**Chapter 5**

**DEEP FEEDFORWARD NETWORKS**

* The goal of a feedforward network is to approximate some function f∗. For example, for a classifier, y=f∗ (x) maps an input x to a category y. A feedforward network defines a mapping y = f (x; θ) and learns the value of the parameters θ that result in the best function approximation.
* These models are called **feedforward**because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y.
* When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks**.
* Feedforward neural networks are called **networks**because they are typically represented by composing together many different functions. For example, we might have three functions f (1), f (2), and f (3) connected in a chain, to form f(x)=f(3)(f(2)(f(1)(x))). These chain structures are the most commonly used structures of neural networks. In this case, f (1) is called the **first layer**of the network, f (2) is called the **second layer**, and so on. The overall length of the chain gives the depth of the model.
* We drive f(x) to match f ∗(x). The training data provides us with noisy, approximate examples of f ∗(x) evaluated at different training points. Each example x is accompanied by a label y ≈ f∗(x). The training examples specify directly what the output layer must do at each point.
* the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f∗. And so, these layers are called **hidden layers**.

* One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they may be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.
* To extend linear models to represent nonlinear functions of x, we can apply the linear model not to x itself but to a transformed input φ(x), where φ is a nonlinear transformation. We can think of φ as providing a set of features describing x, or as providing a new representation for x. The question is then how to choose the mapping φ.

1. One option is to use a very generic φ, such as the infinite-dimensional φ. If φ(x ) is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor.
2. Another option is to manually engineer φ. Until the advent of deep learning, this was the dominant approach. This approach requires decades of human effort.
3. The strategy of deep learning is to learn φ. In this approach, we have a model

y = f (x;θ,w) = φ(x; θ)Tw

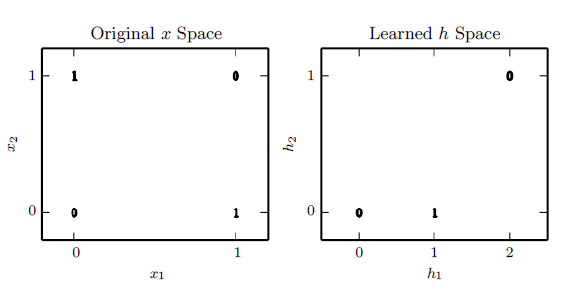
We now have parameters θ that we use to learn φ from a broad class of functions, and parameters w that map from φ( x) to the desired output. This is an example of a deep feedforward network, with φ defining a hidden layer.

* 1. **Example: Learning XOR**
* The XOR function provides the target function y = f∗(x) that we want to learn. Our model provides a function y = f(x;θ) and our learning algorithm will adapt the parameters θ to make f as similar as possible to f∗.
* We want our network to perform correctly on the four points

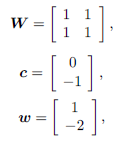
X = { [0, 0] T, [0, 1] T, [1, 0] T, and [1, 1]T }.

We will train the network on all four of these points. We can treat this problem as a regression problem and use a mean squared error loss function.

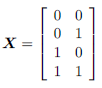
* Now we must choose the form of our model, f (x; θ). Suppose that we choose a linear model, with θ consisting of w and b. Our model is defined to be
* After solving the normal equations, we obtain w = 0 and b = 12. The linear model simply outputs 0.5 everywhere. Why does this happen? Below figure shows how a linear model is not able to represent the XOR function.



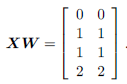
* Specifically, we will introduce a very simple feedforward network with one hidden layer containing two hidden units. See figure below for an illustration of this model. This feedforward network has a vector of hidden units h that are computed by a function f(1)(x;W, c).
* The output layer is still just a linear regression model, but now it is applied to h rather than to x. The network now contains two functions chained together: h = f(1)(x;W, c) and y = f(2)(h;w, b), with the complete model being f(x;W, c,w, b) = f(2)(f(1)(x)). What function should f(1) compute?
* If f(1) were linear, then the feedforward network as a whole would remain a linear function of its input.
* Clearly, we must use a nonlinear function to describe the features. Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an activation function. We use that strategy here, by defining h = g(WTx + c) , where W provides the weights of a linear transformation and c the biases.
* The activation function g is typically chosen to be a function that is applied element-wise, with hi = g(xTW:,i + ci ). In modern neural networks, the default recommendation is to use the rectified linear unit or ReLU.
* We can now specify our complete network as
* We can now specify a solution to the XOR problem. Let



