CHAPTER 8

Neural Networks

Unless you live under a rock with no internet access, you've 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 completely well equipped to understand neural networks in detail.

This, 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 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, which 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) (regularization of high-dimensional hypotheses by seeking to maximize the margin) and kernel methods (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

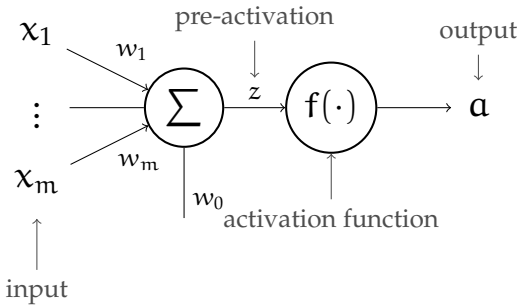
of data in very complex domains.

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

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 ($f(x) = x$), 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.

The function represented by the neuron is expressed as:

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

Before thinking about a whole network, we can consider how to train a single unit. Given a loss function $L(\textit{guess}, \textit{actual})$ 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

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

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

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

We have already studied two special cases of the neuron: linear logistic classifiers (LLC) 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 $L(g, a) = (g - a)^2$. Derive a gradient descent update for w and w_0.

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

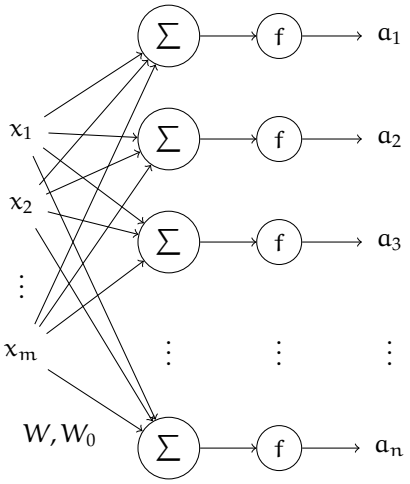
neurons; the inputs of each neuron might be elements of x and/or outputs of other neurons. The outputs are generated by n *output units*.

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 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.

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, the inputs to each unit in the layer are the same (i.e. $x_1, x_2, \ldots x_m$ in this case). 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

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

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

What can we do with a single layer? We have already seen single-layer networks, in the form of linear separators and linear regressors. All we can do with a single layer is make a linear hypothesis. The whole reason for moving to neural networks is to move in the direction of *non-linear* hypotheses. To do this, we will have to consider multiple layers, where we can view the last layer as still being a linear classifier or regressor, but where we interpret the previous layers as learning a non-linear feature transformation $\phi(x)$, rather than having us hand-specify it.

We have used a step or sigmoid function to transform the linear output value for classification, but it's important to be clear that the resulting *separator* is still linear.

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. Let f^l be the activation function of layer l. Then, the pre-activation outputs are the $n^l \times 1$ vector

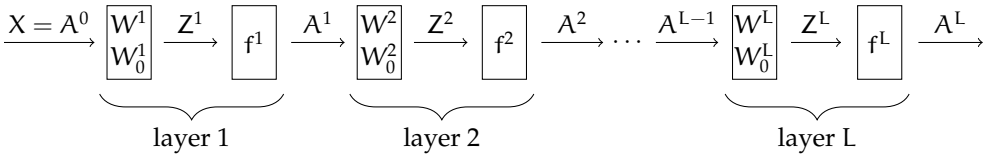
$$Z^l = {W^l}^T A^{l-1} + W_0^l$$

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

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

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.

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.



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.

