Chapter 1

Backpropagation

A key idea of neural nets is to decompose computation into a series of "layers". In this chapter we will think of layers as modular blocks that can be chained together into a **computation graph**. Here's the computation graph for the two-layer MLP from Chapter **??**:

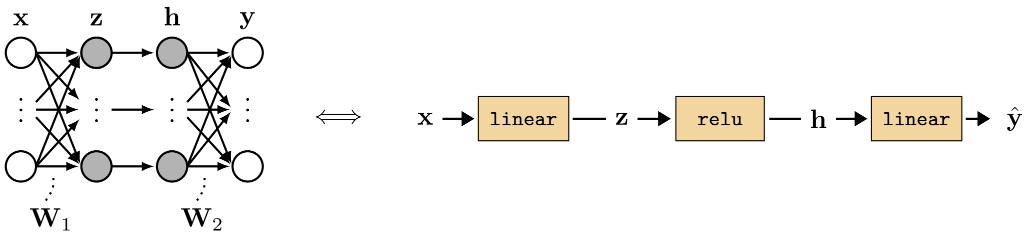
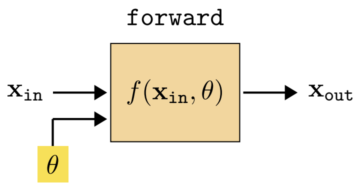


Figure 1.1

Each layer takes in some inputs and transforms them into some outputs. We call this the `forward` pass through the layer. If the layer has parameters, we will consider the parameters to be an *input* to a parameter-free transformation:

$$\mathbf{x}_{\texttt{out}} = f(\mathbf{x}_{\texttt{in}}, \theta) \tag{1.1}$$

Graphically, we will depict the forward operation of a layer like this:



forward
$\mathbf{x}_{\texttt{in}}$
$f(\mathbf{x}_{\texttt{in}}, \theta)$
$\mathbf{x}_{\texttt{out}}$
θ

We will use the color 🟨 to indicate *free* parameters, which are set via learning and are not the result of any other processing. These parameters, along with the input training data, are the leaves in the computation graph.

The learning problem is to find the parameters θ that achieve a desired mapping. Usually we will solve this problem via gradient descent. The question of this chapter is: how do we compute the gradients?

Backpropagation is an algorithm that efficiently calculates the gradient of the loss w.r.t. each and every parameter in a computation graph. It relies on a special new operation, called `backward` that, just like `forward`, can be defined for each layer, and acts in isolation from the rest of the graph. But first, before we get to defining `backward`, we will build up some intuition about the key trick backprop will exploit.

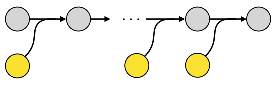
1.1 The trick of backprop: reuse of computation

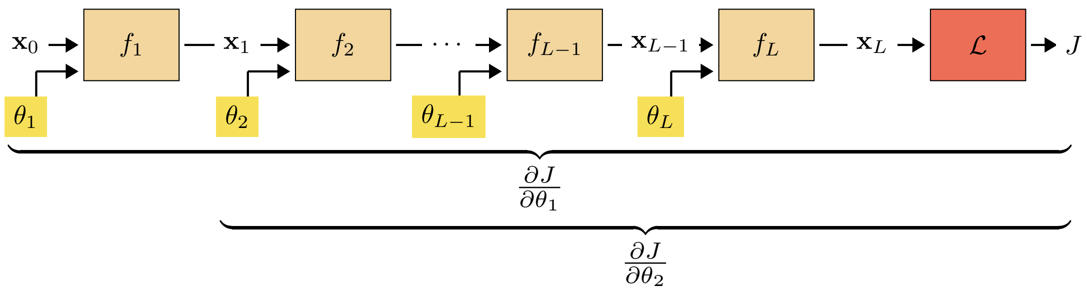
To start, we will consider a simple computation graph that is a chain of functions $f_L \circ f_{L-1} \circ \cdots f_2 \circ f_1$, with each function f_l parameterized by θ_l. We aim to optimize the parameterize

Such a computation graph could represent an MLP, for example, which we will see in the next section.

w.r.t. a loss function $\mathcal{L}$. The loss can be treated as another node in our computation graph, which takes in $\mathbf{x}_L$ (the output of f_L) and outputs a scalar J, the loss. This computation graph looks as follows:

This computation graph is a narrow tree; the parameters live on branches of length 1. This can be easier to see when we plot it with data and parameters as nodes and edges as the functions:





Our goal is to update all the values highlighted in yellow: θ_1, θ_2 and so forth. To do so we need to compute the gradients $\frac{\partial J}{\partial \theta_1}$, $\frac{\partial J}{\partial \theta_2}$, etc. Each of these gradients can be calculated via the chain rule. Here is the chain rule written out for the gradients for θ_1 and θ_2:

$$\frac{\partial J}{\partial \theta_1} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdots \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \theta_1} \tag{1.2}$$

$$\frac{\partial J}{\partial \theta_2} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdots \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \theta_2} \tag{1.3}$$

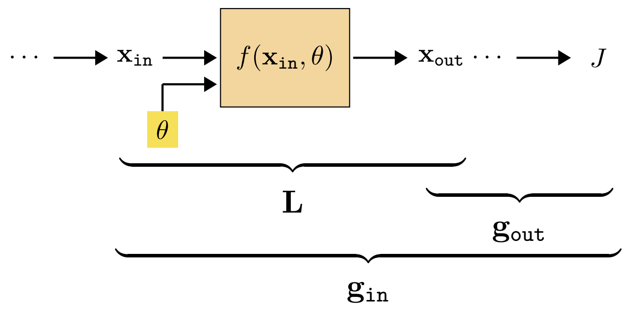
Rather than evaluating both equations separately, we notice that all the terms highlighted in blue are shared. We only need to evaluate this product once, and then can use it to compute both $\frac{\partial J}{\partial \theta_1}$ and $\frac{\partial J}{\partial \theta_2}$. Now notice that this pattern of reuse can be applied in the same way for θ_3, θ_4, and so on. This is the whole trick of backpropagation: rather than computing each layer's gradients independently, observe that they share many of the same terms, so we might as well calculate each shared term once and reuse them.

