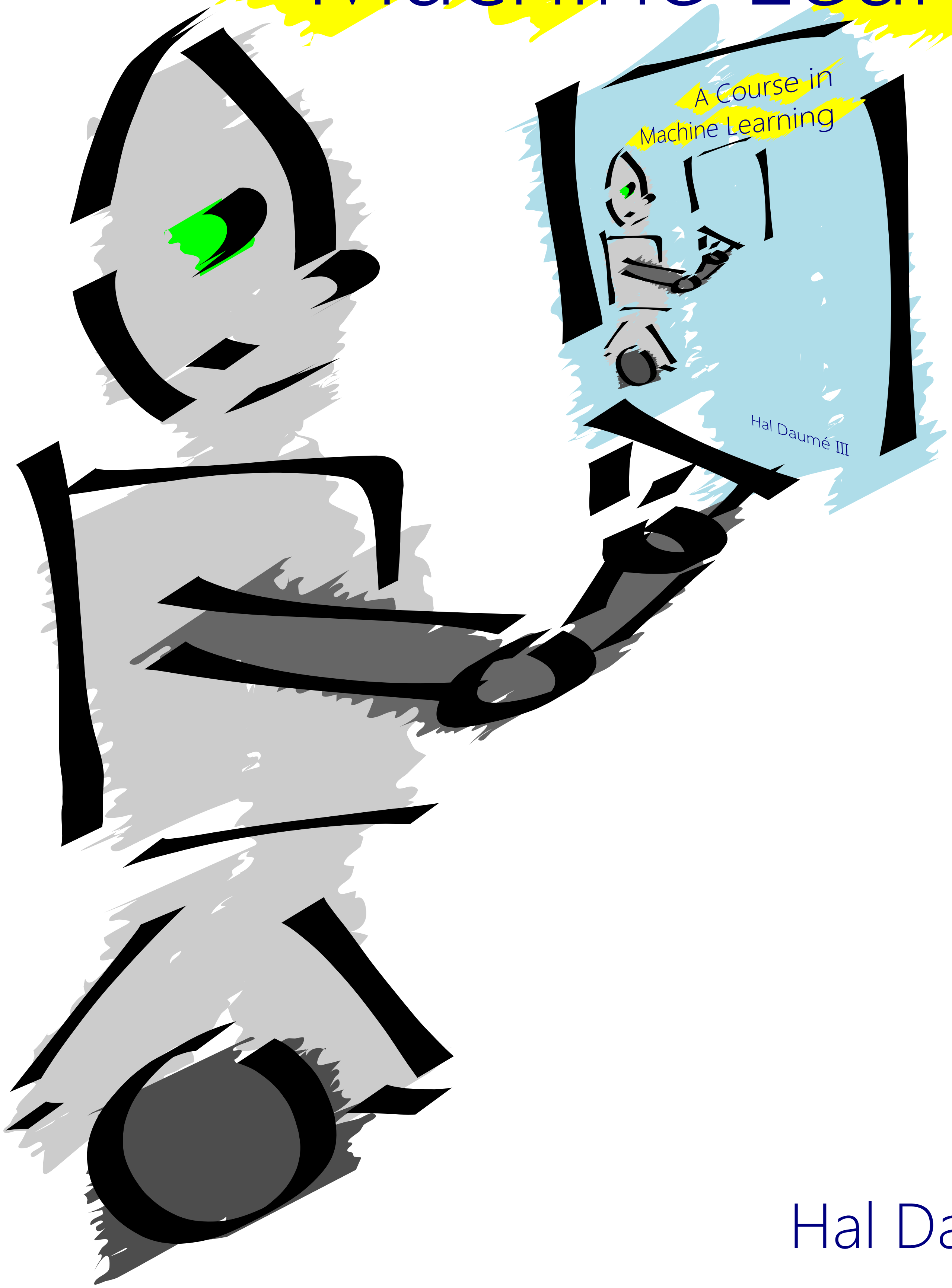
A Course in Machine Learning



Hal Daumé III

version 0.8 , August 2012

8 | Neural Networks

--

Learning Objectives:

- Explain the biological inspiration for multi-layer neural networks.
- Construct a two-layer network that can solve the XOR problem.
- Implement the back-propogation algorithm for training multi-layer networks.
- Explain the trade-off between depth and breadth in network structure.
- Contrast neural networks with radial basis functions with k-nearest neighbor learning.

