Machine Learning Notes

Karthik Seetharaman

Spring 2021

Contents

1 Chapter 1: Stochastic Gradient Descent

§1.1 Perceptrons

A **neural network** is how computers learn to do tasks.

Definition 1.1.1. A **perceptron** is an artificial neuron that takes in several binary inputs $x_1, x_2, \ldots, x_n$ and outputs a single binary output.

Definition 1.1.2. Each input x_i has some **weight** w_i.

A perceptron always outputs 0 or 1. In order to decide what it outputs, we assign some **threshold value** to the sum $\sum w_i x_i$ over all inputs to the perceptron. If this sum is above the threshold value, the perceptron outputs 1, else it outputs 0.

One perceptron alone doesn't have much decision-making power, but a network of perceptrons can make some complex decisions.

We can draw a network of perceptrons by representing each one as a circle, with input and output arrows to each one. Multiple output arrows from one perceptron simply means the output from that perceptron is being used as an input to many other perceptrons.

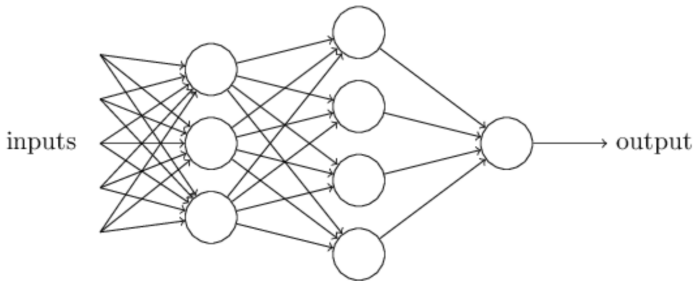

Figure 1.1: A network of perceptrons capable of making more complex decisions. This can, of course, be expanded further.

In order to make this less cumbersome, we write w and x to be the vectors of weights and inputs respectively.

Definition 1.1.3. Let the **bias** b of a perceptron be -1 times its threshold value.

This allows us to write the output of a perceptron as 0 if $w \cdot x + b \leq 0$, and 1 otherwise.

Lemma 1.1.4

We can use a network of perceptrons to compute any logical function.

The idea of machine learning is to continually change the weights and biases of a neural network in order to better complete tasks. However, perceptrons are not the best way to do this.

§1.2 Sigmoid Neurons

In order to actually learn with networks, we would like a small change in weight or bias to map to a small change in the output. This is not the case with perceptrons, as they have discrete outputs. Thus, we need a new type of neuron.

Definition 1.2.1. A **sigmoid neuron** is an artificial neuron modeled by the **sigmoid function** $\sigma(w \cdot x + b)$, where

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

Remark 1.2.2. Consider the asymptotic behavior of the sigmoid neuron. As z gets large, the neuron gets close to 1, and as z gets small, the neuron gets close to 0. This means the asymptotic behavior of the sigmoid neuron is similar to the behavior of the perceptron. This is further shown in Figure 1.2, where one can see its similarity at the edges to a step function.

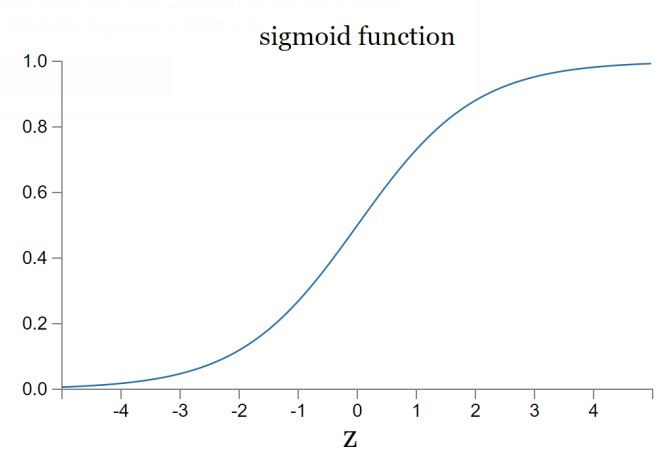

Figure 1.2: The graph of the sigmoid function. Notice the similarity to the step function near the ends.

