Feedforward Neural Networks

Arthur Tilley

1 Basic Definitions

- Our continuing mission (in supervised ML): Given a set of data $X = \{(\bar{x}^{(i)}, \bar{y}^{(i)})\}_{1 \leq i \leq M}$ for $\bar{x} \in \mathbb{R}^n$ and $\bar{y} \in \mathbb{R}^k$, learn a function $f : \mathbb{R}^n \to \mathbb{R}^k$ that approximates this data as well as possible.
- Before, we restricted f to live in various simple families of functions (recall linear and logistic regression). While these models were simple to train, they had low *capacity* (chapter 3) and had a hard time learning complex functions.
- *Neural Networks* (NN) and particularly the *Feedforward Neural Networks* (FFNN) of this chapter expand the families of functions in linear regression and logistic regression to a much more general family have much greater capacity.
- FFNN are called *networks* because they are usually defined by composing together many simpler functions

$$f(\bar{x}) = f_d \circ f_{d-1} \circ \dots f_2 \circ f_1(\bar{x}) = f_d(f_{d-1}(\dots(f_2(f_1(\bar{x}))\dots)$$

  where each $f_i, (1 \leq i \leq d)$ is some function $f : \mathbb{R}^{r_{i-1}} \to \mathbb{R}^{r_i}$ for $r_i \in \mathbb{N}$.
- These functions are frequently referred to as *layers* of the NN. We call f_d the *output layer*. Layers other than the output layer are called *hidden layers*. We can say that layer i has *width* r_i and that the model as a whole has *depth* d. We say that we are doing *deep learning* when we consider our d to be large.
- Since each f_k will typically rely on some set of parameters $\bar{\theta}_k$ we may sometimes write the above expression as

$$f(\bar{x}; \bar{\theta}) = f_{\bar{\theta}}(\bar{x}) = f_{d,\bar{\theta}_d}(f_{d-1,\bar{\theta}_{d-1}}(\dots(f_{2,\bar{\theta}_2}(f_{1,\bar{\theta}_1}(\bar{x}))\dots)$$

- Our goal is to find useful combinations of layers and to find ways of learning values $\hat{\bar{\theta}}$ for the a whole family of parameters $\bar{\theta}$ such that the resulting function $f(\bar{x}; \hat{\theta})$ performs well on our data X.

- Since each layer f_i will generally be a vector valued function $f : \mathbb{R}^{r_{i-1}} \to \mathbb{R}^{r_i}$ it may be represented as r_i different, scalar-valued functions $(f_i)_k : \mathbb{R}^{r_{i-1}} \to \mathbb{R}$. These individual scalar-valued functions are sometimes called *units* or *neurons*. Nevermind why.

2 A simple function that cannot be learned with linear models

- To guide us in our investigation of complex models, it will be helpful to consider a simple function that cannot be approximated with linear regression or logistic regression. The textbook example of this is the function

$$XOR : \{\langle 1,1\rangle, \langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\} \to \{0,1\}$$

  defined by

$$XOR(x,y) = 1 \text{ if } x \neq y \text{ else } 0$$

- Our goal will be to show that linear models such as linear regression and logistic regression are unable to even approximate XOR as a function $f : \mathbb{R}^2 \to \mathbb{R}$. Of course there are simpler functions (for instance of the form $f : \mathbb{R} \to \mathbb{R}$) that cannot be learned with linear models, but XOR is textbook because it is a commonly used boolean function.

- You could, as the book does, show this failure by providing a perfect training set and showing that even then, for example, the normal equations for linear regression fail to give satisfactory weights.

- But it's probably quicker to simply notice that any linear or logistic regression model requires that we be able to separate the 1s from the 0s in $\mathbb{R}^2$ with a line.

- In particular, for linear regression, with $h(\bar{x}) = \bar{w}^T\bar{x} + b$, either $\bar{w} = \bar{0}$, in which case h cannot distinguish the 1s and 0s at all, or $\bar{w} \neq \bar{0}$ and setting $h(\bar{x}) = .5$ defines line in $\mathbb{R}^2$. One side of this line will be points that h maps to a value greater than .5 and on the other side of the line we have values that h maps less than .5. Since we cannot fit just the 1s on either side of any line in $\mathbb{R}^2$, we can't get a linear regression model that is closer to 1 on just the 1s and closer to 0 on just the 0s.

- Similarly with logistic regression, $h(\bar{x}) = \sigma(\bar{w}^T\bar{x} + b) = .5$ is equivalent to $\bar{w}^T\bar{x} + b = 0$, and again, one side of this line will be points that h maps to a value greater than .5 and on the other side of the line we have values that h maps less than .5. Again, since the 1s and 0s are not linearly separable, we cannot hope to find a logistic regression model that is closer to 1 than 0 on all 1s and closer to 0 than 1 on all 0s.

- Now let's see if we can do better with by adding one hidden layer with a function
$$f(\bar{x}) = f_2(f_1(\bar{x}))$$
where
$$f_1 : \mathbb{R}^2 \to \mathbb{R}^w$$
and
$$f_2 : \mathbb{R}^w \to \mathbb{R}$$
for some positive integer w.

- It makes sense to start as simple as possible, but notice that using two consecutive linear (or affine) layers will not get us anywhere. Can you see why?

