computed for all neurons in the first hidden layer by performing the relevant weighted summation and activation function evaluations. These outputs are the inputs for the neurons in the second hidden layer. Again the relevant weighted summation and activation function evaluation are performed to compute the outputs of the neurons in the second hidden layer. This continues layer by layer until reaching the output layer and the outputs are generated. Inside FNNs, there are no feedback connections in which outputs of the model are fed back into itself.

Other neural network architectures such as recurrent neural networks with feedback connections and convolutional neural networks with translation invariance characteristics could be referred to Lecture 9. Here in this brief recap we do not cover those details.

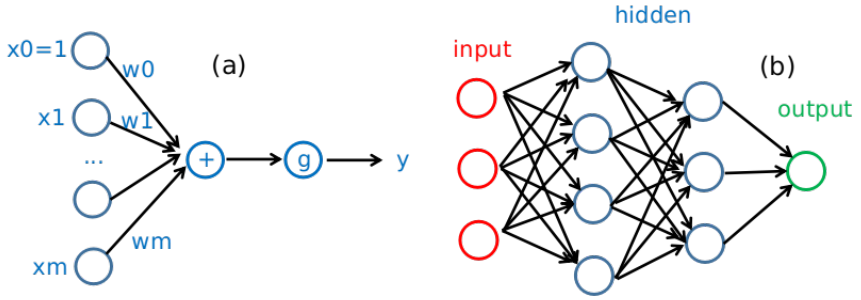


Figure 10.1: (a)A schematic artificial neuron, with x_i representing input signals, w_i representing corresponding weights. g is the activation function and y is the output. Notice that the input $x_0 = 1$ actually added the bias w_0 into the system. (b)A schematic feedforward neural network, with the input layer, hidden layers and output layer.

10.2 Objective Functions

In order to train the parameters $\boldsymbol{w}$ of the neural network, we need to have an objective function. When the neural network is used to predict a numerical value, the objective function is simply the averaged square error with respect to input samples.

$$J = \frac{1}{2N}\sum_{\boldsymbol{X}}(y - \widehat{y}(\boldsymbol{X}, \boldsymbol{w}))^2,$$

where $\boldsymbol{X}$'s are the training sets, with N examples. y's are the true output and the $\widehat{y}$'s are the estimations from the neural network.

When the neural network is used to do binary classification, the objective function is

$$J = -\sum_{(\boldsymbol{X},y)\in S} y \log \widehat{y}(\boldsymbol{X}, \boldsymbol{w}) + (1-y)\log(1 - \widehat{y}(\boldsymbol{X}, \boldsymbol{w})),$$

where $y \in \{0, 1\}$ is the true classifier while $\widehat{y} \in \{0, 1\}$ is the estimated classifier. Similarly for problems with multiple classes, the objective function is

$$J = -\sum_{(\boldsymbol{X},y_k)\in S}\sum_{k=1}^{K} y_k \log \widehat{y}_k(\boldsymbol{X}, \boldsymbol{w}).$$

A central problem in neural networks and other machine learning techniques is how to prevent overfitting, so that the model performs well not only for training set but also for testing set. Regularization is used to modify the algorithm to reduce its variance significantly while not overly increasing the bias. Here, we perform parameter norm penalties which limits the capacity of the model by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J. The L^2 parameter regularization is as the following.

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^T\boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

as is used in usual way, the L^2 regularization would insure the direction along which the parameters contribute significantly to reducing the objective function are preserved relatively intact, while

the direction along which the parameters do not contribute to reducing the objective function significantly are decayed away. This could shrink the weights on features having low correlation with the output target[GBC16].

An alternative is to use L^1 regularization,

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha||\boldsymbol{w}||_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

which could result in a solution in which some parameters have an optimal value of zero. This property induced by L^1 norm could be used as a feature selection mechanism, suggesting some features could be safely discarded[GBC16].

10.3 Backpropagation Algorithm

We now have the objective function J, the next step is to obtain the derivatives of J with respect to all the neural network parameters $\boldsymbol{w}$, in order to use them in the successive optimization with gradient based method. This is achieved by recursively using chain rule.

