Lecture 15
Weight Initialization, Momentum and Learning Rates

09 March 2016

Taylor B. Arnold
Yale Statistics
STAT 365/665

Notes:

- Problem set 5 is due on Monday, March 14th

- Today:
    - modular neural networks
    - a bit of review
    - initializing weights
    - quasi second order SGD
    - learning rate schedule
    - hyper-parameter selection heuristics

Modular view of neural networks

We have so far thought of neural networks as being primarily defined by a network of individual neurons. This is helpful both for its simplicity and the historical development of the estimators. However, there is another way of thinking about neural networks that more closely mimics popular implementations such as Caffe, Keras, and Torch.

Modular view of neural networks, cont.

For an input vector x and a response y, we can view the network as simply being something like:

$$z^1 = W^1 a^0 + b^1 \tag{1}$$

$$a^1 = \sigma(z^1) \tag{2}$$

$$z^2 = W^2 a^1 + b^2 \tag{3}$$

$$a^2 = \sigma(z^2) \tag{4}$$

$$z^3 = W^3 a^2 + b^3 \tag{5}$$

$$a^3 = \sigma(z^3) \tag{6}$$

$$\text{Cost} = (y - a^3)^2 \tag{7}$$

Modular view of neural networks, cont.

For an input vector x and a response y, we can view the network as simply being something like:

$$\begin{align} z^1 &= W^1 a^0 + b^1 \tag{1} \\ a^1 &= \sigma(z^1) \tag{2} \\ z^2 &= W^2 a^1 + b^2 \tag{3} \\ a^2 &= \sigma(z^2) \tag{4} \\ z^3 &= W^3 a^2 + b^3 \tag{5} \\ a^3 &= \sigma(z^3) \tag{6} \\ \text{Cost} &= (y - a^3)^2 \tag{7} \end{align}$$

Each level can be described by a *module*. The z layers are called linear layers, the a's as sigmoid layers, and the last line is simply the cost function. Notice that the activation functions are now their own layers, which actually simplifies things mathematically.

Modular view of neural networks, cont.

Each type of module needs to be able to:

- take an input and return an output for the current tuning parameters
- calculate the matrix $\frac{\partial \text{output}_i}{\partial \text{input}_j}$
- calculate the set $\frac{\partial \text{output}_i}{\partial \text{parameters}_j}$
- store the tuning parameters
- update parameters from a minibatch

If I have all of these implemented, I can simply chain together modules and have a built-in algorithm for learning from input data.

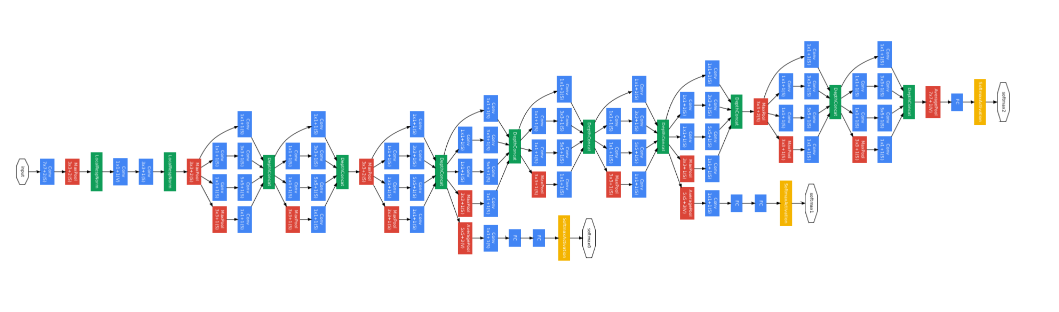
Modular view of neural networks, cont.

An example, using the **keras** library, describes this well:

```
model = Sequential()
model.add(Dense(64, input_dim=20, init='uniform'))
model.add(Activation('sigmoid'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(Activation('sigmoid'))
model.add(Dropout(0.5))
model.add(Dense(10, init='uniform'))
model.add(Activation('softmax'))
```

Where dense refers to a linear connected layer.

