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

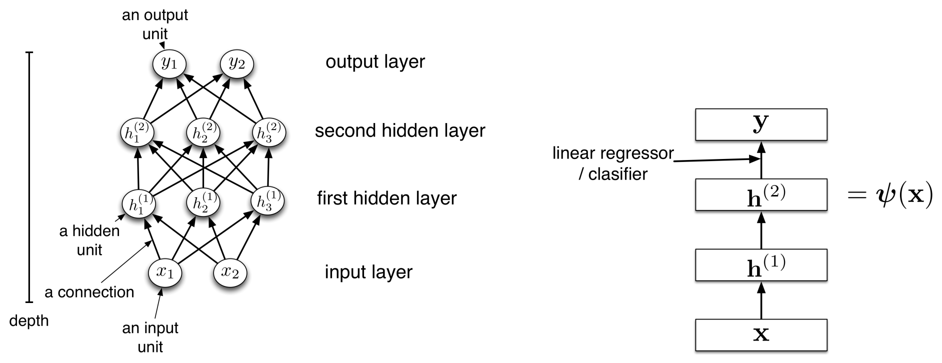


Figure 1: A multilayer perceptron with two hidden layers. **Left:** with the units written out explicitly. **Right:** representing layers as boxes.

2 Multilayer Perceptrons

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

$$a = \phi\left(\sum_j w_j x_j + b\right),$$

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.

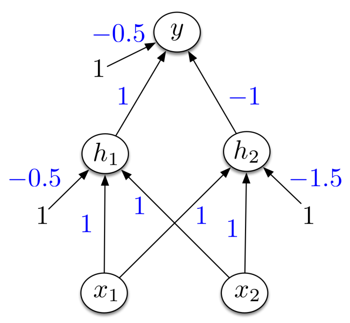


Figure 2: An MLP that computes the XOR function. All activation functions are binary thresholds at 0.

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:

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

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:

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

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):

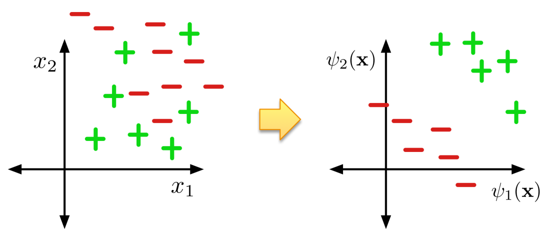
If it's hard to remember when a matrix or vector is transposed, fear not. You can usually figure it out by making sure the dimensions match up.

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

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:



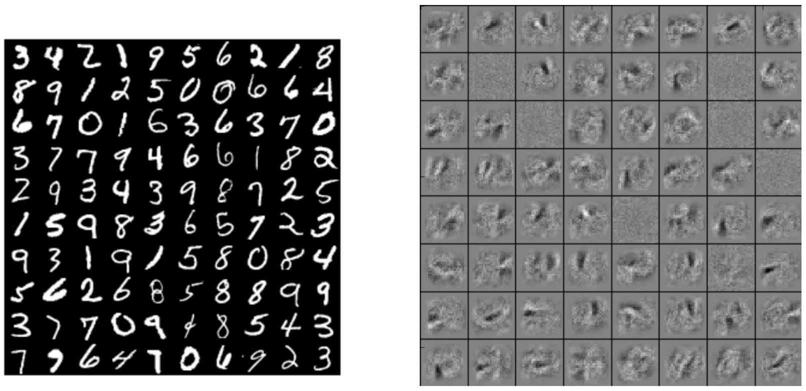


Figure 3: **Left:** Some training examples from the MNIST handwritten digit dataset. Each input is a 28×28 grayscale image, which we treat as a 784-dimensional vector. **Right:** A subset of the learned first-layer features. Observe that many of them pick up oriented edges.

