Lecture 3. Neural Networks and Backpropagation

Disclaimer: This note was modified from cs231n lecture notes by Prof. Li Fei-Fei at Stanford University.

Table of Contents:

Quick intro

It is possible to introduce neural networks without appealing to brain analogies. In the section on linear classification we computed scores for different visual categories given the image using the formula $s = Wx$, where W was a matrix and x was an input column vector containing all pixel data of the image. In the case of CIFAR-10, x is a [3072x1] column vector, and W is a [10x3072] matrix, so that the output scores is a vector of 10 class scores.

An example neural network would instead compute $s = W_2 \max(0, W_1 x)$. Here, W_1 could be, for example, a [100x3072] matrix transforming the image into a 100-dimensional intermediate vector. The function $max(0, -)$ is a non-linearity that is applied elementwise. There are several choices we could make for the non-linearity (which we'll study below), but this one is a common choice and simply thresholds all activations that are below zero to zero. Finally, the matrix W_2 would then be of size [10x100], so that we again get 10 numbers out that we interpret as the class scores. Notice that the non-linearity is critical computationally - if we left it out, the two matrices could be collapsed to a single matrix, and therefore the predicted class scores would again be a linear function of the input. The non-linearity is where we get the *wiggle*. The parameters W_2, W_1 are learned with stochastic gradient descent, and their gradients are derived with chain rule (and computed with backpropagation).

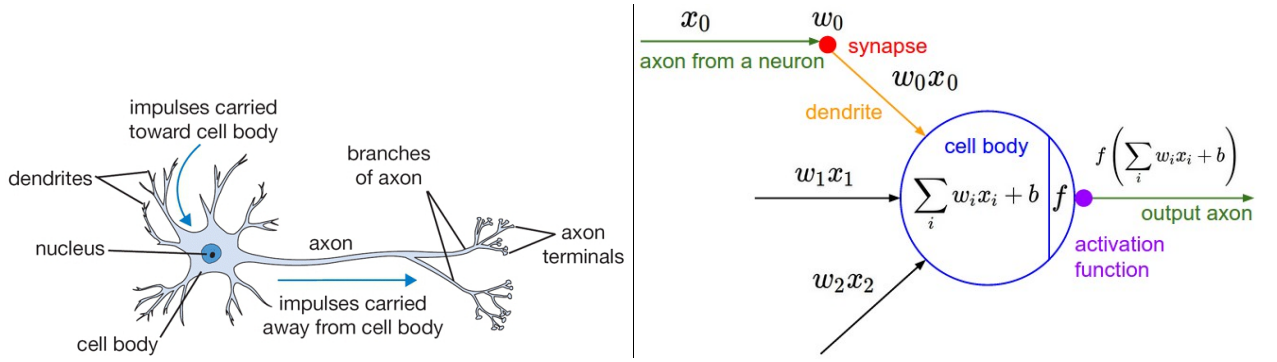
A three-layer neural network could analogously look like $s = W_3 \max(0, W_2 \max(0, W_1 x))$, where all of W_3, W_2, W_1 are parameters to be learned. The sizes of the intermediate hidden vectors are hyperparameters of the network and we'll see how we can set them later. Lets now look into how we can interpret these computations from the neuron/network perspective.

Modeling one neuron

The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems, but has since diverged and become a matter of engineering and achieving good results in Machine Learning tasks. Nonetheless, we begin our discussion with a very brief and high-level description of the biological system that a large portion of this area has been inspired by.

Biological motivation and connections

The basic computational unit of the brain is a **neuron**. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^14 - 10^15 **synapses**. The diagram below shows a cartoon drawing of a biological neuron (left) and a common mathematical model (right). Each neuron receives input signals from its **dendrites** and produces output signals along its (single) **axon**. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model of a neuron, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. $w_0 x_0$) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory (positive weight) or inhibitory (negative weight)) of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can *fire*, sending a spike along its axon. In the computational model, we assume that the precise timings of the spikes do not matter, and that only the frequency of the firing communicates information. Based on this *rate code* interpretation, we model the *firing rate* of the neuron with an **activation function** f, which represents the frequency of the spikes along the axon. Historically, a common choice of activation function is the **sigmoid function** σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0 and 1. We will see details of these activation functions later in this section.



