11 Feedforward Neural Network and Backpropagation

Feedforward Neural Networks (FFNNs) are perhaps the simplest kind of deep nets and are characterized by the following properties:

- There are nodes connected with no cycles.
- Nodes are partitioned into layers numbered 1 to k for some k. The nodes in the first layer receive input of the model and output some values. Then the nodes in layer $i + 1$ receive output of the nodes in layer i as their input and output some values. The output of the model can be computed with the output of the nodes in layer k.
- No outputs are passed back to lower layers.

Now, we only consider fully-connected layers — a special case of a layer in feedforward neural networks.

Definition 11.0.1 (Fully-Connected Layer)**.** *A **fully-connected layer** is a neural network layer in which all the nodes from one layer are fully connected to every node of the next layer.*

Note that not all layers of feedforward neural networks are necessarily fully-connected (a typical case is a Convolutional Neural Network, which we will explore in Chapter 12). However, feedforward neural networks with fully-connected layers are very common and also easy to implement.

11.1 Forward Propagation: An Example

Forward propagation refers to how the network converts a specific input to the output, specifically the calculation and storage of intermediate variables from the input layer to the output layer. In this section, we use concrete examples to motivate the topic. We will provide a more general formula in the next section. Readers who have a stronger background in math may feel to skip this section altogether.

11.1.1 *One Output Node*

We start with the network in Figure 11.1 as an example. The network receives three inputs x_1, x_2, x_3 and has a first hidden layer with two nodes $h_1^{(1)}, h_2^{(1)}$, a second hidden layer with two nodes $h_1^{(2)}, h_2^{(2)}$, and a final output layer with one node o. We assign the *ReLU* activation function to the hidden units, and define weights as shown in Figure 11.1.

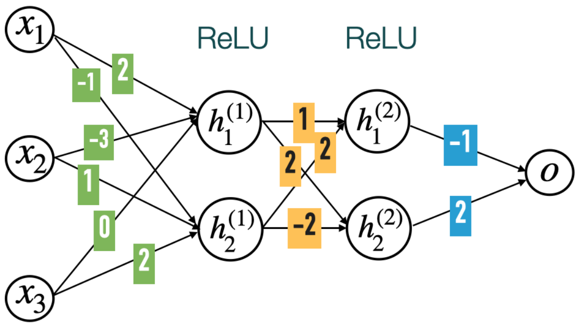

Figure 11.1: A sample feedforward neural network with two hidden layers and one output node.

The two hidden nodes in the first hidden layer are characterized by the following equations:

$$\begin{aligned} h_1^{(1)} &= ReLU(2x_1 - 3x_2) \\ h_2^{(1)} &= ReLU(-x_1 + x_2 + 2x_3) \end{aligned} \tag{11.1}$$

and the two hidden nodes in the second hidden layer are characterized by the following equations:

$$\begin{aligned} h_1^{(2)} &= ReLU(h_1^{(1)} + 2h_2^{(1)}) \\ h_2^{(2)} &= ReLU(2h_1^{(1)} - 2h_2^{(1)}) \end{aligned} \tag{11.2}$$

and the output node is characterized by the following equation:

$$o = -h_1^{(2)} + 2h_2^{(2)}$$

Therefore, if we know the input values x_1, x_2, x_3, we can first calculate the values $h_1^{(1)}, h_2^{(1)}$, then using these values, calculate $h_1^{(2)}, h_2^{(2)}$, and finally using these values, we can calculate the output o of the network.

Example 11.1.1. *If the provided input vector to the neural network in Figure 11.1 is* $\vec{\mathbf{x}} = (1, 1, 1)$, *we can calculate the first hidden layer as*

$$\begin{aligned} h_1^{(1)} &= ReLU(2 - 3) = 0 \\ h_2^{(1)} &= ReLU(-1 + 1 + 2) = 2 \end{aligned}$$

and the second hidden layer as

$$h_1^{(2)} = ReLU(0 + 2 \cdot 2) = 4$$
$$h_2^{(2)} = ReLU(0 - 2 \cdot 2) = 0$$

and the output as

$$o = -4 + 0 = -4$$

11.1.2 *Multiple Output Nodes*

Networks can have more than one output node. An example is the network in Figure 11.2.

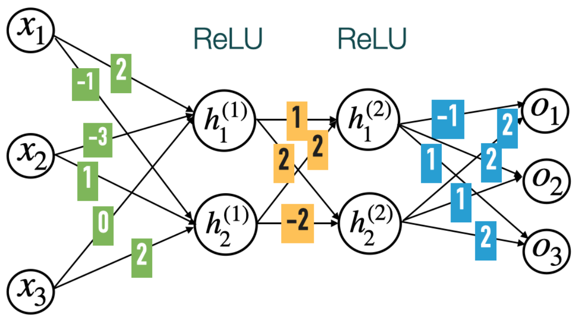

Figure 11.2: A sample feedforward neural network with two hidden layers and three output nodes.

The networks in Figure 11.1 and Figure 11.2 are the same except for the output layer; the former has one output node, while the latter has three output nodes. Now the output values of the network in Figure 11.2 can be calculated as:

$$\begin{aligned} o_1 &= -h_1^{(2)} + 2h_2^{(2)} \\ o_2 &= 2h_1^{(2)} + h_2^{(2)} \\ o_3 &= h_1^{(2)} + 2h_2^{(2)} \end{aligned} \tag{11.3}$$

Recall from the previous Chapter 10 that a FFNN with multiple output nodes is used for multi-class classfication. After the naive output values are calculated, the output nodes will use the softmax activation function to transform the values into the probabilities for each of the three classes. That is, the probability for predicting each class will be calculated as:

$$\begin{aligned} \widehat{o}_1 &= softmax(o_1, o_2, o_3)_1 \\ \widehat{o}_2 &= softmax(o_1, o_2, o_3)_2 \\ \widehat{o}_3 &= softmax(o_1, o_2, o_3)_3 \end{aligned} \tag{11.4}$$

Example 11.1.2. *If the provided input vector to the neural network in Figure 11.1 is $\vec{\mathbf{x}} = (1, 1, 1)$, we can calculate the output layer as*

$$\begin{aligned} o_1 &= -4 + 0 = -4 \\ o_2 &= 2 \cdot 4 + 0 = 8 \\ o_3 &= 4 + 0 = 4 \end{aligned}$$

and the probabilities of each class as

$$\begin{aligned} \widehat{o}_1 &= softmax(-4, 8, 4)_1 = \frac{e^{-4}}{e^{-4} + e^8 + e^4} \simeq 0.00 \\ \widehat{o}_2 &= softmax(-4, 8, 4)_2 = \frac{e^{8}}{e^{-4} + e^8 + e^4} \simeq 0.98 \\ \widehat{o}_3 &= softmax(-4, 8, 4)_3 = \frac{e^{4}}{e^{-4} + e^8 + e^4} \simeq 0.02 \end{aligned}$$