More rigorously, if o is the output of the sigmoid neuron:

Lemma 1.2.3

$$\Delta o \approx \sum_j \frac{\partial o}{\partial w_j} \Delta w_j + \frac{\partial o}{\partial b} \Delta b$$

The continuous output of sigmoid neurons does not lend itself well to discrete answers, but this can be dealt with by using threshold values for the output of the neuron, similar to the idea of bias.

§1.2.1 Sigmoid Neurons Simulating Perceptrons I

We would like to show that, when all weights and biases are multiplied by a positive constant $c > 0$, the behavior of the network stays the same. Note that $w \cdot x + b$ becomes $c(w \cdot x + b)$ when all weights and biases are multiplied by c. Since c is positive, $w \cdot x + b > 0$ iff $c(w \cdot x + b) > 0$ and $w \cdot x + b \leq 0$ iff $c(w \cdot x + b) \leq 0$. This means each individual

neuron in the network stays the same, which means the behavior of the network stays the same, as desired.

§1.2.2 Sigmoid Neurons Simulating Perceptrons II

Again, consider one individual neuron. If $w \cdot x + b > 0$, $\lim_{c \to \infty} c(w \cdot x + b) = \infty$, and the sigmoid function approaches 1 as its input goes to infinity - this is the same behavior as a perceptron. Similarly, if $w \cdot x + b < 0$, then $\lim_{c \to \infty} c(w \cdot x + b) = -\infty$, and the sigmoid function approaches 0 as its input goes to negative infinity - this is, once again, the same behavior as a perceptron, as desired.

If $w \cdot x + b = 0$, then $c(w \cdot x + b) = 0$ and $\sigma(c(w \cdot x + b)) = \frac{1}{2}$, regardless of the value of the constant c.

§1.3 The Architecture of Neural Networks

Definition 1.3.1. The **input layer** consists of all the input neurons and the **output layer** consists of all the output neurons. The **hidden layers** are the layers in between. Consult Figure 1.3 for an example.

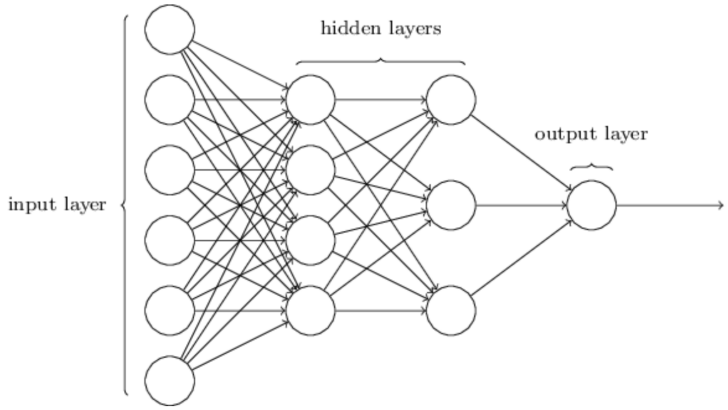

Figure 1.3: This is an example of a neural network with four layers, the middle two of which are hidden.

Designing the input and output layers of a neural network isn't very difficult, but designing the hidden layers can be.

Definition 1.3.2. A **feedforward neural network** is a neural network where all the outputs are used as inputs in the next layer.

All networks discussed here are feedforward neural networks.

§1.4 Learning With Gradient Descent

When learning with a neural network, we would like a way to quantify how well we are doing. This motivates us to introduce a **cost function**. We will use the following cost function for now:

Definition 1.4.1. The **quadratic cost function** is defined by

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a||^2.$$

Here, w and b are the collection of all weights and biases in the network respectively, x is some input (so the function sums over all inputs), n is the number of inputs, $y(x)$ is the desired output for some input x, and a is what the network outputs when x is input. This function is also sometimes known as a **mean square error**.