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

An example code for forward-propagating a single neuron might look as follows:

```
class Neuron(object):
  # ...
  def forward(self, inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

In other words, each neuron performs a dot product with the input and its weights, adds the bias and applies the non-linearity (or activation function), in this case the sigmoid $\sigma(x) = 1/(1+e^{-x})$. We will go into more details about different activation functions at the end of this section.

Coarse model. It's important to stress that this model of a biological neuron is very coarse: For example, there are many different types of neurons, each with different properties. The dendrites in biological neurons perform complex nonlinear computations. The synapses are not just a single weight, they're a complex non-linear dynamical system. The exact timing of the output spikes in many systems is known to be important, suggesting that the rate code approximation may not hold. Due to all these and many other simplifications, be prepared to hear groaning sounds from anyone with some neuroscience background if you draw analogies between Neural Networks and real brains. See this review (pdf), or more recently this review if you are interested.

Single neuron as a linear classifier

The mathematical form of the model Neuron's forward computation might look familiar to you. As we saw with linear classifiers, a neuron has the capacity to "like" (activation near one) or "dislike" (activation near zero) certain linear regions of its input space. Hence, with an appropriate loss function on the neuron's output, we can turn a single neuron into a linear classifier:

Binary Softmax classifier. For example, we can interpret $\sigma(\sum_i w_i x_i + b)$ to be the probability of one of the classes $P(y_i = 1 \mid x_i; w)$. The probability of the other class would be $P(y_i = 0 \mid x_i; w) = 1 - P(y_i = 1 \mid x_i; w)$, since they must sum to one. With this interpretation, we can formulate the cross-entropy loss as we have seen in the Linear Classification section, and optimizing it would lead to a binary Softmax classifier (also known as *logistic regression*). Since the sigmoid function is restricted to be between 0-1, the predictions of this classifier are based on whether the output of the neuron is greater than 0.5.

Binary SVM classifier. Alternatively, we could attach a max-margin hinge loss to the output of the neuron and train it to become a binary Support Vector Machine.

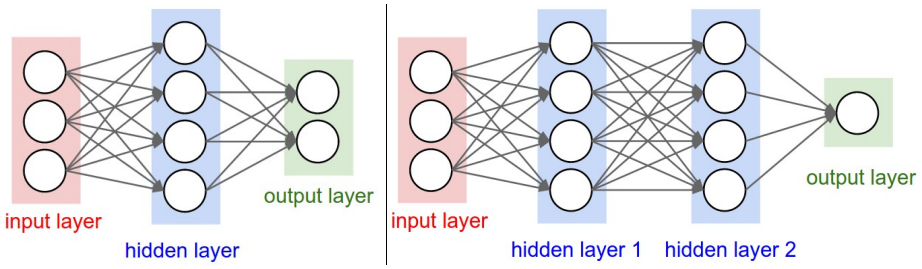
Regularization interpretation. The regularization loss in both SVM/Softmax cases could in this biological view be interpreted as *gradual forgetting*, since it would have the effect of driving all synaptic weights w towards zero after every parameter update.

> *A single neuron can be used to implement a binary classifier (e.g. binary Softmax or binary SVM classifiers)*

Neural Network architectures

Layer-wise organization

Neural Networks as neurons in graphs. Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. Below are two example Neural Network topologies that use a stack of fully-connected layers:



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. **Right:** A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

Naming conventions. Notice that when we say N-layer neural network, we do not count the input layer. Therefore, a single-layer neural network describes a network with no hidden layers (input directly mapped to output). In that sense, you can sometimes hear people say that logistic regression or SVMs are simply a special case of single-layer Neural Networks. You may also hear these networks interchangeably referred to as *"Artificial Neural Networks"* (ANN) or *"Multi-Layer Perceptrons"* (MLP). Many people do not like the analogies between Neural Networks and real brains and prefer to refer to neurons as *units*.

Output layer. Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).

Sizing neural networks. The two metrics that people commonly use to measure the size of neural networks are the number of neurons, or more commonly the number of parameters. Working with the two example networks in the above picture:

- The first network (left) has 4 + 2 = 6 neurons (not counting the inputs), [3 x 4] + [4 x 2] = 20 weights and 4 + 2 = 6 biases, for a total of 26 learnable parameters.
- The second network (right) has 4 + 4 + 1 = 9 neurons, [3 x 4] + [4 x 4] + [4 x 1] = 12 + 16 + 4 = 32 weights and 4 + 4 + 1 = 9 biases, for a total of 41 learnable parameters.

To give you some context, modern Convolutional Networks contain on orders of 100 million parameters and are usually made up of approximately 10-20 layers (hence *deep learning*). However, as we will see the number of *effective* connections is significantly greater due to parameter sharing. More on this in the Convolutional Neural Networks module.

Example feed-forward computation

Repeated matrix multiplications interwoven with activation function. One of the primary reasons that Neural Networks are organized into layers is that this structure makes it very simple and efficient to evaluate Neural Networks using matrix vector operations. Working with the example three-layer neural network in the diagram above, the input would be a [3x1] vector. All connection strengths for a layer can be stored in a single matrix. For example, the first hidden layer's weights `w1` would be of size [4x3], and the biases for all units would be in the vector `b1`, of size [4x1]. Here, every single neuron has its weights in a row of `w1`, so the matrix vector multiplication `np.dot(w1,x)` evaluates the activations of all neurons in that layer. Similarly, `w2` would be a [4x4] matrix that stores the connections of the second hidden layer, and `w3` a [1x4] matrix for the last (output) layer. The full forward pass of this 3-layer neural network is then simply three matrix multiplications, interwoven with the application of the activation function:

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(w1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(w2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(w3, h2) + b3 # output neuron (1x1)
```

In the above code, `w1,w2,w3,b1,b2,b3` are the learnable parameters of the network. Notice also that instead of having a single input column vector, the variable `x` could hold an entire batch of training data (where each input example would be a column of `x`) and then all examples would be efficiently evaluated in parallel. Notice

that the final Neural Network layer usually doesn't have an activation function (e.g. it represents a (real-valued) class score in a classification setting).