11.1.3 *Matrix Notation*

Let $w_{i,j}^{(1)}$ be the weight between the i-th node $h_i^{(1)}$ in the first hidden layer and the j-th input x_j. Then (11.1) can be rewritten as

$$\begin{aligned} h_1^{(1)} &= ReLU(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 + w_{1,3}^{(1)} x_3) \\ h_2^{(1)} &= ReLU(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + w_{2,3}^{(1)} x_3) \end{aligned}$$

Notice that if we set $\vec{\mathbf{x}} = (x_1, x_2, x_3) \in \mathbb{R}^3$ and $\vec{\mathbf{h}}^{(1)} = (h_1^{(1)}, h_2^{(1)}) \in \mathbb{R}^2$ and define a matrix $\mathbf{W}^{(1)} \in \mathbb{R}^{2\times 3}$ where its (i, j) entry is $w_{i,j}^{(1)}$, then we can further rewrite (11.1) as [1]

$$\vec{\mathbf{h}}^{(1)} = ReLU\left(\mathbf{W}^{(1)} \vec{\mathbf{x}}\right) \tag{11.5}$$

[1] Here we interpret the vectors $\vec{\mathbf{x}}$, $\vec{\mathbf{h}}^{(1)}$ as a column vector, or equivalently a 3×1 matrix and a 2×1 matrix respectively. This will be a convention throughout this chapter.

where the *ReLU* function is applied element-wise.

Similarly, if we let $w_{i,j}^{(2)}$ be the weight between the i-th node $h_i^{(2)}$ in the second hidden layer and the j-th node $h_j^{(1)}$ in the first hidden layer, (11.2) can be rewritten as

$$\begin{aligned} h_1^{(2)} &= ReLU(w_{1,1}^{(2)} h_1^{(1)} + w_{1,2}^{(2)} h_2^{(1)}) \\ h_2^{(2)} &= ReLU(w_{2,1}^{(2)} h_1^{(1)} + w_{2,2}^{(2)} h_2^{(1)}) \end{aligned}$$

or in a matrix notation as

$$\vec{\mathbf{h}}^{(2)} = ReLU\left(\mathbf{W}^{(2)} \vec{\mathbf{h}}^{(1)}\right) \tag{11.6}$$

where $\vec{\mathbf{h}}^{(2)} = (h_1^{(2)}, h_2^{(2)}) \in \mathbb{R}^2$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{2\times 2}$ is a matrix whose (i, j) entry is $w_{i,j}^{(2)}$.

Next, if we let $w_{i,j}^{(o)}$ be the weight between the i-th output node o_i (before softmax) and the j-th node $h_j^{(2)}$ in the second hidden layer, (11.3) can be rewritten as

$$\begin{aligned} o_1 &= w_{1,1}^{(o)} h_1^{(2)} + w_{1,2}^{(o)} h_2^{(2)} \\ o_2 &= w_{2,1}^{(o)} h_1^{(2)} + w_{2,2}^{(o)} h_2^{(2)} \\ o_3 &= w_{3,1}^{(o)} h_1^{(2)} + w_{3,2}^{(o)} h_2^{(2)} \end{aligned}$$

or in a matrix notation as

$$\vec{\mathbf{o}} = \mathbf{W}^{(o)} \vec{\mathbf{h}}^{(2)} \tag{11.7}$$

where $\vec{\mathbf{o}} = (o_1, o_2, o_3) \in \mathbb{R}^3$ and $\mathbf{W}^{(o)} \in \mathbb{R}^{3\times 2}$ is a matrix whose (i,j) entry is $w_{i,j}^{(o)}$.

Finally, if we let $\vec{\hat{\mathbf{o}}} = (\hat{o}_1, \hat{o}_2, \hat{o}_3) \in \mathbb{R}^3$, then (11.4) can be rewritten as

$$\vec{\hat{\mathbf{o}}} = softmax(\vec{\mathbf{o}}) \tag{11.8}$$

We summarize the results above into the following matrix equations

$$\begin{aligned} \vec{\mathbf{h}}^{(1)} &= ReLU\left(\mathbf{W}^{(1)}\vec{\mathbf{x}}\right) \\ \vec{\mathbf{h}}^{(2)} &= ReLU\left(\mathbf{W}^{(2)}\vec{\mathbf{h}}^{(1)}\right) \\ \vec{\mathbf{o}} &= \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(2)} \\ \vec{\hat{\mathbf{o}}} &= softmax(\vec{\mathbf{o}}) \end{aligned} \tag{11.9}$$

Example 11.1.3. *If the provided input vector to the neural network in Figure 11.2 is* $\vec{\mathbf{x}} = (1,1,1)$, *we can calculate the first hidden layer as*

$$\vec{\mathbf{h}}^{(1)} = \mathbf{W}^{(1)}\vec{\mathbf{x}} = ReLU\left(\begin{bmatrix} 2 & -3 & 0 \\ -1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

and the second hidden layer as

$$\vec{\mathbf{h}}^{(2)} = \mathbf{W}^{(2)}\vec{\mathbf{h}}^{(1)} = ReLU\left(\begin{bmatrix} 1 & 2 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

and the output layer $\vec{\mathbf{o}}$ *(before the softmax) as*

$$\vec{\mathbf{o}} = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(2)} = \begin{bmatrix} -1 & 2 \\ 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 4 \\ 0 \end{bmatrix} = \begin{bmatrix} -4 \\ 8 \\ 4 \end{bmatrix}$$

The probability distribution $\vec{\hat{\mathbf{o}}}$ *of the three classes can then be calculated as*

$$\vec{\hat{\mathbf{o}}} = softmax(\vec{\mathbf{o}}) = \left(\frac{e^{-4}}{e^{-4}+e^{8}+e^{4}}, \frac{e^{8}}{e^{-4}+e^{8}+e^{4}}, \frac{e^{4}}{e^{-4}+e^{8}+e^{4}}\right)$$

11.2 Forward Propagation: The General Case

We now consider an arbitrary feedforward neural network with $L \geq 1$ layers. Let $\vec{\mathbf{x}} \in \mathbb{R}^{d_0}$ be the vector of d_0 *inputs* to the network. For $k = 1, 2, \ldots, L$, let $\vec{\mathbf{h}}^{(k)} = (h_1^{(k)}, h_2^{(k)}, \ldots, h_{d_k}^{(k)}) \in \mathbb{R}^{d_k}$ represent the d_k nodes of the *k-th hidden layer*. The L-th hidden layer is also known as the *output layer*, and we alternatively denote $d_0 = d_{in}$ and $d_L = d_{out}$ to emphasize that they are respectively the number of inputs and the number of output nodes.

