CHAPTER 6

Neural Networks

You've probably been hearing a lot about "neural networks." Now that we have several useful machine-learning concepts (hypothesis classes, classification, regression, gradient descent, regularization, etc.) we are well equipped to understand neural networks in detail.

This is, in some sense, the "third wave" of neural nets. The basic idea is founded on the 1943 model of neurons of McCulloch and Pitts and the learning ideas of Hebb. There was a great deal of excitement, but not a lot of practical success: there were good training methods (e.g., perceptron) for linear functions, and interesting examples of non-linear functions, but no good way to train non-linear functions from data. Interest died out for a while, but was re-kindled in the 1980s when several people came up with a way to train neural networks with "back-propagation," which is a particular style of implementing gradient descent, that we will study here. By the mid-90s, the enthusiasm waned again, because although we could train non-linear networks, the training tended to be slow and was plagued by a problem of getting stuck in local optima. Support vector machines (SVMs) that use regularization of high-dimensional hypotheses by seeking to maximize the margin, and kernel methods that are an efficient and beautiful way of using feature transformations to non-linearly transform data into a higher-dimensional space, provided reliable learning methods with guaranteed convergence and no local optima.

As with many good ideas in science, the basic idea for how to train non-linear neural networks with gradient descent was independently developed by more than one researcher.

However, during the SVM enthusiasm, several groups kept working on neural networks, and their work, in combination with an increase in available data and computation, has made them rise again. They have become much more reliable and capable, and are now the method of choice in many applications. There are many, many variations of neural networks, which we can't even begin to survey. We will study the core "feed-forward" networks with "back-propagation" training, and then, in later chapters, address some of the major advances beyond this core.

The number increases daily, as may be seen on `arxiv.org`.

We can view neural networks from several different perspectives:

View 1: An application of stochastic gradient descent for classification and regression with a potentially very rich hypothesis class.

View 2: A brain-inspired network of neuron-like computing elements that learn distributed representations.

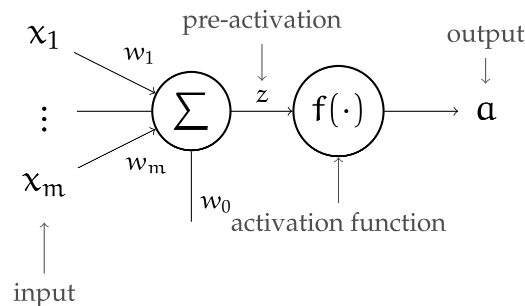
View 3: A method for building applications that make predictions based on huge amounts of data in very complex domains.

We will mostly take view 1, with the understanding that the techniques we develop will enable the applications in view 3. View 2 was a major motivation for the early development of neural networks, but the techniques we will study do not seem to actually account for the biological learning processes in brains.

Some prominent researchers are, in fact, working hard to find analogues of these methods in the brain.

6.1 Basic element

The basic element of a neural network is a "neuron," pictured schematically below. We will also sometimes refer to a neuron as a "unit" or "node."



It is a non-linear function of an input vector $x \in \mathbb{R}^m$ to a single output value $a \in \mathbb{R}$. It is parameterized by a vector of *weights* $(w_1, \ldots, w_m) \in \mathbb{R}^m$ and an *offset* or *threshold* $w_0 \in \mathbb{R}$. In order for the neuron to be non-linear, we also specify an *activation function* $f : \mathbb{R} \to \mathbb{R}$, which can be the identity ($f(x) = x$, in that case the neuron is a linear function of x), but can also be any other function, though we will only be able to work with it if it is differentiable.

Sorry for changing our notation here. We were using d as the dimension of the input, but we are trying to be consistent here with many other accounts of neural networks. It is impossible to be consistent with all of them though—there are many different ways of telling this story.

This should remind you of our θ and θ_0 for linear models.

The function represented by the neuron is expressed as:

$$a = f(z) = f\left(\left(\sum_{j=1}^{m} x_j w_j\right) + w_0\right) = f(w^T x + w_0) \ .$$

Before thinking about a whole network, we can consider how to train a single unit. Given a loss function $\mathcal{L}(\text{guess}, \text{actual})$ and a dataset $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$, we can do (stochastic) gradient descent, adjusting the weights w, w_0 to minimize

$$J(w, w_0) = \sum_i \mathcal{L}\left(\text{NN}(x^{(i)}; w, w_0), y^{(i)}\right) \ ,$$

where NN is the output of our single-unit neural net for a given input.

We have already studied two special cases of the neuron: linear logistic classifiers (LLCs) with NLL loss and regressors with quadratic loss! The activation function for the LLC is $f(x) = \sigma(x)$ and for linear regression it is simply $f(x) = x$.

Study Question: Just for a single neuron, imagine for some reason, that we decide to use activation function $f(z) = e^z$ and loss function $\mathcal{L}(\text{guess}, \text{actual}) = (\text{guess} - \text{actual})^2$. Derive a gradient descent update for w and w_0.

