Lecture 5: Multilayer Perceptrons

Roger Grosse

1 Introduction

So far, we've only talked about linear models: linear regression and linear binary classifiers. We noted that there are functions that can't be represented by linear models; for instance, linear regression can't represent quadratic functions, and linear classifiers can't represent XOR. We also saw one particular way around this issue: by defining features, or basis functions. E.g., linear regression can represent a cubic polynomial if we use the feature map $\boldsymbol{\psi}(x) = (1, x, x^2, x^3)$. We also observed that this isn't a very satisfying solution, for two reasons:

1. The features need to be specified in advance, and this can require a lot of engineering work.

2. It might require a very large number of features to represent a certain set of functions; e.g. the feature representation for cubic polynomials is cubic in the number of input features.

In this lecture, and for the rest of the course, we'll take a different approach. We'll represent complex nonlinear functions by connecting together lots of simple processing units into a *neural network*, each of which computes a linear function, possibly followed by a nonlinearity. In aggregate, these units can compute some surprisingly complex functions. By historical accident, these networks are called *multilayer perceptrons*.

Some people would claim that the methods covered in this course are really "just" adaptive basis function representations. I've never found this a very useful way of looking at things.

1.1 Learning Goals

- Know the basic terminology for neural nets
- Given the weights and biases for a neural net, be able to compute its output from its input
- Be able to hand-design the weights of a neural net to represent functions like XOR
- Understand how a hard threshold can be approximated with a soft threshold
- Understand why shallow neural nets are universal, and why this isn't necessarily very interesting

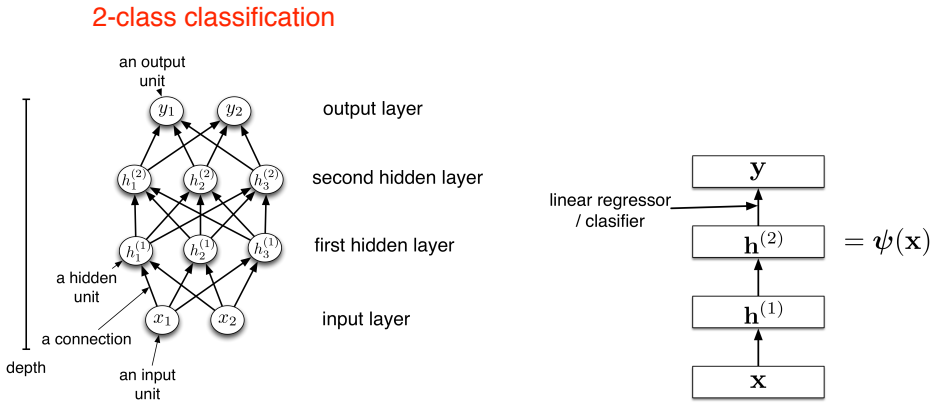


Figure 1: A multilayer perceptron with two hidden layers. **Left:** with the units written out explicitly. **Right:** representing layers as boxes.

2 Multilayer Perceptrons

In the first lecture, we introduced our general neuron-like processing unit:

$$\text{activation} \quad a = \phi\left(\sum_j w_j x_j + b\right),$$

where the x_j are the inputs to the unit, the w_j are the weights, b is the bias, ϕ is the nonlinear activation function, and a is the unit's activation. We've seen a bunch of examples of such units:

- Linear regression uses a linear model, so $\phi(z) = z$.
- In binary linear classifiers, ϕ is a hard threshold at zero.
- In logistic regression, ϕ is the logistic function $\sigma(z) = 1/(1 + e^{-z})$.

A **neural network** is just a combination of lots of these units. Each one performs a very simple and stereotyped function, but in aggregate they can do some very useful computations. For now, we'll concern ourselves with **feed-forward neural networks**, where the units are arranged into a graph without any cycles, so that all the computation can be done sequentially. This is in contrast with **recurrent neural networks**, where the graph can have cycles, so the processing can feed into itself. These are much more complicated, and we'll cover them later in the course.

The simplest kind of feed-forward network is a **multilayer perceptron (MLP)**, as shown in Figure 1. Here, the units are arranged into a set of **layers**, and each layer contains some number of identical units. Every unit in one layer is connected to every unit in the next layer; we say that the network is **fully connected**. The first layer is the **input layer**, and its units take the values of the input features. The last layer is the **output layer**, and it has one unit for each value the network outputs (i.e. a single unit in the case of regression or binary classifiation, or K units in the case of K-class classification). All the layers in between these are known as **hidden layers**, because we don't know ahead of time what these units should compute, and this needs to be discovered during learning. The units

MLP is an unfortunate name. The *perceptron* was a particular algorithm for binary classification, invented in the 1950s. Most multilayer perceptrons have very little to do with the original perceptron algorithm.