This strategy, in general, is called **dynamic programming**.

1.2 `Backward` for a generic layer

To come up with a general algorithm for reusing all the shared computation, we will first look at one generic layer in isolation, and see what we need in order to update its parameters.

The braces represent the part of the computation graph we need to consider in order to evaluate $\mathbf{g}_{\texttt{out}}$, $\mathbf{L}$, and $\mathbf{g}_{\texttt{in}}$


$\cdots$
$\mathbf{x}_{\texttt{in}}$
$f(\mathbf{x}_{\texttt{in}}, \theta)$
$\mathbf{x}_{\texttt{out}}$ $\cdots$
J
θ
$\mathbf{L}$
$\mathbf{g}_{\texttt{out}}$
$\mathbf{g}_{\texttt{in}}$

Here we have introduced two new shorthands, $\mathbf{L}$ and $\mathbf{g}$ – these represent arrays of partial derivatives, defined below, and they are the key arrays we need to keep track of to do backprop. They are defined as:

$$\mathbf{g}_l \triangleq \frac{\partial J}{\partial \mathbf{x}_l} \quad \triangleleft \text{ grad of cost w.r.t. } \mathbf{x}_l \quad [1 \times |\mathbf{x}_l|] \tag{1.4}$$

$$\mathbf{L} \triangleq \frac{\partial \mathbf{x}_{\texttt{out}}}{\partial [\mathbf{x}_{\texttt{in}}, \theta]} \quad \triangleleft \text{ grad of layer} \tag{1.5}$$

$$\mathbf{L}^{\mathbf{x}} \triangleq \frac{\partial \mathbf{x}_{\texttt{out}}}{\partial \mathbf{x}_{\texttt{in}}} \quad \triangleleft \text{ ... w.r.t. layer input data} \quad [|\mathbf{x}_{\texttt{out}}| \times |\mathbf{x}_{\texttt{in}}|] \tag{1.6}$$

$$\mathbf{L}^{\theta} \triangleq \frac{\partial \mathbf{x}_{\texttt{out}}}{\partial \theta} \quad \triangleleft \text{ ... w.r.t. layer params} \quad [|\mathbf{x}_{\texttt{out}}| \times |\theta|] \tag{1.7}$$

All these arrays represent the gradient *at a single operating point* – that of the current value of the data and parameters.

These arrays give a simple formula for computing the gradient we need – $\frac{\partial J}{\partial \theta}$ – in order to `update` θ to minimize the cost:

$$\frac{\partial J}{\partial \theta} = \underbrace{\frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}}}_{\mathbf{g}_{\texttt{out}}} \underbrace{\frac{\partial \mathbf{x}_{\texttt{out}}}{\partial \theta}}_{\mathbf{L}^{\theta}} = \mathbf{g}_{\texttt{out}} \mathbf{L}^{\theta} \tag{1.8}$$

$$\theta^{i+1} \leftarrow \theta^i - \eta \left(\frac{\partial J}{\partial \theta} \right)^T \quad \triangleleft \quad \texttt{update} \tag{1.9}$$

The transpose is because, by convention θ is a column vector while $\frac{\partial J}{\partial \theta}$ is a row vector; see Appendix.

The remaining question is just: how do we get $\mathbf{g}_l$ and $\mathbf{L}_l^{\theta}$ for each layer l?

Computing $\mathbf{L}$ is an entirely local process: for each layer, we just need to know the functional form of its derivative, f', which we then evaluate at the operating point $[\mathbf{x}_{\texttt{in}}, \theta]$ to obtain $\mathbf{L} = f'(\mathbf{x}_{\texttt{in}}, \theta)$.

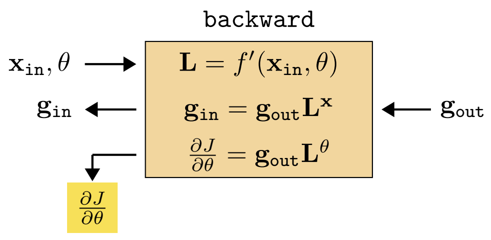
Computing $\mathbf{g}$ is a bit trickier; it requires evaluating the chain rule, and depends on all the layers between $\mathbf{x}_{\texttt{out}}$ and J. However, this can be computed iteratively: once we know $\mathbf{g}_l$, computing $\mathbf{g}_{l-1}$ is just one more matrix multiply! This can be summarized with the following recurrence relation:

$$\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}} \mathbf{L}^{\mathbf{x}} \quad \triangleleft \text{ “backpropagation of errors”} \tag{1.10}$$

This recurrence is is essence of backprop: it sends “error signals” (gradients) backwards through the network, starting at the last layer and iteratively applying Equation 1.10 to compute $\mathbf{g}$ for each previous layer.

Deep learning libraries like Pytorch have a `.grad` field associated with each variable (data, activations, parameters). This field reprsents $\frac{\partial J}{\partial v}$ for each variable v.