  (Pause for group participation)

- The reason is that, in general, if $f(\bar{x}) = U\bar{x} + \bar{a}$ is any linear (or affine) function $f : \mathbb{R}^s \to \mathbb{R}^t$, and if $g(\bar{y}) = V\bar{y} + \bar{b}$ is any linear (or affine) function from $g : \mathbb{R}^t \to \mathbb{R}^r$, then $g(f(\bar{x})) = V(U\bar{x} + \bar{a}) + \bar{b} = VU\bar{x} + (V\bar{a} + \bar{b})$ is an affine function from $\mathbb{R}^s$ to $\mathbb{R}^r$. It follows that two consecutive, purely affine layers in a FFNN are not able to define any function that couldn't be defined with one affine layer.[1]

- Since we've already shown that linear models fail to correctly model XOR, it follows that two (or any number) of consecutive affine layers will also fail.

- To remedy this we need to add a non-linearity somewhere. A very simple non-linear function will do.
  Define the $ReLU$ function $g : \mathbb{R}^n \to \mathbb{R}^n$ (for any n) componentwise by
$$g(\bar{z})_k = max(0, z_k)$$
  or expressed via componentwise max
$$g(\bar{z}) = max(\bar{0}, \bar{z})$$

- In particular we will let
$$f(\bar{x}) = f_2(f_1(\bar{x}))$$
  as before where $f_1 : \mathbb{R}^2 \to \mathbb{R}^w$ and $f_2 : \mathbb{R}^w \to \mathbb{R}$. In addition we set $w = 2$, and let $f_2(\bar{y}) = \bar{w}^T\bar{y} + b$ be an ordinary affine output layer. But

[1] It will sometimes still be advantageous to allow two consecutive affine layers, simply for computational efficiency. For example if we had a layer given by multiplication by the $m \times n$ matrix A, where m, n are large but A was low rank, then we might replace A by BC where B is $m \times p$ and C is $p \times n$ for some small p.

instead of letting $f_1(\bar{x})$ be a generic affine function $f_1(\bar{x}) = U\bar{x} + \bar{a}$ from $\mathbb{R}^2$ into $\mathbb{R}^2$, we compose this last function with the ReLU, That is, we let $f_1(\bar{x}) = \max(\bar{0}, U\bar{x} + \bar{a})$, and $f_2(\bar{y}) = \bar{w}^T\bar{y} + b$. Altogether (making the function parameters explicit arguments) this is:

$$f(\bar{x}; U, \bar{w}, \bar{a}, b) = \bar{w}^T(\max(\bar{0}, U\bar{x} + \bar{a})) + b$$

- Then, as you can verify, the following weights for f get the XOR function exactly on our dataset X

$$\bar{U} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\bar{a} = \langle 0, -1 \rangle,$$

$$\bar{w} = \langle 1, -2 \rangle,$$

$$\bar{b} = \bar{0}$$

3 Loss Functions

- There will generally be no analytical (closed-form) solution to minimize a cost function $C_f(\bar{\theta})$ for models $f(\bar{x}; \bar{\theta})$ like the FFNNs described above. For this reason we will almost always be using an iterative method of cost-function minimization, and these will almost always require us to be able to compute the gradient $\nabla C_f(\bar{\theta})$ in order to make some sort of update to the parameters $\bar{\theta}$.

- The gradient of a cost function C_f for a general network

$$f(\bar{x}) = f_n(f_{n-1}(\ldots f_2(f_1(\bar{x}))\ldots))$$

with several layers is more complicated than the cost functions of linear models like we examined earlier. It follows that the gradients are more complicated to compute (Hint: Gradients are derivatives: Do you remember the chain rule for finding derivative $\frac{d}{dx}f$ of the composition of two functions $f(x) = g(h(x))$?).

- We won't touch on the method of computing these gradients until the end of the chapter (probably two meetings from now), but a few observations are worth making now to motivate how we design our output layer. Recall that the magnitude $|\nabla C_f|$ is proportional to the rate of change of C_f in the direction of the steepest ascent.

For this reason, if our C_f becomes very flat, then our gradient will shrink, and any updates of our parameters in terms of the gradient will grind to a snail's pace. For this reason, we must be very careful in the choice of our output layer and our cost function to that our cost function does not *saturate*, that is, become very too flat.

- Here's the good news: Although it may take a variety of forms, and although we may have to design our output layer to avoid saturation, there is pretty much only one cost function.

  More specifically, let our model space be

  $$\mathscr{F} = \{Q_f\}_f = \{\{Q_{f,\bar{x}}\}_{\bar{x}}\}_f = \{\{Q(\cdot; f(\bar{x}))\}_{\bar{x}}\}_f$$

  some parameterized family of families of probability distributions where

  $$Q_{f,\bar{x}}(\bar{y}) = Q(\bar{y}; f(\bar{x}))$$

  is the probability of the label $\bar{y}$, given parameters $f(\bar{x})$.

  Furthermore, suppose we have some set of data

  $$D = \{(\bar{x}^{(i)}, \bar{y}^{(i)})\}_{1 \leq i \leq M}$$

  consisting of M observations of *features* $\bar{x}^{(i)}$ paired with *labels* $\bar{y}^{(i)}$.