a trick to designing weights to binary classification
write +/- in input space, draw curve that separate + from -
compute w and b from the line

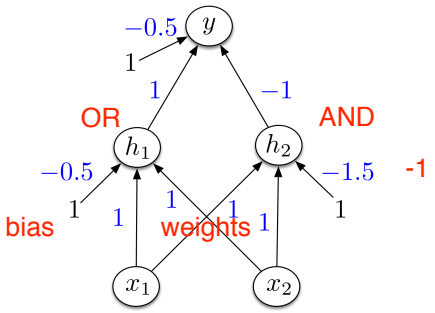


Figure 2: An MLP that computes the XOR function. All activation functions are binary thresholds at 0.

in these layers are known as **input units**, **output units**, and **hidden units**, respectively. The number of layers is known as the **depth**, and the number of units in a layer is known as the **width**. As you might guess, "deep learning" refers to training neural nets with many layers.

Terminology for the depth is very inconsistent. A network with one hidden layer could be called a one-layer, two-layer, or three-layer network, depending if you count the input and output layers.

As an example to illustrate the power of MLPs, let's design one that computes the XOR function. Remember, we showed that linear models cannot do this. We can verbally describe XOR as "one of the inputs is 1, but not both of them." So let's have hidden unit h_1 detect if at least one of the inputs is 1, and have h_2 detect if they are both 1. We can easily do this if we use a hard threshold activation function. You know how to design such units — it's an exercise of designing a binary linear classifier. Then the output unit will activate only if $h_1 = 1$ and $h_2 = 0$. A network which does this is shown in Figure 2.

Let's write out the MLP computations mathematically. Conceptually, there's nothing new here; we just have to pick a notation to refer to various parts of the network. As with the linear case, we'll refer to the activations of the input units as x_j and the activation of the output unit as y. The units in the ℓth hidden layer will be denoted $h_i^{(\ell)}$. Our network is fully connected, so each unit receives connections from all the units in the previous layer. This means each unit has its own bias, and there's a weight for every *pair* of units in two consecutive layers. Therefore, the network's computations can be written out as:

superscript l represent layer

$$
\begin{aligned}
h_i^{(1)} &= \phi^{(1)}\left(\sum_j w_{ij}^{(1)} x_j + b_i^{(1)}\right) \\
h_i^{(2)} &= \phi^{(2)}\left(\sum_j w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)}\right) \\
y_i &= \phi^{(3)}\left(\sum_j w_{ij}^{(3)} h_j^{(2)} + b_i^{(3)}\right)
\end{aligned}
\tag{1}
$$

Note that we distinguish $\phi^{(1)}$ and $\phi^{(2)}$ because different layers may have different activation functions.

Since all these summations and indices can be cumbersome, we usually

write the computations in vectorized form. Since each layer contains multiple units, we represent the activations of all its units with an **activation vector** $\mathbf{h}^{(\ell)}$. Since there is a weight for every pair of units in two consecutive layers, we represent each layer's weights with a **weight matrix** $\mathbf{W}^{(\ell)}$. Each layer also has a **bias vector** $\mathbf{b}^{(\ell)}$. The above computations are therefore written in vectorized form as:

W maps from width of layer l-1 to width of layer l, so W: l x (l-1)

input to nonlinearity is l x 1, l activations for 1 sample

$$\begin{aligned}\mathbf{h}^{(1)} &= \phi^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \\ \mathbf{h}^{(2)} &= \phi^{(2)}\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right) \\ \mathbf{y} &= \phi^{(3)}\left(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}\right)\end{aligned} \tag{2}$$

b has dimension l x 1

When we write the activation function applied to a vector, this means it's applied independently to all the entries.

Recall how in linear regression, we combined all the training examples into a single matrix $\mathbf{X}$, so that we could compute all the predictions using a single matrix multiplication. We can do the same thing here. We can store all of each layer's hidden units for all the training examples as a matrix $\mathbf{H}^{(\ell)}$. Each row contains the hidden units for one example. The computations are written as follows (note the transposes):

H has dimension N x l

If it's hard to remember when a matrix or vector is transposed, fear not. You can usually figure it out by making sure the dimensions match up.