Step function

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

Rectified linear unit

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

Sigmoid function Also known as a *logistic* function, can be interpreted as probability, because for any value of z the output is in $[0, 1]$

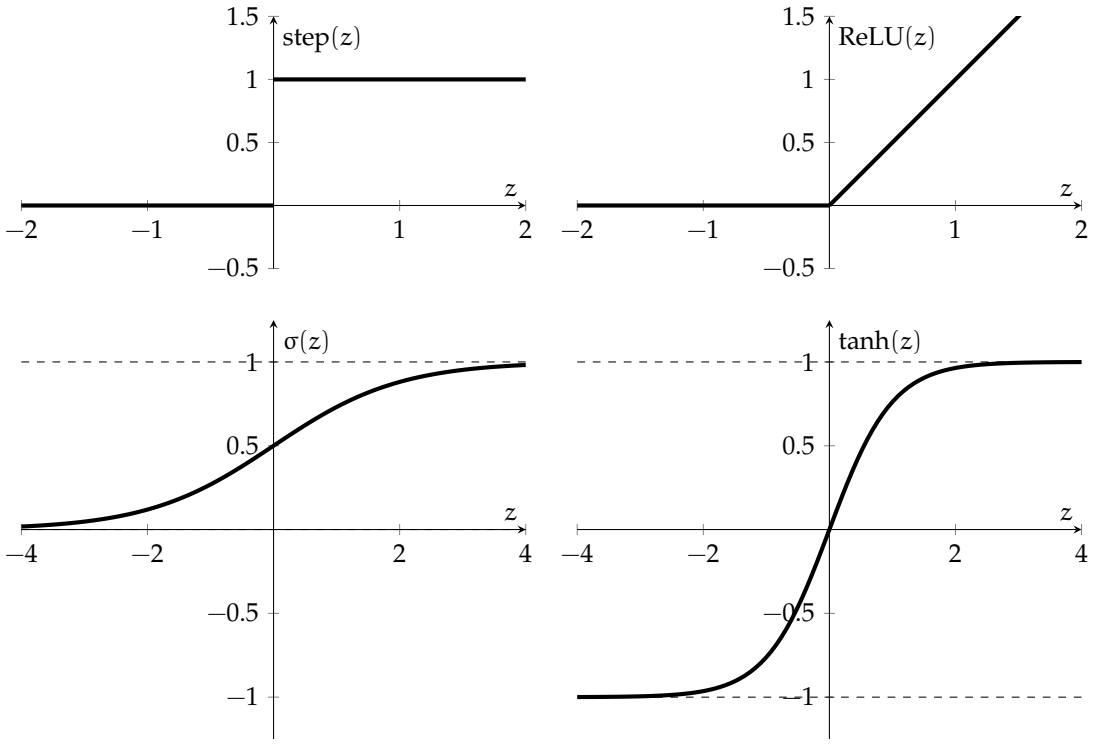
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

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 is discontinuous, we won't be able to use gradient-descent methods to tune the weights in a network with step functions, 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, and sigmoid activations are common for the output for binary classification and softmax for multi-class classification (see section 4 for an explanation).

4 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 2 of chapter 6) 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 4 of chapter 6).

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 $\nabla_W \text{Loss}(\text{NN}(x; W), y)$, 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.

First, let's see how the loss depends on the weights in the final layer, W^L. Remembering that our output is A^L, and using the shorthand loss to stand for $\text{Loss}((\text{NN}(x; W), y)$ which is equal to $\text{Loss}(A^L, y)$, and finally that $A^L = f^L(Z^L)$ and $Z^L = {W^L}^T A^{L-1}$, we can use the chain rule:

$$\frac{\partial \text{loss}}{\partial W^L} = \underbrace{\frac{\partial \text{loss}}{\partial A^L}}_{\text{depends on loss function}} \cdot \underbrace{\frac{\partial A^L}{\partial Z^L}}_{f^{L\prime}} \cdot \underbrace{\frac{\partial Z^L}{\partial W^L}}_{A^{L-1}} .$$

To actually get the dimensions to match, we need to write this a bit more carefully, and note that it is true for any l, including $l = L$:

$$\underbrace{\frac{\partial \text{loss}}{\partial W^l}}_{m^l \times n^l} = \underbrace{A^{l-1}}_{m^l \times 1} \underbrace{\left(\frac{\partial \text{loss}}{\partial Z^l}\right)^T}_{1 \times n^l} \tag{8.1}$$

It might reasonably bother you that $\partial Z^L / \partial W^L = A^{L-1}$. We're somehow thinking about the derivative of a vector with respect to a matrix, which seems like it might need to be a three-dimensional thing. But note that $\partial Z^L / \partial W^L$ is really $(\partial {W^L}^T A^{L-1}) / \partial W^L$ and it seems okay in at least an informal sense that it's A^{L-1}.