  Then maximizing data likelihood, that is picking the f which makes D most likely, is equivalent to each of the following:

  1. Minimizing the average cross-entropy

     $$\frac{1}{M} \sum_{i=1}^{M} H(P_i, Q_{f,\bar{x}^{(i)}})$$

     between the model distributions $Q_{f,\bar{x}^{(i)}} = Q(\cdot; f(\bar{x}^{(i)}))$ and the one-hot sample distributions defined as

     $$P_i(\bar{y}) = \mathbf{1}_{\bar{y}^{(i)}}(\bar{y}) = \begin{cases} 1 & \text{if } \bar{y} = \bar{y}_i \\ 0 & \text{if } \bar{y} \neq \bar{y}_i \end{cases}$$

  2. Minimizing the average cross-entropy

     $$E_{\bar{x} \sim P} H(P_{\bar{x}}, Q_{f,\bar{x}}) = \sum_{\bar{x} \in D_{\bar{x}}} P(\bar{x}) H(P_{\bar{x}}, Q_{f,\bar{x}})$$

     where

     (a) $D_{\bar{x}}$ is just the set of all $\bar{x}$ that occur anywhere in the data:

     $$D_{\bar{x}} = \{\bar{x} : \bar{x} = \bar{x}^{(i)} \text{ for some } 1 \leq i \leq M\}$$

(b) $P(\bar{x})$ is the sample distribution of $\bar{x}$ in D:

$$P(\bar{x}) = \frac{\#_D(\bar{x})}{M} = \frac{|\{i : 1 \leq i \leq M \text{ and } \bar{x}^{(i)} = \bar{x}\}|}{M}$$

(c) $P_{\bar{x}}(\bar{y})$ is the sample distribution of $\bar{y}$ *given* $\bar{x}$ in D:

$$P_{\bar{x}}(\bar{y}) := \frac{\#_D(\bar{x}, \bar{y})}{\#_D(\bar{x})} = \frac{|\{i : 1 \leq i \leq M \text{ and } (\bar{x}^{(i)}, \bar{y}^{(i)} = (\bar{x}, \bar{y})\}|}{|\{i : 1 \leq i \leq M \text{ and } \bar{x}^{(i)} = \bar{x}\}|}.$$

Concisely, this is to say that

1. $$argmax_f Pr(D|f) = argmin_f \frac{1}{M} \sum_{i=1}^{M} H(P_i, Q_{f,\bar{x}^{(i)}}), \text{ and}$$

2. $$argmax_f Pr(D|f) = argmin_f E_{\bar{x} \sim P} H(P_{\bar{x}}, Q_{f,\bar{x}}).$$

We relegate the proof of this equivalence to an appendix (also in the team drive).

- These requirements sound more restrictive than they really are. Note that we can almost always phrase our learning problem in terms of computing the parameters of a probability distribution. For example,

  1. In the case where the end goal is to predict a numerical value in $\mathbb{R}$ (such as some stock's price tomorrow) given features $\bar{x}$, we can couch this in terms of computing the means $f(\bar{x}) = \mu(\bar{x})$ of a family of Gaussian distributions $N(price; \mu(\bar{x}), \sigma^2)$ for fixed variance σ^2. (Recall chapter 5.)
  2. In pretty much any classification problem, say between the classes $C_1, \ldots, C_k$, we are attempting to learn the vector-valued parameter function

     $$f(\bar{x}) = \{\mu_i(\bar{x})\}_{1 \leq i \leq k} = \{Pr(class = C_i | X = \bar{x})\}_{1 \leq i \leq k},$$

     which determine a categorical (or multinoulli) distribution (or a Bernoulli distribution in the case where $k = 2$).

- In summary, to maximize data likelihood we may minimize average cross-entropy between the sample distribution and the model distribution. The word "equivalent" is important here: We could continue to observe the principal of maximum likelihood by using any cost function whose minimization is equivalent to that of cross-entropy. Although in fact we will typically use cross-entropy itself.

4 Output Layers

- The form of the cost function will inform our choice of output layer. This is mostly to avoid the saturation of the cost function mentioned in the previous section.

- Below, let us assume that our network

$$f(\bar{x}) = f_n(f_{n-1}(\dots f_2(f_1(\bar{x}))\dots))$$

can be expressed as

$$f(\bar{x}) = F(h(\bar{x})),$$

where $F(\bar{z}) = f_n(\bar{z})$ is the output layer, and $h(\bar{x}) = f_{n-1}(\dots f_2(f_1(\bar{x}))\dots)$ is the composition of all hidden layers.

- Recall our first principles from the previous section: We will view $f(\bar{x})$ as giving parameters for a probability distribution Q, and our cost function will be the cross entropy average cross-entropy

$$-\frac{1}{M}\sum_i \log Q(\bar{y}^i; f(\bar{x}^{(i)})).$$

. Our goal will be to chose an output layer F such that

1. $f = F \circ h$ is able to cover the parameter space we are interested in (problem dependent), and
2. The function $\log Q(\bar{y}^i; f(\bar{x}^{(i)}))$ does not saturate.