Dependencies:

The first learning models you learned about (decision trees and nearest neighbor models) created complex, **non-linear** decision boundaries. We moved from there to the perceptron, perhaps the most classic linear model. At this point, we will move *back* to non-linear learning models, but using all that we have learned about linear learning thus far.

This chapter presents an extension of perceptron learning to non-linear decision boundaries, taking the biological inspiration of neurons even further. In the perceptron, we thought of the input data point (eg., an image) as being directly connected to an output (eg., label). This is often called a **single-layer network** because there is one layer of weights. Now, instead of directly connecting the inputs to the outputs, we will insert a layer of "hidden" nodes, moving from a single-layer network to a **multi-layer network**. But introducing a non-linearity at inner layers, this will give us non-linear decision boundaires. In fact, such networks are able to express almost any function we want, not just linear functions. The trade-off for this flexibility is increased complexity in parameter tuning and model design.

8.1 Bio-inspired Multi-Layer Networks

One of the major weaknesses of linear models, like perceptron and the regularized linear models from the previous chapter, is that they are linear! Namely, they are unable to learn arbitrary decision boundaries. In contrast, decision trees and KNN *could* learn arbitrarily complicated decision boundaries.

One approach to doing this is to chain together a collection of perceptrons to build more complex **neural networks**. An example of a **two-layer network** is shown in Figure 8.1. Here, you can see five inputs (features) that are fed into two **hidden units**. These hidden units are then fed in to a single **output unit**. Each edge in this figure corresponds to a different weight. (Even though it looks like there are three layers, this is called a two-layer network because we don't count the inputs as a real layer. That is, it's two layers of *trained* weights.)

Prediction with a neural network is a straightforward generaliza-

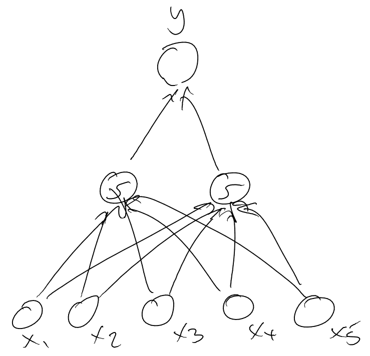


Figure 8.1: picture of a two-layer network with 5 inputs and two hidden units

tion of prediction with a perceptron. First you compute activations of the nodes in the hidden unit based on the inputs and the input weights. Then you compute activations of the output unit given the hidden unit activations and the second layer of weights.

The only major difference between this computation and the perceptron computation is that the hidden units compute a *non-linear* function of their inputs. This is usually called the **activation function** or **link function**. More formally, if $w_{i,d}$ is the weights on the edge connecting input d to hidden unit i, then the activation of hidden unit i is computed as:

$$h_i = f\left(\boldsymbol{w}_i \cdot \boldsymbol{x}\right) \tag{8.1}$$

Where f is the link function and $\boldsymbol{w}_i$ refers to the vector of weights feeding in to node i.

One example link function is the **sign** function. That is, if the incoming signal is negative, the activation is -1. Otherwise the activation is $+1$. This is a potentially useful activiation function, but you might already have guessed the problem with it: it is non-differentiable.

EXPLAIN BIAS!!!

A more popular link function is the **hyperbolic tangent** function, tanh. A comparison between the sign function and the tanh function is in Figure 8.2. As you can see, it is a reasonable approximation to the sign function, but is convenient in that it is differentiable.[1] Because it looks like an "S" and because the Greek character for "S" is "Sigma," such functions are usually called **sigmoid** functions.

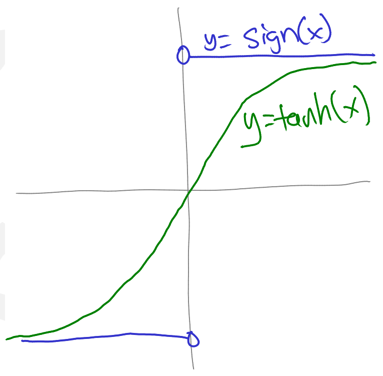


Figure 8.2: picture of sign versus tanh

[1] It's derivative is just $1 - \tanh^2(x)$.