> *The forward pass of a fully-connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function.*

Representational power

One way to look at Neural Networks with fully-connected layers is that they define a family of functions that are parameterized by the weights of the network. A natural question that arises is: What is the representational power of this family of functions? In particular, are there functions that cannot be modeled with a Neural Network?

It turns out that Neural Networks with at least one hidden layer are *universal approximators*. That is, it can be shown (e.g. see *Approximation by Superpositions of Sigmoidal Function* from 1989 (pdf), or this intuitive explanation from Michael Nielsen) that given any continuous function $f(x)$ and some $\epsilon > 0$, there exists a Neural Network $g(x)$ with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, \mid f(x) - g(x) \mid < \epsilon$. In other words, the neural network can approximate any continuous function.

If one hidden layer suffices to approximate any function, why use more layers and go deeper? The answer is that the fact that a two-layer Neural Network is a universal approximator is, while mathematically cute, a relatively weak and useless statement in practice. In one dimension, the "sum of indicator bumps" function $g(x) = \sum_i c_i 1(a_i < x < b_i)$ where a, b, c are parameter vectors is also a universal approximator, but no one would suggest that we use this functional form in Machine Learning. Neural Networks work well in practice because they compactly express nice, smooth functions that fit well with the statistical properties of data we encounter in practice, and are also easy to learn using our optimization algorithms (e.g. gradient descent). Similarly, the fact that deeper networks (with multiple hidden layers) can work better than a single-hidden-layer networks is an empirical observation, despite the fact that their representational power is equal.

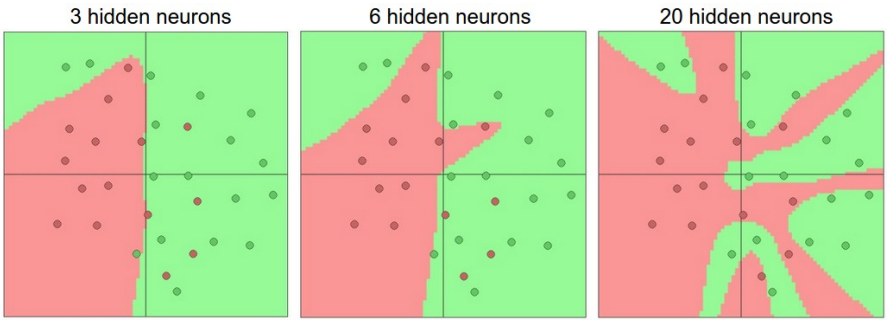
As an aside, in practice it is often the case that 3-layer neural networks will outperform 2-layer nets, but going even deeper (4,5,6-layer) rarely helps much more. This is in stark contrast to Convolutional Networks, where depth has been found to be an extremely important component for a good recognition system (e.g. on order of 10 learnable layers). One argument for this observation is that images contain hierarchical structure (e.g. faces are made up of eyes, which are made up of edges, etc.), so several layers of processing make intuitive sense for this data domain.

The full story is, of course, much more involved and a topic of much recent research. If you are interested in these topics we recommend for further reading:

- Deep Learning book in press by Bengio, Goodfellow, Courville, in particular Chapter 6.4.
- Do Deep Nets Really Need to be Deep?
- FitNets: Hints for Thin Deep Nets

Setting number of layers and their sizes

How do we decide on what architecture to use when faced with a practical problem? Should we use no hidden layers? One hidden layer? Two hidden layers? How large should each layer be? First, note that as we increase the size and number of layers in a Neural Network, the **capacity** of the network increases. That is, the space of representable functions grows since the neurons can collaborate to express many different functions. For example, suppose we had a binary classification problem in two dimensions. We could train three separate neural networks, each with one hidden layer of some size and obtain the following classifiers:



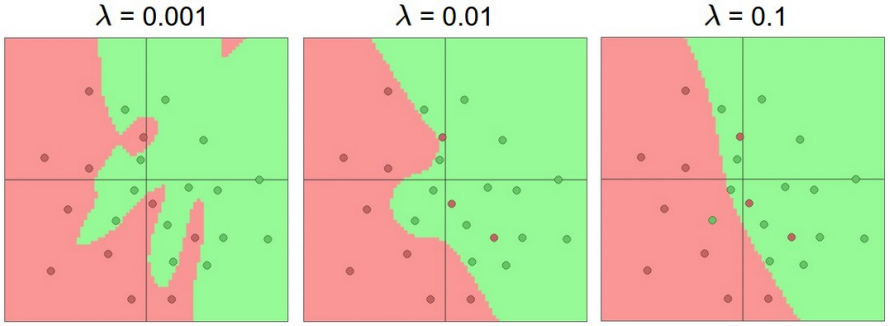
Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath. You can play with these examples in this ConvNetsJS demo.