Yay! So, in order to find the gradient of the loss with respect to the weights in the other layers of the network, we just need to be able to find $\partial \text{loss} / \partial Z^l$.

If we repeatedly apply the chain rule, we get this expression for the gradient of the loss with respect to the pre-activation in the first layer:

$$\frac{\partial \text{loss}}{\partial Z^1} = \underbrace{\underbrace{\frac{\partial \text{loss}}{\partial A^L} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \cdots \cdot \frac{\partial A^2}{\partial Z^2}}_{\partial \text{loss}/\partial Z^2} \cdot \frac{\partial Z^2}{\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1}}_{\partial \text{loss}/\partial A^1} \quad . \tag{8.2}$$

This derivation was informal, to show you the general structure of the computation. In fact, to get the dimensions to all work out, we just have to write it backwards! Let's first understand more about these quantities:

- $\partial \text{loss}/\partial A^L$ is $n^L \times 1$ and depends on the particular loss function you are using.
- $\partial Z^l/\partial A^{l-1}$ is $m^l \times n^l$ and is just W^l (you can verify this by computing a single entry $\partial Z^l_i/\partial A^{l-1}_j$).
- $\partial A^l/\partial Z^l$ is $n^l \times n^l$. It's a little tricky to think about. Each element $a^l_i = f^l(z^l_i)$. This means that $\partial a^l_i/\partial z^l_j = 0$ whenever $i \neq j$. So, the off-diagonal elements of $\partial A^l/\partial Z^l$ are all 0, and the diagonal elements are $\partial a^l_i/\partial z^l_j = f^{l\prime}(z^l_j)$.

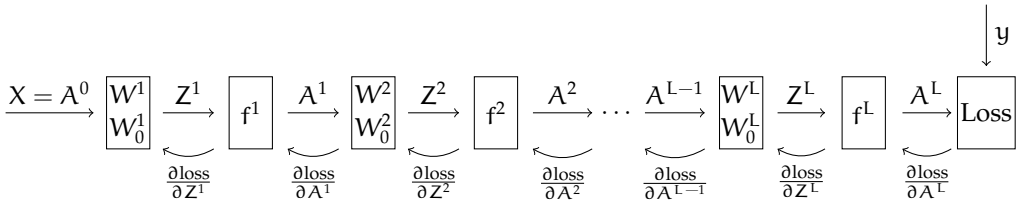
Now, we can rewrite equation 8.2 so that the quantities match up as

$$\frac{\partial \text{loss}}{\partial Z^l} = \frac{\partial A^l}{\partial Z^l} \cdot W^{l+1} \cdot \frac{\partial A^{l+1}}{\partial Z^{l+1}} \cdot \ldots W^{L-1} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot W^L \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{A^L} \quad . \tag{8.3}$$

Using equation 8.3 to compute $\partial \text{loss}/\partial Z^l$ combined with equation 8.1, lets us find the gradient of the loss with respect to any of the weight matrices.

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

This general process 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 on this example. 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.



I like to think of this as "blame propagation". You can think of loss as how mad we are about the prediction that the network 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: $u \rightarrow v$
- backward: $u, v, \partial L/\partial v \rightarrow \partial L/\partial u$
- weight grad: $u, \partial L/\partial v \rightarrow \partial L/\partial W$ 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.