We are finally ready to define the full `backward` function promised at the beginning of this chapter! It consists of the following operation, which has three inputs – $\mathbf{x}_{\texttt{in}}, \theta, \mathbf{g}_{\texttt{out}}$– and two outputs – $\mathbf{g}_{\texttt{in}}$ and $\frac{\partial J}{\partial \theta}$:

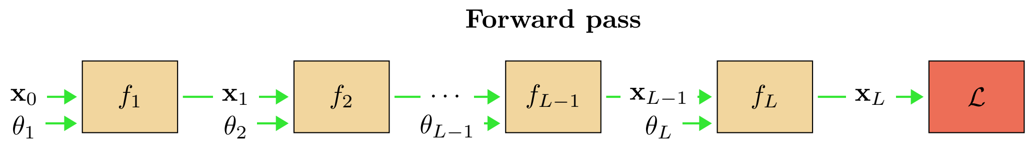

backward
$\mathbf{x}_{\texttt{in}}, \theta$
$\mathbf{L} = f'(\mathbf{x}_{\texttt{in}}, \theta)$
$\mathbf{g}_{\texttt{in}}$
$\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}} \mathbf{L}^{\mathbf{x}}$
$\mathbf{g}_{\texttt{out}}$
$\frac{\partial J}{\partial \theta} = \mathbf{g}_{\texttt{out}} \mathbf{L}^{\theta}$
$\frac{\partial J}{\partial \theta}$

1.3 The full algorithm: forward pass, then backward pass

We are ready now to define the full backprop algorithm. In the last section we saw that we can easily compute the gradient update for θ_l once we have computed $\mathbf{L}_l$ and $\mathbf{g}_l$.

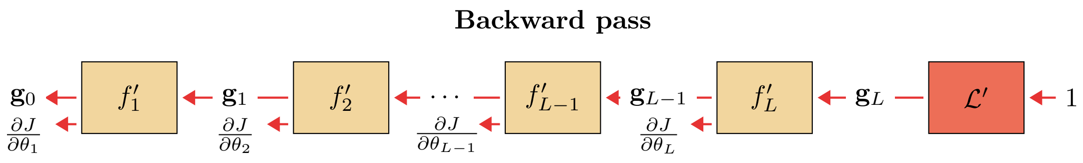
$\mathbf{g}_l$ and $\mathbf{L}_l$ are the $\mathbf{g}$ and $\mathbf{L}$ arrays for layer l.

So, we just need to order our operations so that when we get to updating layer l we have these two arrays ready. The way to do it is to first compute a **forward pass** through the entire network, which means starting with input data $\mathbf{x}_0$ and evaluating layer by layer to produce the sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_L$. Here is what the forward pass looks like:



We needed to run the forward pass before the backward pass because `backward` for each layer requires as input not just $\mathbf{g}_{\texttt{out}}$ but also $\mathbf{x}_{\texttt{in}}$, which we only will know after running the forward pass.

Next, we compute a **backward pass**, iteratively evaluating the $\mathbf{g}$'s and obtaining the sequence $\mathbf{g}_L, \mathbf{g}_{L-1}, \ldots$, as well as the parameter gradients for each layer:


Backward pass
$\mathbf{g}_0$
$\frac{\partial J}{\partial \theta_1}$
f_1'
$\mathbf{g}_1$
$\frac{\partial J}{\partial \theta_2}$
f_2'
$\cdots$
$\frac{\partial J}{\partial \theta_{L-1}}$
f_{L-1}'
$\mathbf{g}_{L-1}$
$\frac{\partial J}{\partial \theta_L}$
f_L'
$\mathbf{g}_L$
$\mathcal{L}'$
1

The full algorithm is summarized in Algorithm **??**.

Algorithm 1: Backpropagation (for chain computation graphs)

```
1  Input: parameter vector $\theta = \{\theta_l\}_{l=1}^{L}$, training datapoint $\{\mathbf{x}_0, \mathbf{y}\}$, "chain"
   computation graph $f_1 \circ \ldots \circ f_L$, Loss function $\mathcal{L} : \mathbb{R}^N \to \mathbb{R}$
2  Output: gradient direction $\frac{\partial J}{\partial \theta} = \{\frac{\partial J}{\partial \theta_l}\}_{l=1}^{L}$
3
4  Forward pass:
5  for $l = 1, \ldots, L$ do
6      $\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \theta_l)$
7
8  Backward pass:
9  $\mathbf{g}_L = \mathcal{L}'(\mathbf{x}_L, y)$
10 for $l = L, \ldots, 1$ do
11     $\mathbf{L}_l = f_l'(\mathbf{x}_{l-1}, \theta_l)$
12     $\mathbf{g}_{l-1} = \mathbf{g}_l \mathbf{L}_l^{\mathbf{x}}$
13     $\frac{\partial J}{\partial \theta_l} = \mathbf{g}_l \mathbf{L}_l^{\theta}$
```

1.4 Example: backprop for an MLP

In order to fully describe backprop for any given architecture, we need $\mathbf{L}$ for each layer in the network. One way to do this is to define the derivative f' for all atomic functions like addition, multiplication, etc, and then expand every layer into a computation graph that involves just these atomic operations. Backprop through the expanded computation graph will then simply make use of all the atomic f's to compute the necessary $\mathbf{L}$ matrices. However, often there are more efficient ways of writing `backward` for standard layers. In this section we will derive a compact `backward` for linear layers and relu layers – the two main layers in MLPs.

1.4.1 Backprop for a linear layer

The definition of a linear layer, in forward direction, is:

$$\mathbf{x}_{\texttt{out}} = \mathbf{W}\mathbf{x}_{\texttt{in}} + \mathbf{b} \tag{1.11}$$

We have separated the parameters into $\mathbf{W}$ and $\mathbf{b}$ for clarity, but remember that we could always rewrite the following in terms of $\theta = \texttt{vec}[\mathbf{W}, \mathbf{b}]$. Let $\mathbf{x}_{\texttt{in}}$ be N-dimensional and $\mathbf{x}_{\texttt{out}}$ be M-dimensional; then $\mathbf{W}$ is an $[M \times N]$ dimensional matrix and $\mathbf{b}$ is an M-dimensional vector.