6.2 Networks

Now, we'll put multiple neurons together into a *network*. A neural network in general takes in an input $x \in \mathbb{R}^m$ and generates an output $a \in \mathbb{R}^n$. It is constructed out of multiple

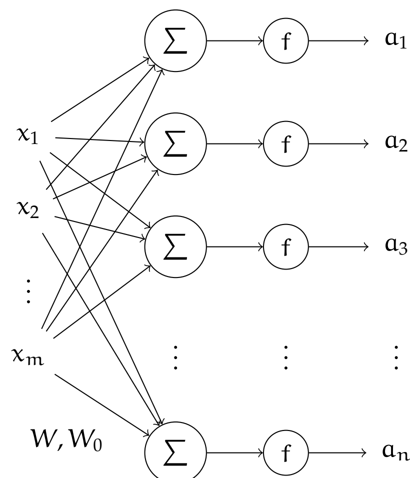
neurons; the inputs of each neuron might be elements of x and/or outputs of other neurons. The outputs are generated by n *output units*.

In this chapter, we will only consider *feed-forward* networks. In a feed-forward network, you can think of the network as defining a function-call graph that is *acyclic*: that is, the input to a neuron can never depend on that neuron's output. Data flows one way, from the inputs to the outputs, and the function computed by the network is just a composition of the functions computed by the individual neurons.

Although the graph structure of a feed-forward neural network can really be anything (as long as it satisfies the feed-forward constraint), for simplicity in software and analysis, we usually organize them into *layers*. A layer is a group of neurons that are essentially "in parallel": their inputs are outputs of neurons in the previous layer, and their outputs are the input to the neurons in the next layer. We'll start by describing a single layer, and then go on to the case of multiple layers.

6.2.1 Single layer

A *layer* is a set of units that, as we have just described, are not connected to each other. The layer is called *fully connected* if, as in the diagram below, all of the inputs (i.e., $x_1, x_2, \dots x_m$ in this case) are connected to every unit in the layer. A layer has input $x \in \mathbb{R}^m$ and output (also known as *activation*) $a \in \mathbb{R}^n$.



Since each unit has a vector of weights and a single offset, we can think of the weights of the whole layer as a matrix, W, and the collection of all the offsets as a vector W_0. If we have m inputs, n units, and n outputs, then

- W is an $m \times n$ matrix,
- W_0 is an $n \times 1$ column vector,
- X, the input, is an $m \times 1$ column vector,
- $Z = W^T X + W_0$, the *pre-activation*, is an $n \times 1$ column vector,
- A, the *activation*, is an $n \times 1$ column vector,

and the output vector is

$$A = f(Z) = f(W^T X + W_0) \ .$$

The activation function f is applied element-wise to the pre-activation values Z.

6.2.2 Many layers

A single neural network generally combines multiple layers, most typically by feeding the outputs of one layer into the inputs of another layer.

We have to start by establishing some nomenclature. We will use l to name a layer, and let m^l be the number of inputs to the layer and n^l be the number of outputs from the layer. Then, W^l and W_0^l are of shape $m^l \times n^l$ and $n^l \times 1$, respectively. Note that the input to layer l is the output from layer $l-1$, so we have $m^l = n^{l-1}$, and as a result A^{l-1} is of shape $m^l \times 1$, or equivalently $n^{l-1} \times 1$. Let f^l be the activation function of layer l. Then, the pre-activation outputs are the $n^l \times 1$ vector

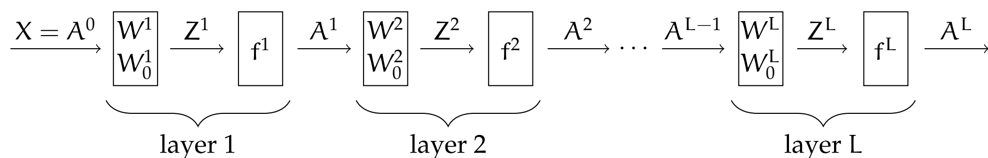
It is technically possible to have different activation functions within the same layer, but, again, for convenience in specification and implementation, we generally have the same activation function within a layer.

$$Z^l = {W^l}^T A^{l-1} + W_0^l$$

and the activation outputs are simply the $n^l \times 1$ vector

$$A^l = f^l(Z^l) \ .$$

Here's a diagram of a many-layered network, with two blocks for each layer, one representing the linear part of the operation and one representing the non-linear activation function. We will use this structural decomposition to organize our algorithmic thinking and implementation.



6.3 Choices of activation function

There are many possible choices for the activation function. We will start by thinking about whether it's really necessary to have an f at all.

What happens if we let f be the identity? Then, in a network with L layers (we'll leave out W_0 for simplicity, but keeping it wouldn't change the form of this argument),

$$A^L = {W^L}^T A^{L-1} = {W^L}^T {W^{L-1}}^T \cdots {W^1}^T X \ .$$

So, multiplying out the weight matrices, we find that

$$A^L = W^{\text{total}} X \ ,$$

which is a *linear* function of X! Having all those layers did not change the representational capacity of the network: the non-linearity of the activation function is crucial.

Study Question: Convince yourself that any function representable by any number of linear layers (where f is the identity function) can be represented by a single layer.

Now that we are convinced we need a non-linear activation, let's examine a few common choices. These are shown mathematically below, followed by plots of these functions.