5 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($\mathcal{D}_n, T, L, (m^1, \ldots, m^L), (f^1, \ldots, f^L)$)

```
1  for l = 1 to L
2      W^l_ij ~ Gaussian(0, 1/m^l)
3      W^l_0j ~ Gaussian(0, 1)
4  for t = 1 to T
5      i = random sample from {1, ..., n}
6      A^0 = x^(i)
7      // forward pass to compute the output A^L
8      for l = 1 to L
9          Z^l = W^lT A^(l-1) + W^l_0
10         A^l = f^l(Z^l)
11     loss = Loss(A^L, y^(i))
12     for l = L to 1:
13         // error back-propagation
14         ∂loss/∂A^l = if l < L then ∂loss/∂Z^(l+1) · ∂Z^(l+1)/∂A^l else ∂loss/∂A^L
15         ∂loss/∂Z^l = ∂loss/∂A^l · ∂A^l/∂Z^l
16         // compute gradient with respect to weights
17         ∂loss/∂W^l = ∂loss/∂Z^l · ∂Z^l/∂W^l
18         ∂loss/∂W^l_0 = ∂loss/∂Z^l · ∂Z^l/∂W^l_0
19         // stochastic gradient descent update
20         W^l = W^l − η(t) · ∂loss/∂W^l
21         W^l_0 = W^l_0 − η(t) · ∂loss/∂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")

$$W^l_{ij} \sim \text{Gaussian}\left(0, \frac{1}{m^l}\right) \quad .$$

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

$$\frac{\partial \text{loss}}{\partial A^L} \quad \text{and} \quad \frac{\partial A^L}{\partial Z^L} \ .$$

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 Loss functions and activation functions

Different loss functions make different assumptions about the range of inputs 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
squared	linear
hinge	linear
NLL	sigmoid
NLLM	softmax

6.1 Two-class classification and log likelihood

For classification, the natural loss function is 0-1 loss, but we have already discussed the fact that it's very inconvenient for gradient-based learning because its derivative is discontinuous.

We have also explored *negative log likelihood* (NLL) in chapter 5. It is nice and smooth, and extends nicely to multiple classes as we will see below.

Hinge loss gives us another way, for binary classification problems, to make a smoother objective, penalizing the *margin*s of the labeled points relative to the separator. The hinge loss is defined to be

$$\mathcal{L}_h(\text{guess}, \text{actual}) = \max(1 - \text{guess} \cdot \text{actual}, 0) \ ,$$

when actual $\in \{+1, -1\}$. It has the property that, if the sign of guess is the same as the sign of actual and the magnitude of guess is greater than 1, then the loss is 0.

It is trying to enforce not only that the guess have the correct sign, but also that it should be some distance away from the separator. Using hinge loss, together with a squared-norm regularizer, actually forces the learning process to try to find a separator that has the maximum *margin* relative to the data set. This optimization set-up is called a *support vector machine*, and was popular before the renaissance of neural networks and gradient descent, because it has a quadratic form that makes it particularly easy to optimize.

6.2 Multi-class classification and log likelihood

We can extend the idea of NLL directly to multi-class classification with K classes, where the training label is represented with the one-hot vector $y = \begin{bmatrix} y_1, \ldots, y_K \end{bmatrix}^T$, where $y_k = 1$ if the example is of class k. Assume that our network uses *softmax* as the activation function

in the last layer, so that the output is $a = \left[a_1, \ldots, a_K\right]^T$, which represents a probability distribution over the K possible classes. Then, the probability that our network predicts the correct class for this example is $\prod_{k=1}^{K} a_k^{y_k}$ and the log of the probability that it is correct is $\sum_{k=1}^{K} y_k \log a_k$, so

$$\mathcal{L}_{\text{nllm}}(\text{guess}, \text{actual}) = -\sum_{k=1}^{K} \text{actual}_k \cdot \log(\text{guess}_k) \ .$$

We'll call this NLLM for *negative log likelihood multiclass.*

Study Question: Show that L_{nllm} for $K = 2$ is the same as L_{nll}.

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.

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.

When we perform *batch* gradient descent, we use the update rule

$$W := W - \eta \nabla_W J(W) \ ,$$

which is equivalent to

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

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:

$$W := W - \eta \nabla_W \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ .$$

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

$$W := W - \eta \sum_{i=1}^{k} \nabla_W \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ .$$

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 shufffling 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])
```

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 $\partial \text{loss}/\partial W_L$. If you look carefully at equation 8.3, 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.

So, we'll 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.

