Chapter 17
Neural Networks & Deep Learning

17.1 What is Deep Learning?

When people talk about **deep learning** they are generally talking about a class of models called **neural networks** that are a loose approximation of how our brains work. Although deep learning has had a lot recent activity in the news, these types of models are not "new" as they have been used for around 50 years. Previously though, they fell in disfavor when simpler models, like the ones that we have already learned in this class, were already performing quite well. Recently, there has been a huge resurgence mainly due to the impressive accuracy these types of models could achieve on benchmark problems. In addition, with the increased amount of data and compute resources (GPUs for example) currently available, training neural networks for complex and large problems has become feasible.

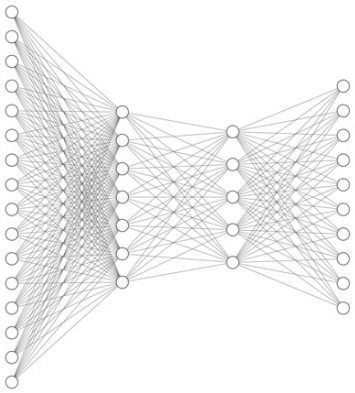

Figure 17.86: Graphical Representation of a Neural Network

17.2 One Unit of a Neural Network

Definition 17.1: Perceptron

A **perceptron** is a single layer neural network that consists of input values also known as the input layer, weights, biases, a summation, and finally an activation function. **Perceptron** used together in multiple layers are called neural networks.

Essentially, a perceptron is a graphical representation of a simple linear classifier. To review, a linear classifier has vector of weights and a bias.

Note that sometimes the bias term is written as part of the weights as w_0 but can also be represented using b for bias. The bias is often referred to as the y intercept in 2D space. This is purely notation and there is no difference in meaning between b and w_0 in this context.

We can more formally define a linear classifier's score as follows:

$$\text{Score}(x) = \sum_{j=1}^{d} w_j x[j] = w_0 + w_1 x[1] + w_q x[2] + ... + w_d x[d] \tag{17.48}$$

This is simply a linear equation which no different than the equation used in a linear regression. This "score", a weighted sum, can be turned into a classifier by using an **activation function**. For example, we could use the *sign* function that outputs 1 if the score is greater than 0 and 0 otherwise. We can see this represented below:

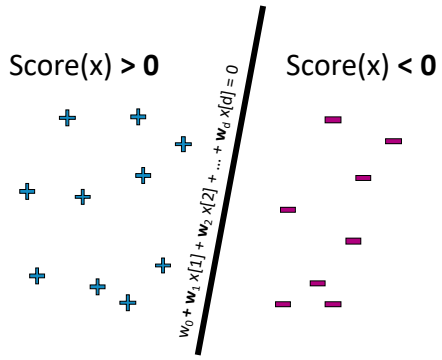

Figure 17.87: Linear Classifier

Definition 17.2: Activation Function

An **activation function** is a function that defines the output of a neuron from the score/output of a weighted sum. There are many different types of activation functions that can help a neural network's training and performance.

In practice, the *sign* function is usually not used as for two main reasons: it is not differentiable and it has no notion of confidence in its output. As the *sign* function only maps to 0 or 1 it is non-differentiable. Then for most classification outputs it would be ideal to have a measure of certainty. It is useful for both training and using a model to know if it has made a sure decision or is uncertain. For those reasons we usually look for activation functions which don't compromise on both of these. However, note that choice of an activation function is highly dependent on its use case. Below are listed a few commonly used activation functions in neural networks:

- **Sigmoid.** There are many different types of sigmoid functions. One for instance is the logistic function as shown above. This was historically extremely popular but has since been used less because the neuron's activation saturates, meaning the weights become increasingly large as the gradient gets smaller. In addition, it is not 0 centered, which can create issues in the gradient steps. Note that when this is applied on the output layer of a neural network this is called **softmax** as it can be interpreted as a class probability (a soft assignment).

$$g(x) = \frac{1}{1 + e^{-\text{Score}(x)}} \tag{17.49}$$

- **Hyperbolic Tangent.** This is also considered a sigmoid function and saturates similarily but has the benefit of being centered at 0.