`vec` is the vectorization operator, which takes numbers in some structured format and rearranges them into a vector.

Next we need the gradients of this function, with respect to its inputs and parameters, i.e. $\mathbf{L}$. Matrix algebra typically hides the details so we will instead first write out all the individual scalar gradients:

Here we define $\mathbf{L}^{\mathbf{W}}$ and $\mathbf{L}^{\mathbf{b}}$ as matrices that store the gradients of each output w.r.t. each weight and bias respectively. The columns of $\mathbf{L}^{\mathbf{W}}$ index over all the MN weights.

$$\mathbf{L}^{\mathbf{x}}[i,j] = \frac{\partial \mathbf{x}_{\texttt{out}}[i]}{\partial \mathbf{x}_{\texttt{in}}[j]} = \frac{\partial \sum_l \mathbf{W}[i,l]\mathbf{x}_{\texttt{in}}[l]}{\partial \mathbf{x}_{\texttt{in}}[j]} = \mathbf{W}[i,j] \tag{1.12}$$

$$\mathbf{L}^{\mathbf{W}}[i,jk] = \frac{\partial \mathbf{x}_{\texttt{out}}[i]}{\partial \mathbf{W}[j,k]} = \frac{\partial \sum_l \mathbf{W}[i,l]\mathbf{x}_{\texttt{in}}[l]}{\partial \mathbf{W}[j,k]} = \begin{cases} \mathbf{x}_{\texttt{in}}[j], & \text{if} \quad i == k \\ 0, & \text{otherwise} \end{cases} \tag{1.13}$$

$$\mathbf{L}^{\mathbf{b}}[i,j] = \frac{\partial \mathbf{x}_{\texttt{out}}[i]}{\partial \mathbf{b}[j]} = \begin{cases} 1, & \text{if} \quad i == j \\ 0, & \text{otherwise} \end{cases} \tag{1.14}$$

Equations 1.12 and 1.14 imply:

$$\boxed{\mathbf{L}^{\mathbf{x}} = \mathbf{W}} \qquad \triangleleft \quad [M \times N] \tag{1.15}$$

$$\boxed{\mathbf{L}^{\mathbf{b}} = \mathbf{I}} \qquad \triangleleft \quad [M \times M] \tag{1.16}$$

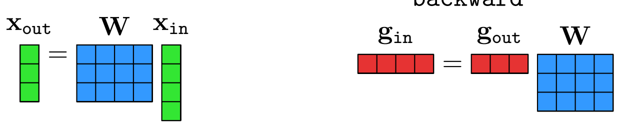
There is no such simply shorthand for $\mathbf{L}^{\mathbf{W}}$, but that is no matter, as we can proceed at this point to implement `backward` for a linear layer by plugging our computed $\mathbf{L}^{\mathbf{x}}$ into Equation 1.10, and $\mathbf{L}^{\mathbf{W}}$ and $\mathbf{L}^{\mathbf{b}}$ into Equation 1.9.

$$\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}}\mathbf{L}^{\mathbf{x}} = \mathbf{g}_{\texttt{out}}\mathbf{W} \tag{1.17}$$

$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{g}_{\texttt{out}}\mathbf{L}^{\mathbf{W}} \tag{1.18}$$

$$\frac{\partial J}{\partial \mathbf{b}} = \mathbf{g}_{\texttt{out}}\mathbf{L}^{\mathbf{b}} = \mathbf{g}_{\texttt{out}} \tag{1.19}$$

To get an intuition for Equation 1.17, it can help to draw the matrices being multiplied. Below, on the left we have the forward operation of the layer (omitting biases) and on the right we have the backward operation in Equation 1.17:


forward
$\mathbf{x}_{\texttt{out}}$
$\mathbf{W}$
$\mathbf{x}_{\texttt{in}}$
=
backward
$\mathbf{g}_{\texttt{in}}$
$\mathbf{g}_{\texttt{out}}$
$\mathbf{W}$
=

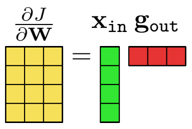
Unlike the other equations, at first glance $\frac{\partial J}{\partial \mathbf{W}}$ does not seem to have a simple form. A naive approach would be to first build out the large sparse matrix $\mathbf{L}^{\mathbf{W}}$ (which is $[M \times MN]$, with zeros wherever $i \neq k$ in $\mathbf{L}^{\mathbf{W}}[i,jk]$), then do the matrix multiply $\mathbf{g}_{\texttt{out}}\mathbf{L}^{\mathbf{W}}$. We can avoid all those multiplications by zero by observing the following simplification:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{W}[i,j]} &= \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}}\frac{\partial \mathbf{x}_{\texttt{out}}}{\partial \mathbf{W}[i,j]} && \triangleleft [1\times M][M\times 1] \rightarrow [1\times 1] \quad (1.20)\\
&= \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}}[\frac{\partial \mathbf{x}_{\texttt{out}}[1]}{\partial \mathbf{W}[i,j]},\dots,\frac{\partial \mathbf{x}_{\texttt{out}}[M]}{\partial \mathbf{W}[i,j]}]^T && (1.21)\\
&= \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}}[\dots,0,\dots,\frac{\partial \mathbf{x}_{\texttt{out}}[i]}{\partial \mathbf{W}[i,j]},\dots,0,\dots]^T && (1.22)\\
&= \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}}[\dots,0,\dots,\mathbf{x}_{\texttt{in}j},\dots,0,\dots]^T && (1.23)\\
&= \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}[i]}\mathbf{x}_{\texttt{in}}[j] && \triangleleft [1\times 1][1\times 1] \rightarrow [1\times 1] \quad (1.24)
\end{aligned}$$