BVLC GoogLeNet: Szegedy et al. (2014)



Review: Cross-entropy and soft-max

Due to the shape of the sigmoid neuron, weights that are very far from their optimal values learn slowly in a plain, vanilla network. One way to fix this is to use the cross-entropy cost-function, defined as:

$$C = -\sum_j \left[y_j \log(a^L_j) + (1 - y_j) \log(1 - a^L_j) \right]$$

For a single sample, and similarly for an entire mini-batch.

Another common approach is to define what is termed a softmax layer. The redefines the activations of the output later, a^L, as follows:

$$a^L_j = \frac{e^{z^L_j}}{\sum_k e^{z^L_k}}$$

This has the additional benefit that the last layer is easily interpreted as a sequence of probabilities.

Review: Regularization in Neural Networks

As the size of neural networks grow, the number of weights and biases can quickly become quite large. State of the art neural networks today often have billions of weight values. In order to avoid over-fitting, one common approach is to add a penalty term to the cost function. Common choices are the ℓ_2-norm, given as:

$$C = C_0 + \lambda \sum_i w_i^2$$

Where C_0 is the unregularized cost, and theℓ_1-norm:

$$C = C_0 + \lambda \sum_i |w_i|.$$

The distinction between these is similar to the differences between lasso and ridge regression.

Review: Dropout

A very different approach to avoiding over-fitting is to use an approach called *dropout*. Here, the output of a randomly chosen subset of the neurons are temporarily set to zero during the training of a given mini-batch. This makes it so that the neurons cannot overly adapt to the output from prior layers as these are not always present. It has enjoyed wide-spread adoption and massive empirical evidence as to its usefulness.

Initial weights

Rather than initializing the weights in the neural network by a standard normal Gaussian, it is better to initialize with Gaussian noise with zero mean but a standard deviation of $1/\sqrt{n}$ where n is the number of nodes from the previous layer that are incoming to this given node. This perhaps makes sense when we consider two layers of a neural network:

$$z^1 = W^1 a^0 + b^1 \tag{8}$$

$$a^1 = \sigma(z^1) \tag{9}$$

This has the effect of making the distribution of all of the weighted inputs, z_j^l, equal to standard Gaussian noise.

Optimization tricks

For the rest of today, I am going to talk about approaches for fitting neural networks that can apply more generally to difficult optimization problems. I think it is easier to use a simple example to illustrate this, rather than directly applying to neural networks. So we'll return to minimizing the sum of squared residuals in a linear models:

$$\begin{aligned} f(\beta) &= \frac{1}{n} \sum_i (y_i - x_i^t \beta)^2 \\ &= \frac{1}{n} \left(y^t y + \beta^t X^t X \beta - 2 y^t X \beta \right) \end{aligned}$$

With the analytic gradient of:

$$\nabla f(\beta) = 2 X^t X \beta - 2 y^t X$$

Gradient Descent

Gradient Descent uses the following set of updates for some learning rate $\eta > 0$:

$$\beta_{t+1} = \beta_t - \eta \cdot \nabla f(\beta_t)$$

Which can be re-written into two parts:

$$\begin{aligned} v_{t+1} &= -\eta \cdot \nabla f(\beta) \\ \beta_{t+1} &= \beta_t + v_{t+1}. \end{aligned}$$

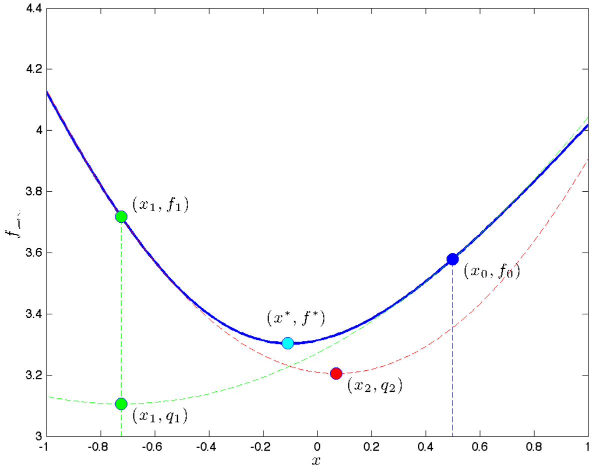
Newton's method

Newton's method uses the Hessian matrix, the matrix of all second derivatives, to generate a quadratic approximation of f near β_t. Generally, the updates are of the form:

$$\beta_{t+1} = \beta_t - \gamma \cdot [H(f(\beta_t))]^{-1} \cdot \nabla f(\beta)$$

Where the learning rate γ function quite a bit differently, primarily because it can reasonably be set to 1 and should never be greater than 1.