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.

x_1	x_2	x_3	t
	⋮		⋮
-1	-1	1	-1
-1	1	-1	1
-1	1	1	1
	⋮		⋮

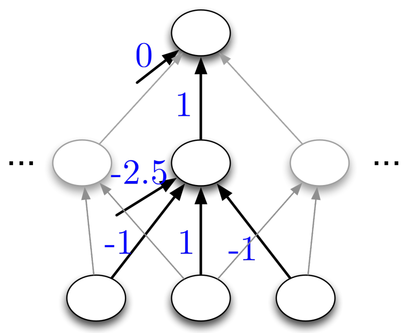


Figure 4: Designing a binary threshold network to compute a particular function.

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:

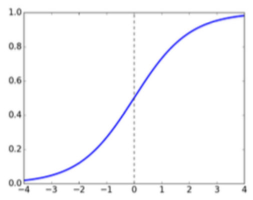
1. We want to be able to compute predictions in a reasonable amount of time.

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.

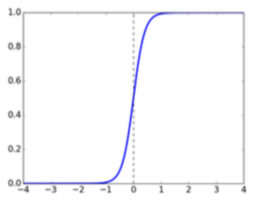
4.3 Soft thresholds

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.

Lecture 6: Backpropagation

Roger Grosse

1 Introduction

So far, we've seen how to train "shallow" models, where the predictions are computed as a linear function of the inputs. We've also observed that deeper models are much more powerful than linear ones, in that they can compute a broader set of functions. Let's put these two together, and see how to train a multilayer neural network. We will do this using backpropagation, the central algorithm of this course. Backpropagation ("backprop" for short) is a way of computing the partial derivatives of a loss function with respect to the parameters of a network; we use these derivatives in gradient descent, exactly the way we did with linear regression and logistic regression.

If you've taken a multivariate calculus class, you've probably encountered the Chain Rule for partial derivatives, a generalization of the Chain Rule from univariate calculus. In a sense, backprop is "just" the Chain Rule — but with some interesting twists and potential gotchas. This lecture and Lecture 8 focus on backprop. (In between, we'll see a cool example of how to use it.) This lecture covers the mathematical justification and shows how to implement a backprop routine by hand. Implementing backprop can get tedious if you do it too often. In Lecture 8, we'll see how to implement an *automatic differentiation* engine, so that derivatives even of rather complicated cost functions can be computed automatically. (And just as efficiently as if you'd done it carefully by hand!)

This will be your least favorite lecture, since it requires the most tedious derivations of the whole course.

1.1 Learning Goals

- Be able to compute the derivatives of a cost function using backprop.

1.2 Background

I would highly recommend reviewing and practicing the Chain Rule for partial derivatives. I'd suggest Khan Academy[1], but you can also find lots of resources on Metacademy[2].

[1] `https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/multivariable-chain-rule/v/multivariable-chain-rule`

[2] `https://metacademy.org/graphs/concepts/chain_rule`

2 The Chain Rule revisited

Before we get to neural networks, let's start by looking more closely at an example we've already covered: a linear classification model. For simplicity, let's assume we have univariate inputs and a single training example (x, t). The predictions are a linear function followed by a sigmoidal nonlinearity. Finally, we use the squared error loss function. The model and loss function are as follows:

$$z = wx + b \tag{1}$$
$$y = \sigma(z) \tag{2}$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2 \tag{3}$$

Now, to change things up a bit, let's add a *regularizer* to the cost function. We'll cover regularizers properly in a later lecture, but intuitively, they try to encourage "simpler" explanations. In this example, we'll use the regularizer $\frac{\lambda}{2}w^2$, which encourages w to be close to zero. (λ is a hyperparameter; the larger it is, the more strongly the weights prefer to be close to zero.) The cost function, then, is:

$$\mathcal{R} = \frac{1}{2}w^2 \tag{4}$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}. \tag{5}$$

In order to perform gradient descent, we wish to compute the partial derivatives $\partial\mathcal{E}/\partial w$ and $\partial\mathcal{E}/\partial b$.

This example will cover all the important ideas behind backprop; the only thing harder about the case of multilayer neural nets will be the cruftier notation.