In the diagram above, we can see that Neural Networks with more neurons can express more complicated functions. However, this is both a blessing (since we can learn to classify more complicated data) and a curse (since it is easier to overfit the training data). **Overfitting** occurs when a model with high capacity fits the noise in the data instead of the (assumed) underlying relationship. For example, the model with 20 hidden neurons fits all the training data but at the cost of segmenting the space into many disjoint red and green decision regions. The model with 3 hidden neurons only has the representational power to classify the data in broad strokes. It models the data as two blobs and interprets the few red points inside the green cluster as **outliers** (noise). In practice, this could lead to better **generalization** on the test set.

Based on our discussion above, it seems that smaller neural networks can be preferred if the data is not complex enough to prevent overfitting. However, this is incorrect - there are many other preferred ways to prevent overfitting in Neural Networks that we will discuss later (such as L2 regularization, dropout, input noise). In practice, it is always better to use these methods to control overfitting instead of the number of neurons.

The subtle reason behind this is that smaller networks are harder to train with local methods such as Gradient Descent: It's clear that their loss functions have relatively few local minima, but it turns out that many of these minima are easier to converge to, and that they are bad (i.e. with high loss). Conversely, bigger neural networks contain significantly more local minima, but these minima turn out to be much better in terms of their actual loss. Since Neural Networks are non-convex, it is hard to study these properties mathematically, but some attempts to understand these objective functions have been made, e.g. in a recent paper The Loss Surfaces of Multilayer Networks. In practice, what you find is that if you train a small network the final loss can display a good amount of variance - in some cases you get lucky and converge to a good place but in some cases you get trapped in one of the bad minima. On the other hand, if you train a large network you'll start to find many different solutions, but the variance in the final achieved loss will be much smaller. In other words, all solutions are about equally as good, and rely less on the luck of random initialization.

To reiterate, the regularization strength is the preferred way to control the overfitting of a neural network. We can look at the results achieved by three different settings:



The effects of regularization strength: Each neural network above has 20 hidden neurons, but changing the regularization strength makes its final decision regions smoother with a higher regularization. You can play with these examples in this ConvNetsJS demo.

The takeaway is that you should not be using smaller networks because you are afraid of overfitting. Instead, you should use as big of a neural network as your computational budget allows, and use other regularization techniques to control overfitting.

Backpropagation

Motivation. In this section we will develop expertise with an intuitive understanding of **backpropagation**, which is a way of computing gradients of expressions through recursive application of **chain rule**. Understanding of this process and its subtleties is critical for you to understand, and effectively develop, design and debug neural networks.

Problem statement. The core problem studied in this section is as follows: We are given some function $f(x)$ where x is a vector of inputs and we are interested in computing the gradient of f at x (i.e. $\nabla f(x)$).

Motivation. Recall that the primary reason we are interested in this problem is that in the specific case of neural networks, f will correspond to the loss function (L) and the inputs x will consist of the training data and the neural network weights. For example, the loss could be the SVM loss function and the inputs are both the training data $(x_i, y_i), i = 1 \ldots N$ and the weights and biases W, b. Note that (as is usually the case in Machine Learning) we think of the training data as given and fixed, and of the weights as variables we have control over. Hence, even though we can easily use backpropagation to compute the gradient on the input examples x_i, in practice we usually only compute the gradient for the parameters (e.g. W, b) so that we can use it to perform a parameter update. However, as we will see later in the class the gradient on x_i can still be useful sometimes, for example for purposes of visualization and interpreting what the Neural Network might be doing.

If you are coming to this class and you're comfortable with deriving gradients with chain rule, we would still like to encourage you to at least skim this section, since it presents a rarely developed view of backpropagation as backward flow in real-valued circuits and any insights you'll gain may help you throughout the class.

Simple expressions and interpretation of the gradient

Lets start simple so that we can develop the notation and conventions for more complex expressions. Consider a simple multiplication function of two numbers $f(x, y) = xy$. It is a matter of simple calculus to derive the partial derivative for either input:

$$f(x, y) = xy \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$$

Interpretation. Keep in mind what the derivatives tell you: They indicate the rate of change of a function with respect to that variable surrounding an infinitesimally small region near a particular point:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

A technical note is that the division sign on the left-hand side is, unlike the division sign on the right-hand side, not a division. Instead, this notation indicates that the operator $\frac{d}{dx}$ is being applied to the function f, and returns a different function (the derivative). A nice way to think about the expression above is that when h is very small, then the function is well-approximated by a straight line, and the derivative is its slope. In other words, the derivative on each variable tells you the sensitivity of the whole expression on its value. For example, if $x = 4, y = -3$ then $f(x, y) = -12$ and the derivative on x $\frac{\partial f}{\partial x} = -3$. This tells us that if we were to increase the value of this variable by a tiny amount, the effect on the whole expression would be to decrease it (due to the negative sign), and by three times that amount. This can be seen by rearranging the above equation (

$f(x+h) = f(x) + h\frac{df(x)}{dx}$). Analogously, since $\frac{\partial f}{\partial y} = 4$, we expect that increasing the value of y by some very small amount h would also increase the output of the function (due to the positive sign), and by $4h$.