Example of Newton's method for optimization



Momentum

The problem with second order methods is that the require learning a p-by-p matrix of values. This can be huge for modern neural networks. An alternative approach is to use a concept similar to the physical idea of momentum. Specifically, we use updates of the form

$$\begin{aligned} v_{t+1} &= \mu \cdot v_t - \eta \cdot \nabla f(\beta_t) \\ \beta_{t+1} &= \beta_t + v_{t+1}. \end{aligned}$$

Where the value $\mu \in [0, 1]$ serves as proxy for friction.

This conveniently requires only a minimal increase in complexity but drastically increases the performance of fitting neural networks.

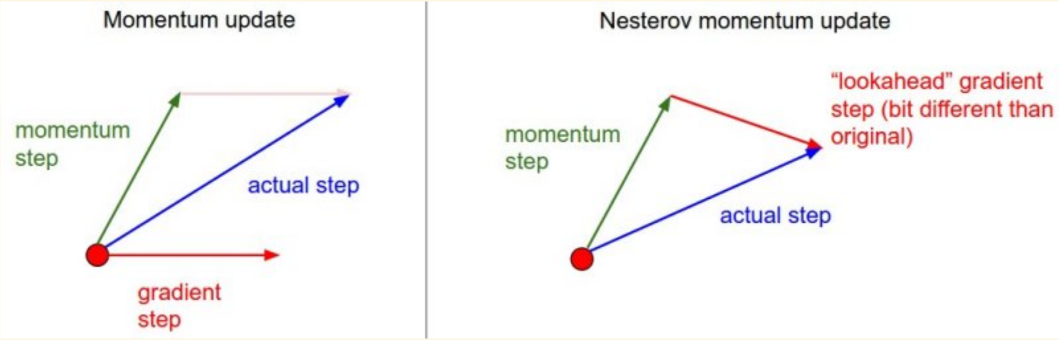
Nesterov's Method

A slight tweak of standard momentum, which is also important when optimizing non-convex surfaces, is to use Nesterov's Method. This calculates the gradient at the spot that would be arrived at with the current momentum. Specifically:

$$\begin{aligned} v_{t+1} &= \mu \cdot v_t - \eta \cdot \nabla f(\beta_t + \mu \cdot v_t) \\ \beta_{t+1} &= \beta_t + v_{t+1}. \end{aligned}$$

A picture helps considerably in understanding this.

Nesterov's Method



Learning Rate Annealing

As we have already seen, it is necessary to decrease the learning rate η over time in stochastic gradient descent. In the SVM case we have used a $1/t$ decay, where t is the epoch number. Other popular suggestions are to use e^{-kt} or to use $(1 - ts)$ for some step size s.

Karpathy suggests that the fixed step size is the most popular at the moment, when only using a single learning rate.

Adaptive learning rates

Another approach is to adaptively decay the learning rate for each parameter by tracking how fast the gradient is changing. A popular example is **Adagrad**, seen here:

> *Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. The Journal of Machine Learning Research, 12, 2121-2159. Chicago*

Or **Adam**, given in this paper:

> *Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.*

Both are quite readable and I suggest taking a look if you would like more details. Currently, many other heuristics are also used and this is an active area of interest (with a lack of theoretical understanding).

A look ahead

We now actually have a clean picture of the basic structure of a neural network. After the past two classes, we have also seen many of the standard tricks for learning the high-dimensional, non-convex optimization function required for fitting the weights and biases in a neural network.

After the break, we will cover:

- the keras library, a python module for building and fitting neural networks much more efficiently
- convolutional modules (or CNNs)
- recurent modules (or RNNs)

Note that the last two things are simply different module types, other than the linear, activation, or cost layers. Historically they are described as different *types* of neural networks, but I do not feel that is a fruitful way of thinking about them.