2.1 How you would have done it in calculus class

Recall that you can calculate partial derivatives the same way you would calculate univariate derivatives. In particular, we can expand out the cost function in terms of w and b, and then compute the derivatives using repeated applications of the univariate Chain Rule.

$$
\begin{aligned}
\mathcal{L}_{\text{reg}} &= \frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2 \\
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial w}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) + \lambda w \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) + \lambda w \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)x + \lambda w \\
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial b}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) + 0 \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)
\end{aligned}
$$

This gives us the correct answer, but hopefully it's apparent from this example that this method has several drawbacks:

1. The calculations are very cumbersome. In this derivation, we had to copy lots of terms from one line to the next, and it's easy to accidentally drop something. (In fact, I made such a mistake while writing these notes!) While the calculations are doable in this simple example, they become impossibly cumbersome for a realistic neural net.

2. The calculations involve lots of redundant work. For instance, the first three steps in the two derivations above are nearly identical.

3. Similarly, the final expressions have lots of repeated terms, which means lots of redundant work if we implement these expressions directly. For instance, $wx + b$ is computed a total of four times between $\partial\mathcal{E}/\partial w$ and $\partial\mathcal{E}/\partial b$. The larger expression $(\sigma(wx + b) - t)\sigma'(wx + b)$ is computed twice. If you happen to notice these things, then perhaps you can be clever in your implementation and factor out the repeated expressions. But, as you can imagine, such efficiency improvements might not always jump out at you when you're implementing an algorithm.

Actually, even in this derivation, I used the "efficiency trick" of not expanding out σ'. If I had expanded it out, the expressions would be even more hideous, and would involve *six* copies of $wx + b$.

The idea behind backpropagation is to share the repeated computations wherever possible. We'll see that the backprop calculations, if done properly, are very clean and modular.

2.2 Multivariable chain rule: the easy case

We've already used the univariate Chain Rule a bunch of times, but it's worth remembering the formal definition:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(g(t)) = f'(g(t))g'(t). \tag{6}$$

Roughly speaking, increasing t by some infinitesimal quantity h_1 "causes" g to change by the infinitesimal $h_2 = g'(t)h_1$. This in turn causes f to change by $f'(g(t))h_2 = f'(g(t))g'(t)h_1$.

The multivariable Chain Rule is a generalization of the univariate one. Let's say we have a function f in two variables, and we want to compute $\frac{d}{dt} f(x(t), y(t))$. Changing t slightly has two effects: it changes x slightly, and it changes y slightly. Each of these effects causes a slight change to f. For infinitesimal changes, these effects combine additively. The Chain Rule, therefore, is given by:

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{7}$$

2.3 An alternative notation

It will be convenient for us to introduce an alternative notation for the derivatives we compute. In particular, notice that the left-hand side in all of our derivative calculations is $\mathrm{d}\mathcal{L}/\mathrm{d}v$, where v is some quantity we compute in order to compute $\mathcal{L}$. (Or substitute for $\mathcal{L}$ whichever variable we're trying to compute derivatives of.) We'll use the notation

$$\overline{v} \triangleq \frac{\partial \mathcal{L}}{\partial v}. \tag{8}$$

This notation is less crufty, and also emphasizes that $\overline{v}$ is a value we compute, rather than a mathematical expression to be evaluated. This notation is nonstandard; see the appendix if you want more justification for it.

We can rewrite the multivariable Chain rule (Eqn. 7) using this notation:

$$\overline{t} = \overline{x}\frac{\mathrm{d}x}{\mathrm{d}t} + \overline{y}\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{9}$$

Here, we use $\mathrm{d}x/\mathrm{d}t$ to mean we should actually evaluate the derivative algebraically in order to determine the formula for $\overline{t}$, whereas $\overline{x}$ and $\overline{y}$ are values previously computed by the algorithm.

