**Slide 4-3**

* Where we are right now:
  + We have the score function
  + We looked at Loss Functions such as SVM Loss
  + We looked at the Full Loss achieve with any particular set of weights over the training data.
    - There are two components to this:

1. Data Loss
2. Regularization Loss
   * What we want to do now is derive the gradient expression of the loss function wrt the weights and we want to do this so we can perform the Optimization process.

**Slide 4-4**

* Gradient descent where we iterate evaluating the gradient on your weights, doing a parameter update and repeating over and over again so that we are converging to the low points of the loss function
* When we arrive at a low loss that is equivalent to making good predictions over our training data in terms of the scores that come out.

**Slide 4-5**

* We saw that there are two ways of evaluating the gradient

1. Numerical gradient
2. Analytic gradient (obtained with calculus)

**Slide 4-6**

* When looking at the equations for the loss you might be tempted to just write out the full expression of the loss and begin calculating gradients.
* It would be wise to look at this as a Computational Graph instead of thinking of this as one giant expression to calculate the gradient as you would do with pen and paper.
* We look at this as inputs flowing through functional pieces which perform transformations on the data and we end up with a value at the end.
* This helps when the expressions become very large

**Slide 4-7**

* A Convolutional Neural Network will have hundreds of operations
* We have our image flowing through a very large computational graph.
* Therefore it becomes impractical to write out these expressions.

**Slide 4-8**

* Worse in complexity than CNNs is a Neural Turing Machine
* The computational graph for this is huge.

**Slide 4-9**

* We end up unrolling the Neural Turing Machine Computational Graph.
* We end up with a giant monster of hundreds of thousands of nodes and little computational units
* Therefore as you see it is impossible to write out the function of the loss of the Neural Turing Machine

**Slide 4-10 – BACK PROPAGATION**

* Lets start out simple and concrete
* We start out with a very small computational graph
* We have 3 scalar inputs and very small graph or circuit that gives us an output
* Here the forward pass of the graph is already prefilled where we set the inputs and compute the output
* We would like to now derive the gradients of the expression on the inputs

**Slide 4-11**

* Intermediate variable “q” introduced after plus gate
* “f” is the multiplication of q\*z
* We want the gradients, the derivatives
* We now perform a backward pass computing the gradients for all the intermediate values until we arrive at our inputs

**Slide 4-13**

* We start off at right
* Df/df is the identity function gradient = 1

**Slide 4-15**

* Df/dz = q = 3
* What 3 is intuitively telling us is that the influence of z on the final value is positive with a force of 3
* If we increment z by h the output of the circuit will react by increasing by 3h

**Slide 4-17**

* Df/dq = z = -4
* This tells us that if q were to increase by h, the value of the circuit would decrease by 4h

**Slide 4-19**

* Df/dy = (df/dq)\*(dq/dy) through the chain rule
* Dq/dy is the local influence of y on q
* Dq/dy = 1 for the gate
* This is the crux of backpropagation
* Dq/dx and dq/dy are saying that both x and y have a positive influence on q with a slope of 1
* We would like to know the influence of y on the final output of the circuit
* We see that increasing y increases q which in turn decreases f

**Slide 4-21**

* The influence of x on the final output of the circuit is calculated the same way
* Df/dx = (df/dq)\*(dq/dx)

**Slide 4-22**

* To generalize a bit and think about it is as follows:
  + Forward Pass
    - You are a gate embedded in a circuit
    - You receive some inputs x and y
    - You perform some function f on them and output a value z
    - Z goes into computational graph
  + Backward Pass
    - We proceed recursively backwards

**Slide 4-23**

* Once we have x and y we can compute the local gradients

**Slide 4-24**

* After the loss is computed and we are going backwards we will eventually learn what is my influence on the final output of the circuit
* We learn what is dL/dz
* We have to chain the gradient through our operation

**Slide 4-25 and 26**

* We must multiply this gradient by the local gradients
* This gives us the influence of x and y on the final output of the circuit

**Slide 4-27**

* We end up recurrsing this process throughout the entire computational graph or circuit
* This process is called “Backpropagation”
  + It’s a way of computing through a recursive application of chain rule through a computational graph
  + The influence of every intermediate value in that graph on the loss function

**Slide 4-28**

* Here we have another example

Slide 4-29

* We write out the derivatives for all the local gates.
* So we know what the gradients will look like
* We start out at the end of the circuit
* 1.00 already filled in. This is the gradient of the identity function
* We now back propagate and apply chain rule for each operation

**Slide 4-31**

* We multiply the gradient result of the gate by the gradient result of the previous gate