Assuming for now that we are using tanh as the link function, the overall prediction made by a two-layer network can be computed using Algorithm 8.1. This function takes a matrix of weights $\mathbf{W}$ corresponding to the first layer weights and a vector of weights $\boldsymbol{v}$ corresponding to the second layer. You can write this entire computation out in one line as:

$$\hat{y} = \sum_i v_i \tanh(\boldsymbol{w}_i \cdot \hat{\boldsymbol{x}}) \tag{8.2}$$

$$= \boldsymbol{v} \cdot \tanh(\mathbf{W}\hat{\boldsymbol{x}}) \tag{8.3}$$

Where the second line is short hand assuming that tanh can take a vector as input and product a vector as output.

? Is it necessary to use a link function at all? What would happen if you just used the identify function as a link?

Algorithm 24 TwoLayerNetworkPredict($\mathbf{W}, \boldsymbol{v}, \hat{\boldsymbol{x}}$)

```
1: for $i$ = 1 to number of hidden units do
2:     $h_i \leftarrow \tanh(\boldsymbol{w}_i \cdot \hat{\boldsymbol{x}})$        // compute activation of hidden unit $i$
3: end for
4: return $\boldsymbol{v} \cdot \boldsymbol{h}$        // compute output unit
```

The claim is that two-layer neural networks are more expressive than single layer networks (i.e., perceptrons). To see this, you can construct a very small two-layer network for solving the XOR problem. For simplicity, suppose that the data set consists of four data points, given in Table 8.1. The classification rule is that $y = +1$ if an only if $x_1 = x_2$, where the features are just ± 1.

y	x_0	x_1	x_2
+1	+1	+1	+1
+1	+1	-1	-1
-1	+1	+1	-1
-1	+1	-1	+1

Table 8.1: Small XOR data set.

You can solve this problem using a two layer network with two hidden units. The key idea is to make the first hidden unit compute an "or" function: $x_1 \vee x_2$. The second hidden unit can compute an "and" function: $x_1 \wedge x_2$. The the output can combine these into a single prediction that mimics XOR. Once you have the first hidden unit activate for "or" and the second for "and," you need only set the output weights as -2 and $+1$, respectively.

? Verify that these output weights will actually give you XOR.

To achieve the "or" behavior, you can start by setting the bias to -0.5 and the weights for the two "real" features as both being 1. You can check for yourself that this will do the "right thing" if the link function were the sign function. Of course it's not, it's tanh. To get tanh to mimic sign, you need to make the dot product either really really large or really really small. You can accomplish this by setting the bias to $-500,000$ and both of the two weights to $1,000,000$. Now, the activation of this unit will be just slightly above -1 for $\boldsymbol{x} = \langle -1, -1 \rangle$ and just slightly below $+1$ for the other three examples.

? This shows how to create an "or" function. How can you create an "and" function?

At this point you've seen that one-layer networks (aka perceptrons) can represent any linear function and only linear functions. You've also seen that two-layer networks can represent non-linear functions like XOR. A natural question is: do you get additional representational power by moving beyond two layers? The answer is partially provided in the following Theorem, due originally to George Cybenko for one particular type of link function, and extended later by Kurt Hornik to arbitrary link functions.

Theorem 9 (Two-Layer Networks are Universal Function Approximators)**.** *Let F be a continuous function on a bounded subset of D-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate F arbitrarily well. Namely, for all $\boldsymbol{x}$ in the domain of F, $\left|F(\boldsymbol{x}) - \hat{F}(\boldsymbol{x})\right| < \epsilon$.*

Or, in colloquial terms "two-layer networks can approximate any function."

This is a remarkable theorem. Practically, it says that if you give me a function F and some error tolerance parameter ϵ, I can construct a two layer network that computes F. In a sense, it says that going from one layer to two layers completely changes the representational capacity of your model.

When working with two-layer networks, the key question is: how many hidden units should I have? If your data is D dimensional and you have K hidden units, then the total number of parameters is $(D+2)K$. (The first $+1$ is from the bias, the second is from the second layer of weights.) Following on from the heuristic that you should have one to two examples for each parameter you are trying to estimate, this suggests a method for choosing the number of hidden units as roughly $\lfloor \frac{N}{D} \rfloor$. In other words, if you have tons and tons of examples, you can safely have lots of hidden units. If you only have a few examples, you should probably restrict the number of hidden units in your network.

The number of units is both a form of inductive bias and a form of regularization. In both view, the number of hidden units controls how complex your function will be. Lots of hidden units $\Rightarrow$ very complicated function. Figure **??** shows training and test error for neural networks trained with different numbers of hidden units. As the number increases, training performance continues to get better. But at some point, test performance gets worse because the network has overfit the data.