Step function:

$$\text{step}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$$

Rectified linear unit (ReLU):

$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} = \max(0, z)$$

Sigmoid function: Also known as a *logistic* function. This can sometimes be interpreted as probability, because for any value of z the output is in $(0, 1)$:

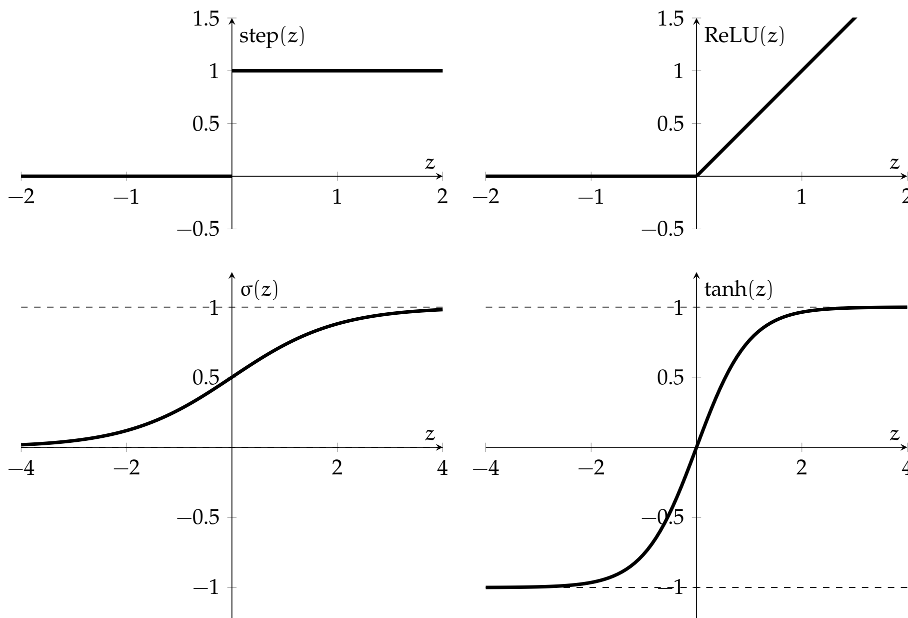
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Hyperbolic tangent: Always in the range $(-1, 1)$:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Softmax function: Takes a whole vector $Z \in \mathbb{R}^n$ and generates as output a vector $A \in (0, 1)^n$ with the property that $\sum_{i=1}^n A_i = 1$, which means we can interpret it as a probability distribution over n items:

$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_n)/\sum_i \exp(z_i) \end{bmatrix}$$



The original idea for neural networks involved using the **step** function as an activation, but because the derivative of the step function is zero everywhere except at the discontinuity (and there it is undefined), gradient-descent methods won't be useful in finding a good setting of the weights, and so we won't consider them further. They have been replaced, in a sense, by the sigmoid, ReLU, and tanh activation functions.

Study Question: Consider sigmoid, ReLU, and tanh activations. Which one is most like a step function? Is there an additional parameter you could add to a sigmoid that would make it be more like a step function?

Study Question: What is the derivative of the ReLU function? Are there some values of the input for which the derivative vanishes?

ReLUs are especially common in internal ("hidden") layers, sigmoid activations are common for the output for binary classification, and softmax activations are common for the output for multi-class classification (see Section 4.3.3 for an explanation).

6.4 Loss functions and activation functions

Different loss functions make different assumptions about the range of values they will get as input and, as we have seen, different activation functions will produce output values in different ranges. When you are designing a neural network, it's important to make these things fit together well. In particular, we will think about matching loss functions with the activation function in the last layer, f^L. Here is a table of loss functions and activations that make sense for them:

Loss	f^L	task
squared	linear	regression
NLL	sigmoid	binary classification
NLLM	softmax	multi-class classification

We explored squared loss in Chapter 2 and (NLL and NLLM) in Chapter 4.

6.5 Error back-propagation

We will train neural networks using gradient descent methods. It's possible to use *batch* gradient descent, in which we sum up the gradient over all the points (as in Section 3.2 of chapter 3) or stochastic gradient descent (SGD), in which we take a small step with respect to the gradient considering a single point at a time (as in Section 3.4 of Chapter 3).

Our notation is going to get pretty hairy pretty quickly. To keep it as simple as we can, we'll focus on computing the contribution of one data point $x^{(i)}$ to the gradient of the loss with respect to the weights, for SGD; you can simply sum up these gradients over all the data points if you wish to do batch descent.

So, to do SGD for a training example (x, y), we need to compute $\nabla_W \mathcal{L}(\text{NN}(x; W), y)$, where W represents all weights W^l, W_0^l in all the layers $l = (1, \ldots, L)$. This seems terrifying, but is actually quite easy to do using the chain rule.

Remember the chain rule! If $a = f(b)$ and $b = g(c)$, so that $a = f(g(c))$, then $\frac{da}{dc} = \frac{da}{db} \cdot \frac{db}{dc} = f'(b)g'(c) = f'(g(c))g'(c)$.

Remember that we are always computing the gradient of the loss function *with respect to the weights* for a particular value of (x, y). That tells us how much we want to change the weights, in order to reduce the loss incurred on this particular training example.

6.5.1 First, suppose everything is one-dimensional

To get some intuition for how these derivations work, we'll first suppose everything in our neural network is one-dimensional. In particular, we'll assume there are $m^l = 1$ inputs and $n^l = 1$ outputs at every layer. So layer l looks like:

$$a^l = f^l(z^l), \quad z^l = w^l a^{l-1} + w_0^l.$$

In the equation above, we're using the lowercase letters $a^l, z^l, w^l, a^{l-1}, w_0^l$ to emphasize that all of these quantities are scalars just for the moment. We'll look at the more general matrix case below.