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

7.2.1 Running averages

We'll start by looking at the notion of a *running average*. It's a computational strategy for estimating a possibly weighted average of a sequence of data. Let our data sequence be $a_1, a_2, \ldots$; then we define a sequence of running average values, $A_0, A_1, A_2, \ldots$ using the equations

$$\begin{aligned} A_0 &= 0 \\ A_t &= \gamma_t A_{t-1} + (1-\gamma_t) a_t \end{aligned}$$

where $\gamma_t \in (0,1)$. If γ_t is a constant, then this is a *moving* average, in which

$$\begin{aligned} A_T &= \gamma A_{T-1} + (1-\gamma) a_T \\ &= \gamma(A_{T-2} + (1-\gamma) a_{T-1}) + (1-\gamma) a_T \\ &= \sum_{t=0}^{T} \gamma^{T-t} (1-\gamma) a_t \end{aligned}$$

So, you can see that inputs a_t closer to the end of the sequence have more effect on A_t than early inputs.

If, instead, we set $\gamma_t = (t-1)/t$, then we get the actual average.

Study Question: Prove to yourself that the previous assertion holds.

7.2.2 Momentum

Now, we can use methods that are a bit like running averages to describe strategies for computing η. The simplest method is *momentum*, in which we try to "average" recent gradient updates, so that if they have been bouncing back and forth in some direction, we take out that component of the motion. For momentum, we have

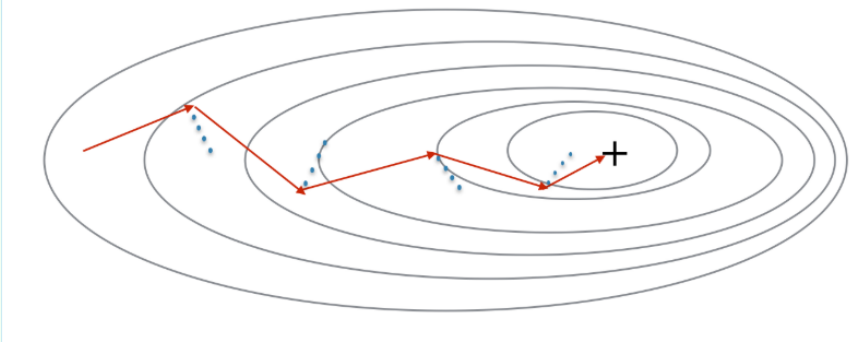
$$\begin{aligned} V_0 &= 0 \\ V_t &= \gamma V_{t-1} + \eta \nabla_W J(W_{t-1}) \\ W_t &= W_{t-1} - V_t \end{aligned}$$

This doesn't quite look like an adaptive step size. But what we can see is that, if we let $\eta = \eta'(1-\gamma)$, then the rule looks exactly like doing an update with step size η' on a moving average of the gradients with parameter γ:

$$\begin{aligned} V_0 &= 0 \\ M_t &= \gamma M_{t-1} + (1-\gamma)\nabla_W J(W_{t-1}) \\ W_t &= W_{t-1} - \eta' M_t \end{aligned}$$

Study Question: Prove to yourself that these formulations are equivalent.

We will find that V_t will be bigger in dimensions that consistently have the same sign for ∇_θ and smaller for those that don't. Of course we now have *two* parameters to set (η and γ), but the hope is that the algorithm will perform better overall, so it will be worth trying to find good values for them. Often γ is set to be something like 0.9.



The red arrows show the update after one step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient with respect to the mini-batch at each step. Momentum smooths the path taken towards the local minimum and leads to faster convergence.

Study Question: If you set $\gamma = 0.1$, would momentum have more of an effect or less of an effect than if you set it to 0.9?