Remark 1.4.2. This function satisfies the basic constraints for a cost function. It is small when many network outputs are equal to the desired output, and large when few network outputs are equal to the desired output.

The aim of our training function is now to optimize the weights and biases in such a way that $C(w,b)$ is minimized. We can do this with something known as **gradient descent**.

To get the idea of gradient descent, consider a general case of trying to find the global minimum of a function $C(v)$ with inputs $v_1, v_2, \ldots, v_n$. The partial derivatives of v give us a local picture of how C looks, which gives us an idea of how to modify the inputs in order to locally decrease C. Repeating this should, theoretically, put us somewhere close to the global minimum of C. This is the key idea of gradient descent.

More mathematically, let v be the vector of inputs. Then, recall that $\Delta C \approx \nabla C \cdot \Delta v$. In order to make ΔC negative, choose $\Delta v = -\eta \nabla C$, where η is some small positive number (known as the **learning rate** in the context of neural networks). Then,

$$\Delta C \approx \nabla C \cdot \Delta v = \nabla C \cdot (-\eta \nabla C) = -\eta ||\nabla C||^2 < 0,$$

as desired. This gives us our algorithm: keep computing Δv in terms of ∇C and changing by that amount until we get close to a global minimum.

Remark 1.4.3. The choice of η is a subtle part of gradient descent. If it is too large, the approximation of ΔC will not be good. If it is too small, the algorithm will run too slowly.

Remark 1.4.4. In some sense, gradient descent is "optimal". If we constrain $||\Delta v|| = \epsilon$, then the choice for Δv that decreases C the most is actually $\eta \nabla C$, where $\eta = \frac{\epsilon}{||\nabla C||}$.

To apply gradient descent to a neural network, replace the vector of inputs with the vectors of weights and biases, and apply the same change rule to both using the quadratic cost function.

In practice, calculating the gradient for every single input then averaging them all is very slow. This motivates us to define the following:

Definition 1.4.5. The **stochastic gradient descent** algorithm is gradient descent, but with a small twist: instead of calculating the exact gradient, we calculate the exact gradient of a small batch of training examples then use this to approximate the whole thing.

Let's rigorize stochastic gradient descent:

Definition 1.4.6. A **mini-batch** of size m is a small subset of inputs $x_1, x_2, \ldots, x_m$.

Then, the idea is that

$$\frac{\sum_{j=1}^{m} \nabla C_{x_j}}{m} \approx \frac{sum_{j=1}^{n} \nabla C_{x_j}}{n} = \nabla C.$$

Picking a good mini-batch size will result in a lot of speed up for the neural network.

§1.4.1 Proving Remark 1.4.4

From the Cauchy-Schwarz Inequality,

$$||\Delta v||||\nabla C|| \geq |\Delta v \cdot \nabla C|.$$

Since $||v|| = \epsilon$, we have $|\Delta v \cdot \nabla C| \leq \epsilon ||\nabla C||$. The minimum value of $\Delta v \cdot \nabla C$ is then $-\epsilon ||\nabla C||$, which leads to $\Delta v \geq \frac{\epsilon \nabla C}{||\nabla C||}$, as desired.

§1.4.2 One-Dimensional Gradient Descent

Gradient descent, in this case, is just going to the end of a line by continuously moving towards it.

2 Chapter 2: Backpropagation

The backpropagation algorithm is how we compute the gradient of the cost function quickly.

§2.1 A Matrix-Based Approach to Computing The Output of a Neural Network

We first introduce some notation. Let w^l_{jk} denote the weight of the connection from the kth neuron in the $(l-1)$th layer to the jth neuron in the lth layer. Refer to Figure 2.1 for an example of this notation.

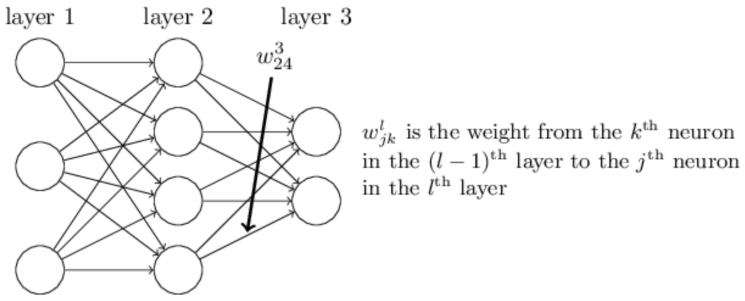

Figure 2.1: An example of the above weight notation.