Additionally, we consider $\mathbf{W}^{(k)} \in \mathbb{R}^{d_k \times d_{k-1}}$ to represent the weights for the k-th hidden layer. Its (i, j) entry is the weight between the i-th node $h_i^{(k)}$ of the k-th hidden layer and the j-th node $h_j^{(k-1)}$ of the $(k-1)$-th hidden layer. We also alternatively denote $\mathbf{W}^{(L)} = \mathbf{W}^{(o)}$ to emphasize that it represents the weights for the output layer.

Finally, let $f^{(k)}$ be the nonlinear activation function for layer k. For instance, consider the output layer. If $d_{out} = 1$ (*i.e.*, there is one output node), we can assume that $f^{(L)}$ is the identity function. On the other hand, if $d_{out} > 1$ (*i.e.*, there are multiple output nodes), we can assume that $f^{(L)}$ is the softmax function. It is also possible to use different activation functions for each layer.

With all these new notations in mind, we can express the nodes of layer k as:

$$\vec{\mathbf{h}}^{(k)} = f^{(k)}(\mathbf{W}^{(k)}\vec{\mathbf{h}}^{(k-1)})$$

for each $k = 1, 2, \ldots, L$.

If $d_{out} = 1$, we let $o = \mathbf{W}^{(L)}\vec{\mathbf{h}}^{L-1}$ denote the final output of the model. If $d_{out} > 1$, we let $\vec{\mathbf{o}} = \mathbf{W}^{(L)}\vec{\mathbf{h}}^{L-1}$ denote the output layer before the softmax and $\vec{\bar{\mathbf{o}}} = f^{(L)}(\vec{\mathbf{o}})$ denote the output layer after the softmax.

11.2.1 Number of Weights

We now briefly consider the number of weights in a feedforward network. There are $d_{in} \cdot d_1$ weights (or variables) for the first hidden layer. Similarly, there are $d_1 \cdot d_2$ weights for the second hidden layer. In total, the number of weights is $\sum\limits_{i=0}^{L-1} d_i \cdot d_{i+1}$.

Example 11.2.1. *The number of weights in the model in Figure 11.2 can be calculated as*

$$3 \times 2 + 2 \times 2 + 2 \times 3 = 16$$

11.2.2 What If We Remove Nonlinearity?

If we removed the nonlinear activation function *ReLU* in our model from (11.9), we would have the following forward propagation equations:

$$\vec{\mathbf{h}}^{(1)} = \mathbf{W}^{(1)}\vec{\mathbf{x}}$$
$$\vec{\mathbf{h}}^{(2)} = \mathbf{W}^{(2)}\vec{\mathbf{h}}^{(1)}$$
$$\vec{\mathbf{o}} = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(2)}$$
$$\vec{\tilde{\mathbf{o}}} = softmax(\vec{\mathbf{o}})$$

Notice that if we set $\mathbf{W}' = \mathbf{W}^{(o)}\mathbf{W}^{(2)}\mathbf{W}^{(1)} \in \mathbb{R}^{3\times3}$, then

$$\vec{\tilde{\mathbf{o}}} = softmax(\mathbf{W}'\vec{\mathbf{x}})$$

We have thus reduced our neural net to a standard multi-classification logistic regression model! As we have discussed the limitation of linear models earlier, this indicates the importance of having nonlinear activation functions between layers.

11.2.3 *Training Loss*

Just like we have defined a loss function for ML models so far, we also define an appropriate loss function for neural networks, where the objective of the network becomes finding a set of weights that minimize the loss. While there are many different definitions of loss functions, here we present two — one that is more appropriate when there is a single output node, and another that is more appropriate for multi-class classification.

When there is only one scalar node in the output layer (*i.e.*, $d_{out} = 1$), we can use a *squared error loss*, similar to the least squares loss from (1.4):

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} (y-o)^2 \qquad \text{(Squared Error Loss)}$$

where $\vec{\mathbf{x}} \in \mathbb{R}^{d_{in}}$ is the input vector, y is its gold value (*i.e.*, actual value in the training data), and $o = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(L-1)}$ is the final output (*i.e.*, prediction) of the neural network.

Example 11.2.2. *If the provided input vector to the neural network in Figure 11.1 is* $\vec{\mathbf{x}} = (1, 1, 1)$ *and the training output is* $y = 0$, *we can calculate the squared error loss as*

$$(y-o)^2 = (0-(-4))^2 = 16$$

When there are multiple nodes in the output layer (*e.g.*, for multi-class classification), we can use a loss function that is similar to the logistic loss. Recall that in logistic regression, we defined the logistic loss as a sum of log loss over a set of data points:

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} -\log \Pr[\text{label } y \text{ on input } \vec{\mathbf{x}}] \qquad \text{(4.5 revisited)}$$

where $y \in \{-1, 1\}$ denotes the gold label. For neural networks, we can analogously define the *cross-entropy loss*:

$$\sum_{(\vec{\mathbf{x}},y)\in\mathcal{D}} -\log \widehat{o}_y \qquad \text{(Cross-Entropy Loss)}$$

where $y \in \{1, \ldots, d_{out}\}$ denotes the gold label, and $\widehat{o}_y$ denotes the probability that the model assigns to class y — that is, the y-th coordinate of the output vector $\vec{\widehat{\mathbf{o}}} = softmax(\vec{\mathbf{o}})$ after applying the softmax function.

Example 11.2.3. *If the provided input vector to the neural network in Figure 11.2 is $\vec{\mathbf{x}} = (1, 1, 1)$ and the training output is $y = 2$, we can calculate the cross-entropy loss for this data point as*

$$-\log \widehat{o}_y = -\log \frac{e^8}{e^{-4} + e^8 + e^4} \simeq 4.02$$

11.3 Backpropagation: An Example

Just like in previous ML models we have learned, the process of training a neural network model involves three steps:

1. Defining an an appropriate loss function.
2. Calculating the gradient of the loss with respect to the training data and the model parameters.
3. Iteratively updating the parameters via the gradient descent algorithm.

But once a neural network grows in size, the second step of calculating the gradients starts to become a problem. Naively trying to calculate each of the gradients separately becomes inefficient. Instead, *Backpropagation* [2] is an efficient way to calculate the gradients with respect to the network parameters such that the number of steps for the computation is *linear* in the size of the nueral network.

[2] Reference: `https://www.nature.com/articles/323533a0`

The key idea is to perform the computation in a very specific sequence — from the output layer to the input layer. By the Chain Rule, we can use the already computed gradient value of a node in a higher layer in the computation of the gradient of a node in a lower layer. This way, the gradient values *propagate* back from the top layer to the bottom layer.

11.3.1 The Delta Method: Reasoning from First Principles

The main goal of backpropagation is to compute the partial derivative $\partial f / \partial w$ where f is the loss and w is the weight of an edge. This will allow us to apply the gradient descent algorithm and appropriately update the weight w. Students often find backpropagation a confusing idea, but it is actually just a simple application of Chain Rule in multivariate calculus.