8.2 The Back-propagation Algorithm

The back-propagation algorithm is a classic approach to training neural networks. Although it was not originally seen this way, based on what you know from the last chapter, you can summarize back-propagation as:

$$\text{back-propagation} = \text{gradient descent} + \text{chain rule} \tag{8.4}$$

More specifically, the set up is *exactly* the same as before. You are going to optimize the weights in the network to minimize some objective function. The only difference is that the predictor is no longer linear (i.e., $\hat{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b$) but now non-linear (i.e., $\boldsymbol{v} \cdot \tanh(\mathbf{W}\hat{\boldsymbol{x}})$). The only question is how to do gradient descent on this more complicated objective.

For now, we will ignore the idea of regularization. This is for two reasons. The first is that you already know how to deal with regularization, so everything you've learned before applies. The second is that *historically*, neural networks have not been regularized. Instead, people have used **early stopping** as a method for controlling overfitting. Presently, it's not obvious which is a better solution: both are valid options.

To be completely explicit, we will focus on optimizing squared error. Again, this is mostly for historic reasons. You could easily replace squared error with your loss function of choice. Our overall objective is:

$$\min_{\mathbf{W},\boldsymbol{v}} \quad \sum_n \frac{1}{2}\left(y_n - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)\right)^2 \tag{8.5}$$

Here, f is some link function like tanh.

The easy case is to differentiate this with respect to $\boldsymbol{v}$: the weights for the output unit. Without even doing any math, you should be able to guess what this looks like. The way to think about it is that from $\boldsymbol{v}$s perspective, it is just a linear model, attempting to minimize squared error. The only "funny" thing is that its inputs are the activations $\boldsymbol{h}$ rather than the examples $\boldsymbol{x}$. So the gradient with respect to $\boldsymbol{v}$ is just as for the linear case.

To make things notationally more convenient, let e_n denote the *error* on the nth example (i.e., the blue term above), and let $\boldsymbol{h}_n$ denote the vector of hidden unit activations on that example. Then:

$$\nabla_{\boldsymbol{v}} = -\sum_n e_n \boldsymbol{h}_n \tag{8.6}$$

This is exactly like the linear case. One way of interpreting this is: how would the output weights have to change to make the prediction better? This is an easy question to answer because they can easily measure how their changes affect the output.

The more complicated aspect to deal with is the weights corresponding to the *first* layer. The reason this is difficult is because the weights in the first layer aren't necessarily trying to produce specific values, say 0 or 5 or -2.1. They are simply trying to produce activations that get fed to the output layer. So the change they want to make depends crucially on how the output layer interprets them.

Thankfully, the chain rule of calculus saves us. Ignoring the sum over data points, we can compute:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right)^2 \tag{8.7}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_i} = \frac{\partial \mathcal{L}}{\partial f_i}\frac{\partial f_i}{\partial \boldsymbol{w}_i} \tag{8.8}$$

$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right) v_i = -e v_i \tag{8.9}$$

$$\frac{\partial f_i}{\partial \boldsymbol{w}_i} = f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x} \tag{8.10}$$

Algorithm 25 TwoLayerNetworkTrain($\mathbf{D}$, η, K, $MaxIter$)

```
W ← D×K matrix of small random values              // initialize input layer weights
v ← K-vector of small random values                // initialize output layer weights
for iter = 1 ... MaxIter do
    G ← D×K matrix of zeros                        // initialize input layer gradient
    g ← K-vector of zeros                          // initialize output layer gradient
    for all (x,y) ∈ D do
        for i = 1 to K do
            $a_i \leftarrow \mathbf{w}_i \cdot \hat{\mathbf{x}}$
            $h_i \leftarrow \tanh(a_i)$             // compute activation of hidden unit i
        end for
        $\hat{y} \leftarrow \mathbf{v} \cdot \mathbf{h}$                          // compute output unit
        $e \leftarrow y - \hat{y}$                                // compute error
        $\mathbf{g} \leftarrow \mathbf{g} - e\mathbf{h}$                       // update gradient for output layer
        for i = 1 to K do
            $\mathbf{G}_i \leftarrow \mathbf{G}_i - ev_i(1 - \tanh^2(a_i))\mathbf{x}$   // update gradient for input layer
        end for
    end for
    $\mathbf{W} \leftarrow \mathbf{W} - \eta\mathbf{G}$                         // update input layer weights
    $\mathbf{v} \leftarrow \mathbf{v} - \eta\mathbf{g}$                         // update output layer weights
end for
return W, v
```