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Here we consider acyclic networks with directed edges, such as FNNs. Without loss of generality, the network is structured in different layers, with neurons in layer $(t+1)$ getting all their inputs from the outputs of neurons in layer t. Assume the general output of the network is $f \in \boldsymbol{R}$, which could represent the cost function J. Then we have the following theorem.

Theorem 10.1. *To compute the derivative of f with respect to the parameters $\boldsymbol{w}$, it suffices to compute $\partial f/\partial p$ for every neuron, where p is the input value of the corresponding neuron.*

We know that the input value p of a neuron at layer t is calculated as $p(t) = \sum_j w_j o_j(t-1)$, where $o_j(t-1)$ is the the output of neuron j in layer $(t-1)$ that is connected to the target neuron in layer t. Then it is obvious by the chain rule

$$\frac{\partial f}{\partial w_i} = \frac{\partial f}{\partial p(t)} \frac{\partial p(t)}{\partial w_i} = \frac{\partial f}{\partial p(t)} o_i(t-1),$$

where the $o_i(t-1)$'s are already known through forward propagation.

Backpropagation algorithm emerges by computing the partial derivatives in the reverse direction. The messages are passed through the backward direction from the higher-index layers to lower-index layers, based on the following fashion.

Theorem 10.2. *The neuron u receives a message from each of its child neuron from a higher-index layer. It sums these message to get a number S (if u is the output of the entire network, define $S = \partial f/\partial p_f$) and sends the following message to any of its parent neuron z in a lower-index layer: $S \cdot \frac{\partial p_u}{\partial p_z}$. Then at each neuron z, the value S is exactly $\partial f/\partial p_z$.*

The proof of the theorem could be achieved through inductions[AM16]. First look at the base case, at the output layer, this is true, since $\partial f/\partial p_f = S$. Suppose the claim to be true for neurons $u_1, u_2, ...u_m$ at layer $(t+1)$. Then for their common parent neuron z at layer t, by hypothesis, neuron z receives $\frac{\partial f}{\partial p_{u_j}}\frac{\partial p_{u_j}}{\partial p_z}$ from each of the u_j. By chain rule, the S value for z is $S = \sum_{j=1}^{m} \frac{\partial f}{\partial p_{u_j}}\frac{\partial p_{u_j}}{\partial p_z} = \partial f/\partial p_z$. This completes the induction proof. Then we take a look at the message sent.

$$S \cdot \frac{\partial p_u}{\partial p_z} = S \cdot \frac{\partial p_u}{\partial o_z}\frac{\partial o_z}{\partial p_z} = S \cdot w_{zu} \cdot \frac{\partial}{\partial p_z} g(p_z),$$

where g is the activation function of the neurons. As stated in Section 10.1, the common choices for the activation function are hyperbolic tangent function and the sigmoid function, the derivatives of both could be obtained easily.

Clearly, the amount of work done by each neuron (receiving messages, summation, derivative computation and sending messages) is proportional to its degree, thus the overall work done are the summation of the degree of the network. The running time is linear $O(V + E)$. The above theorems give the following backpropagation algorithm. This deals with a simplified version where all variables are scalars. The more general case involving tensors could be referred to [GBC16].

Algorithm 1 The Backpropagation Algorithm

Require: Observations $\boldsymbol{X} = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}^T$, neural network $\boldsymbol{w}$ with n layers.
Ensure: The derivatives of output f with respect to parameter set $\boldsymbol{w}$
1: Forward propagation to calculate the output with X as the input.
2: Initialization **derivatives**[n] $= \partial f/\partial p_f$
3: **for** j=n-1 down to 1 **do**
4: **derivatives**$[j] = \sum_{i \in Children(j)}$ **derivatives**$[i]\frac{\partial p_i}{\partial p_j}$
5: $= \sum_{i \in Children(j)}$ **derivatives**$[i] \cdot w_{ji} \cdot \frac{\partial}{\partial p_j} g(p_j)$
6: **end for**

10.4 Gradient Descent based Optimization Methods

We have obtained the derivatives of the loss function (or output) with respect to the neural network parameters $\boldsymbol{w}$, now we need to optimize these parameters based on the gradient, to achieve the minimum loss. Gradient descent method is one of the most popular optimization algorithms for neural networks. Although it is widely used, there are still some challenges. Here we introduce the gradient descent variants and other algorithms developped based upon it.