In this subsection, we motivate the topic with the *Delta Method*, which is an intuitive way to compute $\partial f / \partial w$. We perturb a weight w by a small amount Δ and measure how much the output changes. In doing so, we also measure how the rest of the network changes. As we will see later, the process of computing the partial derivative of the form $\partial f / \partial w$ requires us to also compute the partial derivative of the form $\partial f / \partial h$ where h is the value at a node.

Readers who are familiar with Chain Rule can quickly browse through the rest of this subsection.

Example 11.3.1. *Consider the model from Figure 11.2 but now with the inputs $\vec{\mathbf{x}} = (3, 1, 2)$. We use the same notation for the nodes and the weights that we used throughout Section 11.1. The goal of this simple example is to illustrate what the derivatives mean.*

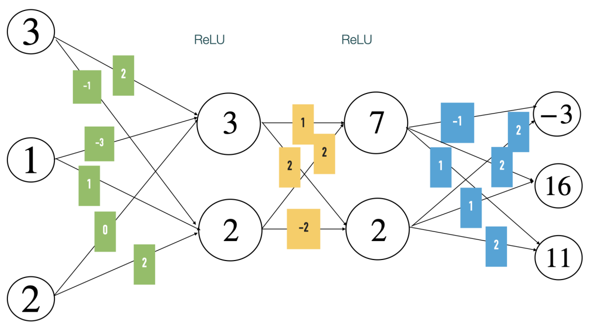

Figure 11.3: The model from Figure 11.2 with inputs $\vec{\mathbf{x}} = (3, 1, 2)$.

Suppose we want to take partial derivative of first output node o_1 with respect to the weight $w^{(1)}_{2,2}$ (i.e., the weight on the edge between the second input x_2 and the second node $h^{(1)}_2$ of first hidden layer). This is denoted as $\partial o_1 / \partial w^{(1)}_{2,2}$. Its definition involves considering how changing $w^{(1)}_{2,2}$ by an infinitesimal amount Δ changes o_1, whose current value is -3.

Adding Δ to $w^{(1)}_{2,2}$ will change the values of the first hidden layer to

$$h^{(1)}_1 = ReLU(2 \cdot 3 + (-3) \cdot 1 + 0 \cdot 2) = 3$$
$$h^{(1)}_2 = ReLU((-1) \cdot 3 + (1 + \Delta) \cdot 1 + 2 \cdot 2) = 2 + \Delta$$

Letting $\Delta \to 0$, we see that the rate of change of $h^{(1)}_1$ and $h^{(1)}_2$ with respect to change of $w^{(1)}_{2,2}$ is 0 and 1 respectively. In more formal terms, $\partial h^{(1)}_1 / \partial w^{(1)}_{2,2} = 0$ and $\partial h^{(1)}_2 / \partial w^{(1)}_{2,2} = 1$.

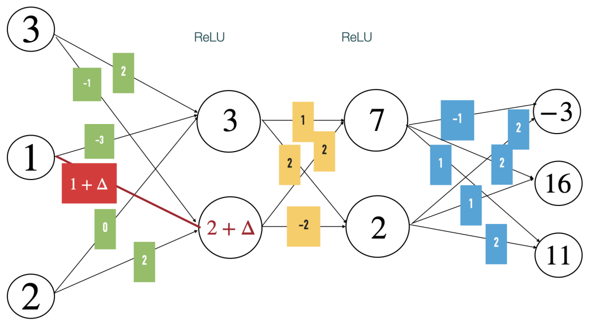

Figure 11.4: The model from Figure 11.2 with inputs $\vec{\mathbf{x}} = (3, 1, 2)$. When $w_{2,2}^{(1)}$ is changed by a small number Δ, the first hidden layer is also affected.

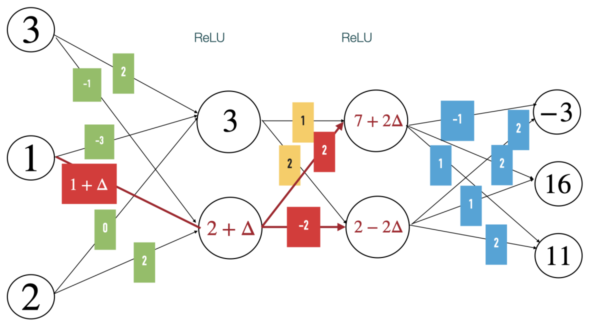

Figure 11.5: The model from Figure 11.2 with inputs $\vec{\mathbf{x}} = (3, 1, 2)$. When $w_{2,2}^{(1)}$ is changed by a small number Δ, the second hidden layer is also affected.

Using the updated values of the first hidden layer, the second hidden layer will be calculated as

$$h_1^{(2)} = ReLU(1 \cdot 3 + 2 \cdot (2 + \Delta)) = 7 + 2\Delta$$
$$h_2^{(2)} = ReLU(2 \cdot 3 + (-2) \cdot (2 + \Delta)) = 2 - 2\Delta$$

This shows that the rate of change of $h_1^{(2)}$ and $h_2^{(2)}$ with respect to change of $w_{2,2}^{(1)}$ is 2 and -2 respectively.

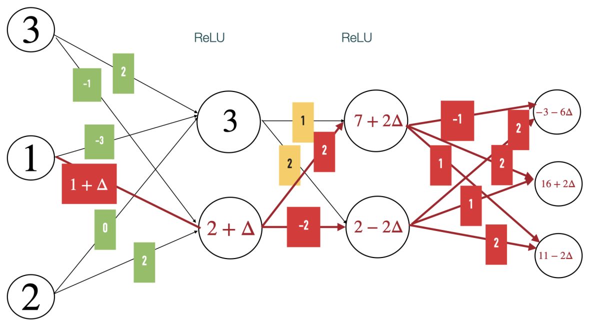

Figure 11.6: The model from Figure 11.2 with inputs $\vec{\mathbf{x}} = (3, 1, 2)$. When $w_{2,2}^{(1)}$ is changed by a small number Δ, the output layer is also affected.

Finally the output layer can now be calculated as

$$\begin{aligned}
o_1 &= (-1) \cdot (7 + 2\Delta) + 2 \cdot (2 - 2\Delta) = -3 - 6\Delta \\
o_2 &= 2 \cdot (7 + 2\Delta) + 1 \cdot (2 - 2\Delta) = 16 + 2\Delta \\
o_3 &= 1 \cdot (7 + 2\Delta) + 2 \cdot (2 - 2\Delta) = 11 - 2\Delta
\end{aligned}$$

This shows that the rate of change of o_1 with respect to change of $w^{(1)}_{2,2}$ is -6.