2.4 Using the computation graph

In this section, we finally introduce the main algorithm for this course, which is known as **backpropagation**, or **reverse mode automatic differentiation (autodiff)**.[3]

[3]Automatic differentiation was invented in 1970, and backprop in the late 80s. Originally, backprop referred to the special case of reverse mode autodiff applied to neural nets, although the derivatives were typically written out by hand (rather than using an autodiff package). But in the last few years, neural nets have gotten so diverse that we basically think of them as compositions of functions. Also, very often, backprop is now implemented using an autodiff software package. For these reasons, the distinction between autodiff and backprop has gotten blurred, and we will use the terms interchangeably in this course. Note that there is also a forward mode autodiff, but it's rarely used in neural nets, and we won't cover it in this course.

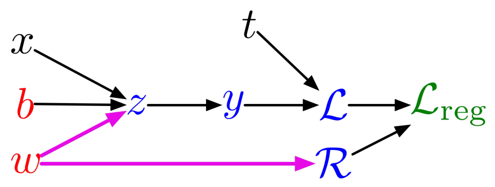


Figure 1: Computation graph for the regularized linear regression example in Section 2.4. The magenta arrows indicate the case which requires the multivariate chain rule because w is used to compute both z and $\mathcal{R}$.

Now let's return to our running example, written again for convenience:

$$\begin{aligned} z &= wx + b \\ y &= \sigma(z) \\ \mathcal{L} &= \frac{1}{2}(y - t)^2 \\ \mathcal{R} &= \frac{1}{2}w^2 \\ \mathcal{L}_{\text{reg}} &= \mathcal{L} + \lambda \mathcal{R}. \end{aligned}$$

Let's introduce the **computation graph**. The nodes in the graph correspond to all the values that are computed, with edges to indicate which values are computed from which other values. The computation graph for our running example is shown in Figure 1.

Note that the computation graph is *not* the network architecture. The nodes correspond to values that are computed, rather than to units in the network.

The goal of backprop is to compute the derivatives $\overline{w}$ and $\overline{b}$. We do this by repeatedly applying the Chain Rule (Eqn. 9). Observe that to compute a derivative using Eqn. 9, you first need the derivatives for its *children* in the computation graph. This means we must start from the result of the computation (in this case, $\mathcal{E}$) and work our way backwards through the graph. It is because we work backward through the graph that backprop and reverse mode autodiff get their names.

Let's start with the formal definition of the algorithm. Let $v_1, \ldots, v_N$ denote all of the nodes in the computation graph, in a topological ordering. (A topological ordering is any ordering where parents come before children.) We wish to compute all of the derivatives $\overline{v_i}$, although we may only be interested in a subset of these values. We first compute all of the values in a **forward pass**, and then compute the derivatives in a **backward pass**. As a special case, v_N denotes the result of the computation (in our running example, $v_N = \mathcal{E}$), and is the thing we're trying to compute the derivatives of. Therefore, by convention, we set $\overline{v_N} = 1$. The algorithm is as follows:

$\overline{\mathcal{E}} = 1$ because increasing the cost by h increases the cost by h.

```
For $i = 1, \ldots, N$
    Compute $v_i$ as a function of $\mathrm{Pa}(v_i)$
$\overline{v_N} = 1$
For $i = N - 1, \ldots, 1$
    $\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$
```

Here $\mathrm{Pa}(v_i)$ and $\mathrm{Ch}(v_i)$ denote the parents and children of v_i.

This procedure may become clearer when we work through the example in full:

$$\begin{aligned}
\overline{\mathcal{L}_{\text{reg}}} &= 1 \\
\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}} \, \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{R}} \\
&= \overline{\mathcal{L}_{\text{reg}}} \, \lambda \\
\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \, \frac{\mathrm{d}\mathcal{L}_{\text{reg}}}{\mathrm{d}\mathcal{L}} \\
&= \overline{\mathcal{L}_{\text{reg}}} \\
\overline{y} &= \overline{\mathcal{L}} \, \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y} \\
&= \overline{\mathcal{L}} \, (y - t) \\
\overline{z} &= \overline{y} \, \frac{\mathrm{d}y}{\mathrm{d}z} \\
&= \overline{y} \, \sigma'(z) \\
\overline{w} &= \overline{z} \, \frac{\partial z}{\partial w} + \overline{\mathcal{R}} \frac{\mathrm{d}\mathcal{R}}{\mathrm{d}w} \\
&= \overline{z} \, x + \overline{\mathcal{R}} \, w \\
\overline{b} &= \overline{z} \, \frac{\partial z}{\partial b} \\
&= \overline{z}
\end{aligned}$$