10.4.1 Gradient Descent Variants

Batch gradient descent computes the gradient of the cost function with respect to the parameter $\boldsymbol{w}$ for the entire training dataset:

$$\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

We need to calculate the gradients for the whole dataset to perform only one update. This shows the batch gradient descent could be quite slow and intractable for data not fit in memory. It also

does not allow updating the parameters with new coming examples on-the-fly. Another problem with the batch gradient descent is that it may got stuck at the local minima, especially when the objective function is non-convex.

Stochastic gradient descent (SGD) instead performs a parameter update for each training example $\boldsymbol{x^{(i)}}$ and $y^{(i)}$.

$$\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{x}^{(i)}, y^{(i)})$$

Note that shuffle the training data at each epoch is required. SGD removes the redundant gradient calculation by performing one update at a time. It is usually much faster and could be used for on-line learning. The frequent updates of SGD could have a high variance, causing the objective function to fluctuate heavily[Rud16]. This fluctuation may cause SGD to jump out of the current local minima and moves to new and potentially better local minima. On the other hand, the fluctuation may also lead to overshooting so we never get a converged solution.

Mini-batch gradient descent takes the n samples each time and performs an update, where $n = 50 \sim 256$.

$$\boldsymbol{w} = \boldsymbol{w} - \eta \cdot \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{x}^{(i:i+n)}, \boldsymbol{y}^{(i:i+n)})$$

By using part of the examples, the mini-batch gradient descent could still maintain a fast approach, at the same time reducing the variance of the parameter update, leading to more stable convergence. Mini-batch gradient descent is typically the choice when training a neural network. However, it still faces some difficulties as (1) It is difficult to choose a proper learning rate to achieve fast convergence and avoid overshooting. (2) The same learning rate applied to all parameter updates may not be able to fit the characteristics of the dataset, like sparsity. (3) The algorithm may get trapped in local minima or saddle points for highly non-convex objective functions. We may need some other extentions.

10.4.2 Stochastic Gradient Descent Extentions

Momentum method is a method dampens oscillations and helps accelerate SGD convergence in the vicinity of local optima where the surface curves are much steeper in one direction than another[Qia99].

$$\boldsymbol{v_t} = \gamma \boldsymbol{v_{t-1}} + \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{x}^{(i)}, y^{(i)})$$

$$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{v_t}$$

where γ is usually set to be 0.9 or a similar value. The name momentum stems from an analogy to momentum in physics: the weight vector $\boldsymbol{w}$, thought of as a particle traveling through parameter space, incurs acceleration from the gradient of the loss. Unlike in classical stochastic gradient descent, it tends to keep traveling in the same direction, preventing oscillations.

Nesterov accelerated gradient (NAG) could prevent from going too fast by momentum method and increase responsiveness of the results[Nes83].

$$\boldsymbol{v_t} = \gamma \boldsymbol{v_{t-1}} + \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w} - \gamma \boldsymbol{v_{t-1}}; \boldsymbol{x}^{(i)}, y^{(i)})$$

$$\boldsymbol{w} = \boldsymbol{w} - \boldsymbol{v_t}$$

where we get an approximation of the next position of the parameters by calculating $\boldsymbol{w} - \gamma \boldsymbol{v_{t-1}}$, and then update. Again the value of γ is 0.9 or similar.

Now we could adapt the parameter update to the slope of the cost function and speed up SGD in turn. We would also like to adapt the parameter update to each features of the input examples, to make a decision about large or small updates depending on their importance.