- In the example of predicting a scalar numerical value such as stock price, as we saw in the previous chapter, we will typically take the model distributions to be a family of Gaussian distributions, perhaps with fixed variance σ^2 parameterized like

$$Q = N(y; f(\bar{x}), \sigma^2).$$

Here the parameter that is being learned is the mean of the Gaussian for a given input $\bar{x}$, which can be any value in $\mathbb{R}$.
For this reason, our output layer can simply be an affine layer

$$F(\bar{z}) = \bar{w}^T\bar{z} + b.$$

Furthermore, since affine functions don't frequently saturate, then f will not saturate (so long as h doesn't).
Lastly, recall from chapter 5 that minimizing cross-entropy of such a distribution is equivalent to minimizing the mean-squared error, in this case we can take our cost function to be simply

$$MSE = \sum_i \|\bar{w}^T h(\bar{x}) + b - y^{(i)}\|^2.$$

- In the case of classification, our model distribution will typically be a Bernoulli distribution $Bern(y; f(\bar{x}))$ (in the case of binary classification), or, more generally, a categorical (multinoulli) distribution over k possible classes $C_1 \dots C_k$ (if we just call them $1 \dots k$ the notation will work out cleaner below):

$$Q = Cat(y; f_1(\bar{x}), \dots, f_k(\bar{x})).$$

Let's first examine binary classification and then turn to multi-class classification. Here our model distribution is $Bern(y; f(\bar{x}))$ and so we are trying to learn the (single, scalar) Bernoulli parameter

$$\mu(\bar{x}) = Pr(y = 1 | X = x)$$

as a function of $\bar{x}$. Thus our average cross-entropy is

$$-\frac{1}{M} \sum_{i=1}^{M} \log Bern(y^{(i)}; f(\bar{x})) =$$

$$-\frac{1}{M} \sum_{i=1}^{M} \log \left(f(\bar{x}) * \mathbf{1}_{y^{(i)}=1} + (1 - f(\bar{x})) * \mathbf{1}_{y^{(i)}=0} \right).$$

Now we can define our output function F. First we assume that the output of the hidden units $h(\bar{x})$ is one dimensional, ie. a scalar in $\mathbb{R}$ (if this is not already the case we can insert one more affine unit with width one).

Then we let F be the sigmoid function from earlier chapters,

$$F(z) = \sigma(z) = \frac{1}{1 + \exp(-z)}.$$

Note that this function already satisfies the first of the desired properties above since its range is in the (open) unit interval, which is the parameter space for our Bernoulli distribution.

With regard to the second property, notice that the average cross-entropy becomes

$$-\frac{1}{M} \sum_{i} \log \left(\sigma(h(\bar{x}^{(i)})) * \mathbf{1}_{y^{(i)}=1} + (1 - \sigma(h(\bar{x}^{i}))) * \mathbf{1}_{y^{(i)}=0} \right).$$

As we've already seen, $1 - \sigma(x) = \sigma(-z)$ and so we can rewrite this as

$$-\frac{1}{M} \sum_{i} \log \left(\sigma \left((2y^{(i)} - 1) h(\bar{x}^{(i)}) \right) \right).$$

Here we can see the benefit of using the equation for average cross-entropy as our loss, instead of something equivalent like mean squared error: The

sigmoid function by itself does saturate for very positive and very negative inputs, but fortunately the $-\log$ of the cross-entropy will help with this. Indeed we can write the last expression above in terms of the softplus function from Chapter 3 ($\zeta(z) := \log(1+\exp(z)) = -\log(\sigma(-z))$) as follows

$$\frac{1}{M}\sum_i \zeta((1-2y^{(i)})h(\bar{x}^{(i)})).$$

Notice that, since $\zeta(z)$ does not become flat unless z is very negative, the only way for the gradient of this cost function to become small is if either $y = 1$ and the hidden layers are outputting a very positive value or if y = 0 and the hidden layers are outputting a very negative value. So in this case the gradients only shrink away to zero when the function is already doing well.

Let's turn to the multivariate case. Here our model distributions are

$$Cat(y; f_1(\bar{x}), \ldots, f_d(\bar{x})),$$

and the parameters function we are trying to learn are

$$f_k(\bar{x}) = Pr(y = k|\bar{x}).$$

Minimizing average cross-entropy reduces to minimizing

$$-\frac{1}{M}\sum_i \log Cat(y^i; f_1(\bar{x}^{(i)}), \ldots, f_d(\bar{x}^{(i)})) =$$

$$-\frac{1}{M}\sum_i \log f_{y^{(i)}}(\bar{x}^{(i)}).$$

(Here $y^{(i)}$ has value in the set $\{1, 2, \ldots, k\}$ of classes.)
Unlike the binary classification case, here we assume that our hidden layers $h(\bar{x})$ output a vector in $\mathbb{R}^k$ (again, if this is not already true, we can insert one affine layer of width k). The function that is typically chosen for the output layer is known as *softmax* and is defined componentwise by

$$(F(\bar{z}))_j = (softmax(\bar{z}))_j = \frac{\exp(z_j)}{\sum_{i=1}^k \exp(z_i)}.$$

Softmax has a variety of nice properties.

1. Notice that, each component $(softmax(\bar{z}))_k$ is in the interval $[0, 1]$,[2] and furthermore that the individual components sum to 1, so the output is always a proper set of parameters for a categorical distribution.