To use SGD, then, we want to compute $\partial \mathcal{L}(\text{NN}(x; W), y)/\partial w^l$ and $\partial \mathcal{L}(\text{NN}(x; W), y)/\partial w_0^l$

Check your understanding: why do we need exactly these quantities for SGD?

for each layer l and each data point (x, y). Below we'll write "loss" as an abbreviation for $\mathcal{L}(\text{NN}(x; W), y)$. Then our first quantity of interest is $\partial\text{loss}/\partial w^l$. The chain rule gives us the following. First, let's look at the case $l = L$:

$$\begin{aligned}
\frac{\partial \text{loss}}{\partial w^L} &= \frac{\partial \text{loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L} \\
&= \frac{\partial \text{loss}}{\partial a^L} \cdot (f^L)'(z^L) \cdot a^{L-1}.
\end{aligned}$$

Now we can look at the case of general l:

$$\begin{aligned}
\frac{\partial \text{loss}}{\partial w^l} &= \frac{\partial \text{loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdots \frac{\partial z^{l+1}}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial w^l} \\
&= \frac{\partial \text{loss}}{\partial a^L} \cdot (f^L)'(z^L) \cdot w^L \cdot (f^{L-1})'(z^{L-1}) \cdot \cdots \cdot w^{l+1} \cdot (f^l)'(z^l) \cdot a^{l-1} \\
&= \frac{\partial \text{loss}}{\partial z^l} \cdot a^{l-1}.
\end{aligned}$$

Note that every multiplication above is scalar multiplication because every term in every product above is a scalar. And though we solved for all the other terms in the product, we haven't solved for $\partial\text{loss}/\partial a^L$ because the derivative will depend on which loss function you choose. Once you choose a loss function though, you should be able to compute this derivative.

Study Question: Suppose you choose squared loss. What is $\partial\text{loss}/\partial a^L$?

Study Question: Check the derivations above yourself. You should use the chain rule and also solve for the individual derivatives that arise in the chain rule.

Study Question: Check that the the final layer ($l = L$) case is a special case of the general layer l case above.

Study Question: Derive $\partial\mathcal{L}(\text{NN}(x; W), y)/\partial w_0^l$ for yourself, for both the final layer ($l = L$) and general l.

Study Question: Does the $L = 1$ case remind you of anything from earlier in this course?

Study Question: Write out the full SGD algorithm for this neural network.

It's pretty typical to run the chain rule from left to right like we did above. But, for where we're going next, it will be useful to notice that it's completely equivalent to write it in the other direction. So we can rewrite our result from above as follows:

$$\frac{\partial \text{loss}}{\partial w^l} = a^{l-1} \cdot \frac{\partial \text{loss}}{\partial z^l} \tag{6.1}$$

$$\frac{\partial \text{loss}}{\partial z^l} = \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^{l+1}}{\partial a^l} \cdots \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial \text{loss}}{\partial a^L} \tag{6.2}$$

$$= \frac{\partial a^l}{\partial z^l} \cdot w^{l+1} \cdots \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot w^L \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial \text{loss}}{\partial a^L}. \tag{6.3}$$

6.5.2 The general case

Next we're going to do everything that we did above, but this time we'll allow any number of inputs m^l and outputs n^l at every layer. First, we'll tell you the results that correspond to our derivations above. Then we'll talk about why they make sense. And finally we'll derive them carefully.

OK, let's start with the results! Again, below we'll be using "loss" as an abbreviation for $\mathcal{L}(\text{NN}(x; W), y)$. Then,

$$\underbrace{\frac{\partial \text{loss}}{\partial W^l}}_{m^l \times n^l} = \underbrace{A^{l-1}}_{m^l \times 1} \underbrace{\left(\frac{\partial \text{loss}}{\partial Z^l}\right)^T}_{1 \times n^l} \tag{6.4}$$

$$\frac{\partial \text{loss}}{\partial Z^l} = \frac{\partial A^l}{\partial Z^l} \cdot \frac{\partial Z^{l+1}}{\partial A^l} \cdot \ldots \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{\partial A^L} \tag{6.5}$$

$$= \frac{\partial A^l}{\partial Z^l} \cdot W^{l+1} \cdot \ldots \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot W^L \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{\partial A^L} . \tag{6.6}$$

First, compare each equation to its one-dimensional counterpart, and make sure you see the similarities. That is, compare the general weight derivatives in Eq. 6.4 to the one-dimensional case in Eq. 6.1. Compare the intermediate derivative of loss with respect to the pre-activations Z^l in Eq. 6.5 to the one-dimensional case in Eq. 6.2. And finally compare the version where we've substituted in some of the derivatives in Eq. 6.6 to Eq. 6.3. Hopefully you see how the forms are very analogous. But in the matrix case, we now have to be careful about the matrix dimensions. We'll check these matrix dimensions below.