$$g(x) = \tanh(x) \tag{17.50}$$

- **Rectified Linear Unit (ReLU).** This is the most popular activation function uses in neural networks as it is easy to implement, fast to compute, and has a true 0 value. However, neurons can "die off" as their outputs are 0. Note that there are variants of this which are "leaky" and "noisy" which can be beneficial depending on the use case.

$$g(x) = \max\{x, 0\} \tag{17.51}$$

- **Softplus.** Softplus is a smooth approximation of ReLU.

$$g(x) = \log(1 + \exp(x)) \tag{17.52}$$

The two components of a linear weighted sum and an activation function make up what we call a single **neuron**. Then combined with the input layer with a single neuron into the linear model we get a perceptron.

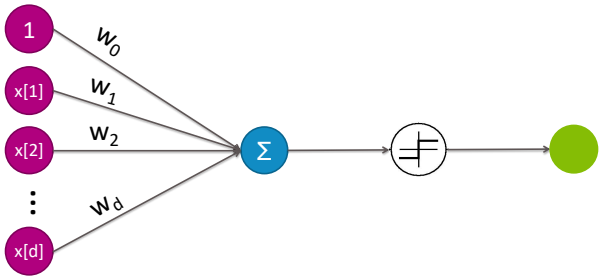

Figure 17.88: A perceptron.

17.3 Perceptron Learning

As we have learned, a perceptron is essential a linear model. We will take a look at two functions that a perception can learn. A perceptron can find weights which can solve these two problems with the 3 trained weights (bias, x_1 and x_2) and the *sign* activation function.

- **The OR Function.**

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Here is one solution: $w_0 = -0.5$, $w_1 = 1$, and $w_2 = 1$.

To solve this we simply find the weighted sum:

$$sign(\sum_{j=1}^{d} w_j x[j]) = sign(w_0 + w_1 * x_1 + w_2 * x_2) \tag{17.53}$$

Then let's check if this works for the second row of the table above using $x_1 = 0$ and $x_2 = 1$:

$$sign(-0.5 + 1 * 0 + 1 * 1) = sign(0.5) = 1 \tag{17.54}$$

The rest of these are also valid and can be checked if you wish. Clearly there are multiple solutions to this problem as there are only 4 data points to fit here but this is one that separates the data correctly.

- **The AND Function.**

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

  Here is one solution: $w_0 = -1.5, w_1 = 1,$ and $w_2 = 1$. Again, similar to the previous problem there are multiple solutions.

However, as you likely have predicted, a single perception has its limitations in that it can only fit linear classification problems. For instance, a single perceptron cannot fit to the XOR function because it is not linearly separable:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

In order to create a neural network complex enough for nonlinear problems we need to combine multiple perceptrons in layers.

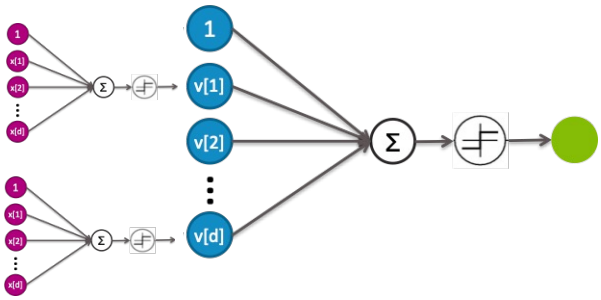

Figure 17.89: Perceptrons in Layers (a Neural Network)

Definition 17.3: Hidden Layer

A **hidden layer** is any layer in a neural network that is between the input and output layers where the neuron takes weighted inputs and produces an output using an activation function.

Let's design a two layer (one hidden layer) neural network that can fit the XOR function. We will start by breaking down the XOR function:

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } !x_2) \text{ OR } (!x_1 \text{ AND } x_2) \tag{17.55}$$

Using this we can design a neural network to fit XOR. We will use the first half of the OR statement as the first layer and the second half as the second layer. We can also refer to the first layer as the **input layer** and the second layer as the **hidden layer**.

We will define $v_1 = (x_1 \text{ AND } !x_2)$ and $v_2 = (!x_1 \text{ AND } x_2)$. We can now create a neural network which in the second layer uses the same OR perceptron we fit earlier. Then we define the first layer's weights to apply

the correct AND for both v_1 and v_2. We are also still using the *sign* activation function for all neurons. Combined together we have a valid XOR fit neural network.