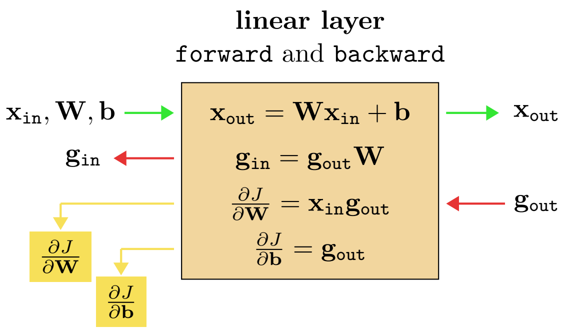
Now we can just arrange all these scalar derivatives into the matrix for $\frac{\partial J}{\partial \mathbf{W}}$, and obtain the following:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{W}} &= \begin{bmatrix} \frac{\partial J}{\partial \mathbf{W}[1,1]} & \cdots & \frac{\partial J}{\partial \mathbf{W}[N,1]} \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial \mathbf{W}[1,M]} & \cdots & \frac{\partial J}{\partial \mathbf{W}[N,M]} \end{bmatrix} && (1.25)\\
&= \begin{bmatrix} \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}1}}\mathbf{x}_{\texttt{in}}[1] & \cdots & \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}[N]}\mathbf{x}_{\texttt{in}}[1] \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}[1]}\mathbf{x}_{\texttt{in}}[M] & \cdots & \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}[N]}\mathbf{x}_{\texttt{in}}[M] \end{bmatrix} && (1.26)\\
&= \mathbf{x}_{\texttt{in}}\frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}} && (1.27)\\
&= \mathbf{x}_{\texttt{in}}\mathbf{g}_{\texttt{out}} && (1.28)
\end{aligned}$$

So we see that in the end this gradient has the simple form of an outer product between two vectors, $\mathbf{x}_{\texttt{in}}$ and $\mathbf{g}_{\texttt{out}}$:


$\frac{\partial J}{\partial \mathbf{W}}$
$=$
$\mathbf{x}_{\texttt{in}}$
$\mathbf{g}_{\texttt{out}}$

We can summarize all these operations in the following `forward` and `backward` diagram for linear layer:


linear layer
`forward` and `backward`
$\mathbf{x}_{\texttt{in}}, \mathbf{W}, \mathbf{b}$
$\mathbf{x}_{\texttt{out}} = \mathbf{W}\mathbf{x}_{\texttt{in}} + \mathbf{b}$
$\mathbf{x}_{\texttt{out}}$
$\mathbf{g}_{\texttt{in}}$
$\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}}\mathbf{W}$
$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{x}_{\texttt{in}}\mathbf{g}_{\texttt{out}}$
$\mathbf{g}_{\texttt{out}}$
$\frac{\partial J}{\partial \mathbf{b}} = \mathbf{g}_{\texttt{out}}$
$\frac{\partial J}{\partial \mathbf{W}}$
$\frac{\partial J}{\partial \mathbf{b}}$

Notice that all these operations are simple expressions, mainly involving matrix multiplies. `Forward` and `backward` for a linear layer are also very easy to write in code, using any library that provides matrix multiplication (`matmul`) as a primitive. Here is Python code for this layer:

```
class linear():
    def __init__(self, W, b, lr):
        self.W = W
        self.b = b
        self.lr = lr # learning rate

    def forward(self, x_in):
        self.x_in = x_in
        return matmul(W,x)+b

    def backward(self,J_out):
        J_in = matmul(J_out,W)
        dJdW = matmul(self.x_in,J_out)
        dJdb = J_out
        return J_in, dJdW, dJdb

    def update(self, dJdW, dJdb):
        self.W -= self.lr*dJdW.transpose()
        self.b -= self.lr*dJdb
```

1.4.2 Backprop for a pointwise nonlinearity

Pointwise nonlinearities have very simple `backward` functions. Let a (parameterless) scalar nonlinearity be $h : \mathbb{R} \to \mathbb{R}$ with derivative function $h' : \mathbb{R} \to \mathbb{R}$. Define a pointwise layer using h as $f(\mathbf{x}_{\texttt{in}}) = [h(\mathbf{x}_{\texttt{in}}[1]), \dots, h(\mathbf{x}_{\texttt{in}}[N])]^T$. Then we have

$$\mathbf{L}^{\mathbf{x}} = f'(\mathbf{x}_{\texttt{in}}) = \texttt{diag}([h'(\mathbf{x}_{\texttt{in}}[1]), \dots, h'(\mathbf{x}_{\texttt{in}}[N])]^T) \triangleq \mathbf{H}' \tag{1.29}$$

`diag` is the operator that places a vector on the diagonal of a matrix, whose other entries are all zero.