[2]More specifically, in the interval $(0, 1]$ and if $k \geq 2$ in the interval $(0, 1)$.

2. Softmax plays well with cross entropy since the exponentiation cancels with the logarithm of the cross-entropy. Specifically, writing our model function as

$$f(\bar{x}) = F(h(\bar{x})) = softmax(h(\bar{x})),$$

the average cross-entropy above reduces to

$$-\frac{1}{M}\sum_i \log f_{y^{(i)}}(\bar{x}^{(i)}) =$$

$$-\frac{1}{M}\sum_i \log\left(softmax_{y^{(i)}}(h(\bar{x}^{(i)}))\right) =$$

$$-\frac{1}{M}\sum_i \log\frac{\exp\left(h(\bar{x}^{(i)})_{y^{(i)}}\right)}{\sum_{j=1}^{d}\exp\left(h(\bar{x}^{(i)})_j\right)} =$$

$$-\frac{1}{M}\sum_i \left(h(\bar{x}^{(i)})_{y^{(i)}} - \log\sum_{j=1}^{d}\exp\left(h(\bar{x}^{(i)})_j\right)\right) \approx$$

$$-\frac{1}{M}\sum_i \left(h(\bar{x}^{(i)})_{y^{(i)}} - \max_{j=1}^{d} h(\bar{x}^{(i)})_j\right) =$$

$$\frac{1}{M}\sum_i \left(\max_{j=1}^{d} h(\bar{x}^{(i)})_j - h(\bar{x}^{(i)})_{y^{(i)}}\right)$$

This seems like a reasonable cost function since, for each example $(\bar{x}^{(i}, y^{(i)})$ we are effectively penalizing to the degree to which the correct class $y^{(i)}$ is not the greatest scoring component in the output of the hidden layers $h(\bar{x}^{(i)})$. Furthermore, this error function is not likely to become saturated (have shrinking gradient) as long as h does not.

5 Hidden Units

- Now we turn to the h in $f = F \circ h$.
- We will almost always be taking our hidden layers $h_i(\bar{z}) = g(\mathbf{W}\bar{z} + \bar{b})$ to be affine layers with some sort of activation function g. Thus the choice of activation function is where the variety appears.
- While being able to compute the gradient of our cost function does technically require that our model function be differentiable at any point, a function that has just a small finite number of points at which it is nondiffentiable (e.g. the ReLU) usually doesn't cause problems in practice.
- We generally initialize our biases $\bar{b}$ to some small positive numbers.

- Some activation functions:

  1. The *Generalized ReLU* $g : \mathbb{R}^s \to \mathbb{R}^s$ has a different not-necessarily-zero left-part slope for each component:

     $$g(\bar{z})_i = max(0, z_i) + \alpha_i min(0, z_i).$$

     .
  2. The ordinary ReLU is of course the special case where $\alpha_i = 0$ for all i
  3. The *Absolute Value Rectifier* is the special case where $\alpha_i = -1$ for all i
  4. A *Leaky ReLU* is where $\alpha_i = \alpha$ for all all for some small, fixed α.
  5. We may not treat these slopes α_i as constants, but rather variables that can be learned during training. This is called a *parametric* ReLU or $PReLU$.
  6. A *maxout* function $g : \mathbb{R}^s \to \mathbb{R}^t$ where $s = tk$, is defined componentwise by

     $$g(\bar{z})_i = max_{j \in G_i} z_j$$

     where G_i is the set of k consecutive indices for the inputs $G_i = \{(i-1)k+1, \ldots, ik\}$. Notice that in this case a maxout *unit* $h_i(\bar{z}) = g(\mathbf{W}\bar{z} + \bar{b})$ as a whole can learn to act like a variety of activation functions. For example (graphic1).
  7. More generally (but still restricting ourselves to a scalar-valued activation function), consider any unit $h' = g'(B\bar{x} + \bar{d})$ with $B \in \mathbf{R}^{m \times n}$,$\bar{d} \in \mathbf{R}^m$ and where $g' : \mathbb{R}^m \to \mathbb{R}$ is an arbitrary convex, and piecewise-linear function, expressed generally as

     $$g'(\bar{z}) = \begin{cases} \bar{w}_1^T \bar{z} + a_1 & \text{for } \bar{z} \in C_1 \\ \bar{w}_2^T \bar{z} + a_2 & \text{for } \bar{z} \in C_2 \\ \vdots & \vdots \\ \bar{w}_K^T \bar{z} + a_k & \text{for } \bar{z} \in C_k \end{cases}$$

     for some K-piece convex partition of $\mathbb{R}^m = \bigcup_{j=1}^K C_i$ with the $C_s \cap C_t = \emptyset$ for $s \neq t$. (graphic2) Note that to say that this function is convex is to say that for all $\bar{z} \in C_i$ we have

     $$\bar{w}_i^T \bar{z} + a_i = \max_{1 \leq j \leq K} (\bar{w}_j^T \bar{z} + a_j)$$