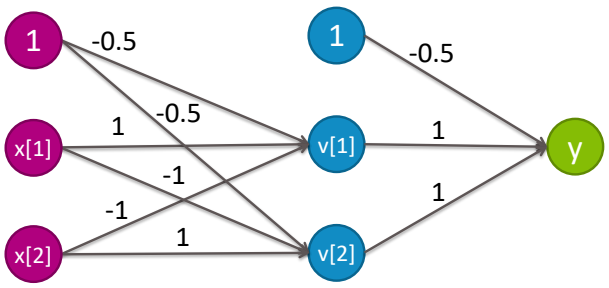

Figure 17.90: Two Layer XOR Solution

We have now defined our first two layer (one hidden layer) neural network. Keep in mind normally this neural network will be trained using data instead of assigned like above. Training a neural network will be discussed below.

We will now try the neural network's prediction by running a forward pass using the sample input $x_1 = 1$ and $x_2 = 0$.

Following the model as node v_1 we get $-0.5 + 1 * 1 + -1 * 0 = 0.5$ which is positive so node $v_1 = 1$. For node v_2 we get $-0.5 + -1 * 1 + 1 * 0 = -1.5$ which is negative so we $v_2 = 0$. In the hidden layer we sum to get $b + v_1 * 1 + v_2 * 1 = -0.5 + 1 * 1 + 0 * 1 = 0.5$ which again is positive so the output is 1 which matches XOR.

Surprisingly any two layer neural network can represent any function if enough nodes are allowed in the hidden layer. Although this is ideal in that we can fit complex functions well it has the consequence that neural networks are likely to overfit however there are some methods to try to avoid overfitting similar to other models. Having lots of training data is important along with regularization and/or dropout. Also, keeping the network shallower and with fewer nodes can help to avoid overfitting, going deeper only helps if you are very careful.

17.4 Applications

17.4.1 Regression or Classification?

We have showed that using a binary final output function (0 or 1) in neural network can be used for classification tasks like AND, OR, and XOR. For more complex classification tasks, like previously stated, we can use softmax to transform scores from the neural network into a probability output. This can be used in a variety of settings, such as in computer vision.

For instance, let us decide that we wish to classify images into one of 5 different classes with our newfound understanding of neural networks. We should design our neural network to have 5 output nodes corresponding to the 5 different classes. Then when using the softmax activation function on the output scores we can turn those 5 scores into 5 probabilities. Then as in this scenario we wish to just decide one classification, we can choose the class with the highest probability and assign the image to some class that way.

Example(s)

One example of where classification is used in neural networks is for the MNIST dataset. These are hand-written digit images that are labelled with the correct value. In a neural network design for

this there will be 10 nodes in the output layer corresponding to the probability of some image (the input) being classified as some digit (0-9). That means that number of classes should be equal to the number of outputs.

Neural networks are not limited to classification tasks though. For instance, in a regression task for a neural net, they will have one output node as the single number prediction.

17.4.2 Learning Features

As we have learned through using LASSO, it can be immensely helpful to decide which features are important to take use. Sometimes we don't know if the best features are even from the set of the current ones or if it would be better to use combinations of multiple of them. This is a key feature of neural networks that makes complex tasks much easier.

Previously, computer vision methods used hand crafted features to make decisions about images. For example, if trying to classify images of people it might look for the face, nose, mouth, etc. These are major generalizations though and not very easy to simply plug an image in and know these features directly. In addition this relies on coming up with these hand picked features which often we might not know what will be the best indicators for complex classification problems