Let's start by talking through each of the terms in the matrix version of these equations. Recall that loss is a scalar, and W^l is a matrix of size $m^l \times n^l$. You can read about the conventions in the course for derivatives starting in this chapter in Appendix A. By these conventions (not the only possible conventions!), we have that $\partial \text{loss}/\partial W^l$ will be a matrix of size $m^l \times n^l$ whose (i, j) entry is the scalar $\partial \text{loss}/\partial W^l_{i,j}$. In some sense, we're just doing a bunch of traditional scalar derivatives, and the matrix notation lets us write them all simultaneously and succinctly. In particular, for SGD, we need to find the derivative of the loss with respect to every scalar component of the weights because these are our model's parameters and therefore are the things we want to update in SGD.

The next quantity we see in Eq. 6.4 is A^{l-1}, which we recall has size $m^l \times 1$ (or equivalently $n^{l-1} \times 1$ since it represents the outputs of the $l-1$ layer). Finally, we see $\partial \text{loss}/\partial Z^l$. Again, loss is a scalar, and Z^l is a $n^l \times 1$ vector. So by the conventions in Appendix A, we have that $\partial \text{loss}/\partial Z^l$ has size $n^l \times 1$. The transpose then has size $1 \times n^l$. Now you should be able to check that the dimensions all make sense in Eq. 6.4; in particular, you can check that inner dimensions agree in the matrix multiplication and that, after the multiplication, we should be left with something that has the dimensions on the lefthand side.

Now let's look at Eq. 6.6. We're computing $\partial \text{loss}/\partial Z^l$ so that we can use it in Eq. 6.4. The weights are familiar. The one part that remains is terms of the form $\partial A^l/\partial Z^l$. Checking out Appendix A, we see that this term should be a matrix of size $n^l \times n^l$ since A^l and Z^l both have size $n^l \times 1$. The (i, j) entry of this matrix is $\partial A^l_j/\partial Z^l_i$. This scalar derivative is something that you can compute when you know your activation function. If you're not using a softmax activation function, A^l_j typically is a function only of Z^l_j, which means that $\partial A^l_j/\partial Z^l_i$ should equal 0 whenever $i \neq j$, and that $\partial A^l_j/\partial Z^l_j = (f^l)'(Z^l_j)$.

Study Question: Compute the dimensions of every term in Eqs. 6.5 and 6.6 using Appendix A. After you've done that, check that all the matrix multiplications work; that is, check that the inner dimensions agree and that the lefthand side and right-hand side of these equations have the same dimensions.

Study Question: If I use the identity activation function, what is $\partial A^l_j/\partial Z^l_j$ for any j? What is the full matrix $\partial A^l/\partial Z^l$?

6.5.3 Derivations for the general case

You can use everything above without deriving it yourself. But if you want to find the gradients of loss with respect to W_0^l (which we need for SGD!), then you'll want to know how to actually do these derivations. So next we'll work out the derivations.

The key trick is to just break every equation down into its scalar meaning. For instance, the (i, j) element of $\partial\text{loss}/\partial W^l$ is $\partial\text{loss}/\partial W^l_{i,j}$. If you think about it for a moment (and it might help to go back to the one-dimensional case), the loss is a function of the elements of Z^l, and the elements of Z^l are a function of the $W^l_{i,j}$. There are n^l elements of Z^l, so we can use the chain rule to write

$$\frac{\partial\text{loss}}{\partial W^l_{i,j}} = \sum_{k=1}^{n^l} \frac{\partial\text{loss}}{\partial Z^l_k} \frac{\partial Z^l_k}{\partial W^l_{i,j}}. \tag{6.7}$$

To figure this out, let's remember that $Z^l = (W^l)^\top A^{l-1} + W^l_0$. We can write one element of the Z^l vector, then, as $Z^l_b = \sum_{a=1}^{m^l} W^l_{a,b} A^{l-1}_a + (W^l_0)_b$. It follows that $\partial Z^l_k / \partial W^l_{i,j}$ will be zero except when $k = j$ (check you agree!). So we can rewrite Eq. 6.7 as

$$\frac{\partial\text{loss}}{\partial W^l_{i,j}} = \frac{\partial\text{loss}}{\partial Z^l_j} \frac{\partial Z^l_j}{\partial W^l_{i,j}} = \frac{\partial\text{loss}}{\partial Z^l_j} A^{l-1}_i. \tag{6.8}$$

Finally, then, we match entries of the matrices on both sides of the equation above to recover Eq. 6.4.

Study Question: Check that Eq. 6.8 and Eq. 6.4 say the same thing.

Study Question: Convince yourself that $\partial Z^l/\partial A^{l-1} = W^l$ by comparing the entries of the matrices on both sides on the equality sign.

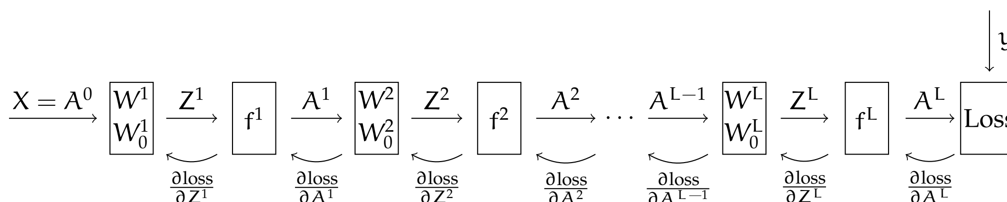
Study Question: Convince yourself that Eq. 6.5 is true.