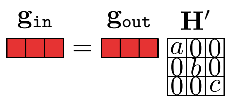
There are no parameters to update, so we just have to calculate $\mathbf{g}_{\texttt{in}}$ in the `backward` operation, using Equation 1.10:

$$\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}}\mathbf{H}' \tag{1.30}$$

As an example, for a `relu` layer we have:

$$h'(x) = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.31}$$

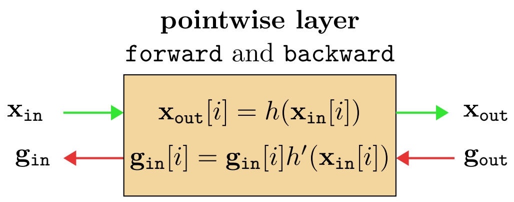
As a matrix multiply, the `backward` operation looks like this:



with $a = h'(\mathbf{x}_{\texttt{in}1})$, $a = h'(\mathbf{x}_{\texttt{in}2})$, and $a = h'(\mathbf{x}_{\texttt{in}3})$. We can simplify this equation as follows:

$$\mathbf{g}_{\texttt{in}}[i] = \mathbf{g}_{\texttt{in}}[i]h'(\mathbf{x}_{\texttt{in}}[i]) \quad \forall i \tag{1.32}$$

The full set of operations for a pointwise layer then looks like this:


pointwise layer
forward and backward
$\mathbf{x}_{\texttt{in}}$
$\mathbf{x}_{\texttt{out}}[i] = h(\mathbf{x}_{\texttt{in}}[i])$
$\mathbf{x}_{\texttt{out}}$
$\mathbf{g}_{\texttt{in}}$
$\mathbf{g}_{\texttt{in}}[i] = \mathbf{g}_{\texttt{in}}[i]h'(\mathbf{x}_{\texttt{in}}[i])$
$\mathbf{g}_{\texttt{out}}$

1.4.3 Backprop for loss layers

The last layer we need to define for a complete MLP is the loss layer. As a simple example, we will derive backprop for an L_2 loss function: $\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$, where $\hat{\mathbf{y}}$ is the output of the network (prediction) and $\mathbf{y}$ is the ground truth.

This layer has no parameters so we only need to derive Equation 1.10 for this layer:

$$\mathbf{L}^{\mathbf{x}} = \frac{\partial \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2}{\partial \hat{\mathbf{y}}} = \hat{\mathbf{y}} - \mathbf{y} \qquad \triangleleft \quad [1 \times |\mathbf{y}|] \tag{1.33}$$

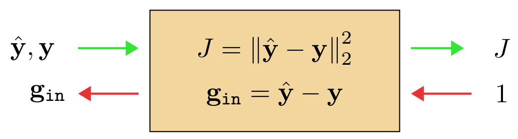
$$\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}}(\hat{\mathbf{y}} - \mathbf{y}) = \hat{\mathbf{y}} - \mathbf{y} \tag{1.34}$$

Here we have made use of the fact that $\mathbf{g}_{\texttt{out}} = \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}} = \frac{\partial J}{\partial J} = 1$, since the output of the loss layer *is* the cost J.

So, the backward signal sent by the L_2 loss layer is a row vector of per-dimension errors between the prediction and the target.

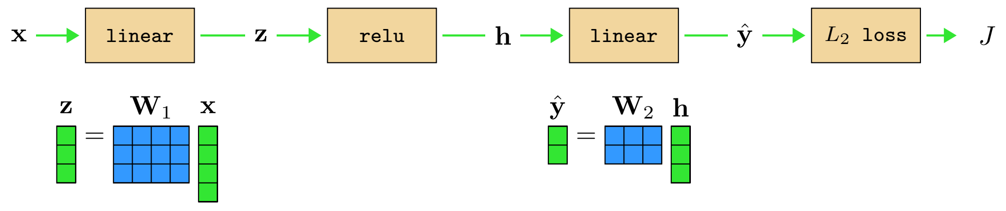
This completes our derivation of `forward` and `backward` for a L_2 loss layer:

L_2 loss layer
`forward` and `backward`


$\hat{\mathbf{y}}, \mathbf{y}$
$\mathbf{g}_{\texttt{in}}$
$J = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$
$\mathbf{g}_{\texttt{in}} = \hat{\mathbf{y}} - \mathbf{y}$
J
1

1.4.4 Putting it all together: backprop through an MLP

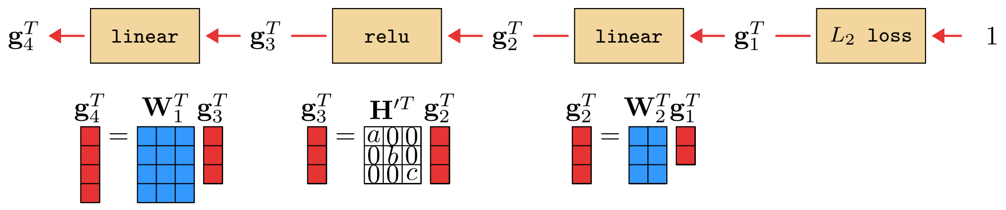
Let's see what happens when we put all these operations together in an MLP. We will start with the MLP in Figure 1.1. For simplicity, we will omit biases. Let $\mathbf{x}$ be 4-dimensional and $\mathbf{z}$ and $\mathbf{h}$ be 3-dimensional, and $\hat{\mathbf{y}}$ be 2-dimensional. Then the forward pass looks like this:



For the backward pass, we will here make a slight change in convention, which will clarify an interesting connection between the forward and backward directions. Rather than representing gradients $\mathbf{g}$ as row vectors, we will transpose them and treat them as column vectors. The `backward` operation for transposed vectors follows from the matrix identity that $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$:

$$\mathbf{g}_{\texttt{in}}^T = (\mathbf{g}_{\texttt{out}}\mathbf{W})^T = \mathbf{W}^T\mathbf{g}_{\texttt{out}}^T \tag{1.35}$$

Now we will write the backward pass using these transposed $\mathbf{g}$'s:


$\mathbf{g}_4^T$
linear
$\mathbf{g}_3^T$
relu
$\mathbf{g}_2^T$
linear
$\mathbf{g}_1^T$
L_2 loss
1
$\mathbf{g}_4^T = \mathbf{W}_1^T \mathbf{g}_3^T$
$\mathbf{g}_3^T = \mathbf{H}'^T \mathbf{g}_2^T$
a 0 0
0 b 0
0 0 c
$\mathbf{g}_2^T = \mathbf{W}_2^T \mathbf{g}_1^T$

This reveals an interesting connection between `forward` and `backward` for linear layers: `backward` for a linear layer is the same operation as `foward`, just with the weights transposed! We have omitted the bias terms here, but recall from Equation 1.17 that the backward pass to the activations ignores biases anyway.