input to nonlinearity is N x l, l activations for each N samples

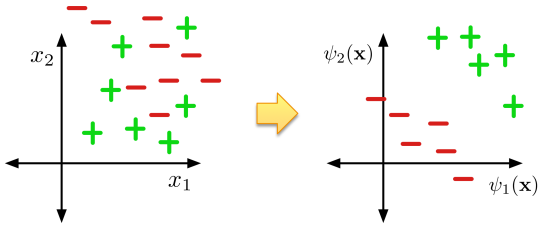
$$\begin{aligned}\mathbf{H}^{(1)} &= \phi^{(1)}\left(\mathbf{X}\mathbf{W}^{(1)\top} + \mathbf{1}\mathbf{b}^{(1)\top}\right) \\ \mathbf{H}^{(2)} &= \phi^{(2)}\left(\mathbf{H}^{(1)}\mathbf{W}^{(2)\top} + \mathbf{1}\mathbf{b}^{(2)\top}\right) \\ \mathbf{Y} &= \phi^{(3)}\left(\mathbf{H}^{(2)}\mathbf{W}^{(3)\top} + \mathbf{1}\mathbf{b}^{(3)\top}\right)\end{aligned} \tag{3}$$

l has dimension N x 1

These equations can be translated directly into NumPy code which efficiently computes the predictions over the whole dataset.

3 Feature Learning

We already saw that linear regression could be made more powerful using a feature mapping. For instance, the feature mapping $\boldsymbol{\psi}(x) = (1, x, x^2, x^e)$ can represent third-degree polynomials. But static feature mappings were limited because it can be hard to design all the relevant features, and because the mappings might be impractically large. Neural nets can be thought of as a way of learning nonlinear feature mappings. E.g., in Figure 1, the last hidden layer can be thought of as a feature map $\boldsymbol{\psi}(\mathbf{x})$, and the output layer weights can be thought of as a linear model using those features. But the whole thing can be trained end-to-end with backpropagation, which we'll cover in the next lecture. The hope is that we can learn a feature representation where the data become linearly separable:



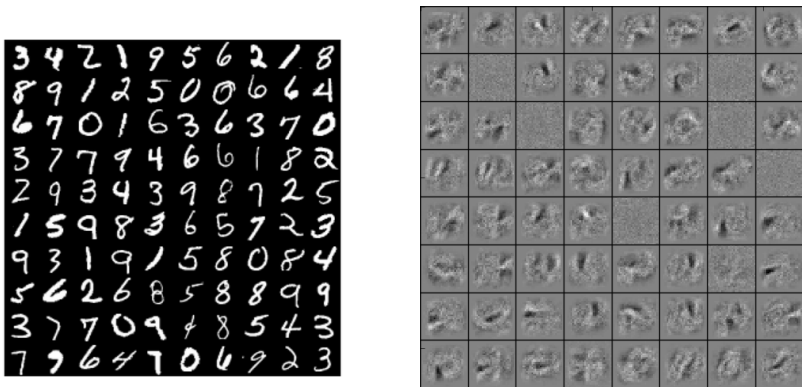


Figure 3: **Left:** Some training examples from the MNIST handwritten digit dataset. Each input is a 28×28 grayscale image, which we treat as a 784-dimensional vector. **Right:** A subset of the learned first-layer features. Observe that many of them pick up oriented edges.

Consider training an MLP to recognize handwritten digits. (This will be a running example for much of the course.) The input is a 28×28 grayscale image, and all the pixels take values between 0 and 1. We'll ignore the spatial structure, and treat each input as a 784-dimensional vector. This is a multiway classification task with 10 categories, one for each digit class. Suppose we train an MLP with two hidden layers. We can try to understand what the first layer of hidden units is computing by visualizing the weights. Each hidden unit receives inputs from each of the pixels, which means the weights feeding into each hidden unit can be represented as a 784-dimensional vector, the same as the input size. In Figure 3, we display these vectors as images.

Later on, we'll talk about convolutional networks, which use the spatial structure of the image.

In this visualization, positive values are lighter, and negative values are darker. Each hidden unit computes the dot product of these vectors with the input image, and then passes the result through the activation function. So if the light regions of the filter overlap the light regions of the image, and the dark regions of the filter overlap the dark region of the image, then the unit will activate. E.g., look at the third filter in the second row. This corresponds to an **oriented edge**: it detects vertical edges in the upper right part of the image. This is a useful sort of feature, since it gives information about the locations and orientation of strokes. Many of the features are similar to this; in fact, oriented edges are a very commonly learned by the first layers of neural nets for visual processing tasks.

It's harder to visualize what the second layer is doing. We'll see some tricks for visualizing this in a few weeks. We'll see that higher layers of a neural net can learn increasingly high-level and complex features.

4 Expressive Power

Linear models are fundamentally limited in their expressive power: they can't represent functions like XOR. Are there similar limitations for MLPs? It depends on the activation function.