Study Question: Apply the same reasoning to find the gradients of loss with respect to W^l_0.

6.5.4 Reflecting on backpropagation

This general process of computing the gradients of the loss with respect to the weights is called *error back-propagation*. The idea is that we first do a *forward pass* to compute all the a and z values at all the layers, and finally the actual loss. Then, we can work backward and compute the gradient of the loss with respect to the weights in each layer, starting at layer L and going back to layer 1.

We could call this "blame propagation". Think of loss as how mad we are about the prediction just made. Then $\partial\text{loss}/\partial A^L$ is how much we blame A^L for the loss. The last module has to take in $\partial\text{loss}/\partial A^L$ and compute $\partial\text{loss}/\partial Z^L$, which is how much we blame Z^L for the loss. The next module (working backwards) takes in $\partial\text{loss}/\partial Z^L$ and computes $\partial\text{loss}/\partial A^{L-1}$. So every module is accepting its blame for the loss, computing how much of it to allocate to each of its inputs, and passing the blame back to them.



If we view our neural network as a sequential composition of modules (in our work so far, it has been an alternation between a linear transformation with a weight matrix, and a component-wise application of a non-linear activation function), then we can define a simple API for a module that will let us compute the forward and backward passes, as

well as do the necessary weight updates for gradient descent. Each module has to provide the following "methods." We are already using letters a, x, y, z with particular meanings, so here we will use u as the vector input to the module and v as the vector output:

- forward: $u \to v$
- backward: $u, v, \partial L/\partial v \to \partial L/\partial u$
- weight grad: $u, \partial L/\partial v \to \partial L/\partial W$ only needed for modules that have weights W

In homework we will ask you to implement these modules for neural network components, and then use them to construct a network and train it as described in the next section.

6.6 Training

Here we go! Here's how to do stochastic gradient descent training on a feed-forward neural network. After this pseudo-code, we motivate the choice of initialization in lines 2 and 3. The actual computation of the gradient values (e.g., $\partial \text{loss}/\partial A^L$) is not directly defined in this code, because we want to make the structure of the computation clear.

Study Question: What is $\partial Z^l/\partial W^l$?

Study Question: Which terms in the code below depend on f^L?

SGD-NEURAL-NET($\mathcal{D}_n, T, L, (m^1, \dots, m^L), (f^1, \dots, f^L), \text{Loss}$)

1 **for** $l = 1$ **to** L
2 $\quad W^l_{ij} \sim \text{Gaussian}(0, 1/m^l)$
3 $\quad W^l_{0j} \sim \text{Gaussian}(0, 1)$
4 **for** $t = 1$ **to** T
5 $\quad i = \text{random sample from } \{1, \dots, n\}$
6 $\quad A^0 = x^{(i)}$
7 $\quad$ // forward pass to compute the output A^L
8 $\quad$ **for** $l = 1$ **to** L
9 $\quad\quad Z^l = W^{l^T} A^{l-1} + W^l_0$
10 $\quad\quad A^l = f^l(Z^l)$
11 $\quad \text{loss} = \text{Loss}(A^L, y^{(i)})$
12 $\quad$ **for** $l = L$ **to** 1:
13 $\quad\quad$ // error back-propagation
14 $\quad\quad \partial\text{loss}/\partial A^l =$ **if** $l < L$ **then** $\partial Z^{l+1}/\partial A^l \cdot \partial\text{loss}/\partial Z^{l+1}$ **else** $\partial\text{loss}/\partial A^L$
15 $\quad\quad \partial\text{loss}/\partial Z^l = \partial A^l/\partial Z^l \cdot \partial\text{loss}/\partial A^l$
16 $\quad\quad$ // compute gradient with respect to weights
17 $\quad\quad \partial\text{loss}/\partial W^l = A^{l-1} \cdot \left(\partial\text{loss}/\partial Z^l\right)^T$
18 $\quad\quad \partial\text{loss}/\partial W^l_0 = \partial\text{loss}/\partial Z^l$
19 $\quad\quad$ // stochastic gradient descent update
20 $\quad\quad W^l = W^l - \eta(t) \cdot \partial\text{loss}/\partial W^l$
21 $\quad\quad W^l_0 = W^l_0 - \eta(t) \cdot \partial\text{loss}/\partial W^l_0$

Initializing W is important; if you do it badly there is a good chance the neural network training won't work well. First, it is important to initialize the weights to random values. We want different parts of the network to tend to "address" different aspects of the problem; if they all start at the same weights, the symmetry will often keep the values from moving in useful directions. Second, many of our activation functions have (near)

zero slope when the pre-activation z values have large magnitude, so we generally want to keep the initial weights small so we will be in a situation where the gradients are non-zero, so that gradient descent will have some useful signal about which way to go.

One good general-purpose strategy is to choose each weight at random from a Gaussian (normal) distribution with mean 0 and standard deviation $(1/m)$ where m is the number of inputs to the unit.

Study Question: If the input x to this unit is a vector of 1's, what would the expected pre-activation z value be with these initial weights?