In contrast, the `relu` layer is not a `relu` on the backward pass. Instead, it becomes a sort of "gating" matrix, parameterized by functions of the activations from the forward pass (a, b, and c). This matrix is all zeros except for ones on the diagonal where the activation was non-negative. This layer acts to mask out gradients for variables on the negative side of the `relu`. Notice that this operation is a matrix multiply – in fact, *all* `backward` operations are matrix multiplies, no matter what the `forward` operation might be, as you can observe in Algorithm 13.

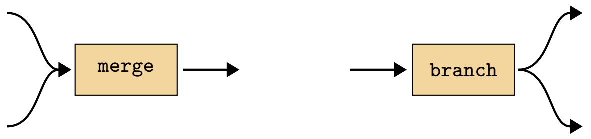
1.5 Backprop through DAGs: branch and merge

So far we have only seen chain-like graphs, –[]–[]–[]→. Can backprop handle other graphs? It turns out the answer is *yes*. Presently we will consider **directed acyclic graphs**, or **DAGs**. In Chapter **??**, we will see that neural nets can also include cycles and still be trained with variants of backprop (e.g., backprop through time).

In a DAG, nodes can have multiple inputs and multiple outputs. In fact, we have already seen several examples of such nodes in the preceding sections. For example, a linear layer can be thought of as having two inputs, $\mathbf{x}_{\texttt{in}}$ and θ, and one output $\mathbf{x}_{\texttt{out}}$; or it can be thought of as having $N = |\mathbf{x}_{\texttt{in}}| + |\theta|$ inputs and $M = |\mathbf{x}_{\texttt{out}}|$ outputs, if we count up each dimension of the input and output vectors. So we have already seen DAG computation graphs.

However, to work with general DAGs, it helps to introduce two new special modules, which act to construct the topology of the graph. We will call these special operators `merge` and `branch`:

We only consider binary branching and merging here, but branching and merging N ways can be done analogously; or you can simply apply `branch`/`merge` N times in a row to achieve the same effect.



We define them mathematically as variable concatenation and copying, respectively:

$$\texttt{merge}(\mathbf{x}^a_{\texttt{in}}, \mathbf{x}^b_{\texttt{in}}) \triangleq [\mathbf{x}^a_{\texttt{in}}, \mathbf{x}^b_{\texttt{in}}] \triangleq \mathbf{x}_{\texttt{out}} \tag{1.36}$$

$$\texttt{branch}(\mathbf{x}_{\texttt{in}}) \triangleq [\mathbf{x}_{\texttt{in}}, \mathbf{x}_{\texttt{in}}] \triangleq [\mathbf{x}^a_{\texttt{out}}, \mathbf{x}^b_{\texttt{out}}] \tag{1.37}$$

What if $\mathbf{x}^a_{\texttt{in}}$ and $\mathbf{x}^b_{\texttt{in}}$ are tensors, or other objects, with different shapes? Can we still concatenate them? The answer is yes. The shape of the data tensor has no impact on the math. We pick the shape just as a notational convenience, e.g., it's natural to think about images as 2D arrays.

`merge` takes two inputs and concatenates them. This results in a new multidimensional variable. The backward pass equation is trivial. To compute the gradient of the cost w.r.t. $\mathbf{x}^a_{\texttt{in}}$, i.e. $\mathbf{g}^a_{\texttt{in}}$, we have

$$\mathbf{g}^a_{\texttt{in}} = \mathbf{g}_{\texttt{out}}\mathbf{L}^{\mathbf{x}^a} = \frac{\partial J}{\partial \mathbf{x}_{\texttt{out}}}\frac{\partial \mathbf{x}_{\texttt{out}}}{\partial \mathbf{x}^a_{\texttt{in}}} \tag{1.38}$$

$$= \mathbf{g}_{\texttt{out}}[\frac{\partial \mathbf{x}^a_{\texttt{in}}}{\partial \mathbf{x}^a_{\texttt{in}}}, \frac{\partial \mathbf{x}^b_{\texttt{in}}}{\partial \mathbf{x}^a_{\texttt{in}}}]^T \tag{1.39}$$

$$= \mathbf{g}_{\texttt{out}}[1, 0]^T \tag{1.40}$$

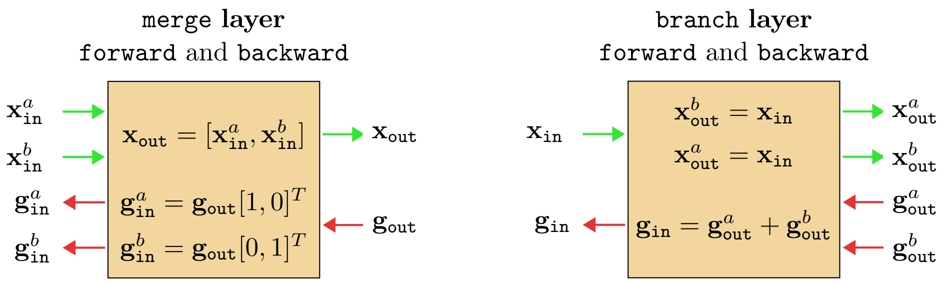
and likewise for $\mathbf{g}^b_{\texttt{in}}$. That is, we just pick out the first half of the $\mathbf{g}_{\texttt{out}}$ gradient vector for $\mathbf{g}^a_{\texttt{in}}$ and the second half for $\mathbf{g}^b_{\texttt{in}}$. There is really nothing new here. We already defined backprop for multidimensional variables above, and `merge` is just an explicit way of constructing multidimensional variables.