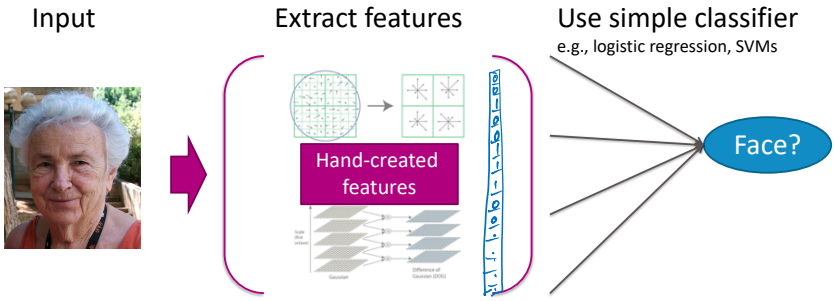

Figure 17.91: Manual Feature Selection in Computer Vision

Using neural networks fixes this problem. We can directly input images as multi-dimension arrays (you will see how to do this in the next chapter) and the neural network can learn features across the multiple layers of the network. Each layer will learn subsequently more complex features starting from the pixel level as the input to understanding features in later layers that we might have hand picked or more likely features that we might not understand, but help to make an accurate classifier.

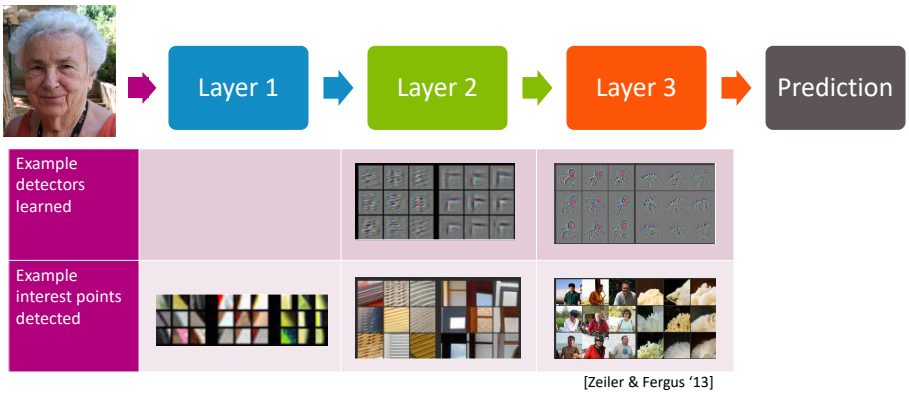

Figure 17.92: Manual Feature Selection in Computer Vision

17.5 Training

Now that we understand the basics of what make of neural networks are, what they can learn, and what they can be used for we should learn about how to actually train one of these models. The first step, similar to the other models we have learned, is that we must define a cost function.

- For regression tasks, typically the residual sum of squares (RSS) or Root Mean Square Error (RMSE) is used
- For classification, typically Cross Entropy loss is used.

With a cost function defined, we can train a neural network, similar to other models, using gradient descent in three main steps:

1. Do a forward pass of the data through the network to get predictions.
2. Compare predictions to true values using the cost function.
3. Backpropagate errors by pushing the calculation of the gradient through the back of the network so the weights make better predictions.

To train neural networks well, usually there is a large amount of data. To speed training up instead of following the three steps from above for the entire dataset at once, the data is broken up in to **batches**. Generally, going through the entire data set once is not enough to train the network so multiple passes through the same data is required. We call an iteration that goes over every batch (the entire training set) once an **epoch**.

Here is how the training of the neural network would look like in code across a set number of epochs and batches:

```
for i in range(num_epochs):
  for batch in batches(training_data):
    preds = model.predict(batch.data) # Forward pass
    diffs = compare(preds, batch.labels)  # Compare
    model.backprop(diffs)         # Backpropagation
```

Figure 17.93: Training a Neural Network

One major challenge of training neural networks is that there are many hyperparameters. We must consider at least the shape of the network (how many hidden layers and hidden neurons), the activation functions, the learning rate for gradient descent, the batch size, and the number of epochs. Unfortunately, there is no best solution to determining these values and often is lots of trial and error. Grid search and random search can be useful and Bayesian Optimization can also help with this.

In general, loss functions with neural networks are not convex meaning that the backpropagation algorithm for gradient descent will only converge to a local optima. Similar to k-means the initialization is important and could affect the final result. However, there is not one known way and usually random initialization is used. The same applies for the learning rate if the step sizes are too large when running gradient descent.

There is still ongoing research about understanding neural networks and why they work so well on certain problems, the best ways to train, and the caveats of using them. It is important to know that while neural networks are useful for fitting complex problems, they are not the one and done solution as they are often quick to overfit and overly complex for many tasks.