We write this choice (where $\sim$ means "is drawn randomly from the distribution") as $W^l_{ij} \sim \text{Gaussian}\left(0, \frac{1}{m^l}\right)$. It will often turn out (especially for fancier activations and loss functions) that computing $\frac{\partial \text{loss}}{\partial Z^L}$ is easier than computing $\frac{\partial \text{loss}}{\partial A^L}$ and $\frac{\partial A^L}{\partial Z^L}$. So, we may instead ask for an implementation of a loss function to provide a backward method that computes $\partial \text{loss}/\partial Z^L$ directly.

6.7 Optimizing neural network parameters

Because neural networks are just parametric functions, we can optimize loss with respect to the parameters using standard gradient-descent software, but we can take advantage of the structure of the loss function and the hypothesis class to improve optimization. As we have seen, the modular function-composition structure of a neural network hypothesis makes it easy to organize the computation of the gradient. As we have also seen earlier, the structure of the loss function as a sum over terms, one per training data point, allows us to consider stochastic gradient methods. In this section we'll consider some alternative strategies for organizing training, and also for making it easier to handle the step-size parameter.

6.7.1 Batches

Assume that we have an objective of the form

$$J(W) = \sum_{i=1}^{n} \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ ,$$

where h is the function computed by a neural network, and W stands for all the weight matrices and vectors in the network.

Recall that, when we perform *batch* (or the vanilla) gradient descent, we use the update rule

$$W_t = W_{t-1} - \eta \nabla_W J(W_{t-1}) \ ,$$

which is equivalent to

$$W_t = W_{t-1} - \eta \sum_{i=1}^{n} \nabla_W \mathcal{L}(h(x^{(i)}; W_{t-1}), y^{(i)}) \ .$$

So, we sum up the gradient of loss at each training point, with respect to W, and then take a step in the negative direction of the gradient.

In *stochastic* gradient descent, we repeatedly pick a point $(x^{(i)}, y^{(i)})$ at random from the data set, and execute a weight update on that point alone:

$$W_t = W_{t-1} - \eta \nabla_W \mathcal{L}(h(x^{(i)}; W_{t-1}), y^{(i)}) \ .$$

As long as we pick points uniformly at random from the data set, and decrease η at an appropriate rate, we are guaranteed, with high probability, to converge to at least a local optimum.

These two methods have offsetting virtues. The batch method takes steps in the exact gradient direction but requires a lot of computation before even a single step can be taken, especially if the data set is large. The stochastic method begins moving right away, and can sometimes make very good progress before looking at even a substantial fraction of the whole data set, but if there is a lot of variability in the data, it might require a very small η to effectively average over the individual steps moving in "competing" directions.

An effective strategy is to "average" between batch and stochastic gradient descent by using *mini-batches*. For a mini-batch of size K, we select K distinct data points uniformly at random from the data set and do the update based just on their contributions to the gradient

$$W_t = W_{t-1} - \eta \sum_{i=1}^{K} \nabla_W \mathcal{L}(h(x^{(i)}; W_{t-1}), y^{(i)}) \ .$$

Most neural network software packages are set up to do mini-batches.

Study Question: For what value of K is mini-batch gradient descent equivalent to stochastic gradient descent? To batch gradient descent?

Picking K unique data points at random from a large data-set is potentially computationally difficult. An alternative strategy, if you have an efficient procedure for randomly shuffling the data set (or randomly shuffling a list of indices into the data set) is to operate in a loop, roughly as follows:

```
MINI-BATCH-SGD(NN, data, K)
1  n = length(data)
2  while not done:
3      RANDOM-SHUFFLE(data)
4      for i = 1 to ⌈n/K⌉
5          BATCH-GRADIENT-UPDATE(NN, data[(i − 1)K : iK])
```

See note on the ceiling[1] function, for the case when n/K is not an integer.

6.7.2 Adaptive step-size

Picking a value for η is difficult and time-consuming. If it's too small, then convergence is slow and if it's too large, then we risk divergence or slow convergence due to oscillation. This problem is even more pronounced in stochastic or mini-batch mode, because we know we need to decrease the step size for the formal guarantees to hold.

It's also true that, within a single neural network, we may well want to have different step sizes. As our networks become *deep* (with increasing numbers of layers) we can find that magnitude of the gradient of the loss with respect the weights in the last layer, $\partial\text{loss}/\partial W_L$, may be substantially different from the gradient of the loss with respect to the weights in the first layer $\partial\text{loss}/\partial W_1$. If you look carefully at Eq. 6.6, you can see that the output gradient is multiplied by all the weight matrices of the network and is "fed back" through all the derivatives of all the activation functions. This can lead to a problem of *exploding* or *vanishing* gradients, in which the back-propagated gradient is much too big or small to be used in an update rule with the same step size.

[1] In line 4 of the algorithm above, $\lceil \cdot \rceil$ is known as the *ceiling* function; it returns the smallest integer greater than or equal to its input. E.g., $\lceil 2.5 \rceil = 3$ and $\lceil 3 \rceil = 3$.

So, we can consider having an independent step-size parameter *for each weight*, and updating it based on a local view of how the gradient updates have been going. Some common strategies for this include *momentum* ("averaging" recent gradient updates), *Adadelta* (take larger steps in parts of the space where $J(W)$ is nearly flat), and *Adam* (which combines these two previous ideas). Details of these approaches are described in Appendix B.0.1.

