**Slide 6-3: Where We Are**

* We are training Neural Networks which is really a 4 step process
  1. Sample data set of data and labels. We sample that data set
  2. We Forward propagate through the graph to get the loss which is telling us how well we are classifying this data
  3. We Back propagate to compute the gradient and all the weights. This gradient is telling us how we should nudge every single weight in the network so we are better classifying these images
  4. We update all the parameters using the gradient. This is the nudge

**Slide 6-4**

* We looked into activation functions and their pros and cons in using them
* If we don’t use an activation function then the capacity of the network is equivalent to a linear classifier

**Slide 6-5**

* We talked briefly about Data preprocessing

**Slide 6-6**

* We talked about Weight initialization
* Problem is we have to choose initialization for the weights in particular the scale of how large we want them to be in the beginning
  1. We saw that if the weights are too small then the activation in a neural network goes all toward zero
  2. If we set the weights too high then the activation will explode instead
* We end up with super saturated networks or networks that output all zeros
* Xavier initialization gives a reasonable activation

**Slide 6-7**

* Batch Normalization makes the initialization process more robust so that we don’t have to get it right

**Slide 6-8**

* We looked in to how to baby sit the learning process and what to watch out for

**Slide 6-12: Parameter Updates**

* We specifically look at the last line
* We want to make this more complex

**Slide 6-13**

* SGD is the slowest and is hardly used in practice

**Slide 6-14**

* Why is SGD so slow
* Lets say we have this surface where the loss function is very shallow horizontally but very steep vertically
* Q: What is the trajectory along which we converge toward the minimum with SGD
* A: trajectory bounces up and down a lot but does not progress horizontally quickly

**Slide 6-15**

* Local gradients have this form
* Vertically we have a large gradient because it’s a very steep function
* Hortizontally we have a small gradient because its shallow horizontally

**Slide 6-16**

* Way too slow the horizontal direction
* Way too fast in the vertical direction

**Slide 6-17: Momentum Update**

* One way to remedy is momentum update
* In Gradient Decent we take the gradient and we are integrating our current position by that gradient
* In Momentum update we take the gradient and instead of integrating position directly we increment the variable “v” (v for velocity)
* We build up an exponential sum of gradients in the past
* “mu” is a hyperparameter between 0 and 1
* Using Momentum Update interprets the trajectory behavior as a ball rolling across a landscape
  1. The gradient in this case is a force that the particle is feeling
  2. Learning rate \* dx is the acceleraton we integrate the velocity
  3. Mu \* v is interpreted as friction because after every iteration we are slightly slowing down
     + If this was not there then the ball would never come to a rest

**Slide 6-18**

* This ends up improving the convergence in many cases since
  1. We end up damping the oscillations in steep directions
  2. We end up encouraging progress in shallow directions
* **Velocity usually initialized to zero**

**Slide 6-19**

* In comparison Momentum Update performs better than SGD

**Slide 6-20: Nesterov Momentum Update**

* Nesterov Momentum or Nesterov Accelerated Gradient Descent
* In the ordinary Momentup Update X is incremented by two parts
  1. Direction the momentum is trying to carry you
  2. Direction of the gradient
  3. The actual step taken is the vector sum of the two

**Slide 6-21**

* Nesterov Momentum says why don’t we look ahead since we know we are already going to be there
  1. One step lookahead
  2. This enjoys better guarantees on convergence rates

**Slide 6-23**

* Normally we only think of doing a backward pass and forward pass
* We usually have a parameter vector theta and the gradient at that point
* But Nesterov asks us to have the gradient vector theta and another gradient at another point
* Slightly inconvenient

**Slide 6-25**

* Variable transform helps where we donote the new look ahead as a new variable phi which puts the equations in the form of a regular update

**Slide 6-26**

* Nesterov Accelerated Gradient doesn’t over shoot as much as momentum update and will therefore converge faster

**Slide 6-27: AdaGrad Update**

* This is very common to see in practice originally developed in Convex Optimization Literature and then ported over to Neural Networks
* We have the update as we normally see it. Basic stochastic gradient descent learning rate \* gradient
* but then we are scaling it by this variable that we keep accumulating
* note that the cache which we are building up and is the sum of gradients squared contains positive numbers only
* cache is a giant vector the same size as the parameter vector
* As we keep building up the cache variable we divide the step (learning rate\* gradient) by the square root of the cache
* This is called the Per Parameter Adaptive learning rate method since every dimension in the parameter space now has its own learning rate which is scaled dynamically based off of the gradients seen

**Slide 6-28**

* Q: What happens with AdaGrad with the earlier example?
* A:In the steep directons we have a very small learning rate and in the shallow directions we have a very fast learning rate.