Example 11.3.2. *Now we consider the meaning of $\partial o_1 / \partial h^{(2)}_1$: how changing the value of $h^{(2)}_1$ by an infinitesimal Δ affects o_1. Note that this is a thought experiment that does not correspond to a change that is possible if the net were a physical object constructed of nodes and wires — the value of $h^{(2)}_1$ is completely decided by the previous layers and cannot be changed in isolation without touching the previous layers. However, treating these values as variables, it is possible to put on a calculus hat and and think about the rate of change of one with respect to the other.*

If only the value of $h^{(2)}_1$ is changed from 7 to $7 + \Delta$ in Figure 11.3, then o_1 can be calculated as

$$o_1 = (-1) \cdot (7 + \Delta) + 2 \cdot 2 = -3 - \Delta$$

which shows that $\partial o_1 / \partial h^{(2)}_1 = -1$.

Problem 11.3.3. *Following the calculations in Example 11.3.1 and Example 11.3.2, calculate $\partial o_1 / \partial h^{(2)}_2$, $\partial h^{(2)}_1 / \partial w^{(1)}_{2,2}$, and $\partial h^{(2)}_2 / \partial w^{(1)}_{2,2}$. Verify that*

$$\frac{\partial o_1}{\partial w^{(1)}_{2,2}} = \frac{\partial o_1}{\partial h^{(2)}_1} \cdot \frac{\partial h^{(2)}_1}{\partial w^{(1)}_{2,2}} + \frac{\partial o_1}{\partial h^{(2)}_2} \cdot \frac{\partial h^{(2)}_2}{\partial w^{(1)}_{2,2}}$$

Problem 11.3.4. *Following the calculations in Example 11.3.1 and Example 11.3.2, calculate $\partial h^{(2)}_1 / \partial h^{(1)}_2$, $\partial h^{(2)}_2 / \partial h^{(1)}_2$, and $\partial h^{(1)}_2 / \partial w^{(1)}_{2,2}$. Verify that*

$$\frac{\partial o_1}{\partial w^{(1)}_{2,2}} = \frac{\partial o_1}{\partial h^{(2)}_1} \cdot \frac{\partial h^{(2)}_1}{\partial h^{(1)}_2} \cdot \frac{\partial h^{(1)}_2}{\partial w^{(1)}_{2,2}} + \frac{\partial o_1}{\partial h^{(2)}_2} \cdot \frac{\partial h^{(2)}_2}{\partial h^{(1)}_2} \cdot \frac{\partial h^{(1)}_2}{\partial w^{(1)}_{2,2}} \tag{11.10}$$

Some readers may notice that (11.10) is just the result of chain rule in multivariable calculus. It shows that the effect of $w^{(1)}_{2,2}$ on o_1 is the sum of its effect through all possible paths that the values propagate through the network, and the amount of effect for each path can be calculated by multiplying the appropriate partial derivative between each layer.

In this section, we calculated by hand one partial derivative $\partial o_1 / \partial w^{(1)}_{2,2}$. But in general, to compute the loss gradient, we see below that we want to calculate the partial derivative of each output node o_i with respect to each weight in the network. Manually applying chain rule for each partial derivative as in (11.10) is too inefficient.[3] Instead,

[3] Putting on your COS 226 hat, you can check that the computational cost of this naive method scales quadratically in the size of the network.

in the next section, we will learn how to utilize matrix operations to combine the computation for multiple partial derivatives into one process.[4]

[4] This efficiency holds even without taking into account the fact that today's GPUs are optimized for fast matrix operations.

11.4 Backpropagation: The General Case

11.4.1 Jacobian Matrix

Suppose some vector $\vec{\mathbf{y}} = (y_1, y_2, \ldots, y_m) \in \mathbb{R}^m$ is a function of $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ — that is, there is a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ such that $\vec{\mathbf{y}} = f(\vec{\mathbf{x}})$, or equivalently, if there are m functions $f_i : \mathbb{R}^n \to \mathbb{R}$ for each $i = 1, 2, \ldots, m$ such that $y_i = f_i(\vec{\mathbf{x}})$.

Then the *Jacobian matrix* of $\vec{\mathbf{y}}$ with respect to $\vec{\mathbf{x}}$, denoted as $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$, is an $m \times n$ matrix whose (i, j) entry is the partial derivative $\partial y_i / \partial x_j$. Note that each entry of this matrix is itself a function of $\vec{\mathbf{x}}$. A bit confusingly, a Jacobian matrix is also often denoted as $\partial \vec{\mathbf{y}} / \partial \vec{\mathbf{x}}$ when it is clear from the context that $\vec{\mathbf{x}}, \vec{\mathbf{y}}$ are vectors and hence this object is not a partial derivative or gradient. [5]

[5] Note that the i-th row of the Jacobian matrix contains the gradient of y_i, *i.e.* the i-th coordinate of $\vec{\mathbf{y}}$.

The mathematical interpretation of the Jacobian matrix is that if we change $\vec{\mathbf{x}}$ such that each coordinate x_i is updated to $x_i + \delta_i$ for an infinitesimal value δ_i, then the output $\vec{\mathbf{y}}$ changes to $\vec{\mathbf{y}} + J(\vec{\mathbf{y}}, \vec{\mathbf{x}})\vec{\delta}$.

Example 11.4.1. *Suppose $\vec{\mathbf{y}}$ is a linear function of $\vec{\mathbf{x}}$ — that is, there exists a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ such that $\vec{\mathbf{y}} = \mathbf{A}\vec{\mathbf{x}}$. Then notice that y_i, the i-th coordinate of $\vec{\mathbf{y}}$, can be expressed as*

$$y_i = \mathbf{A}_{i,*}\vec{\mathbf{x}} = A_{i,1}x_1 + A_{i,2}x_2 + \ldots + A_{i,n}x_n$$

Notice that the partial derivative $\partial y_i / \partial x_j$ is equal to A_{ij}. This means that the (i, j) entry of the Jacobian matrix is the (i, j) entry of the matrix $\mathbf{A}$, and hence $J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = \mathbf{A}$.

Problem 11.4.2. *If $\vec{\mathbf{y}} \in \mathbb{R}^2$ is a function of $\vec{\mathbf{x}} \in \mathbb{R}^3$ such that*

$$\begin{aligned} y_1 &= 2x_1 - x_2 + 3x_3 \\ y_2 &= -x_1 + 2x_3 \end{aligned}$$

then what is the Jacobian matrix $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$?