We similarly define b^l_j to be the bias of the jth neuron in the lth layer and a^l_j to be the activation of the jth neuron in the lth layer. We can also define the **weight matrix** w^l, **bias vector** b^l, and the **activation vector** a^l of each layer in the obvious way.

Definition 2.1.1. To **vectorize** a function f, we apply it to every element of a vector.

We then have the following important equation:

$$a^l = \sigma(a^{l-1}w^l + b^l).$$

Definition 2.1.2. The **weighted input** is the vector $z^l = a^{l-1}w^l + b^l$. This can, of course, be componentized.

§2.2 The Four Equations of Backpropagation

Definition 2.2.1. The **error** δ^l_j of the jth neuron the lth layer is defined as

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}.$$

The first equation is an equation for the error in the output layer:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j).$$

This is the component form, so we can rewrite the equation in vector form with the Hadamard product:

$$\delta^L = \nabla_a C \circ z^L.$$

Remark 2.2.2. Everything in that equation is easily computed.

The second equation is determining δ^l from δ^{l+1} - moving backwards. We have

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l).$$

Combining this equation with the first allows us to compute the error at any layer in the network.

The third equation of backpropagation is quite elegant:

$$\delta^l_j = \frac{\partial C}{\partial b^l_j}.$$

Thus, we can compute the error at the lth layer with the first two equations, which gives us free information about the change in the biases through this equation.

The final equation we need is the following:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j.$$

§2.3 Proving The Four Equations of Backpropagation

We will prove each equation componentwise.

Recall that $\delta^L_j = \frac{\partial C}{\partial z^L_j}$. By the chain rule, this is

$$\sum_k \frac{\partial C}{\partial a^L_k} \frac{\partial a^L_k}{\partial z^L_j},$$

summing over all neurons in the output layer. As $a^L_j = \sigma(z^L_j)$ and none of the other activations depend on z^L_j, the sum becomes

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j),$$

which is the first equation componentized.

To prove the second equation, once again begin with $\delta^l_j = \frac{\partial C}{\partial z^l_j}$, and instead write

$$\delta^l_j = \sum_k \frac{\partial C}{\partial z^{l+1}_k} \frac{\partial z^{l+1}_k}{\partial z^l_j} = \sum_k \delta^{l+1}_k \frac{\partial z^{l+1}_k}{\partial z^l_j}.$$

However, we have that

$$z^{l+1}_k = \sum_j w^{l+1}_{kj} a^l_j + b^{l+1}_k = \sum_j w^{l+1}_{kj} \sigma(z^l_j) + b^{l+1}_k,$$

which, when differentiated, yields

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'(z_j^l).$$

Thus,

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1}\sigma'(z_j^l),$$

which is the second equation componentized.

To prove the third equation, write $b_j^l = z_j^l - \sum_k w_{jk}^l a_k^{l-1}$, so that $\frac{\partial b_j^l}{\partial z_j^l} = 1$. Then,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial b_k^l}\frac{\partial b_k^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l},$$

since all other terms of the sum disappear as only b_j^l depends on z_j^l.

To prove the fourth equation, recall that $z_j^l = \sum_m w_{jm}^l a_m^{l-1} + b_j^l$. Taking a specific input neuron k to the jth neuron in layer l, we get that $\frac{\partial w_{jk}^l}{\partial z_j^l} = \frac{1}{a_k^{l-1}}$. Then, using the chain rule with respect to a specific weight, we get

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial w_{jk}^l}\frac{\partial w_{jk}^l}{\partial z_j^l} = \frac{\partial C}{\partial w_{jk}^l}\left(\frac{1}{a_k^{l-1}}\right),$$

which gives the fourth equation when rearranged.