Adagrad is an algorithm for gradient-based optimization that performs larger updates for infrequent parameters and smaller updates for frequent parameters[DHS11]. For this reason it is well suited to deal with sparse data. Unlike previously used the same learning rate η for all parameter $\boldsymbol{w}$, Adagrad uses a different learning rate for each parameter w_i at each time step t.

$$g_{t,i} = \nabla_{\boldsymbol{w}_t} J(w_{t,i}; \boldsymbol{x}^{(i)}, y^{(i)})$$

$$G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i}^2$$

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where $G_t \in \boldsymbol{R}^{d \times d}$ is a diagonal matrix with its ith diagonal element to be the sum of the squares of the gradient with respect to w_i up to time step t, while ϵ is a smoothing term avoiding division by zero. Now we could vectorize the implementation

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\boldsymbol{G}_t + \epsilon}} \odot \boldsymbol{g}_t$$

Adagrad's main benefits is that it does not need to ajust the learning rates for different features manually, the learning rates will ajust themselves automatically depending on how important the corresponding features are. The main weakness of Adagrad is its accumulation of the squared gradients in the denominator, which could end up as a huge value, making the learning rate to shrink to some value close to zero, and the objective function could no longer be able to proceed toward the minima.

Adadelta is an extension of Adagrad seeking to reduce its aggressive, monotonically decreasing learning rate[Zei12]. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size k.

$$\boldsymbol{g}_t = \nabla_{\boldsymbol{w}_t} J(\boldsymbol{w}_t; \boldsymbol{x}^{(i)}, y^{(i)})$$

$$E[\boldsymbol{g}^2]_t = \gamma E[\boldsymbol{g}^2]_{t-1} + (1-\gamma) \boldsymbol{g}_t^T \boldsymbol{g}_t$$

$$\Delta \boldsymbol{w}_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \boldsymbol{g}_t = -\frac{\eta}{RMS[g]_t} \boldsymbol{g}_t$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta \boldsymbol{w}_t$$

It should be noted that the units in this update do not match, the update does not have the same hypothetical units as the parameter. To resolve this, the updat scheme is modified as

$$E[\Delta \boldsymbol{w}^2]_t = \gamma E[\Delta \boldsymbol{w}^2]_{t-1} + (1-\gamma) \Delta \boldsymbol{w}_t^2$$

$$RMS[\Delta \boldsymbol{w}]_t = \sqrt{E[\Delta \boldsymbol{w}^2]_t + \epsilon}$$

$$\Delta \boldsymbol{w}_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t}\boldsymbol{g}_t$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta \boldsymbol{w}_t$$

We do not even need to set a default learning rate, since it has been eliminated from the update rule.

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton[HSS]. It was developed independently from Adadelta around the same time in order to solve Adagrad's radically diminishing learning rates. The update scheme is as the following.

$$E[\boldsymbol{g}^2]_t = \gamma E[\boldsymbol{g}^2]_{t-1} + (1-\gamma)\boldsymbol{g}_t^T\boldsymbol{g}_t$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{E[\boldsymbol{g}^2]_t + \epsilon}}\boldsymbol{g}_t$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. The suggestions for the default values are $\gamma = 0.9$ and $\eta = 0.001$.

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter[KB14]. In addition to storing an exponentially decaying average of past squared gradients $\boldsymbol{v}_t$ like Adadelta, Adam also keeps an exponentially decaying average of past gradients $\boldsymbol{m}_t$.

$$\boldsymbol{m_t} = \beta_1\boldsymbol{m_{t-1}} + (1-\beta_1)\boldsymbol{g_t}$$

$$\boldsymbol{v_t} = \beta_2\boldsymbol{v_{t-1}} + (1-\beta_2)\boldsymbol{g_t^2}$$

where $\boldsymbol{m_t}$ and $\boldsymbol{v_t}$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. As $\boldsymbol{m_t}$ and $\boldsymbol{v_t}$ are initialized as vectors of 0's, during the initial steps they are found to be biased towards zeros. So a bias-corrected estimates are given, instead.

$$\widehat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-(\beta_1)^t}$$

$$\widehat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-(\beta_2)^t}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\widehat{v}_t} + \epsilon}\widehat{\boldsymbol{m}}_t$$

The proposed default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Empirically the Adam works well in practice and compares favorably to other adaptive learning methods.