This section is very strongly influenced by Sebastian Ruder's excellent blog posts on the topic: ruder.io/optimizing-gradient-descent

6.8 Regularization

So far, we have only considered optimizing loss on the training data as our objective for neural network training. But, as we have discussed before, there is a risk of overfitting if we do this. The pragmatic fact is that, in current deep neural networks, which tend to be very large and to be trained with a large amount of data, overfitting is not a huge problem. This runs counter to our current theoretical understanding and the study of this question is a hot area of research. Nonetheless, there are several strategies for regularizing a neural network, and they can sometimes be important.

6.8.1 Methods related to ridge regression

One group of strategies can, interestingly, be shown to have similar effects to each other: early stopping, weight decay, and adding noise to the training data.

Result is due to Bishop, described in his textbook and here doi.org/10.1162/neco.1995.7.1.108.

Early stopping is the easiest to implement and is in fairly common use. The idea is to train on your training set, but at every *epoch* (a pass through the whole training set, or possibly more frequently), evaluate the loss of the current W on a *validation set*. It will generally be the case that the loss on the training set goes down fairly consistently with each iteration, the loss on the validation set will initially decrease, but then begin to increase again. Once you see that the validation loss is systematically increasing, you can stop training and return the weights that had the lowest validation error.

Another common strategy is to simply penalize the norm of all the weights, as we did in ridge regression. This method is known as *weight decay*, because when we take the gradient of the objective

$$J(W) = \sum_{i=1}^{n} \mathcal{L}(\text{NN}(x^{(i)}), y^{(i)}; W) + \lambda \|W\|^2$$

we end up with an update of the form

$$\begin{aligned} W_t &= W_{t-1} - \eta \left(\left(\nabla_W \mathcal{L}(\text{NN}(x^{(i)}), y^{(i)}; W_{t-1}) \right) + 2\lambda W_{t-1} \right) \\ &= W_{t-1}(1 - 2\lambda\eta) - \eta \left(\nabla_W \mathcal{L}(\text{NN}(x^{(i)}), y^{(i)}; W_{t-1}) \right) \quad . \end{aligned}$$

This rule has the form of first "decaying" W_{t-1} by a factor of $(1 - 2\lambda\eta)$ and then taking a gradient step.

Finally, the same effect can be achieved by perturbing the $x^{(i)}$ values of the training data by adding a small amount of zero-mean normally distributed noise before each gradient computation. It makes intuitive sense that it would be more difficult for the network to overfit to particular training data if they are changed slightly on each training step.

6.8.2 Dropout

Dropout is a regularization method that was designed to work with deep neural networks. The idea behind it is, rather than perturbing the data every time we train, we'll perturb the network! We'll do this by randomly, on each training step, selecting a set of units in each layer and prohibiting them from participating. Thus, all of the units will have to take a

kind of "collective" responsibility for getting the answer right, and will not be able to rely on any small subset of the weights to do all the necessary computation. This tends also to make the network more robust to data perturbations.

During the training phase, for each training example, for each unit, randomly with probability p temporarily set $a_j^\ell = 0$. There will be no contribution to the output and no gradient update for the associated unit.

Study Question: Be sure you understand why, when using SGD, setting an activation value to 0 will cause that unit's weights not to be updated on that iteration.

When we are done training and want to use the network to make predictions, we multiply all weights by p to achieve the same average activation levels.

Implementing dropout is easy! In the forward pass during training, we let

$$a^\ell = f(z^\ell) * d^\ell$$

where $*$ denotes component-wise product and d^ℓ is a vector of 0's and 1's drawn randomly with probability p. The backwards pass depends on a^ℓ, so we do not need to make any further changes to the algorithm.

It is common to set p to 0.5, but this is something one might experiment with to get good results on your problem and data.

6.8.3 Batch normalization

Another strategy that seems to help with regularization and robustness in training is *batch normalization*. It was originally developed to address a problem of *covariate shift*: that is, if you consider the second layer of a two-layer neural network, the distribution of its input values is changing over time as the first layer's weights change. Learning when the input distribution is changing is extra difficult: you have to change your weights to improve your predictions, but also just to compensate for a change in your inputs (imagine, for instance, that the magnitude of the inputs to your layer is increasing over time—then your weights will have to decrease, just to keep your predictions the same).

For more details see `arxiv.org/abs/1502.03167`.

So, when training with mini-batches, the idea is to *standardize* the input values for each mini-batch, just in the way that we did it in Section 5.3.3 of Chapter 5, subtracting off the mean and dividing by the standard deviation of each input dimension. This means that the scale of the inputs to each layer remains the same, no matter how the weights in previous layers change. However, this somewhat complicates matters, because the computation of the weight updates will need to take into account that we are performing this transformation. In the modular view, batch normalization can be seen as a module that is applied to z^l, interposed after the product with W^l and before input to f^l.

We follow here the suggestion from the original paper of applying batch normalization before the activation function. Since then it has been shown that, in some cases, applying it after works a bit better. But there aren't any definite findings on which works better and when.

Although batch-norm was originally justified based on the problem of covariate shift, it's not clear that that is actually why it seems to improve performance. Batch normalization can also end up having a regularizing effect for similar reasons that adding noise and dropout do: each mini-batch of data ends up being mildly perturbed, which prevents the network from exploiting very particular values of the data points. For those interested, the equations for batch normalization, including a derivation of the forward pass and backward pass, are described in Appendix B.0.2.