§2.4 The Backpropagation Algorithm

1. First, set the input x to get the activation at the first layer a^1.
2. Until the output layer, compute $a^l = w^l a^{l-1} + b^l$ and $z^l = \sigma(a^l)$.
3. Compute the error at the output layer $\delta^L = \nabla_a(C) \circ \sigma'(z^L)$.
4. Backpropagate and calculate $\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l)$.
5. Calculate the outputs: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

3 Chapter 3: Improving The Way Neural Networks Learn

§3.1 The Cross-Entropy Cost Function

Let's first introduce some motivation for introducing a new cost function. The idea is that, when the initial output is very badly wrong, the quadratic cost function does not learn quickly as we would expect.

To see this, say $x = 1$ is our input and $y = 0$ is the desired output. We have $C = \frac{(y-a)^2}{2}$. Recall that $a = (z) = (wx + b)$, so

$$\frac{\partial C}{\partial w} = a'(z) = \frac{\partial C}{\partial b}.$$

When the neuron's output is very wrong and close to 1, $\sigma'(z)$ is very close to 0, which gives us slow learning with the expressions for $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ given above (these expressions determine learning rate, roughly).

To attempt to address this issue, we define the **cross-entropy cost function**

$$C = -\frac{1}{n}\sum_x [y \log a + (1-y)\log(1-a)].$$

It is worth noting that this function is always positive (the sum is a sum of negative terms, times a negative number). It can also be seen that if the function outputs are similar to the desired output for a lot of inputs, the cost function is small. Thus, this function could conceivably be a cost function.

Now, let's see if this addresses learning slowdown. Substituting $a = \sigma(z)$ and computing the partial derivative of C gives:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z) - y).$$

However, we can note that $\sigma'(z) = \sigma(z)(1-\sigma(z))$, so the expression becomes

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x (\sigma(z) - y).$$

In other words, the learning rate is completely controlled by the error in the output, which is exactly what we want.

Remark 3.1.1. We can generalize the cross-entropy cost function to many neurons:

$$C = -\frac{1}{n}\sum_x \sum_j [y_j \log a_j^L + (1-y_j)\log(1-a_j^L)].$$

§3.1.1 Verifying the Sigmoid Relation

Let's quickly verify that $\sigma'(z) = \sigma(z)(1-\sigma(z))$. By the quotient rule,

$$\sigma'(z) = \frac{(1)'(1+e^{-z}) - (1)(1+e^{-z})'}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2} = \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}}{1+e^{-z}}\right) = \sigma(z)(1-\sigma(z)),$$

as desired.

§3.1.2 Maximizing the Cross-Entropy Cost Function

It's worth checking that the cross-entropy cost function is maximized when $a = y$ for all neurons. Consider a specific neuron (we will avoid indices, but recall that we are referring to only one neuron). Its contribution to the cost function is $y \log a + (1-y)\log(1-a)$. Call this function f. Then,

$$\frac{\partial f}{\partial a} = \frac{y}{a} - \frac{1-y}{1-a} = \frac{y-a}{a(1-a)}.$$

Thus, $a = y$ is the only critical value of f. It can be checked that this maximizes f, as desired.

§3.2 Softmax

Here, we define a new type of output layer for neural networks. Specifically, we define the **softmax function**:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}.$$

Note that this means the output activations of the last layer always sum to 1, and that they are positive. In other words, they form a probability distribution, which is nice: the activation of each neuron can be thought of as the probability that the output is that neuron.

In order to use the softmax layer, we use the **log-likelihood cost function**:

$$C = -\log(a_y^L).$$

It can be verified that this behaves like a cost function.

§3.3 Overfitting and Regularization

Definition 3.3.1. In general, a network is **overfitting** to the training data if the classification accuracy does not improve with the test data, even if it does with the training data. The network is not generalizing past the training data.

We need some way of identifying when overfitting is occurring, because it can be a very large problem if left unchecked.