AdaMax borrows the idea from Adam that the update rule scales the gradient inversely proportional to the l_2 norm of the past gradients[KB14]. Now generalize this update to l_p norm, with the updating rate β's also parametrized.

$$\boldsymbol{v}_t = \beta_2^p\boldsymbol{v}_{t-1} + (1-\beta_2^p)|\boldsymbol{g}_t|^p$$

Norms for large p values generally become numerically unstable. However, l_∞ also generally exhibits stable behavior.

$$\boldsymbol{u}_t = \beta_2^\infty\boldsymbol{v}_{t-1} + (1-\beta_2^\infty)|\boldsymbol{g}_t|^\infty = \max(\beta_2\boldsymbol{v}_{t-1}, |\boldsymbol{g}_t|)$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\boldsymbol{u}_t}\widehat{\boldsymbol{m}}_t$$

Notice that $\boldsymbol{u}_t$ relies on the *max* operation and we do not need to compute a bias correction for it. Good default values are $\eta = 0.002, \beta_1 = 0.9$ and $\beta_2 = 0.999$.

Nesterov-accelerated Adaptive Moment Estimation (Nadam) combines Adam and NAG[Doz16]. The NAG is first modified as the following.

$$\boldsymbol{g}_t = \nabla_{\boldsymbol{w}_t} J(\boldsymbol{w}_t)$$

$$\boldsymbol{m}_t = \gamma \boldsymbol{m}_{t-1} + \eta \boldsymbol{g}_t$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - (\gamma \boldsymbol{m}_t + \eta \boldsymbol{g}_t)$$

Instead of the old NAG method shown before, we now apply the look-ahead momentum vector directly to update the current parameters. In order to add Nesterov momentum to Adam, we could similarly replace the previous momentum vector with the current momentum vector. Recall that the expanded form of Adam is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\widehat{v}_t} + \epsilon}\left(\frac{\beta_1 \boldsymbol{m}_{t-1}}{1 - (\beta_1)^t} + \frac{(1-\beta_1)\boldsymbol{g}_t}{1 - (\beta_1)^t}\right)$$

Notice that the term $\frac{\boldsymbol{m}_{t-1}}{1-(\beta_1)^t}$ is just the bias-corrected estimate of the momentum vector of the previous time step. We replace it with $\widehat{\boldsymbol{m}}_{t-1}$.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\widehat{v}_t} + \epsilon}\left(\beta_1 \widehat{\boldsymbol{m}}_{t-1} + \frac{(1-\beta_1)\boldsymbol{g}_t}{1 - (\beta_1)^t}\right)$$

Now we could add Nesterov momentum just as we did before by replacing the term $\widehat{\boldsymbol{m}}_{t-1}$ with the current version $\widehat{\boldsymbol{m}}_t$. The final update rule is as the following.

$$\boldsymbol{m_t} = \beta_1 \boldsymbol{m_{t-1}} + (1-\beta_1)\boldsymbol{g_t}$$

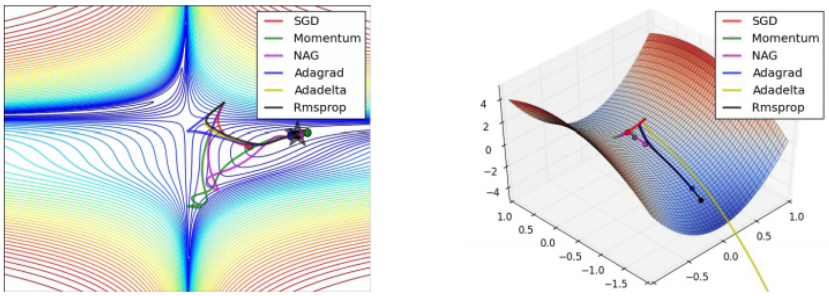
$$\boldsymbol{v_t} = \beta_2 \boldsymbol{v_{t-1}} + (1-\beta_2)\boldsymbol{g_t^2}$$

$$\widehat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - (\beta_1)^t}$$

$$\widehat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - (\beta_2)^t}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\widehat{v}_t} + \epsilon}\left(\beta_1 \widehat{\boldsymbol{m}}_t + \frac{(1-\beta_1)\boldsymbol{g}_t}{1 - (\beta_1)^t}\right)$$

Regarding the above listed algorithms, two tesing case could be found in [Rud16] and the figure is adaped here shown as the following. Figure 10.2(a) shows a complex loss contours and all the algorithms start at hte same point and take different path to reach the minimum. It shows that Adagrad, Adadelta and RMSprop go immediately in the right direction and converge quickly, while Momentum and NAG are going to the off-track directions. Finally NAG is able to correct its direction due to the increased responsiveness. Figure 10.2(b) shows the behaviors of different algorithms at a saddle points. SGD, Momentum and NAG find themselves difficult to break the