`branch` is only slightly more complicated. In branching, we send *copies* of the same output to multiple downstream nodes. Therefore, we have multiple gradients coming back to the `branch` module, each from different downstream paths. So the inputs to this module on the backward pass are $\frac{\partial J}{\partial \mathbf{x}^a_{\texttt{in}}}, \frac{\partial J}{\partial \mathbf{x}^b_{\texttt{in}}}$, which we can write as the gradient vector $\mathbf{g}_{\texttt{out}} = \frac{\partial J}{\partial [\mathbf{x}^a_{\texttt{in}}, \mathbf{x}^b_{\texttt{in}}]} = [\frac{\partial J}{\partial \mathbf{x}^a_{\texttt{in}}}, \frac{\partial J}{\partial \mathbf{x}^b_{\texttt{in}}}] = [\mathbf{g}^a_{\texttt{out}}, \mathbf{g}^b_{\texttt{out}}]$. Let's compute the backwards pass output:

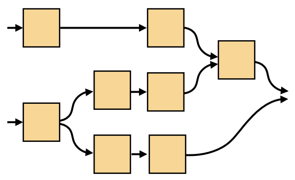
$$\begin{aligned}
\mathbf{g}_{\texttt{in}} &= \mathbf{g}_{\texttt{out}} \mathbf{L}^{\mathbf{x}} && (1.41)\\
&= [\mathbf{g}^a_{\texttt{out}}, \mathbf{g}^b_{\texttt{out}}] \frac{\partial [\mathbf{x}^a_{\texttt{out}}, \mathbf{x}^b_{\texttt{out}}]}{\partial \mathbf{x}_{\texttt{in}}} && (1.42)\\
&= [\mathbf{g}^a_{\texttt{out}}, \mathbf{g}^b_{\texttt{out}}] \frac{\partial [\mathbf{x}_{\texttt{in}}, \mathbf{x}_{\texttt{in}}]}{\partial \mathbf{x}_{\texttt{in}}} && (1.43)\\
&= [\mathbf{g}^a_{\texttt{out}}, \mathbf{g}^b_{\texttt{out}}][1, 1]^T && (1.44)\\
&= \mathbf{g}^b_{\texttt{out}} + \mathbf{g}^b_{\texttt{out}} && (1.45)
\end{aligned}$$

So, branching just sums both the gradients passed backwards to it.

Both `merge` and `branch` have no parameters, so there is no parameter gradient to define. Thus, we have fully specified the forward and backward behavior of these layers. These diagrams summarize the behavior:

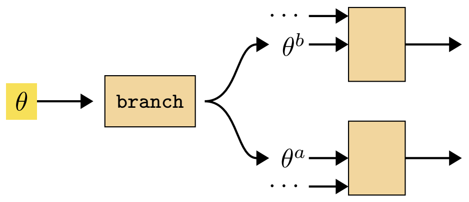

merge layer
forward and backward
$\mathbf{x}^a_{\texttt{in}}$
$\mathbf{x}^b_{\texttt{in}}$
$\mathbf{g}^a_{\texttt{in}}$
$\mathbf{g}^b_{\texttt{in}}$
$\mathbf{x}_{\texttt{out}} = [\mathbf{x}^a_{\texttt{in}}, \mathbf{x}^b_{\texttt{in}}]$
$\mathbf{g}^a_{\texttt{in}} = \mathbf{g}_{\texttt{out}}[1, 0]^T$
$\mathbf{g}^b_{\texttt{in}} = \mathbf{g}_{\texttt{out}}[0, 1]^T$
$\mathbf{x}_{\texttt{out}}$
$\mathbf{g}_{\texttt{out}}$
branch layer
forward and backward
$\mathbf{x}_{\texttt{in}}$
$\mathbf{g}_{\texttt{in}}$
$\mathbf{x}^b_{\texttt{out}} = \mathbf{x}_{\texttt{in}}$
$\mathbf{x}^a_{\texttt{out}} = \mathbf{x}_{\texttt{in}}$
$\mathbf{g}_{\texttt{in}} = \mathbf{g}^a_{\texttt{out}} + \mathbf{g}^b_{\texttt{out}}$
$\mathbf{x}^a_{\texttt{out}}$
$\mathbf{x}^b_{\texttt{out}}$
$\mathbf{g}^a_{\texttt{out}}$
$\mathbf{g}^b_{\texttt{out}}$

With `merge` and `branch`, we can constuct any DAG computation graph by simply inserting these layers wherever we want a layer to have multiple inputs or multiple outputs.


An example of a DAG computation graph that we can construct, and do backprop through, with the tools defined above.

1.6 Parameter sharing

Parameter sharing consists of a single parameter being sent as input to multiple different layers. We can consider this as a branching operation, like so:



Then, from the previous section, it is clear that gradients summate for shared parameters. Let $\{\theta^i\}_{i=1}^N$ be a set of variables that are all copies of one free parameter θ. Then,

$$\frac{\partial J}{\partial \theta} = \sum_i \frac{\partial J}{\partial \theta^i} \tag{1.46}$$

Neural net layers that have their parameters shared in this way are sometimes said to use **tied weights**.

1.7 Backprop to the data

Backprop does not distinguish between parameters and data — it treats both as generic inputs to parameterless modules. Therefore, we can use backprop to optimize data inputs to the graph just like we can use backprop to optimize parameter inputs to the graph.

This can be useful for lots of different applications. One example is visualizing the input image that most activates a given neuron in a neural net. Here is what that looks like:

1.8 Concluding remarks

Bibliography