> *The derivative on each variable tells you the sensitivity of the whole expression on its value.*

As mentioned, the gradient ∇f is the vector of partial derivatives, so we have that $\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}] = [y, x]$. Even though the gradient is technically a vector, we will often use terms such as *"the gradient on x"* instead of the technically correct phrase *"the partial derivative on x"* for simplicity.

We can also derive the derivatives for the addition operation:

$$f(x,y) = x + y \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$$

that is, the derivative on both x, y is one regardless of what the values of x, y are. This makes sense, since increasing either x, y would increase the output of f, and the rate of that increase would be independent of what the actual values of x, y are (unlike the case of multiplication above). The last function we'll use quite a bit in the class is the *max* operation:

$$f(x,y) = \max(x,y) \qquad \rightarrow \qquad \frac{\partial f}{\partial x} = 1(x >= y) \qquad \frac{\partial f}{\partial y} = 1(y >= x)$$

That is, the (sub)gradient is 1 on the input that was larger and 0 on the other input. Intuitively, if the inputs are $x = 4, y = 2$, then the max is 4, and the function is not sensitive to the setting of y. That is, if we were to increase it by a tiny amount h, the function would keep outputting 4, and therefore the gradient is zero: there is no effect. Of course, if we were to change y by a large amount (e.g. larger than 2), then the value of f would change, but the derivatives tell us nothing about the effect of such large changes on the inputs of a function; They are only informative for tiny, infinitesimally small changes on the inputs, as indicated by the $\lim_{h \to 0}$ in its definition.

Compound expressions with chain rule

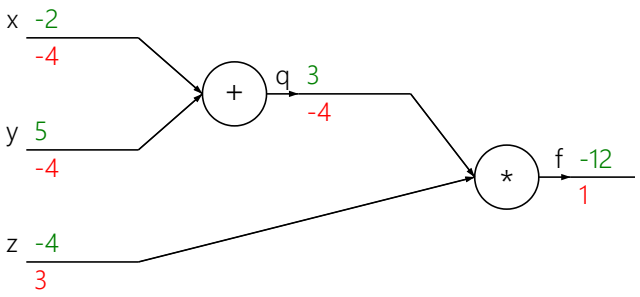
Lets now start to consider more complicated expressions that involve multiple composed functions, such as $f(x,y,z) = (x+y)z$. This expression is still simple enough to differentiate directly, but we'll take a particular approach to it that will be helpful with understanding the intuition behind backpropagation. In particular, note that this expression can be broken down into two expressions: $q = x + y$ and $f = qz$. Moreover, we know how to compute the derivatives of both expressions separately, as seen in the previous section. f is just multiplication of q and z, so $\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$, and q is addition of x and y so $\frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$. However, we don't necessarily care about the gradient on the intermediate value q - the value of $\frac{\partial f}{\partial q}$ is not useful. Instead, we are ultimately interested in the gradient of f with respect to its inputs x, y, z. The **chain rule** tells us that the correct way to "chain" these gradient expressions together is through multiplication. For example, $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$. In practice this is simply a multiplication of the two numbers that hold the two gradients. Lets see this with an example:

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
```

```
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dfdx = 1.0 * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfdy = 1.0 * dfdq # dq/dy = 1
```

We are left with the gradient in the variables `[dfdx,dfdy,dfdz]`, which tell us the sensitivity of the variables `x,y,z` on `f`!. This is the simplest example of backpropagation. Going forward, we will use a more concise notation that omits the `df` prefix. For example, we will simply write `dq` instead of `dfdq`, and always assume that the gradient is computed on the final output.

This computation can also be nicely visualized with a circuit diagram:



The real-valued *"circuit"* on left shows the visual representation of the computation. The **forward pass** computes values from inputs to output (shown in green). The **backward pass** then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the way to the inputs of the circuit. The gradients can be thought of as flowing backwards through the circuit.

Intuitive understanding of backpropagation

Notice that backpropagation is a beautifully local process. Every gate in a circuit diagram gets some inputs and can right away compute two things: 1. its output value and 2. the *local* gradient of its output with respect to its inputs. Notice that the gates can do this completely independently without being aware of any of the details of the full circuit that they are embedded in. However, once the forward pass is over, during backpropagation the gate will eventually learn about the gradient of its output value on the final output of the entire circuit. Chain rule says that the gate should take that gradient and multiply it into every gradient it normally computes for all of its inputs.

> *This extra multiplication (for each input) due to the chain rule can turn a single and relatively useless gate into a cog in a complex circuit such as an entire neural network.*

Lets get an intuition for how this works by referring again to the example. The add gate received inputs [-2, 5] and computed output 3. Since the gate is computing the addition operation, its local gradient for both of its inputs is +1. The rest of the circuit computed the final value, which is -12. During the backward pass in which the chain rule is applied recursively backwards through the circuit, the add gate (which is an input to the multiply gate) learns that the gradient for its output was -4. If we anthropomorphize the circuit as wanting to output a higher value (which can help with intuition), then we can think of the circuit as "wanting" the output of the add gate to be lower (due to negative sign), and with a *force* of 4. To continue the recurrence and to chain the gradient, the add gate takes that gradient and multiplies it to all of the local gradients for its inputs (making the gradient on both **x** and **y** 1 * -4 = -4). Notice that this has the desired effect: If **x,y** were to decrease (responding to their negative gradient) then the add gate's output would decrease, which in turn makes the multiply gate's output increase.

Backpropagation can thus be thought of as gates communicating to each other (through the gradient signal) whether they want their outputs to increase or decrease (and how strongly), so as to make the final output value higher.

Modularity: Sigmoid example

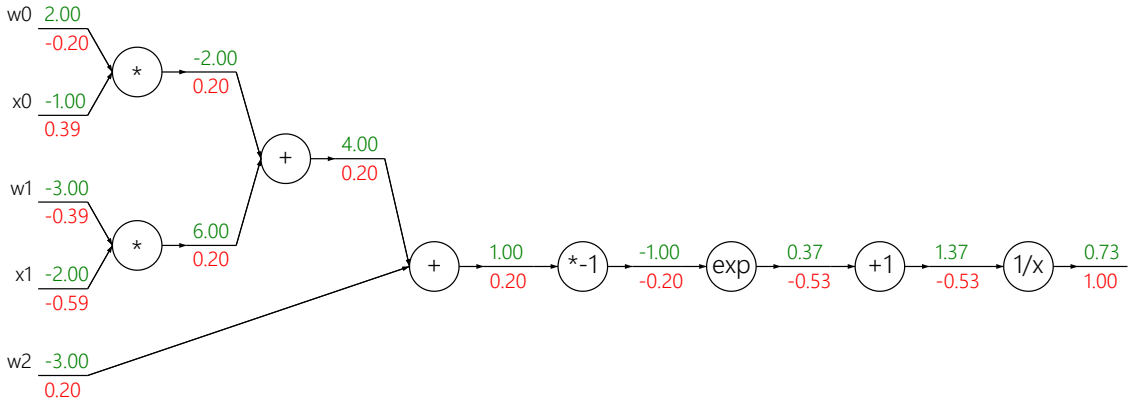
The gates we introduced above are relatively arbitrary. Any kind of differentiable function can act as a gate, and we can group multiple gates into a single gate, or decompose a function into multiple gates whenever it is convenient. Lets look at another expression that illustrates this point:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

as we will see later in the class, this expression describes a 2-dimensional neuron (with inputs **x** and weights **w**) that uses the *sigmoid activation* function. But for now lets think of this very simply as just a function from inputs *w,x* to a single number. The function is made up of multiple gates. In addition to the ones described already above (add, mul, max), there are four more:

$$\begin{aligned}
f(x) &= \frac{1}{x} && \rightarrow && \frac{df}{dx} = -1/x^2 \\
f_c(x) &= c + x && \rightarrow && \frac{df}{dx} = 1 \\
f(x) &= e^x && \rightarrow && \frac{df}{dx} = e^x \\
f_a(x) &= ax && \rightarrow && \frac{df}{dx} = a
\end{aligned}$$

Where the functions f_c, f_a translate the input by a constant of c and scale the input by a constant of a, respectively. These are technically special cases of addition and multiplication, but we introduce them as (new) unary gates here since we do not need the gradients for the constants c, a. The full circuit then looks as follows:



Example circuit for a 2D neuron with a sigmoid activation function. The inputs are [x0,x1] and the (learnable) weights of the neuron are [w0,w1,w2]. As we will see later, the neuron computes a dot product with the input and then its activation is softly squashed by the sigmoid function to be in range from 0 to 1.

In the example above, we see a long chain of function applications that operates on the result of the dot product between **w,x**. The function that these operations implement is called the *sigmoid function* $\sigma(x)$. It turns out that the derivative of the sigmoid function with respect to its input simplifies if you perform the derivation (after a fun tricky part where we add and subtract a 1 in the numerator):

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\rightarrow \quad \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

As we see, the gradient turns out to simplify and becomes surprisingly simple. For example, the sigmoid expression receives the input 1.0 and computes the output 0.73 during the forward pass. The derivation above shows that the *local* gradient would simply be (1 - 0.73) * 0.73 ~= 0.2, as the circuit computed before (see the image above), except this way it would be done with a single, simple and efficient expression (and with less numerical issues). Therefore, in any real practical application it would be very useful to group these operations into a single gate. Lets see the backprop for this neuron in code:

```
w = [2,-3,-3] # assume some random weights and data
x = [-1, -2]

# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function

# backward pass through the neuron (backpropagation)
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient derivation
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
# we're done! we have the gradients on the inputs to the circuit
```

Implementation protip: staged backpropagation. As shown in the code above, in practice it is always helpful to break down the forward pass into stages that are easily backpropped through. For example here we created an intermediate variable `dot` which holds the output of the dot product between `w` and `x`. During backward pass we then successively compute (in reverse order) the corresponding variables (e.g. `ddot`, and ultimately `dw, dx`) that hold the gradients of those variables.

The point of this section is that the details of how the backpropagation is performed, and which parts of the forward function we think of as gates, is a matter of convenience. It helps to be aware of which parts of the expression have easy local gradients, so that they can be chained together with the least amount of code and effort.

Backprop in practice: Staged computation

Lets see this with another example. Suppose that we have a function of the form:

$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

To be clear, this function is completely useless and it's not clear why you would ever want to compute its gradient, except for the fact that it is a good example of backpropagation in practice. It is very important to stress that if you were to launch into performing the differentiation with respect to either x or y, you would end up with very large and complex expressions. However, it turns out that doing so is completely unnecessary because we don't need to have an explicit function written down that evaluates the gradient. We only have to know how to compute it. Here is how we would structure the forward pass of such expression:

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                               #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
```

```
den = sigx + xpysqr # denominator                    #(6)
invden = 1.0 / den                                   #(7)
f = num * invden # done!                             #(8)
```

Phew, by the end of the expression we have computed the forward pass. Notice that we have structured the code in such way that it contains multiple intermediate variables, each of which are only simple expressions for which we already know the local gradients. Therefore, computing the backprop pass is easy: We'll go backwards and for every variable along the way in the forward pass (`sigy, num, sigx, xpy, xpysqr, den, invden`) we will have the same variable, but one that begins with a `d`, which will hold the gradient of the output of the circuit with respect to that variable. Additionally, note that every single piece in our backprop will involve computing the local gradient of that expression, and chaining it with the gradient on that expression with a multiplication. For each row, we also highlight which part of the forward pass it refers to:

```
# backprop f = num * invden
dnum = invden # gradient on numerator                        #(8)
dinvden = num                                                #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                           #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                           #(6)
dxpysqr = (1) * dden                                         #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                   #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                              #(4)
dy = (1) * dxpy                                              #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                             #(2)
dsigy = (1) * dnum                                           #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                            #(1)
# done! phew
```

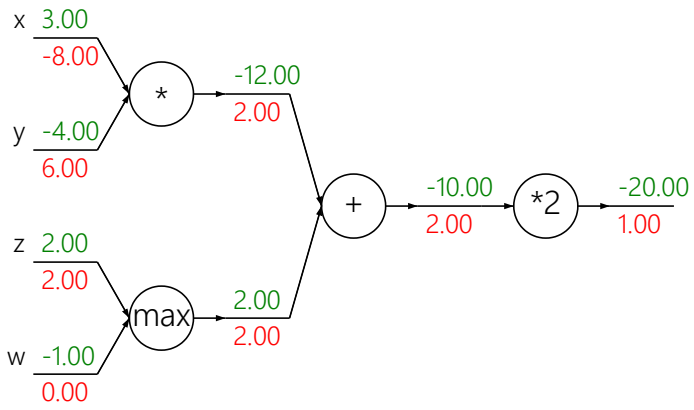
Notice a few things:

Cache forward pass variables. To compute the backward pass it is very helpful to have some of the variables that were used in the forward pass. In practice you want to structure your code so that you cache these variables, and so that they are available during backpropagation. If this is too difficult, it is possible (but wasteful) to recompute them.

Gradients add up at forks. The forward expression involves the variables **x,y** multiple times, so when we perform backpropagation we must be careful to use `+=` instead of `=` to accumulate the gradient on these variables (otherwise we would overwrite it). This follows the *multivariable chain rule* in Calculus, which states that if a variable branches out to different parts of the circuit, then the gradients that flow back to it will add.

Patterns in backward flow

It is interesting to note that in many cases the backward-flowing gradient can be interpreted on an intuitive level. For example, the three most commonly used gates in neural networks (*add,mul,max*), all have very simple interpretations in terms of how they act during backpropagation. Consider this example circuit:



An example circuit demonstrating the intuition behind the operations that backpropagation performs during the backward pass in order to compute the gradients on the inputs. Sum operation distributes gradients equally to all its inputs. Max operation routes the gradient to the higher input. Multiply gate takes the input activations, swaps them and multiplies by its gradient.

Looking at the diagram above as an example, we can see that:

The **add gate** always takes the gradient on its output and distributes it equally to all of its inputs, regardless of what their values were during the forward pass. This follows from the fact that the local gradient for the add operation is simply +1.0, so the gradients on all inputs will exactly equal the gradients on the output because it will be multiplied by x1.0 (and remain unchanged). In the example circuit above, note that the + gate routed the gradient of 2.00 to both of its inputs, equally and unchanged.