(a) SGD optimization on loss surface contours

(b) SGD optimization on saddle point

Figure 10.2: Test cases with different optimization algorithms. (a). Contours of a loss surface (the Beale function). (b). Saddle point. The figure is adapted from [Rud16], with original source from Alec Radford.

symmetry, although the latter to finally manage to escape the saddle point. However, Adagrad, RMSprop and Adadelta quickly head down the negative slope.

In general, when the input data is sparse, or the fast convergence is required for training a deep/complex neural network, we should choose one of the adaptive learning rate algorithm. Among all these kind of algorithms, Adagrad suffers from its radically diminishing learning rates. RMSprop, Adadelta and Adam are developed to overcome this shortage, and they are very similar algorithms doing well under similar conditions. Kingma et al.[KB14] show that the bias-correction helps Adam slightly outperform RMSprop toward the end of optimization as gradients become sparser. So normally Adam might be the best choice.

10.4.3 Additional Materials: A Brief Introduction to TensorFlow*

TensorFlow is Google's recent open-sourc framework for the implementation and training of large scale machine learning models, especially for deep neural networks[AAB+16]. The computational model for TensorFlow is a directed graph, where nodes are functions/computations and edges are numbers/matrices. Many common machine learning models especially neural networks are visualized as directed graphs already, using graph computation model makes it natural for machine learning practitioners. Also by splitting up computation into small, easily differentiable pieces, TensorFlow could automatically compute the derivative of any node with respect to any other node affecting the first node's output using backpropagations. Last it is easy to split the large graph into several smaller graphs and give each computing unit (CPU or GPU) a separate part of the graph to work on by having the computation separated, making it much easier to distribute work loads across multiple computational devices. The characteristic execution model for TensorFlow is briefly summarized as the following.

1. Dataflow graph elements: In TensorFlow graph, each vertex represents a unit of local computation, and each edge represents the output from or input to, a vertex. The values that flow along edges are referred to as **tensors**. The input/output data (usually n-dimensional arrays) is modeled as tensors, because tensors naturally represent the inputs to and results of the common mathematical operations in machine learning algorithms. The computation at vertices are referred to as **operations**. An operation takes $m \geq 0$ tensors as input and

produce $n \geq 0$ tensors as output. The graph also has stateful operations containing mutable states, that could be shared between different executions of the graph. Data flow with mutable state enables TensorFlow to mimic the functionality of a parameter server with additional flexibility.

2. Partial and concurrent execution: TensorFlow uses a dataflow graph to represent all possible computations in a particular application. It allows the user to specify declaratively the subgraph that should be executed. TensorFlow also supports multiple concurrent executionss on the same graph through shared variables and queues. The implementations of mutable state and coordination via queues make it possible to specify a wide variety of model architectures for users. This increases the TensorFlow's flexibility significantly.

3. Distributed execution: data flow simplifies distributed execution due to the explicit communications between subcomputations. TensorFlow places operations on devices (CPU's or GPU's) subject to implicit or explicit constraints in the graph. The placement algorithm computes a feasible set of devices for each operation, calculates the sets of operations that must be colocated and selects a satisfying device for each colocation group. In addition, users could also specify partial device preferences and the placement algorithm would take those into consideration. This gives TensorFlow great flexibility in how operations in the dataflow graph are mapped to devices. Once the operations in a graph have been placed, and the partial subgraph has been computed for a step, TensorFlow partitions the operations into per-device subgraphs. A per-device subgraph for device d contains all of the operations that were assigned to d, with additional `Send` and `Recv` operations that replace edges across device boundaries.

The above execution model characteristics make TensorFlow a flexible and powerful platform for building and training large scale neural networks.

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[AM16] Sanjeev Arora and Tengyu Ma. back-prop. http://www.offconvex.org/2016/12/20/backprop/, 2016.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[Doz16] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[HSS] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning-lecture 6a-overview of mini-batch gradient descent.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[Nes83] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.