CS 182/282A: Designing, Visualizing and Understanding Deep Neural Networks **Spring 2019**

Section 3: Backpropagation and Optimization

Notes by: Philippe Laban

3.1 Backpropagation

We will walk through an example for forward and backward propagation on a simple example.

3.1.1 Setup:

You are given given an input $x \in R^2$ and are trying to predict a $y \in [0, 1]$.
You setup a simple "neural network":

$$y(x) = \sigma(w_0 x_0 + w_1 x_1 + w_2)$$

With: $\sigma(x) = \frac{1}{1+e^{-x}}$ You setup a loss, which gives you access to: $dy = \frac{dL}{dy}$. You are interested in getting: $dw_i = \frac{dL}{dw_i}$, so you can use these gradients to update your weights, and train your neural network.

3.1.2 Worked example

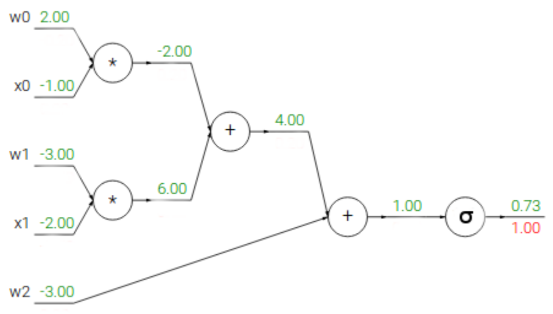

Figure 3.1: Work through getting the gradients for a specific sample: $x = (-1, -2)$ and weights have been initialized to: $w = (2, -3, -3)$.

Hints/Steps:

1. Verify that you understand the forward pass.

2. Compute what the derivative of the sigmoid function with respect to its input ($\frac{d\sigma(x)}{dx}$).
3. Use the chain rule to backpropagate step by step. Write the chain rule explicitly.
4. Verify you get $dw = (-0.20, -0.4, 0.2)$

3.2 Optimization methods

Now that we have obtained gradients for the loss with respect to our weights, we have to use them to update the weights.
There are several optimization methods available.

3.2.1 Gradient Descent

For each weight, we have a $dw = \frac{dL}{d_w}$, we update with the goal to minimize loss, so we go against the gradient.

$$w_{new} = w_{old} - \alpha dw$$

How do you choose the value of α, also called the learning rate? Another problem:

$$L = \sum_{(x_i,y_i)\in D} L_i = L(x_i, y_i, W)$$

Problem, the loss is a sum overall all samples (x_i, y_i) in our dataset (D).

$$dw = \sum_{(x_i,y_i)\in D} \frac{dL(x_i, y_i, W)}{dw}$$

To obtain one full gradient, we need to process the full dataset, this is often too expensive.

3.2.2 Stochastic Gradient Descent (SGD)

At each step, we select a random (stochastic) subsample of the data, called a **minibatch** MB.

$$MB \subset D, \|MB\| = \text{S}$$

For example, the batch size (S) could be 10, 32, 76, etc. Then we only compute the gradients with respect to this minibatch.

$$d\hat{w} = \frac{1}{S} \sum_{(x_i,y_i)\in MB} \frac{dL(x_i, y_i, W)}{dw}$$

$d\hat{w}$ is noise, and depends on the minibatch selected, as well as the minibatch. However we use it as a substitute for the true gradient.

$$w_{new} = w_{old} - \alpha d\hat{w}$$

3.2.3 SGD with momentum

What if we kept the noisy gradients and summed them into a "momentum vector" to reduce noise? At each step, update the momentum with the new gradient:

$$M_{w,t+1} = \mu M_{w,t} - \alpha dw$$

Where $\mu \in [0, 1]$, is the momentum constant, which determines the forgetting rate of old gradients. Then use the momentum to update the weigths:

$$w_{t+1} = w_t + M_{w,t+1}$$

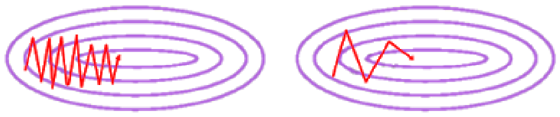

Figure 3.2: Effect of using momentum on a 2-d problem. The lines are equipotential lines of the loss, and the momentum method converges faster, with the intuition that it remembers the direction of the older gradients, which were "pushing" it towards the center, representing the local minimum.