Putting this together, we get that the gradient with respect to $\mathbf{w}_i$ is:

$$\nabla_{\mathbf{w}_i} = -ev_i f'(\mathbf{w}_i \cdot \mathbf{x})\mathbf{x} \tag{8.11}$$

Intuitively you can make sense of this. If the overall error of the predictor (e) is small, you want to make small steps. If v_i is small for hidden unit i, then this means that the output is not particularly sensitive to the activation of the ith hidden unit. Thus, its gradient should be small. If v_i flips sign, the gradient at $\mathbf{w}_i$ should also flip signs. The name **back-propagation** comes from the fact that you propagate gradients backward through the network, starting at the end.

The complete instantiation of gradient descent for a two layer network with K hidden units is sketched in Algorithm 8.2. Note that this really is *exactly* a gradient descent algorithm; the only different is that the computation of the gradients of the input layer is moderately complicated.

As a bit of practical advice, implementing the back-propagation algorithm can be a bit tricky. Sign errors often abound. A useful trick is first to keep $\mathbf{W}$ fixed and work on just training $\mathbf{v}$. Then keep $\mathbf{v}$ fixed and work on training $\mathbf{W}$. Then put them together.

? What would happen to this algorithm if you wanted to optimize exponential loss instead of squared error? What if you wanted to add in weight regularization?

? If you like matrix calculus, derive the same algorithm starting from Eq (8.3).

8.3 Initialization and Convergence of Neural Networks

Based on what you know about linear models, you might be tempted to initialize all the weights in a neural network to zero. You might also have noticed that in Algorithm **??**, this is not what's done: they're initialized to small random values. The question is why?

The answer is because an initialization of $\mathbf{W} = \mathbf{0}$ and $\boldsymbol{v} = \mathbf{0}$ will lead to "uninteresting" solutions. In other words, if you initialize the model in this way, it will eventually get stuck in a bad local optimum. To see this, first realize that on any example $\boldsymbol{x}$, the activation h_i of the hidden units will all be zero since $\mathbf{W} = \mathbf{0}$. This means that on the first iteration, the gradient on the output weights ($\boldsymbol{v}$) will be zero, so they will stay put. Furthermore, the gradient $w_{1,d}$ for the dth feature on the ith unit will be *exactly* the same as the gradient $w_{2,d}$ for the same feature on the second unit. This means that the weight matrix, after a gradient step, will change in *exactly the same way* for every hidden unit. Thinking through this example for iterations $2 \dots$, the values of the hidden units will *always* be exactly the same, which means that the weights feeding in to any of the hidden units will be exactly the same. Eventually the model will converge, but it will converge to a solution that does not take advantage of having access to the hidden units.

This shows that neural networks are *sensitive* to their initialization. In particular, the function that they optimize is **non-convex**, meaning that it might have plentiful local optima. (One of which is the trivial local optimum described in the preceding paragraph.) In a sense, neural networks *must* have local optima. Suppose you have a two layer network with two hidden units that's been optimized. You have weights $\boldsymbol{w}_1$ from inputs to the first hidden unit, weights $\boldsymbol{w}_2$ from inputs to the second hidden unit and weights (v_1, v_2) from the hidden units to the output. If I give you back another network with $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ swapped, and v_1 and v_2 swapped, the network computes *exactly* the same thing, but with a markedly different weight structure. This phenomena is known as **symmetric modes** ("mode" referring to an optima) meaning that there are symmetries in the weight space. It would be one thing if there were lots of modes and they were all symmetric: then finding one of them would be as good as finding any other. Unfortunately there are additional local optima that are *not* global optima.

Random initialization of the weights of a network is a way to address *both* of these problems. By initializing a network with small random weights (say, uniform between -0.1 and 0.1), the network is unlikely to fall into the trivial, symmetric local optimum. Moreover, by training a collection of networks, each with a different random

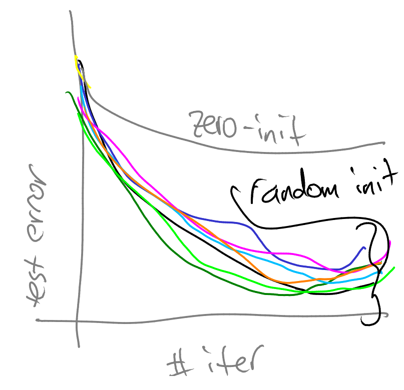


Figure 8.3: convergence of randomly initialized networks

initialization, you can often obtain better solutions that with just one initialization. In other words, you can train ten networks with different random seeds, and then pick the one that does best on held-out data. Figure 8.3 shows prototypical *test-set* performance for ten networks with different random initialization, plus an eleventh plot for the trivial symmetric network initialized with zeros.