The **max gate** routes the gradient. Unlike the add gate which distributed the gradient unchanged to all its inputs, the max gate distributes the gradient (unchanged) to exactly one of its inputs (the input that had the highest value during the forward pass). This is because the local gradient for a max gate is 1.0 for the highest value, and 0.0 for all other values. In the example circuit above, the max operation routed the gradient of 2.00 to the **z** variable, which had a higher value than **w**, and the gradient on **w** remains zero.

The **multiply gate** is a little less easy to interpret. Its local gradients are the input values (except switched), and this is multiplied by the gradient on its output during the chain rule. In the example above, the gradient on **x** is -8.00, which is -4.00 x 2.00.

Unintuitive effects and their consequences. Notice that if one of the inputs to the multiply gate is very small and the other is very big, then the multiply gate will do something slightly unintuitive: it will assign a relatively huge gradient to the small input and a tiny gradient to the large input. Note that in linear classifiers where the weights are dot producted $w^T x_i$ (multiplied) with the inputs, this implies that the scale of the data has an effect on the magnitude of the gradient for the weights. For example, if you multiplied all input data examples x_i by 1000 during preprocessing, then the gradient on the weights will be 1000 times larger, and you'd have to lower the learning rate by that factor to compensate. This is why preprocessing matters a lot, sometimes in subtle ways! And having intuitive understanding for how the gradients flow can help you debug some of these cases.

Gradients for vectorized operations

The above sections were concerned with single variables, but all concepts extend in a straight-forward manner to matrix and vector operations. However, one must pay closer attention to dimensions and transpose operations.

Matrix-Matrix multiply gradient. Possibly the most tricky operation is the matrix-matrix multiplication (which generalizes all matrix-vector and vector-vector) multiply operations:

```
# forward pass
W = np.random.randn(5, 10)
X = np.random.randn(10, 3)
D = W.dot(X)
```

```
# now suppose we had the gradient on D from above in the circuit
dD = np.random.randn(*D.shape) # same shape as D
dW = dD.dot(X.T) #.T gives the transpose of the matrix
dX = W.T.dot(dD)
```

Tip: use dimension analysis! Note that you do not need to remember the expressions for `dW` and `dX` because they are easy to re-derive based on dimensions. For instance, we know that the gradient on the weights `dW` must be of the same size as `W` after it is computed, and that it must depend on matrix multiplication of `X` and `dD` (as is the case when both `X,W` are single numbers and not matrices). There is always exactly one way of achieving this so that the dimensions work out. For example, `X` is of size [10 x 3] and `dD` of size [5 x 3], so if we want `dW` and `W` has shape [5 x 10], then the only way of achieving this is with `dD.dot(X.T)`, as shown above.

Work with small, explicit examples. Some people may find it difficult at first to derive the gradient updates for some vectorized expressions. Our recommendation is to explicitly write out a minimal vectorized example, derive the gradient on paper and then generalize the pattern to its efficient, vectorized form.

Erik Learned-Miller has also written up a longer related document on taking matrix/vector derivatives which you might find helpful. Find it here.

Summary

- We introduced a very coarse model of a biological **neuron**.
- We discussed several types of **activation functions** that are used in practice, with ReLU being the most common choice.
- We introduced **Neural Networks** where neurons are connected with **Fully-Connected layers** where neurons in adjacent layers have full pair-wise connections, but neurons within a layer are not connected.
- We saw that this layered architecture enables very efficient evaluation of Neural Networks based on matrix multiplications interwoven with the application of the activation function.
- We saw that that Neural Networks are **universal function approximators**, but we also discussed the fact that this property has little to do with their ubiquitous use. They are used because they make certain "right" assumptions about the functional forms of functions that come up in practice.
- We discussed the fact that larger networks will always work better than smaller networks, but their higher model capacity must be appropriately addressed with stronger regularization (such as higher weight decay), or they might overfit. We will see more forms of regularization (especially dropout) in later sections.
- We developed intuition for what the gradients mean, how they flow backwards in the circuit, and how they communicate which part of the circuit should increase or decrease and with what force to make the final output higher.
- We discussed the importance of **staged computation** for practical implementations of backpropagation. You always want to break up your function into modules for which you can easily derive local gradients, and then chain them with chain rule. Crucially, you almost never want to write out these expressions on paper and differentiate them symbolically in full, because you never need an explicit mathematical equation for the gradient of the input variables. Hence, decompose your expressions into stages such that you can differentiate every stage independently (the stages will be matrix vector multiplies, or max operations, or sum operations, etc.) and then backprop through the variables one step at a time.

In the next section we will start to define neural networks, and backpropagation will allow us to efficiently compute the gradient of a loss function with respect to its parameters. In other words, we're now ready to train neural nets, and the most conceptually difficult part of this class is behind us! ConvNets will then be a small step away.

References

- Automatic differentiation in machine learning: a survey