     Then this unit can be expressed in terms of maxout as follows:

     $$h'(\bar{x}) = g'(B\bar{x} + \bar{d}) = \begin{cases} \bar{w}_1^T (B\bar{x} + \bar{d}) + a_1 & \text{for } \bar{x} \in B^{-1}(C_1 - \bar{d}) \\ \bar{w}_2^T (B\bar{x} + \bar{d}) + a_2 & \text{for } \bar{x} \in B^{-1}(C_2 - \bar{d}) \\ \vdots & \vdots \\ \bar{w}_K^T (B\bar{x} + \bar{d}) + a_K & \text{for } \bar{x} \in B^{-1}(C_K - \bar{d}) \end{cases}$$

$$= \max_{1 \leq j \leq K} \left(\bar{w}_j^T (B\bar{x} + \bar{d}) + a_j \right) = \max_{1 \leq j \leq K} (D\bar{x} + \bar{c})$$

where $D = WB \in \mathbb{R}^{k \times n}$ and $\bar{c} = W\bar{d} + \bar{a} \in \mathbb{R}^k$, and where

$$W = \begin{bmatrix} \bar{w}_1^T \\ \bar{w}_2^T \\ \vdots \\ \bar{w}_K^T \end{bmatrix}$$

In summary, any unit $h' = g'(B\bar{x} + \bar{d})$ with $B \in \mathbb{R}^{m \times n}$ and g' K-piecewise-linear and convex can be expressed as $\max_{1 \leq j \leq K} (D\bar{x} + \bar{c})$ for $D \in \mathbb{R}^{k \times n}$. We can generalize this to vector valued outputs by taking advantage of multiple groups G_i in the definition of maxout (left to reader).

8. The sigmoid
$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$
and hyperbolic tangent
$$\tanh(z) = 2\sigma(2z) - 1$$
used to be the go-to activation functions for hidden layer. However, unlike the family of ReLU-like functions mentioned previously, these both saturate at very positive and very negative inputs, so ReLU-like functions are generally preferred to them on hidden units. But, as we have seen before, if they are composed with the log function, as in the average cross-entropy cost function, this saturation goes away. Thus we may still use sigmoid and tanh as activation functions for *output units* if we have an appropriate cost function.

9. We may have *no* activation function. As mentioned at the beginning, two consecutive, purely affine layers in a FFNN are not able to define any function that couldn't be defined with one affine layer, but it might still be advantageous to allow two consecutive affine layers, simply for computational efficiency.
For example if we had a layer $h_i(\bar{x}) = A\bar{x}$ for $A \in \mathbb{R}^{m \times n}$ where m and n are large but A is expected to be low-rank (that is $dim(span(A)) \ll m$), then we might replace A by BC where $B \in \mathbb{R}^{m \times p}$ and $C \in \mathbb{R}^{p \times n}$ for some small p.

6 Network Architecture

- Aside from choice of cost function, output layer, and hidden units, the remaining choice when designing a FFNN is the depth of the network and the widths of the various hidden layers.

- The *Universal Approximation Theorem* tells us that in fact *any* continuous function defined on a closed and bounded subset of $\mathbb{R}^n$ can be approximated as closely as we like by a neural network with a linear output layer and at least one hidden layer with an appropriate activation function.

- More specifically,

  > Let f be any continuous function defined on any closed and bounded subset C of $\mathbb{R}^n$. Let $g : \mathbb{R} \to \mathbb{R}$ be any scalar-valued activation function that is non-constant, bounded, continuous, and monotonically-increasing (e.g. the sigmoid). Let $\epsilon \in \mathbb{R}$ be some tolerance with $\epsilon > 0$.
  > Then there exists some $M \in \mathbb{N}$, vector $\bar{v} \in \mathbb{R}^M$, some matrix $W \in \mathbb{R}^{M \times n}$, and some vector $\bar{b} \in \mathbb{R}^M$ such that, letting F be the FFNN defined by $F(\bar{x}) = \bar{v}^T \bar{g}(W\bar{x} + \bar{b})$, we have
  >
  > $$|F(\bar{x}) - f(\bar{x})| < \epsilon$$
  >
  > for all $\bar{x}$ in C.[3] [4]

- While the assumptions here imply that g saturates on large magnitude inputs (see why?), similar theorems have since been proven where the activation function may be ReLU-like.

- Unfortunately, just because such weights $\bar{v}, W, \bar{b}$ exist, doesn't mean that we'll necessarily be able to find them during training.

- Frequently we can describe a more complex space by going deep and narrow than wide and shallow. E.g. Montufar et. al. showed that the number of linear regions carved out by a deep rectifier network with d inputs, depth l, and n units per hidden layer is

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right).$$

7 Backpropagation

- Finally we turn to the problem of actually computing the gradient of our cost function.

- As before we can assume our model function can be expressed in terms of several layers as follows

$$f(\bar{x}; \bar{\theta}) = f_{\bar{\theta}}(\bar{x}) = f_{d,\bar{\theta}_d}(f_{d-1,\bar{\theta}_{d-1}}(\ldots(f_{2,\bar{\theta}_2}(f_{1,\bar{\theta}_1}(\bar{x}))\ldots)$$

[3] Here $\bar{g} : \mathbb{R}^N \to \mathbb{R}^N$ is just the function applying g componentwise

[4] The statement of the theorem is stronger. It makes the claim not just for continuous functions but for all Borel-measurable functions.