One of the typical complaints about neural networks is that they are finicky. In particular, they have a rather large number of knobs to tune:

1. The number of layers
2. The number of hidden units per layer
3. The gradient descent learning rate η
4. The initialization
5. The stopping iteration or weight regularization

The last of these is minor (early stopping is an easy regularization method that does not require much effort to tune), but the others are somewhat significant. Even for two layer networks, having to choose the number of hidden units, and then get the learning rate and initialization "right" can take a bit of work. Clearly it can be automated, but nonetheless it takes time.

Another difficulty of neural networks is that their weights can be difficult to interpret. You've seen that, for linear networks, you can often interpret high weights as indicative of positive examples and low weights as indicative of negative examples. In multilayer networks, it becomes very difficult to try to understand what the different hidden units are doing.

8.4 Beyond Two Layers

The definition of neural networks and the back-propagation algorithm can be generalized beyond two layers to any arbitrary directed acyclic graph. In practice, it is most common to use a layered network like that shown in Figure 8.4 unless one has a very strong reason (aka inductive bias) to do something different. However, the view as a directed graph sheds a different sort of insight on the back-propagation algorithm.

Suppose that your network structure is stored in some directed acyclic graph, like that in Figure 8.5. We index nodes in this graph as u, v. The activation *before* applying non-linearity at a node is a_u and after non-linearity is h_u. The graph has a single sink, which is the output node y with activation a_y (no non-linearity is performed

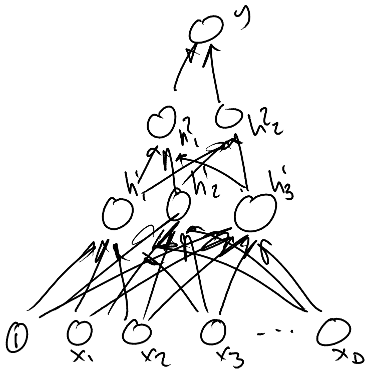


Figure 8.4: multi-layer network

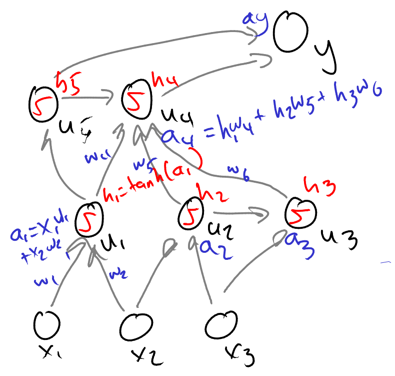


Figure 8.5: DAG network

Algorithm 26 FORWARDPROPAGATION(x)

```
1: for all input nodes $u$ do
2:    $h_u \leftarrow$ corresponding feature of $x$
3: end for
4: for all nodes $v$ in the network whose parent's are computed do
5:    $a_v \leftarrow \sum_{u \in par(v)} w_{(u,v)} h_u$
6:    $h_v \leftarrow \tanh(a_v)$
7: end for
8: return $a_y$
```

Algorithm 27 BACKPROPAGATION(x, y)

```
1: run FORWARDPROPAGATION($x$) to compute activations
2: $e_y \leftarrow y - a_y$                                   // compute overall network error
3: for all nodes $v$ in the network whose error $e_v$ is computed do
4:    for all $u \in par(v)$ do
5:       $g_{u,v} \leftarrow -e_v h_u$                         // compute gradient of this edge
6:       $e_u \leftarrow e_u + e_v w_{u,v}(1 - \tanh^2(a_u))$  // compute the "error" of the parent node
7:    end for
8: end for
9: return all gradients $g_e$
```

on the output unit). The graph has D-many inputs (i.e., nodes with no parent), whose activations h_u are given by an input example. An edge (u, v) is from a parent to a child (i.e., from an input to a hidden unit, or from a hidden unit to the sink). Each edge has a weight $w_{u,v}$. We say that $par(u)$ is the set of parents of u.

There are two relevant algorithms: forward-propagation and back-propagation. Forward-propagation tells you how to compute the activation of the sink y given the inputs. Back-propagation computes derivatives of the edge weights for a given input.