Example 11.4.3. *If $\vec{\mathbf{x}} \in \mathbb{R}^n$ and $\vec{\mathbf{y}} = ReLU(\vec{\mathbf{x}}) \in \mathbb{R}^n$, then notice that*

$$\frac{\partial y_i}{\partial x_i} = \begin{cases} 1 & x_i > 0 \\ 0 & otherwise \end{cases}$$

We can also denote this with an indicator function $\mathbb{1}(x_i > 0)$. Also for any $j \neq i$, we see that $\partial y_i / \partial x_j = 0$. Therefore, the Jacobian matrix $J(\vec{\mathbf{y}}, \vec{\mathbf{x}})$ is a diagonal matrix whose entry down the diagonal is $\mathbb{1}(x_i > 0)$; that is

$$J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = diag(\mathbb{1}(\vec{\mathbf{x}} > 0))$$

where we take the indicator function element-wise to the vector $\vec{\mathbf{x}}$.

Definition 11.4.4 (Jacobian Chain Rule)**.** *Suppose vector $\vec{\mathbf{z}} \in \mathbb{R}^\ell$ is a function of $\vec{\mathbf{y}} \in \mathbb{R}^m$ and $\vec{\mathbf{y}}$ is a function of $\vec{\mathbf{x}} \in \mathbb{R}^n$, then by the chain rule, the Jacobian matrix $J(\vec{\mathbf{z}}, \vec{\mathbf{x}}) \in \mathbb{R}^{\ell \times m}$ is represented as the matrix product:*

$$J(\vec{\mathbf{z}}, \vec{\mathbf{x}}) = J(\vec{\mathbf{z}}, \vec{\mathbf{y}})J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) \tag{11.11}$$

In context of the feedforward neural network, each hidden layer is a function of the previous layer. Specifically, vector of activations of a hidden layer is a function of the vector of activations of the previous layer as well as of the trainable weights within the layer.

Example 11.4.5 (Gradient calculation for a single layer with ReLU's)**.** *If $\vec{\mathbf{x}} \in \mathbb{R}^n, \mathbf{A} \in \mathbb{R}^{m \times n}, \vec{\mathbf{y}} = \mathbf{A}\vec{\mathbf{x}} \in \mathbb{R}^m$ and $\vec{\mathbf{z}} = ReLU(\vec{\mathbf{y}}) \in \mathbb{R}^m$, then the Jacobian matrix $J(\vec{\mathbf{z}}, \vec{\mathbf{x}})$ can be calculated as*

$$J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = J(\vec{\mathbf{z}}, \vec{\mathbf{y}})J(\vec{\mathbf{y}}, \vec{\mathbf{x}}) = diag(\mathbb{1}(\mathbf{A}\vec{\mathbf{x}} > 0))\mathbf{A}$$

11.4.2 Backpropagation — Efficiency Using Jacobian Viewpoint

Now we return to backpropagation, and show how the Jacobian viewpoint allows computing the gradient of the loss (with respect to network parameters) with a number of mathematical operations (*i. e.*, additions and multiplications) proportional to the size of the fully connected net.

Recall that we want to find the weights $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(o)}$ that minimize the cross-entropy loss ℓ. To apply the standard/stochastic gradient descent algorithm, we need to find the partial derivative $\frac{\partial \ell}{\partial \mathbf{W}_{i,j}^{(k)}}$ of the loss function with respect to each weight $\mathbf{W}_{i,j}^{(k)}$ of each layer k.

To simplify notations, we introduce a new matrix $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}$ which has the same dimensions as $\mathbf{W}^{(k)}$ (*e.g.*, $\frac{\partial \ell}{\partial \mathbf{W}^{(1)}} \in \mathbb{R}^{2\times 3}$ in Figure 11.2) and the (i, j) entry of the matrix is:

$$\left(\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}\right)_{i,j} = \frac{\partial \ell}{\partial \mathbf{W}_{i,j}^{(k)}} \tag{11.12}$$

for any layer k. The matrix $\frac{\partial \ell}{\partial \mathbf{W}^{(k)}}$ will be called the gradient with respect to the weights of the k-th layer. [6] Now the update rule for the gradient descent algorithm can be written as the following:

$$\mathbf{W}^{(k)} \to \mathbf{W}^{(k)} - \eta \cdot \frac{\partial \ell}{\partial \mathbf{W}^{(k)}} \tag{11.13}$$

where η is the learning rate. Now the question remains as how to calculate these gradients. As the name "backpropagation" suggests, we will first compute the gradient of the loss ℓ with respect to the output nodes; we then inductively compute the gradient for the previous layers, until we reach the input layer.

[6] Alternatively, you can think of flattening $\mathbf{W}^{(k)}$ into a single vector, then finding the Jacobian matrix $\partial\ell/\partial\mathbf{W}^{(k)}$, and later converting it back to a matrix form.

1. Output Layer: First recall that the cross-entropy loss is

$$\begin{aligned}\ell &= -\log\left(\frac{e^{o_y}}{\sum\limits_{i=1}^{d_{out}} e^{o_i}}\right)\\ &= -\log(e^{o_y}) + \log\left(\sum_{i=1}^{d_{out}} e^{o_i}\right)\\ &= -o_y + \log\left(\sum_{i=1}^{d_{out}} e^{o_i}\right)\end{aligned}$$

where $y \in \{1, 2, \ldots, d_{out}\}$ is the ground truth value. Therefore, the gradient with respect to the output layer is

$$\frac{\partial \ell}{\partial o_i}_{1\leq i\leq d_{out}} = \begin{cases} -1 + \widehat{o}_i & y = i \\ \widehat{o}_i & y \neq i \end{cases}$$

To simplify notations, we introduce a *one-hot encoding vector* $\vec{\mathbf{e}}_y$, which has 1 only at the y-th coordinate and 0 everywhere else. Then, we can rewrite the equation above as: [7]

$$\frac{\partial \ell}{\partial \vec{\mathbf{o}}} = \left(\vec{\widehat{\mathbf{o}}} - \vec{\mathbf{e}}_y\right)^\intercal \in \mathbb{R}^{1\times d_{out}} \tag{11.14}$$

[7] Note that $\partial\ell/\partial\vec{\mathbf{o}}$, the term on the left hand side, is a Jacobian matrix in $\mathbb{R}^{1\times d_{out}}$. But $\vec{\widehat{\mathbf{o}}}$ and $\vec{\mathbf{e}}_y$, the terms on the right hand side, are both column vectors, or equivalently a $d_{out} \times 1$ matrix. We resolve the problem by taking the transpose.

This is the Jacobian matrix of the loss ℓ with respect to the output layer $\vec{\mathbf{o}}$.