Since we've derived a procedure for computing $\overline{w}$ and $\overline{b}$, we're done. Let's write out this procedure without the mess of the derivation, so that we can compare it with the naïve method of Section 2.1:

$$\begin{aligned}
\overline{\mathcal{L}_{\text{reg}}} &= 1 \\
\overline{\mathcal{R}} &= \overline{\mathcal{L}_{\text{reg}}} \, \lambda \\
\overline{\mathcal{L}} &= \overline{\mathcal{L}_{\text{reg}}} \\
\overline{y} &= \overline{\mathcal{L}} \, (y - t) \\
\overline{z} &= \overline{y} \, \sigma'(z) \\
\overline{w} &= \overline{z} \, x + \overline{\mathcal{R}} \, w \\
\overline{b} &= \overline{z}
\end{aligned}$$

The derivation, and the final result, are much cleaner than with the naïve method. There are no redundant computations here. Furthermore, the procedure is *modular*: it is broken down into small chunks that can be reused for other computations. For instance, if we want to change the loss function, we'd only have to modify the formula for $\overline{y}$. With the naïve method, we'd have to start over from scratch.

Actually, there's one redundant computation, since $\sigma(z)$ can be reused when computing $\sigma'(z)$. But we're not going to focus on this point.

3 Backprop on a multilayer net

Now we come to the prototypical use of backprop: computing the loss derivatives for a multilayer neural net. This introduces no new ideas beyond

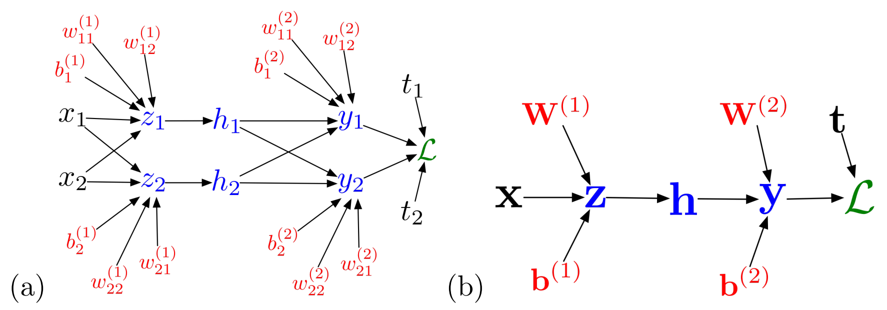


Figure 2: **(a)** Full computation graph for the loss computation in a multilayer neural net. **(b)** Vectorized form of the computation graph.

what we've already discussed, so think of it as simply another example to practice the technique. We'll use a multilayer net like the one from the previous lecture, and squared error loss with multiple output units:

$$\begin{aligned} z_i &= \sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \\ h_i &= \sigma(z_i) \\ y_k &= \sum_i w_{ki}^{(2)} h_i + b_k^{(2)} \\ \mathcal{L} &= \frac{1}{2} \sum_k (y_k - t_k)^2 \end{aligned}$$

As before, we start by drawing out the computation graph for the network. The case of two input dimensions and two hidden units is shown in Figure 2(a). Because the graph clearly gets pretty cluttered if we include all the units individually, we can instead draw the computation graph for the vectorized form (Figure 2(b)), as long as we can mentally convert it to Figure 2(a) as needed.

Based on this computation graph, we can work through the derivations of the backwards pass just as before.