**Slide 4-38**

* The local gradient for the plus gate is 1 for any of its inputs

**Slide 4-39**

* Therefore we end up getting 1 X 0.2
* So a plus gate is like a gradient distributor

**Slide 4-42**

* We could have a sigmoid gate as one gate that does the entire sigmoid function
* Or we could break it up into individual gates has is shown in the example
* If we decide to create one sigmoid gate then we have to find the local gradient of that gate and apply that
* The decision on which to use would depend on which would be more efficient to compute

**Slide 4-44**

* One should try and gain more of an understanding of these networks than just a black box level understanding
* Having an intuitive understanding of these networks aids in the debugging of these networks

**Slide 4-47**

* Implementation of a binary multiplier (Object Oriented Class definition)
* Both a forward and backward pass must be calculated
* Backward pass is computing the chain rule in the backward pass

**Slide 4-48**

* In general when computing the forward pass every intermediate value for each gate must be remembered so that we have access to them during the backward pass.
* This must be kept in mind for memory considerations

**Slide 4-49**

* Torch is a Deep Learning Framework
* Github for Torch shows basically a lot of layers we can use to construct our network
* A library will have a bunch of layers each with a forward and backward implementation

**Slide 4-53**

* Caffe is a Deep Learning Framework for images
* This is CPU code only. There is another file which is for GPUs using CUDA code

**Slide 4-54**

* We are going to work not with scalars but with scalars
* Therefore our inputs and outputs will be full Jacobian Matrices
* dL/dz is a vector
* dz/dx is a Jacobian Matrix
* Should be (dz/dx)\*(dL/dz)
* However, we will not actually ever fully form the Jacobian

**Slide 4-56**

* Q: What is the size of the Jacobian Matrix?
* A: 4096 X 4096
* Q: What does the Jacobian Matrix look like:
* A: The Jacobian Matrix is almost an Identity Matrix however with some values equal to zero
  + Whichever elements had a value less than zero during the forward pass would be made zero
* Therefore we never fully form the Jacobian
* We look at where the inputs are equal to zero and we kill the gradients at those places

**Slide 4-58**

* We usually process the inputs in Mini-batches

**Slide 4-59**

* Hint on design on how to approach problem
  + Think of this as back propagation
  + Stage computation in units that you know the local gradient of and then do the backprop.
  + Compute the scores from W and x
  + Compute the margins, loss
  + Then do the backprop
  + Compute the gradient on scores before gradient on weights

**Slide 4-62 – NEURAL NETWORKS**

* Lets look at score functions
* Lets start to make f more complex

**Slide 4-63**

* 2 layer ANN
* More complex mathematical expression of f
* Activation function now included
  + Here we are using a threshold of zero as the activation function
* Matix Multiply, threshold everything to zero, then we do one more matrix multiply to get scores

**Slide 4-64**

* Using CFAR10 as an example
* Now we have an intermediate representation of a hidden state (hidden layers)
* Before we were matrix multiplying from 3072, straight to 10
* However the hiddlen layer allows us to do more interesting things

**Slide 4-65**

* One example is using the example of a Linear Classifer on CFAR10 to classify images of a car.
  + The linear classifier would try and go through the different modes of cars in different orientations
  + However the classifier would not do well with cars of slightly different color
  + Now with intermediate hidden layer could have a number dealing specifically with i.e. red cars facing forward, red cars facing left, red cars facing right
* The value of h only becomes positive if it finds what its looking for in the image
* We can now have template for different modes
* W2 matrix can sum across all the templates

**Slide 4-66**

* We can extend this to a three layer neural network

**Slide 4-67**

* 2 layer ANN is actually very simple to implement and train
* Random function is creating the first initial random weights to start somewhere

**Slide 4-68**

* We will also be training a 2 layer ANN
* We will not, however, be using logistic regression
* Advice is to stage computation into intermediate results and do proper back prop into every intermediate result

**Slide 4-72**

* Crude model of a neuron
* b is a bias
* activation function computes f

**Slide 4-73**

* Biological models historically ppl like to use the sigmoid function
  + We get a value between zero and one and we can interpret this as the rate at which this neuron is firing
* If the neuron sees something it “like” the rate will begin to spike a lot

**Slide 4-74**

* Dendrite Computation review article

**Slide 4-76**

* There exist an entire set of nonlinearities we can choose from
* Historically sigmoid has been used a lot
* As of 2012, ReLU is now the popular one

**Slide 4-81**

* Usually the more neurons in the hidden layers the better however we must take care to properly regularize.
* Usually regularization is constant throughout