2. Jacobian With Respect To Hidden Layer: We first compute $\partial\ell/\partial\vec{\mathbf{h}}^{(L-1)}$, the Jacobian matrix with respect to the last hidden layer before the output layer. Since $\vec{\mathbf{o}} = \mathbf{W}^{(o)}\vec{\mathbf{h}}^{(L-1)}$, we can apply the result from Example 11.4.1 and get

$$\begin{aligned}\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} &= \frac{\partial \ell}{\partial \vec{\mathbf{o}}} J(\vec{\mathbf{o}}, \vec{\mathbf{h}}^{(L-1)})\\ &= \frac{\partial \ell}{\partial \vec{\mathbf{o}}}\mathbf{W}^{(o)} \end{aligned}\tag{11.15}$$

Now as an inductive hypothesis, assume that we have already computed the gradient (or Jacobian matrix) $\partial\ell/\partial\vec{\mathbf{h}}^{(k+1)}$. We now compute $\partial\ell/\partial\vec{\mathbf{h}}^{(k)}$ using the result from Example 11.4.5.

$$\begin{aligned}\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} &= \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} J(\vec{\mathbf{h}}^{(k+1)}, \vec{\mathbf{h}}^{(k)})\\ &= \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0))\mathbf{W}^{(k+1)}\end{aligned}\tag{11.16}$$

3. Gradient With Respect to Weights: We first compute $\partial\ell/\partial\mathbf{W}^{(o)}$, the gradients with respect to the weights of the output layer. Notice that a particular weight $w_{i,j}^{(o)}$ is only used in computing o_i out of all output nodes:

$$o_i = w_{i,1}^{(o)} h_1^{(L-1)} + \ldots + w_{i,j}^{(o)} h_j^{(L-1)} + \ldots + w_{i,d_{L-1}}^{(o)} h_{d_{L-1}}^{(L-1)}$$

Therefore, the gradient with respect to $w_{i,j}^{(o)}$ can be calculated as

$$\frac{\partial \ell}{\partial w_{i,j}^{(o)}} = \frac{\partial \ell}{\partial o_i} \cdot \frac{\partial o_i}{\partial w_{i,j}^{(o)}} = \frac{\partial \ell}{\partial o_i} \cdot h_j^{(L-1)}$$

We can combine these results into the following matrix form

$$\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} = \left(\frac{\partial \ell}{\partial \vec{\mathbf{o}}}\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(L-1)}\right)^{\mathsf{T}} \tag{11.17}$$

Now as an inductive hypothesis, assume that we have already computed the gradient (or Jacobian) $\partial \ell / \partial \vec{\mathbf{h}}^{(k)}$. We now compute $\partial \ell / \partial \mathbf{W}^{(k)}$.

To do this, we introduce an intermediate variable $\vec{\mathbf{z}}^{(k)} = \mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)}$ such that $\vec{\mathbf{h}}^{(k)} = ReLU(\vec{\mathbf{z}}^{(k)})$. Then the gradient with respect to a particular weight $w_{i,j}^{(k)}$ can be calculated as

$$\frac{\partial \ell}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial z_i^{(k)}} \cdot \frac{\partial z_i^{(k)}}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial z_i^{(k)}} \cdot h_j^{(k-1)}$$

We can combine these results into the following matrix form

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{W}^{(k)}} &= \left(\frac{\partial \ell}{\partial \vec{\mathbf{z}}^{(k)}}\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(k-1)}\right)^{\mathsf{T}} \\
&= \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} J(\vec{\mathbf{h}}^{(k)}, \vec{\mathbf{z}}^{(k)})\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(k-1)}\right)^{\mathsf{T}} \\
&= \left(J(\vec{\mathbf{h}}^{(k)}, \vec{\mathbf{z}}^{(k)})\right)^{\mathsf{T}} \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}}\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(k-1)}\right)^{\mathsf{T}} \\
&= diag(\mathbb{1}(\mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)} > 0)) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}}\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(k-1)}\right)^{\mathsf{T}}
\end{aligned} \tag{11.18}$$

4. Full Backpropagation Process We summarize the results above into the following four steps:

1. Compute the Jacobian matrix with respect to the output layer, $\frac{\partial \ell}{\partial \vec{\mathbf{o}}} \in \mathbb{R}^{1 \times d_{out}}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{o}}} = \left(\vec{\mathbf{o}} - \vec{\mathbf{e}}_y\right)^{\mathsf{T}} \tag{(11.14) revisited}$$

2. Compute the Jacobian matrix with respect to the last hidden layer, $\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} \in \mathbb{R}^{1 \times d_{L-1}}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(L-1)}} = \frac{\partial \ell}{\partial \vec{\mathbf{o}}} \mathbf{W}^{(o)} \tag{(11.15) revisited}$$

Then, compute the gradient with respect to the output weights, $\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} \in \mathbb{R}^{d_{out} \times d_{L-1}}$:

$$\frac{\partial \ell}{\partial \mathbf{W}^{(o)}} = \left(\frac{\partial \ell}{\partial \vec{\mathbf{o}}}\right)^\top \left(\vec{\mathbf{h}}^{(L-1)}\right)^\top \qquad \text{((11.17) revisited)}$$

3. For each successive layer k, calculate the Jacobian matrix with respect to the k-th hidden layer $\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} \in \mathbb{R}^{1 \times d_k}$:

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} = \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0))\mathbf{W}^{(k+1)} \quad \text{((11.16) revisited)}$$

   Next, compute the gradient with respect to the $(k+1)$-th hidden layer weights $\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} \in \mathbb{R}^{d_{k+1} \times d_k}$:

$$\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} = diag(\mathbb{1}(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)} > 0)) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}}\right)^\top \left(\vec{\mathbf{h}}^{(k)}\right)^\top \qquad \text{((11.18) revisited)}$$

4. Repeat Step 3 until we reach the input layer.

Note that these instructions are based on a model that adopts the cross-entropy loss and the *ReLU* activation function. Using alternative losses and/or activation functions would result in a similar form, although the actual derivatives may be slightly different.

Problem 11.4.6. *(i) Show that if $\mathbf{A}$ is an $m \times n$ matrix and $\vec{\mathbf{h}} \in \mathbb{R}^n$ then computing $\mathbf{A}\vec{\mathbf{h}}$ requires mn multiplications and m additions. (ii) Using the previous part, argue that the number of arithmetic operations (additions or multiplications) in backpropagation algorithm on an FC net with ReLU activations is proportional to the number of parameters in the net.*

While the above calculation is in line with your basic algorithmic training, it doesn't exactly describe running time in modern ML environments with GPUs, since certain operations are parallelized, and compilers are optimized to run backpropagation as fast as possible.

11.4.3 *Using a Different Activation Function*

We briefly consider what happens if we choose a different activation function for the hidden layers. Consider the sigmoid activation function $\sigma(z) = \frac{1}{1+e^{-z}}$. Its derivative is given by:

$$\begin{aligned} \sigma'(z) &= \frac{1}{1+e^{-z}} \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \sigma(z) \cdot \left(\frac{e^{-z}}{1+e^{-z}}\right) \\ &= \sigma(z) \cdot (1 - \sigma(z)) \end{aligned} \qquad (11.19)$$

There is also the hyperbolic tangent function $\tanh(z) = \frac{e^{2z}-1}{e^{2z}+1}$.