Let X be the design matrix containing all four points in the binary input space, with one example per row:



The first step in the neural network is to multiply the input matrix by the first layer’s weight matrix:



Next, we add the bias vector c



To finish computing the value of h for each example, we apply the rectified linear transformation:



We finish by multiplying by the weight vector w:

C:\Users\vardan_nadkarni\Desktop\Untitled.png

* 1. **Gradient-Based Learning**
* The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.
* We can of course, train models such as linear regression and support vector machines with gradient descent too, and in fact this is common when the training set is extremely large.
  + 1. **Cost Functions**
* An important aspect of the design of a deep neural network is the choice of the cost function.
* In most cases, our parametric model defines a distribution p(y | x;θ ) and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model’s predictions as the cost function.
* Sometimes, we take a simpler approach, where rather than predicting a complete probability distribution over y, we merely predict some statistic of y conditioned on x.
  + - 1. **Cost Functions**
  1. **Hidden Units**

The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.

Rectified linear units are an excellent default choice of hidden unit. Many other types of hidden units are available.

The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

Some of the hidden units included in this list are not actually differentiable at all input points. For example, the rectified linear function g(z) = max{0 , z} is not differentiable at z = 0. This may seem like it invalidates g for use with a gradientbased learning algorithm. In practice, gradient descent still performs well enough for these models to be used for machine learning tasks.

In general, a function g(z ) has a left derivative defined by the slope of the function immediately to the left of z and a right derivative defined by the slope of the function immediately to the right of z . A function is differentiable at z only if both the left derivative and the right derivative are defined and equal to each other.

In the case of g(z) = max{0, z}, the left derivative at z = 0 is 0 and the right derivative is 1. Software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is undefined or raising an error.

* + 1. **Rectified Linear Units and Their Generalizations**

Rectified linear units are easy to optimize because they are so similar to linear units.

The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active.

Rectified linear units are typically used on top of an affine transformation:

When initializing the parameters of the affine transformation, it can be a good practice to set all elements of b to a small, positive value, such as 0.1. This makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

One drawback to rectified linear units is that they cannot learn via gradient based methods on examples for which their activation is zero. A variety of generalizations of rectified linear units guarantee that they receive gradient everywhere.

Three generalizations of rectified linear units are based on using a non-zero slope αi when zi < 0: hi = g(z,α)i = max(0, zi) + αimin(0, zi). Absolute value rectificationfixes αi = −1 to obtain g(z) = |z|. It is used for object recognition from images (Jarrett *et al.*, 2009), where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.

A **leaky ReLU**(Maas *et al.*, 2013) fixes αi to a small value like 0.01 while a **parametric ReLU**or PReLUtreats αi as a learnable parameter (He *et al.*, 2015).

**Maxout units**(Goodfellow *et al.*, 2013a) generalize rectified linear units further. Instead of applying an element-wise function g(z), maxout units divide z into groups of k values. Each maxout unit then outputs the maximum element of one of these groups:

where G(i) is the indices of the inputs for group i, {(i − 1)k + 1, . . . , ik}.

A maxout unit can learn a piecewise linear, convex function with up to k pieces. Maxout units can thus be seen as **learning the activation function** itself rather than just the relationship between units. With large enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity. In particular, a maxout layer with two pieces can learn to implement the same function of the input x as a traditional layer using the rectified linear activation function, absolute value rectification function, or the leaky or parametric ReLU, or can learn to implement a totally different function altogether.

Each maxout unit is now parametrized by k weight vectors instead of just one, so maxout units typically need more regularization than rectified linear units. They can work well without regularization if the training set is large and the number of pieces per unit is kept low.

Maxout units have a few other benefits. In some cases, one can gain some statistical and computational advantages by requiring fewer parameters. Specifically, if the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights.

Because each unit is driven by multiple filters, maxout units have some redundancy that helps them to resist a phenomenon called *catastrophic forgetting* in which neural networks forget how to perform tasks that they were trained on in the past

* + 1. **Logistic Sigmoid and Hyperbolic Tangent**