Definition 3.3.2. We often test the network on **validation data** to test for overfitting, as to avoid accidentally overfitting paraetesr to the test data during testing.

Increasing the size of the training data is an easy way to prevent overfitting, but this is not always possible. Thus, we must examine other methods.

§3.3.1 L2 Regularization

L2 regularization is a method of reducing overfitting by modifying the cost function. Let C_0 be the original cost function (either quadratic or cross-entropy). Then, we add a term to get a new cost function:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2.$$

Here, λ is called the **regularization parameter**.

We can now compute

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$

and

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}.$$

Remark 3.3.3. The gradient descent rule for w changes by multiplying w by a factor of $1 - \frac{\eta\lambda}{n}$, referred to as a **weight decay**.

Remark 3.3.4. The idea behind L2 regularization is Occam's razor - smaller weights means a simpler model, which we should try to pick most of the time.

§3.3.2 L1 Regularization

L1 regularization is defined by

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|,$$

so that

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} \text{sgn}(w).$$

What this means is that the weights shrink by a constant amount, as opposed to L2 regularization where the weights shrink by a value proportional to the weight itself.

§3.3.3 Dropout

Instead of modifying the cost function, dropout works by modifying the network itself. The idea is that we randomly drop half the neurons from the hidden layer, as shown in Figure 3.1:

We use stochastic gradient descent on this network with some mini-batch, restore the dropped neurons, delete another half of the neurons, and repeat. Averaging over all of these removes some of the overfitting.

Remark 3.3.5. Combining dropout and L2 regularization can give good results.

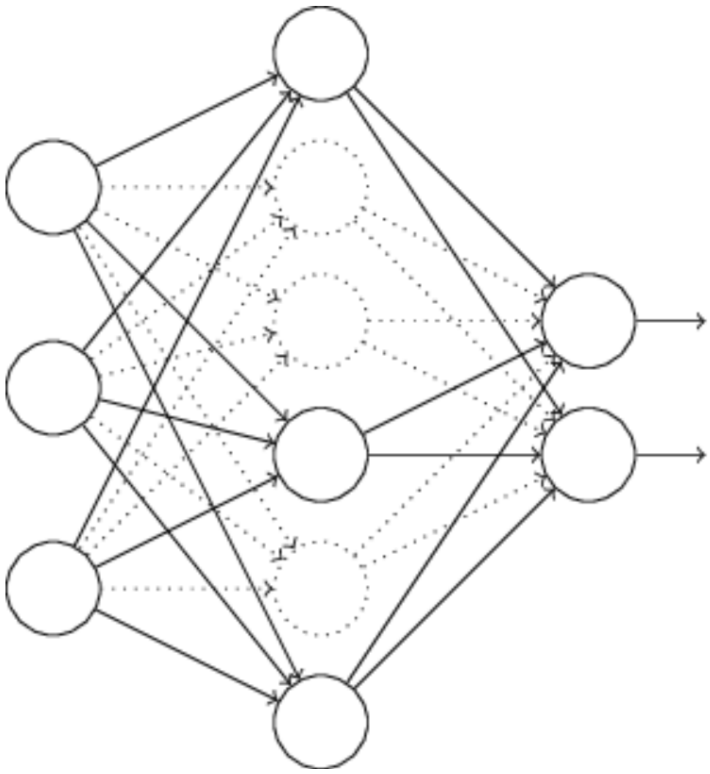

Figure 3.1: An example of dropping half the neurons in the hidden layer (the dropped ones are dotted).

§3.3.4 Weight Initialization

It is worth thinking about how we initialize the weights of our network, before we do any learning. Weights are typically chosen as Gaussian random variables with mean 0, but their standard deviation (which affects the bell curve) is worth analyzing. If we use a standard deviation of 1, it is likely that a lot of the weights will have large magnitude, which goes against the principle of regularization. It is beneficial to choose a standard deviation of $\frac{1}{\sqrt{n_{in}}}$, where n_{in} is the number of input neurons into the network. This will result in a more sharply peaked distribution, making weights closer to 0.