Problem 11.4.7. *Compute $f'(z)$ for $f(z) = \tanh(z)$; show how $f'(z)$ can be written in terms of $f(z)$.*

Problem 11.4.8. *Say a neural network uses an activation function $f(z)$ at layer $k+1$ such that $f'(z)$ is a function of $f(z)$. That is, $f'(z) = g(f(z))$ for some function g. Then verify that (11.16, 11.18) can be rewritten as:*

$$\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k)}} = \frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}} diag(g(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)}))\mathbf{W}^{(k+1)}$$
$$\frac{\partial \ell}{\partial \mathbf{W}^{(k+1)}} = diag(g(\mathbf{W}^{(k+1)}\vec{\mathbf{h}}^{(k)})) \left(\frac{\partial \ell}{\partial \vec{\mathbf{h}}^{(k+1)}}\right)^{\mathsf{T}} \left(\vec{\mathbf{h}}^{(k)}\right)^{\mathsf{T}}$$

11.4.4 Final Remark

Directly following the steps of backpropagation is complicated and involves a lot of calculations. But remember that backpropagation is simply *computing gradients by the chain rule*. At a high level, we can think of the loss as a function of inputs and all the weights and note that backpropagation simply entails calculating derivatives with respect to each variable. The good news is that modern deep learning software does all the gradient calculations for users. All the model designer needs to do is to determine the neural network architecture (*e.g.*, choose number of layers, number of hidden units, and the activation functions).

One note of caution is that the loss function for deep neural nets is highly non-convex with respect to the parameters of the network. Just as we discussed in Chapter 3, the gradient descent algorithm is not guaranteed to find the actual minimizer in such situation, and the choice of the initial values of the parameters matter a lot.

11.5 Feedforward Neural Network in Python

In this section, we discuss how to write Python code to build neural network models and perform forward propagation and backpropagation. As usual, we use the *numpy* package to speed up computation. Additionally, we use the *torch* package to easily design and train the neural network.

```python
# import necessary packages
import numpy as np
import torch
import torch.nn as nn

# define the neural network
class Net(nn.Module):
    def __init__(self, input_size=2, hidden_dim1=2, hidden_dim2=3,
                                       hidden_dim3=2):
```

```
        super(Net, self).__init__()

        self.hidden1 = nn.Linear(input_size, hidden_dim1, bias=False)
        self.hidden1.weight = nn.Parameter(torch.tensor([[-2., 1.], [3., -1.
                                       ]]))

        self.hidden2 = nn.Linear(hidden_dim1, hidden_dim2, bias=False)
        self.hidden2.weight = nn.Parameter(torch.tensor([[0., 1.], [2., -1.]
                                       , [1., 2.]]))

        self.hidden3 = nn.Linear(hidden_dim2, hidden_dim3, bias=False)
        self.hidden3.weight = nn.Parameter(torch.tensor([[-1., 2., 1.], [3.,
                                        0., 0.]]))

        self.activation = nn.ReLU()

    # single step of forward propagation
    def forward(self, x):
        h1 = self.hidden1(x)
        h1 = self.activation(h1)
        h2 = self.hidden2(h1)
        h2 = self.activation(h2)
        h3 = self.hidden3(h2)
        return h3

net = Net()

# forward propagation with sample input
x = torch.tensor([3., 1.])
y_pred = net.forward(x)
print('Predicted value:', y_pred)

# backpropagation with sample input
loss = nn.functional.cross_entropy(y_pred.unsqueeze(0), torch.LongTensor([1]
                                       ))
loss.backward()
print(loss)
print(net.hidden1.weight.grad)
```

We start the code by importing all necessary packages.

```
import numpy as np
impoort torch
import torch.nn as nn
```

With *PyTorch*, we can design the architecture of any neural network by defining the corresponding *class*.

```
class Net(nn.Module):
    def __init__(self, input_size=2, hidden_dim1=2, hidden_dim2=3,
                                    hidden_dim3=2):
        super(Net, self).__init__()

        self.hidden1 = nn.Linear(input_size, hidden_dim1, bias=False)
        self.hidden1.weight = nn.Parameter(torch.tensor([[-2., 1.], [3., -1.
                                       ]]))

        self.hidden2 = nn.Linear(hidden_dim1, hidden_dim2, bias=False)
        self.hidden2.weight = nn.Parameter(torch.tensor([[0., 1.], [2., -1.]
                                       , [1., 2.]]))
        self.hidden3 = nn.Linear(hidden_dim2, hidden_dim3, bias=False)
        self.hidden3.weight = nn.Parameter(torch.tensor([[-1., 2., 1.], [3.,
                                            0., 0.]]))

        self.activation = nn.ReLU()

    def forward(self, x):
        h1 = self.hidden1(x)
        h1 = self.activation(h1)
        h2 = self.hidden2(h1)
        h2 = self.activation(h2)
        h3 = self.hidden3(h2)
        return h3
```

In the constructor, we define all the layers and activation functions we are going to use in the network. In particular, we specify that we need fully-connected layers by making instances of the *nn.Linear* class and that we need *ReLU* activation function by making an instance of the *nn.Relu* class. Then in the *forward()* function, we specify the order in which to apply the layers and activations. See Figure 11.7 for a visualization of this neural network architecture.

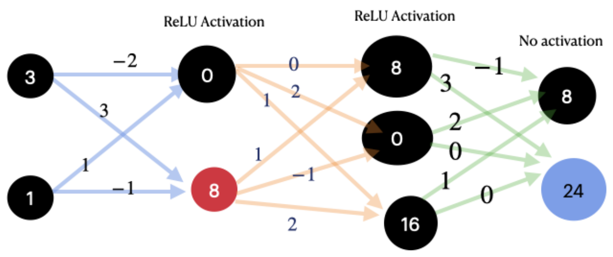

Figure 11.7: A sample feedforward neural network with two hidden layers and two output nodes.

We can simulate one step of forward propagation by calling the *forward()* function of the class *Net* we defined.

```
x = torch.tensor([3., 1.])
y_pred = net.forward(x)
print('Predicted value:', y_pred)
```

Similarly, we can implement backpropagation by specifying which loss function we want to use, and calling its *backward()* function.

```
loss = nn.functional.cross_entropy(y_pred.unsqueeze(0), torch.LongTensor([1]
                                    ))
loss.backward()
print(loss)
print(net.hidden1.weight.grad)
```

Each function call of *backward()* will evaluate the gradients of *loss* with respect to every parameter in the network. Gradients can be manually accessed through the following code.

```
print(net.hidden1.weight.grad)
```

Note that calling *backward()* multiple times will cause gradients to accumulate. While we do not update model weights in this code sample, it is important to periodically clear the gradient buffer when doing so to prevent unintended training. [8] We will discuss how to do this in the next chapter.

[8] For more information, see https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html