7.2.3 Adadelta

Another useful idea is this: we would like to take larger steps in parts of the space where $J(W)$ is nearly flat (because there's no risk of taking too big a step due to the gradient being large) and smaller steps when it is steep. We'll apply this idea to each weight independently, and end up with a method called *adadelta*, which is a variant on *adagrad* (for adaptive gradient). Even though our weights are indexed by layer, input unit and output unit, for simplicity here, just let W_j be any weight in the network (we will do the same thing for all of them).

$$\begin{aligned} g_{t,j} &= \nabla_W J(W_{t-1})_j \\ G_{t,j} &= \gamma G_{t-1,j} + (1-\gamma) g_{t,j}^2 \\ W_{t,j} &= W_{t-1,j} - \frac{\eta}{\sqrt{G_{t,j} + \epsilon}} g_{t,j} \end{aligned}$$

The sequence $G_{t,j}$ is a moving average of the square of the jth component of the gradient. We square it in order to be insensitive to the sign—we want to know whether the magnitude is big or small. Then, we perform a gradient update to weight j, but divide the step size by $\sqrt{G_{t,j} + \epsilon}$, which is larger when the surface is steeper in direction j at point W_{t-1} in weight space; this means that the step size will be smaller when it's steep and larger when it's flat.

7.2.4 Adam

Adam has become the default method of managing step sizes neural networks. It combines the ideas of momentum and and adadelta. We start by writing moving averages of the gradient and squared gradient, which reflect estimates of the mean and variance of the gradient for weight j:

Although, interestingly, it may actually violate the convergence conditions of SGD: arxiv.org/abs/1705.08292

$$\begin{aligned} g_{t,j} &= \nabla_W J(W_{t-1})_j \\ m_{t,j} &= B_1 m_{t-1,j} + (1-B_1) g_{t,j} \\ v_{t,j} &= B_2 v_{t-1,j} + (1-B_2) g_{t,j}^2 \quad . \end{aligned}$$

A problem with these estimates is that, if we initialize $m_0 = v_0 = 0$, they will always be biased (slightly too small). So we will correct for that bias by defining

$$\begin{aligned} \hat{m}_{t,j} &= \frac{m_{t,j}}{1 - B_1^t} \\ \hat{v}_{t,j} &= \frac{v_{t,j}}{1 - B_2^t} \\ W_{t,j} &= W_{t-1,j} - \frac{\eta}{\sqrt{\hat{v}_{t,j} + \epsilon}} \hat{m}_{t,j} \quad . \end{aligned}$$

Note that B_1^t is B_1 raised to the power t, and likewise for B_2^t. To justify these corrections, note that if we were to expand $m_{t,j}$ in terms of $m_{0,j}$ and $g_{0,j}, g_{1,j}, \ldots, g_{t,j}$ the coefficients would sum to 1. However, the coefficient behind $m_{0,j}$ is B_1^t and since $m_{0,j} = 0$, the sum of coefficients of non-zero terms is $1 - B_1^t$, hence the correction. The same justification holds for $v_{t,j}$.

Now, our update for weight j has a step size that takes the steepness into account, as in adadelta, but also tends to move in the same direction, as in momentum. The authors of this method propose setting $B_1 = 0.9, B_2 = 0.999, \epsilon = 10^{-8}$. Although we now have even more parameters, Adam is not highly sensitive to their values (small changes do not have a huge effect on the result).

Study Question: Define $\hat{m}_j$ directly as a moving average of $g_{t,j}$. What is the decay (γ parameter)?

Even though we now have a step-size for each weight, and we have to update various quantities on each iteration of gradient descent, it's relatively easy to implement by maintaining a matrix for each quantity ($m_t^\ell, v_t^\ell, g_t^\ell, g_t^{2\,\ell}$) in each layer of the network.

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.

8.1 Methods related to ridge regression

One group of strategies can, interestingly, be shown to have similar effects: 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* (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

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

we end up with an update of the form

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

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

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.

8.3 Batch Normalization

A more modern alternative to dropout, which tends to achieve better performance, 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 of chapter 4, 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.

Batch normalization ends 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.

Let's think of the batch-norm layer as taking z^l as input and producing an output $\widehat{Z}$ as output. But now, instead of thinking of Z^l as an $n^l \times 1$ vector, we have to explicitly think about handling a mini-batch of data of size K, all at once, so Z^l will be $n^l \times K$, and so will the output $\widehat{Z}^l$.