Slide 6-29

* Q2: What happens to the step size over long time?
* A2: We end up building the cache more and more until the learning rates become very small numbers and the learning comes to a halt

**Slide 6-30: RMS Prop**

* Change to AdaGrad proposed by Jeff Hinton recently
* We add a leaky decay rate to the cache variable
* Instead of just having the sum of squares, we have the sum of squares multiplied by a leaky decay rate

**Slide 6-33**

* AdaGrad and RMSProp converge the fastest

**Slide 6-34: Adam Update**

* We can combine both AdaGrad and Momentum
* Adam

**Slide 6-38**

* We talked about several learning rate updates
* Q: Which one of these learning rates is best to use?
* A: Trick Question – None of these are

**Slide 6-39**

* These are only first order methods since they only use the gradient info at the loss function

**Slide 6-40**

* There are an entire set of second order methods for optimization
* They end up forming a larger approximation to the loss function
* You don’t only need the gradient but also the hessian
* Q: Why do ppl like these methods?
* A: Less hyper parameters – no learning rate
* A: faster convergence
* Q2: Why is this impractical for training Deep Neural Nets?
* A2: Issue with using these practically is the size of the Hessian matrix which we then want to invert
  1. i.e. inverting a 1000000 x 1000000 dimension matrix
* WE WILL NOT BE USING THIS

**Slide 6-41**

* Other methods not required
  1. BFGS
  2. L-BFGS

**Slide 6-44 : In practice**

* Adam is a good default choice in most cases
* Use L-BFGS if you can afford to do full batches

Slide 6-45: Model Ensembles

Slide 6-46

* It turns out that if you train multiple independent models on your training data instead of just one and you average their results at test time, you always get 2% extra performance
* Downside to this is you need to perform forward prop on all of these models as well as train all of these models

**Slide 6-47 Tips and Tricks**

* As you are training your neural network you have check point to figure out the validation performance per epoch per model.
* One method is you can take the check points of the models and do an ensemble over those

**Slide 6-50: Regularization Dropout**

* As you are doing a forward pass of your network you will randomly set some neurons to zero

**Slide 6-51**

* Randomly computer a binary mask to dropout some neurons
* These drops have to be accounted for during back prop (kill the gradient)

**Slide 6-52**

* How does this make sense?

**Slide 6-53**

* Dropout is forcing our code, our representation of what the image was about to be redundant
* We need the redundancy in case half of the network drops off
* We need to be able to base the score off of more features since we may lose one at any time

**Slide 6-54**

* Video has a more detailed explanation
* Each neuron that is dropped off will basically have no effect on the network
* Therefore after each iteration we drop out different neurons and every time we train the network with only half the neurons
* Approx. 50% of the neurons are typically dropped out

Slide 6-55: At test time

* We have integrated all this noise in the forward pass
* We would like to integrate out all the noise
* Monte Carlo approximation would be many forward passes with different masks and then average all the preditions
  1. This is not effiecient

Slide 6-57

* We could do this another way
  1. Do no dropout in the forward pass for test images
  2. Q: Suppose that with all inputs present at test time the ouput of the neuron is x. What would its output be during training time, in expectation (e.g. if p = 0.5)

Slide 6-58

* 1. Assuming we have a linear neuron no nonlinearities
     + During test: a = w\*x + w1\*y
     + During training: (1/2) (w0\*x+w1\*y)
  2. By leaving all the neurons on our network is not used to seeing all the outputs of the neurons so large
  3. We therefore have to chop it in half

Slide 6-59

* We have to scale our activations at test time down so that we get what we expect

Slide 6-60

* At test time we compute the neural network but now at test time we have to multiply by p

Slide 6-62

* Inverted dropout is used more in practice