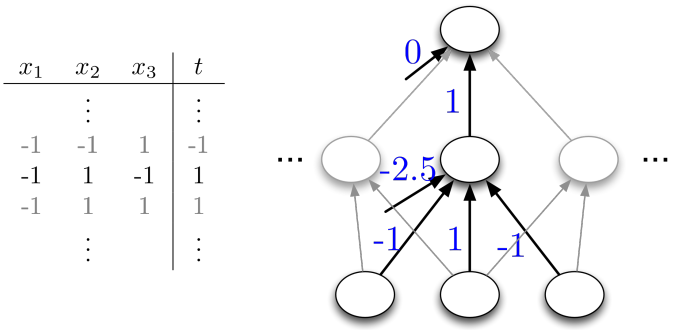


Figure 4: Designing a binary threshold network to compute a particular function.

4.1 Linear networks

Deep linear networks are no more powerful than shallow ones. The reason is simple: if we use the linear activation function $\phi(x) = x$ (and forget the biases for simplicity), the network's function can be expanded out as $\mathbf{y} = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdots \mathbf{W}^{(1)}\mathbf{x}$. But this could be viewed as a single linear layer with weights given by $\mathbf{W} = \mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdots \mathbf{W}^{(1)}$. Therefore, a deep linear network is no more powerful than a single linear layer, i.e. a linear model.

4.2 Universality

As it turns out, nonlinear activation functions give us much more power: under certain technical conditions, even a shallow MLP (i.e. one with a single hidden layer) can represent arbitrary functions. Therefore, we say it is **universal**.

Let's demonstrate universality in the case of binary inputs. We do this using the following game: suppose we're given a function mapping input vectors to outputs; we will need to produce a neural network (i.e. specify the weights and biases) which matches that function. The function can be given to us as a table which lists the output corresponding to every possible input vector. If there are D inputs, this table will have 2^D rows. An example is shown in Figure 4. For convenience, let's suppose these inputs are ± 1, rather than 0 or 1. All of our hidden units will use a hard threshold at 0 (but we'll see shortly that these can easily be converted to soft thresholds), and the output unit will be linear.

This argument can easily be made into a rigorous proof, but this course won't be concerned with mathematical rigor.

Our strategy will be as follows: we will have 2^D hidden units, each of which recognizes one possible input vector. We can then specify the function by specifying the weights connecting each of these hidden units to the outputs. For instance, suppose we want a hidden unit to recognize the input $(-1, 1, -1)$. This can be done using the weights $(-1, 1, -1)$ and bias -2.5, and this unit will be connected to the output unit with weight 1. (Can you come up with the general rule?) Using these weights, any input pattern will produce a set of hidden activations where exactly one of the units is active. The weights connecting inputs to outputs can be set based on the input-output table. Part of the network is shown in Figure 4.

Universality is a neat property, but it has a major catch: the network required to represent a given function might have to be extremely large (in particular, exponential). In other words, not all functions can be represented **compactly**. We desire compact representations for two reasons:

1. We want to be able to compute predictions in a reasonable amount of time.

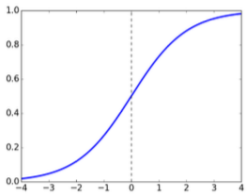
2. We want to be able to train a network to *generalize* from a limited number of training examples; from this perspective, universality simply implies that a large enough network can memorize the training set, which isn't very interesting.

4.3 Soft thresholds

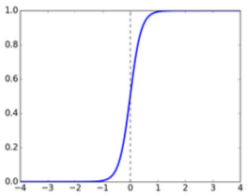
same expressive power as hard threshold

In the previous section, our activation function was a step function, which gives a hard threshold at 0. This was convenient for designing the weights of a network by hand. But recall from last lecture that it's very hard to directly learn a linear classifier with a hard threshold, because the loss derivatives are 0 almost everywhere. The same holds true for multilayer perceptrons. If the activation function for any unit is a hard threshold, we won't be able to learn that unit's weights using gradient descent. The solution is the same as it was in last lecture: we replace the hard threshold with a soft one.

Does this cost us anything in terms of the network's expressive power? No it doesn't, because we can approximate a hard threshold using a soft threshold. In particular, if we use the logistic nonlinearity, we can approximate a hard threshold by scaling up the weights and biases:



$y = \sigma(x)$



$y = \sigma(5x)$

4.4 The power of depth

If shallow networks are universal, why do we need deep ones? One important reason is that deep nets can represent some functions more *compactly* than shallow ones. For instance, consider the parity function (on binary-valued inputs):

$$f_{\text{par}}(x_1, \ldots, x_D) = \begin{cases} 1 & \text{if } \sum_j x_j \text{ is odd} \\ 0 & \text{if it is even.} \end{cases} \tag{4}$$

We won't prove this, but it requires an exponentially large shallow network to represent the parity function. On the other hand, it can be computed by a deep network whose size is *linear* in the number of inputs. Designing such a network is a good exercise.