Our first step will be to compute the *batchwise* mean and standard deviation. Let μ^l be the $n^l \times 1$ vector where

$$\mu_i^l = \frac{1}{K} \sum_{j=1}^{K} Z_{ij}^l \quad ,$$

and let σ^l be the $n^l \times 1$ vector where

$$\sigma_i^l = \sqrt{\frac{1}{K}\sum_{j=1}^{K}(Z_{ij}^l - \mu_i)^2} \ .$$

The basic normalized version of our data would be a matrix, element (i, j) of which is

$$\overline{Z}_{ij}^l = \frac{Z_{ij}^l - \mu_i^l}{\sigma_i^l + \epsilon} \ ,$$

where ϵ is a very small constant to guard against division by zero. However, if we let these be our $\widehat{Z}$ values, we really are forcing something too strong on our data—our goal was to normalize across the data batch, but not necessarily force the output values to have exactly mean 0 and standard deviation 1. So, we will give the layer the "opportunity" to shift and scale the outputs by adding new weights to the layer. These weights are G^l and B^l, each of which is an $n^l \times 1$ vector. Using the weights, we define the final output to be

$$\widehat{Z}_{ij}^l = G_i^l \overline{Z}_{ij}^l + B_i^l \ .$$

That's the forward pass. Whew!

Now, for the backward pass, we have to do two things: given $\partial L/\partial \widehat{Z}^l$,

- Compute $\partial L/\partial Z^l$ for back-propagation, and
- Compute $\partial L/\partial G^l$ and $\partial L/\partial B^l$ for gradient updates of the weights in this layer.

Schematically

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial \widehat{Z}}\frac{\partial \widehat{Z}}{\partial B} \ .$$

It's hard to think about these derivatives in matrix terms, so we'll see how it works for the components. B_i contributes to $\widehat{Z}_{ij}$ for all data points j in the batch. So

$$\begin{aligned}\frac{\partial L}{\partial B_i} &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\frac{\partial \widehat{Z}_{ij}}{\partial B_i} \\ &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}} \ ,\end{aligned}$$

Similarly, G_i contributes to $\widehat{Z}_{ij}$ for all data points j in the batch. So

$$\begin{aligned}\frac{\partial L}{\partial G_i} &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\frac{\partial \widehat{Z}_{ij}}{\partial G_i} \\ &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\overline{Z}_{ij} \ .\end{aligned}$$

Now, let's figure out how to do backprop. We can start schematically:

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial \widehat{Z}}\frac{\partial \widehat{Z}}{\partial Z} \ .$$

And because dependencies only exist across the batch, but not across the unit outputs,

$$\frac{\partial L}{\partial Z_{ij}} = \sum_{k=1}^{K}\frac{\partial L}{\partial \widehat{Z}_{ik}}\frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}} \ .$$

The next step is to note that

$$\begin{aligned}\frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}} &= \frac{\partial \widehat{Z}_{ik}}{\partial \overline{Z}_{ik}} \frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} \\ &= G_i \frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}}\end{aligned}$$

And now that

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \left(\delta_{jk} - \frac{\partial \mu_i}{\partial Z_{ij}}\right) \frac{1}{\sigma_i} - \frac{Z_{ik} - \mu_i}{\sigma_i^2} \frac{\partial \sigma_i}{\partial Z_{ij}} \quad ,$$

where $\delta_{jk} = 1$ if $j = k$ and $\delta_{jk} = 0$ otherwise. Getting close! We need two more small parts:

$$\begin{aligned}\frac{\partial \mu_i}{\partial Z_{ij}} &= \frac{1}{K} \\ \frac{\partial \sigma_i}{\partial Z_{ij}} &= \frac{Z_{ij} - \mu_i}{K\sigma_i}\end{aligned}$$

Putting the whole crazy thing together, we get

$$\frac{\partial L}{\partial Z_{ij}} = \sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}} G_i \frac{1}{K\sigma_i} \left(\delta_{jk} K - 1 - \frac{(Z_{ik} - \mu_i)(Z_{ij} - \mu_i)}{\sigma_i^2}\right)$$