One you get used to it, feel free to skip the step where we write down $\overline{\mathcal{L}}$.

$$\begin{aligned} \overline{\mathcal{L}} &= 1 \\ \overline{y_k} &= \overline{\mathcal{L}}\,(y_k - t_k) \\ \overline{w_{ki}^{(2)}} &= \overline{y_k}\, h_i \\ \overline{b_k^{(2)}} &= \overline{y_k} \\ \overline{h_i} &= \sum_k \overline{y_k} w_{ki}^{(2)} \\ \overline{z_i} &= \overline{h_i}\, \sigma'(z_i) \\ \overline{w_{ij}^{(1)}} &= \overline{z_i}\, x_j \\ \overline{b_i^{(1)}} &= \overline{z_i} \end{aligned}$$

Focus especially on the derivation of $\overline{h_i}$, since this is the only step which actually uses the multivariable Chain Rule.

Once we've derived the update rules in terms of indices, we can find the vectorized versions the same way we've been doing for all our other calculations. For the forward pass:

$$\begin{aligned}
\mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\
\mathbf{h} &= \sigma(\mathbf{z}) \\
\mathbf{y} &= \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \\
\mathcal{L} &= \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2
\end{aligned}$$

And the backward pass:

$$\begin{aligned}
\overline{\mathcal{L}} &= 1 \\
\overline{\mathbf{y}} &= \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t}) \\
\overline{\mathbf{W}^{(2)}} &= \overline{\mathbf{y}}\mathbf{h}^\top \\
\overline{\mathbf{b}^{(2)}} &= \overline{\mathbf{y}} \\
\overline{\mathbf{h}} &= \mathbf{W}^{(2)\top}\overline{\mathbf{y}} \\
\overline{\mathbf{z}} &= \overline{\mathbf{h}} \circ \sigma'(\mathbf{z}) \\
\overline{\mathbf{W}^{(1)}} &= \overline{\mathbf{z}}\mathbf{x}^\top \\
\overline{\mathbf{b}^{(1)}} &= \overline{\mathbf{z}}
\end{aligned}$$

4 Appendix: why the weird notation?

Recall that the partial derivative $\partial\mathcal{E}/\partial w$ means, how much does $\mathcal{E}$ change when you make an infinitesimal change to w, holding everything else fixed? But this isn't a well-defined notion, because it depends what we mean by "holding everything else fixed." In particular, Eqn. 5 defines the cost as a function of two arguments; writing this explicitly,

$$\mathcal{E}(\mathcal{L}, w) = \mathcal{L} + \frac{\lambda}{2}w^2. \tag{10}$$

Computing the partial derivative of this function with respect to w,

$$\frac{\partial\mathcal{E}}{\partial w} = \lambda w. \tag{11}$$

But in the previous section, we (correctly) computed

$$\frac{\partial\mathcal{E}}{\partial w} = (\sigma(wx + b) - t)\sigma'(wx + b)x + \lambda w. \tag{12}$$

What gives? Why do we get two different answers?

The problem is that mathematically, the notation $\partial\mathcal{E}/\partial w$ denotes the partial derivative of a *function* with respect to one of its *arguments*. We make an infinitesimal change to one of the arguments, while holding the rest of the arguments fixed. When we talk about partial derivatives, we need to be careful about what are the arguments to the function. When we compute the derivatives for gradient descent, we treat $\mathcal{E}$ as a function of the parameters of the model — in this case, w and b. In this context,

$\partial \mathcal{E} / \partial w$ means, how much does $\mathcal{E}$ change if we change w while holding b fixed? By contrast, Eqn. 10 treats $\mathcal{E}$ as a function of $\mathcal{L}$ and w; in Eqn. 10, we're making a change to the second argument to $\mathcal{E}$ (which happens to be denoted w), while holding the first argument fixed.

Unfortunately, we need to refer to *both* of these interpretations when describing backprop, and the partial derivative notation just leaves this difference implicit. Doubly unfortunately, our field hasn't consistently adopted any notational conventions which will help us here. There are dozens of explanations of backprop out there, most of which simply ignore this issue, letting the meaning of the partial derivatives be determined from context. This works well for experts, who have enough intuition about the problem to resolve the ambiguities. But for someone just starting out, it might be hard to deduce the meaning from context.

That's why I picked the bar notation. It's the least bad solution I've been able to come up with.