- We will also have some error function $f_{err}(\bar{y}, \hat{y})$ such that, for some set (or mini-batch)
$$D = \{(\bar{x}^{(i)}, \bar{y}^{(i)})\}_{1 \leq i \leq M}$$
of training data, our cost function can be expressed as
$$Cost(D, \bar{\theta}) = \frac{1}{M} \sum_i f_{err}(\bar{y}^{(i)}, f_{\bar{\theta}}(\bar{x}^{(i)})).$$

- As mentioned above, we will almost always take
$$f_{err}(\bar{y}, \hat{y}) = H(P_{\bar{y}}, Q_{\hat{y}}) = -E_{P_{\bar{y}}} \log Q_{\hat{y}},$$
But here we won't make any specific assumptions about the form of f_{err}.

- Note that although $Cost$ is written above as a function of both $\bar{\theta}$ and (some batch of) training examples $D = \{(\bar{x}^{(i)}, \bar{y}^{(i)})\}_{1 \leq i \leq M}$, we will treat the latter as constants and attempt to minimize $Cost$ as a function of $\bar{\theta}$. The process of actually plugging in the batch to get a function $Cost(\bar{\theta})$ of just the model parameters is sometimes called *forward propagation*.

- Thus our general goal is to find a way of computing the gradient of $Cost$. Since the gradient of a linear combination is just the linear combination of the gradients, this reduces to finding the gradient $\nabla_{\bar{\theta}} J(\hat{\theta})$ of an arbitrary function J at some point $\hat{\theta}$, where J is of the form
$$J(\bar{\theta}) = f_{err}(f_n(\bar{\theta}_n, f_{n-1}(\bar{\theta}_{n-1}, f_{n-2}(\dots f_2(\bar{\theta}_2, f_1(\bar{\theta}_1)) \dots))))$$
where $\bar{\theta}$ is the concatenation of all the $\{\bar{\theta}_i\}_{1 \leq i \leq n}$. We can then update $\hat{\theta}$ in terms of this gradient as in some procedure like gradient descent.

- It will be worthwhile to introduce intermediate scalar-valued variables to represent the outputs of the various layers using a system of equations like this:
$$\begin{aligned} J(\bar{\theta}) &= z, \\ z &= f_{err}(\bar{z}_n) \\ \bar{z}_n &= f_n(\bar{\theta}_n, \bar{z}_{n-1}) \\ \bar{z}_{n-1} &= f_{n-1}(\bar{\theta}_{n-1}, \bar{z}_{n-2}) \\ &\vdots \quad \vdots \\ \bar{z}_2 &= f_2(\bar{\theta}_2, \bar{z}_1) \\ \bar{z}_1 &= f_1(\bar{\theta}_1) \end{aligned}$$

- We recall the chain rule:

Let $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^m \to \mathbb{R}^n$ be functions with $z = f(\bar{u})$ and $\bar{u} = g(\bar{v})$. Then for $1 \leq j \leq m$ we have

$$\frac{\partial f}{\partial v_j} = \frac{\partial z}{\partial v_j} = \sum_{i=1}^{n} \frac{\partial u_i}{\partial v_j} \frac{\partial z}{\partial u_i} = \begin{bmatrix} \frac{\partial u_1}{\partial v_j} \\ \frac{\partial u_2}{\partial v_j} \\ \vdots \\ \frac{\partial u_n}{\partial v_j} \end{bmatrix}^T \nabla_{\bar{u}} z$$

And so putting these all together we get

$$\nabla_{\bar{v}} f = \nabla_{\bar{v}} z = \begin{bmatrix} \frac{\partial u_1}{\partial v_1} & \frac{\partial u_2}{\partial v_1} & \cdots & \frac{\partial u_n}{\partial v_1} \\ \frac{\partial u_1}{\partial v_2} & \frac{\partial u_2}{\partial v_2} & \cdots & \frac{\partial u_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial u_1}{\partial v_m} & \frac{\partial u_2}{\partial v_m} & \cdots & \frac{\partial u_n}{\partial v_m} \end{bmatrix} \nabla_{\bar{u}} z = \left(\frac{\partial \bar{u}}{\partial \bar{v}} \right)^T \nabla_{\bar{u}} z$$

Where $\frac{\partial \bar{u}}{\partial \bar{v}}$ is the Jacobian matrix (which we encountered in chapter 2) for the function g.[5]

Notice we could just compute each $\frac{\partial z}{\partial \theta_{i,j}}$ in turn by recursively applying, say, the first version of the chain rule on all possible paths from z to a $\theta_{i,j}$. More precisely, if instead of a single g above we have $\{g_i\}_{1 \leq i \leq d}$, where for $1 \leq i \leq d$ we have $g_i : \mathbb{R}^{r_{i-1}} \to \mathbb{R}^{r_i}$ and

$$\begin{aligned} z &= f(\bar{u}_d) \\ \bar{u}_d &= g_d(\bar{u}_{d-1}) \\ \bar{u}_{d-1} &= g_{d-1}(\bar{u}_{d-2}) \\ \vdots &\quad \vdots \\ \bar{u}_2 &= g_2(\bar{u}_1) \\ \bar{u}_1 &= g_1(\bar{v}) \end{aligned}$$