The key aspect of the **forward-propagation** algorithm is to iteratively compute activations, going deeper and deeper in the DAG. Once the activations of all the parents of a node u have been computed, you can compute the activation of node u. This is spelled out in Algorithm 8.4. This is also explained pictorially in Figure 8.6.

Back-propagation (see Algorithm 8.4) does the opposite: it computes gradients top-down in the network. The key idea is to compute an *error* for each node in the network. The error at the output unit is the "true error." For any input unit, the error is the amount of gradient that we see coming from our children (i.e., higher in the network). These errors are computed backwards in the network (hence the name **back-propagation**) along with the gradients themselves. This is also explained pictorially in Figure 8.7.

Given the back-propagation algorithm, you can directly run gradient descent, using it as a subroutine for computing the gradients.

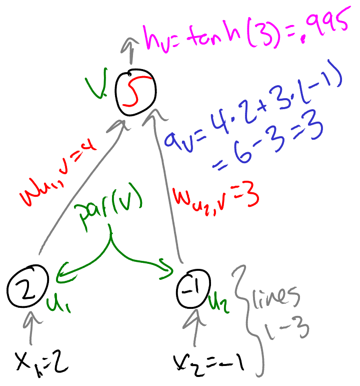
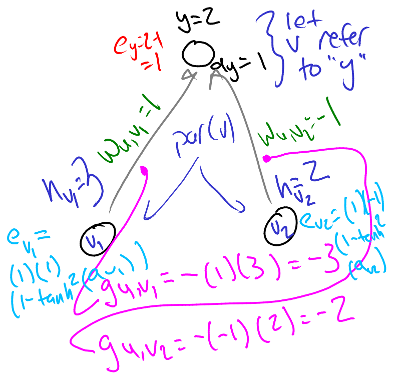


Figure 8.6: picture of forward prop



$y = 2$
$e_y = 2 - 1 = 1$
$a_y = 1$
let v refer to "y"
$w_{u,v_1} = 1$
$w_{u,v_2} = -1$
par(v)
$h_{v_1} = 3$
$h_{v_2} = 2$
v_1
v_2
$e_{v_1} = (1)(1)(1 - \tanh^2(a_{v_1}))$
$e_{v_2} = (1)(-1)(1 - \tanh^2(a_{v_2}))$
$g_{u,v_1} = -(1)(3) = -3$
$g_{u,v_2} = -(-1)(2) = -2$

Figure 8.7: picture of back prop

8.5 Breadth versus Depth

At this point, you've seen how to train two-layer networks and how to train arbitrary networks. You've also seen a theorem that says that two-layer networks are universal function approximators. This begs the question: if two-layer networks are so great, why do we care about deeper networks?

To understand the answer, we can borrow some ideas from CS theory, namely the idea of **circuit complexity**. The goal is to show that there are functions for which it might be a "good idea" to use a deep network. In other words, there are functions that will require a huge number of hidden units if you force the network to be shallow, but can be done in a small number of units if you allow it to be deep. The example that we'll use is the **parity function** which, ironically enough, is just a generalization of the XOR problem. The function is defined over binary inputs as:

$$parity(\boldsymbol{x}) = \sum_d \boldsymbol{x}_d \mod 2 \tag{8.12}$$

$$= \begin{cases} 1 & \text{if the number of 1s in } \boldsymbol{x} \text{ is odd} \\ 0 & \text{if the number of 1s in } \boldsymbol{x} \text{ is even} \end{cases} \tag{8.13}$$

It is easy to define a circuit of depth $\mathcal{O}(\log_2 D)$ with $\mathcal{O}(D)$-many gates for computing the parity function. Each gate is an XOR, arranged in a complete binary tree, as shown in Figure 8.8. (If you want to disallow XOR as a gate, you can fix this by allowing the depth to be doubled and replacing each XOR with an AND, OR and NOT combination, like you did at the beginning of this chapter.)

Figure 8.8: `nnet:paritydeep`: deep function for computing parity

This shows that if you are allowed to be deep, you can construct a circuit with that computes parity using a number of hidden units that is linear in the dimensionality. So can you do the same with shallow circuits? The answer is no. It's a famous result of circuit complexity that parity requires exponentially many gates to compute in constant depth. The formal theorem is below:

Theorem 10 (Parity Function Complexity)**.** *Any circuit of depth $K < \log_2 D$ that computes the parity function of D input bits must contain $\mathcal{O}e^D$ gates.*

This is a very famous result because it shows that constant-depth circuits are less powerful that deep circuits. Although a neural network isn't exactly the same as a circuit, the is generally believed that the same result holds for neural networks. At the very least, this gives a strong indication that depth might be an important consideration in neural networks.