3.2.4 RMSProp

Similar to SGD, this method runs with randomly selected mini-batches. We expect the parameters to be in the same range of scale, however, gradients of some weights might have much larger magnitude than others, which might make them overfluctuate.
In RMSProp, you keep a moving average of of the "norm" of the gradients for each weight, and normalize the gradient by this norm. It is equivalent to having a different learning rate for each parameter, based on its gradient.
At each step we update the norm estimate of the weight:

$$s_{w,t} = \beta s_{w,t-1} + (1 - \beta)(dw)^2$$

Where $\beta \in [0, 1]$ is the decay factor of the norm term. Then we apply the normalized gradient update:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{s_{w,t} + \epsilon}} dw$$

These operations above are applied element-wise to each element of the vectors involved. (If we're dealing with a multi-layer neural network, think of w as representing the concatenation of all parameters together in one giant vector.) For example, $\sqrt{s_{w,t}}$ means taking the square root of each element in the vector $s_{w,t}$.

Note: ADAM is roughly RMSProp with a momemtum update. The two parameters (μ and β) are called β_1 and β_2.

Here's a blog post with an up-to-date list of popular optimization methods for deep-learning: `http://ruder.io/optimizing-gradient-descent/`

3.3 Training advice and strategies

Now we have a way to decide on a neural network architecture, obtain gradients for all the weights (backpropagation), and update the weights (optimization method).

How do we know whether the neural network is training? How do we compare neural networks?
It is usually based on a metric evaluated on the validation dataset, and the validation loss is often used.
Here are things to keep in mind.

3.3.1 Tuning the learning rate

All the methods described above involve a learning rate. How is this rate chosen?
Answer: start with a random learning rate and then increase it or decrease it until finding a good rate.
Can you tell in each situation, whether the rate is too high or too low.

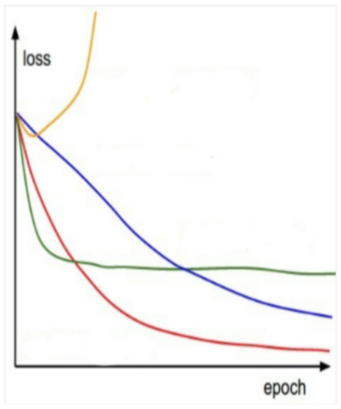

Figure 3.3: Plotting the validation loss for the same architecture and optimization method, with varying learning rates. When is the learning rate too high, too low, and at the right level.

A straightforward way to find a reasonable learning rate is to test every order of magnitude of a parameter, such as $\alpha \in \{1e^{-5}, 1e^{-4}, 1e^{-3}, 1e^{-2}\}$.

3.3.2 Choosing an optimization method

You can start simple with SGD, and try more complicated optimization methods such as ADAM. Note that each method needs the learning rate to be re-tuned, as the best learning rate for SGD might not be the best one for RMSProp.

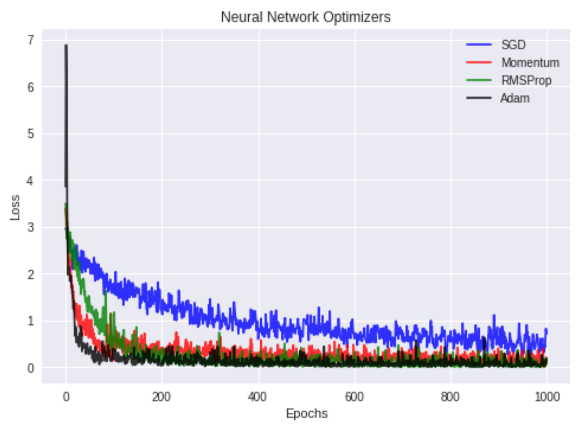

Figure 3.4: The optimization method can both change what value the loss converges to, as well as how fast it converges to it.

3.3.3 Learning rate schedules

There is no reason the learning rate has to be fixed throughout training. Commonly, the learning rate is reduced over time, allowing the network to be fine-tuned with smaller gradients. An exponential decay can be used for the learning rate:

$$\alpha_t = \alpha_0 e^{-\lambda t}$$

Another strategy is to keep the learning rate constant until the validation loss stabilizes, then reduce the learning rate, and keep training.

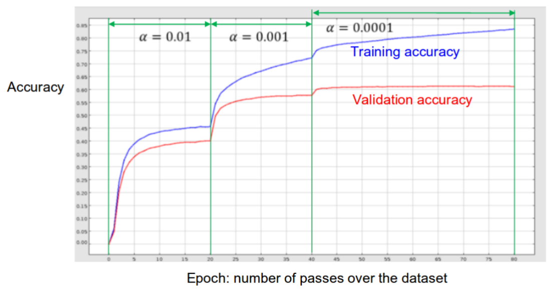

Figure 3.5: The validation accuracy of the training of an AlexNet. Every time the accuracy plateaus, α is divided by 10, and training continues.