Then for $1 \leq j \leq r_0$ we have

$$\frac{\partial f}{\partial v_j} = \frac{\partial z}{\partial v_j} = \sum_{p \in \mathscr{P}_j} \left(\frac{\partial p_1}{\partial v_j} \left(\prod_{k=1}^{d-1} \frac{\partial p_{k+1}}{\partial p_k} \right) \frac{\partial z}{\partial p_d} \right)$$

[5] We use this common notation when discussing derivatives and gradients, but keep in mind that these are *functions*. For example, when we say

$$\frac{\partial z}{\partial v_j} = \sum_{i=1}^{n} \frac{\partial u_i}{\partial v_j} \frac{\partial z}{\partial u_i}$$

We're saying that, for all vector inputs $\hat{v}$ to the function on the left, the value is

$$\frac{\partial z}{\partial v_j}(\hat{v}) = \sum_{i=1}^{n} \left(\frac{\partial u_i}{\partial v_j}(\hat{v}) \frac{\partial z}{\partial u_i}(g(\hat{v})) \right).$$

where for $1 \leq j \leq r_0$ we denote by $\mathscr{P}_i$ the set of all all paths from variables in $\bar{u}_d$ down to $\bar{u}_1$, that is to say each step p_t of the path p is some variable in the vector $\bar{u}_t$.

But this will in general be computationally wasteful (consider how many links in the various paths would be repeated for various different variables in $\bar{v}$). For this reason we will generally take some approach that stores partial derivatives along the way.

The following is a basic backpropagation algorithm.

Suppose

$$J(\bar{\theta}) = z,$$
$$z = f_{err}(\bar{z}_n)$$
$$\bar{z}_n = f_n(\bar{\theta}_n, \bar{z}_{n-1})$$
$$\bar{z}_{n-1} = f_{n-1}(\bar{\theta}_{n-1}, \bar{z}_{n-2})$$
$$\vdots \quad \vdots$$
$$\bar{z}_2 = f_2(\bar{\theta}_2, \bar{z}_1)$$
$$\bar{z}_1 = f_1(\bar{\theta}_1).$$

1. Define tables θ_grad and z_grad with keys the integers $1 \leq i \leq n$. When we are done, the value of θ_grad on each key i will be $\nabla_{\bar{\theta}_i} J$ and value of z_grad on each key i will be $\nabla_{\bar{z}_i} J$
2. Set $z\_grad[n] \leftarrow \nabla_{\bar{z}_n} f_{err}$.
3. For $i \leftarrow n$ down to 2
   (a) Compute the transpose of the Jacobian $\left(\frac{\partial \bar{z}_i}{\partial \bar{\theta}_i \bar{z}_{i-1}}\right)^T$ of the function f_i with respect to all its variables, and divide the rows into two matrices depending on whether they're a θ variable row or a z variable row: $\left(\frac{\partial \bar{z}_i}{\partial \bar{z}_{i-1}}\right)^T$ and $\left(\frac{\partial \bar{z}_i}{\partial \bar{\theta}_i}\right)^T$
   (b) Set $\theta\_grad[i] \leftarrow \left(\frac{\partial \bar{z}_i}{\partial \bar{\theta}_i}\right)^T z\_grad[\bar{z}_i]$
   (c) Set $z\_grad[i-1] \leftarrow \left(\frac{\partial \bar{z}_i}{\partial \bar{z}_{i-1}}\right)^T z\_grad[\bar{z}_i]$
4. Set $\theta\_grad[1] \leftarrow \left(\frac{\partial \bar{z}_1}{\partial \bar{\theta}_1}\right)^T z\_grad[\bar{z}_1]$
5. Return θ_grad.

- Notice that this algorithm generalizes to any J whose structure is given by a directed acyclic graph.

Let $\mathscr{G}$ be a DAG whose nodes are each some scalar valued variable v together with some function f_v for computing v from the variables in any parent nodes $u \in Par_{\mathscr{G}}(v)$. (If the node for v has no parents, we can take f_v to be the constant function outputting v). We can assume also that only one node in $\mathscr{G}$ has no children (this will represent the output of J).

1. Let $v_1 \preccurlyeq v_2 \preccurlyeq \ldots \preccurlyeq v_s$ be any topological ordering of the nodes of $\mathscr{G}$.[6]
2. Let *grad_table* be a table with keys $1, 2, \ldots, s$.
3. Set $grad\_table[s] \leftarrow 1$ (since $v_s = J(\bar{\theta})$ is the output variable)
4. For $j = s - 1$ down to 1:

$$grad\_table[j] \leftarrow \sum_{i:j \in Par(v_i)} grad\_table[i] \frac{\partial v_i}{\partial v_j}$$

5. Return the vector of all $grad\_table[i]$ where v_i is a leaf node.

- If we take a single scalar derivative to have constant computation time, the total runtime of these algorithms is $O(|E_{\mathscr{G}}|)$ where $E_{\mathscr{G}}$ is the set of all edges in $\mathscr{G}$.

[6] A *topological ordering* of a directed graph $\mathscr{G}$ is some linear ordering $\preccurlyeq$ of the nodes of $\mathscr{G}$ such that, for any two nodes v, u from $\mathscr{G}$, if there is an edge from v to u then $v \preccurlyeq u$. Such an ordering will exist if and only if $\mathscr{G}$ is acyclic.