? What is it about neural networks that makes it so that the theorem about circuits does not apply directly?

One way of thinking about the issue of breadth versus depth has to do with the number of *parameters* that need to be estimated. By the heuristic that you need roughly one or two examples for every parameter, a deep model could potentially require exponentially fewer examples to train than a shallow model!

This now flips the question: if deep is potentially so much better, why doesn't everyone use deep networks? There are at least two answers. First, it makes the **architecture selection** problem more significant. Namely, when you use a two-layer network, the only hyperparameter to choose is how many hidden units should go in the middle layer. When you choose a deep network, you need to choose how many layers, and what is the width of all those layers. This can be somewhat daunting.

A second issue has to do with training deep models with back-propagation. In general, as back-propagation works its way down through the model, the sizes of the gradients shrink. You can work this out mathematically, but the intuition is simpler. If you are the beginning of a very deep network, changing one single weight is unlikely to have a significant effect on the output, since it has to go through so many other units before getting there. This directly implies that the derivatives are small. This, in turn, means that back-propagation essentially never moves far from its initialization when run on very deep networks.

? While these small derivatives might make training difficult, they might be *good* for other reasons: what reasons?

Finding good ways to train deep networks is an active research area. There are two general strategies. The first is to attempt to initialize the weights better, often by a **layer-wise** initialization strategy. This can be often done using unlabeled data. After this initialization, back-propagation can be run to tweak the weights for whatever classification problem you care about. A second approach is to use a more complex optimization procedure, rather than gradient descent. You will learn about some such procedures later in this book.

8.6 Basis Functions

At this point, we've seen that: (a) neural networks can mimic linear functions and (b) they can learn more complex functions. A reasonable question is whether they can mimic a KNN classifier, and whether they can do it efficiently (i.e., with not-too-many hidden units).

A natural way to train a neural network to mimic a KNN classifier is to replace the sigmoid link function with a **radial basis function** (RBF). In a **sigmoid network** (i.e., a network with sigmoid links), the hidden units were computed as $h_i = \tanh(\boldsymbol{w}_i, \boldsymbol{x}\cdot)$. In an **RBF network**, the hidden units are computed as:

$$h_i = \exp\left[-\gamma_i \,||\boldsymbol{w}_i - \boldsymbol{x}||^2\right] \tag{8.14}$$

In other words, the hidden units behave like little Gaussian "bumps" centered around locations specified by the vectors $\boldsymbol{w}_i$. A one-dimensional example is shown in Figure 8.9. The parameter γ_i specifies the *width* of the Gaussian bump. If γ_i is large, then only data points that are really close to $\boldsymbol{w}_i$ have non-zero activations. To distinguish sigmoid networks from RBF networks, the hidden units are typically drawn with sigmoids or with Gaussian bumps, as in Figure 8.10.

Figure 8.9: `nnet:rbfpicture`: a one-D picture of RBF bumps

Figure 8.10: `nnet:unitsymbols`: picture of nnet with sigmoid/rbf units

Training RBF networks involves finding good values for the Gassian widths, γ_i, the centers of the Gaussian bumps, $\boldsymbol{w}_i$ and the connections between the Gaussian bumps and the output unit, $\boldsymbol{v}$. This can all be done using back-propagation. The gradient terms for $\boldsymbol{v}$ remain unchanged from before, the the derivates for the other variables differ (see Exercise **??**).

One of the big questions with RBF networks is: where should the Gaussian bumps be centered? One can, of course, apply back-propagation to attempt to find the centers. Another option is to specify them ahead of time. For instance, one potential approach is to have one RBF unit per data point, centered on that data point. If you carefully choose the γs and vs, you can obtain something that looks nearly identical to distance-weighted KNN by doing so. This has the added advantage that you can go futher, and use back-propagation to *learn* good Gaussian widths (γ) and "voting" factors ($\boldsymbol{v}$) for the nearest neighbor algorithm.

? Consider an RBF network with one hidden unit per training point, centered at that point. What bad thing might happen if you use back-propagation to estimate the γs and $\boldsymbol{v}$ on this data if you're not careful? How could you be careful?

8.7 Exercises

Exercise 8.1. TODO. . .