Back propagation

Gradient Decent works by updating weights $\mathbf{W}_{(l)}$ by the derivative of the loss $\mathcal{L}$ with respect to $\mathbf{W}_{(l)}$.

We will now show how the derivatives

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} \text{ for } l = 1, \ldots, L$$

are computed, first mathematically and then by code.

We can do this via a procedure know as back propagation

It is really nothing more than an *iterated* application of the Chain Rule of Calculus.

Let

- $\mathcal{L}$ denote loss (computed after final layer L)
- $\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial y_{(l)}}$ denote the derivative of $\mathcal{L}$ with respect to the output of layer l, i.e., $y_{(l)}$,
    - refer to as **loss gradient** (at output of layer l)

We will show how to compute

- $\frac{\partial \mathcal{L}}{\partial W_{(l)}}$, from $\mathcal{L}'_{(l)}$ for $l \in [1, L]$

We will also show how to compute

- $\mathcal{L}'_{(l-1)}$ from $\mathcal{L}'_l$
    - so that we can continue this process as the previous layer (i.e, *propogate loss gradient backwards*)

Note that $\mathbf{y}_{(l)}$ is a function of $\mathbf{y}_{(l-1)}$ (the output of the previous layer) and $\mathbf{W}_{(l)}$, the parameters of layer l.

We can compute derivatives of $\mathbf{y}_{(l)}$ with respect to each of its inputs

- $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$
- $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}}$

Refer to these as **local gradients**

Note that we can compute the local gradients during the **forward pass** as the derivatives only depend on inputs and not on any value subsequent to layer l

Recall we defined the loss as being pseudo layer $L+1$

$$\mathbf{y}_{(L+1)} = \mathcal{L}$$

Then $\mathcal{L}'_{(L+1)} = 1$ and this is the base for our backwards computation of $\mathcal{L}'_{(l)}, l < L+1$.

Similarly we consider the input $\mathbf{x}$ to be the "output" of layer 0: $\mathbf{y}_{(0)} = \mathbf{x}$

We can use the chain rule to obtain the

- gradient with respect to weights $\mathbf{W}_{(l)}$, given the loss gradient $\mathcal{L}'_{(l)}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}} = \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{W}_{(l)}}$$

That is:

- gradient of $\mathcal{L}$ with respect to weight $\mathbf{W}_{(l)}$
- is the loss gradient (at current step), multiplied by
- a local gradient (with respect to input $W_{(l)}$)

So we have the information required to update $\mathbf{W}_{(l)}$ by Gradient Descent.

Now that we have gradients for layer l, we need to work backward to layer $l-1$.

For the loss gradient:

$$\begin{aligned}\mathcal{L}'_{(l-1)} &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l-1)}} \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} \\ &= \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}\end{aligned}$$

That is, the loss gradient of the preceding layer is the product of

- the loss gradient of the current layer
- the local gradient with respect to the layer's inputs

This "backwards" computation may seem odd.

After all, when we encounter layer l on the forward pass we have **no idea**

- what the subsequent layers look like
- what the loss (which depends on $\mathbf{y}_{(L)}$) will be

None the less:

- all you need to know is we compute $\mathcal{L}'_{(l)}$ from deep l to shallow

Vanishing/exploding gradients

Now that we have a better view of how backward propagation of gradients work, we are equipped to understand the difficulties of training the weights.

Until the problems were understood, and solutions found, the evolution of Deep Learning was extremely slow.

Let's summarize back propagation up until this point

- We compute the loss gradient $\mathcal{L}'_{(l)} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_{(l)}}$ of each layer l in descending order
- The backward step to compute the loss gradient of the preceding layer is:
  - $\mathcal{L}'_{(l-1)} = \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$

When we derived back propagation, we didn't look "inside" of the "local gradient " $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}}$

We will do so now.

Let's look more deeply into the term $\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(i-1)}}$

$$
\begin{aligned}
\frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} &= \frac{\partial a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}))}{\partial \mathbf{y}_{(l-1)}} && (\text{def. of } \mathbf{y}_{(l)}) \\
&= \frac{\partial a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, W_{(l)}))}{\partial f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)})} \frac{\partial f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)})}{\partial \mathbf{y}_{(l-1)}} && (\text{chain rule}) \\
&= a'_{(l)} f'_{(l)}
\end{aligned}
$$

where we define

$$
\begin{aligned}
a'_{(l)} &= \frac{\partial a_{(l)}(f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)}))}{\partial f_{(l)}(\mathbf{y}_{(l-1)}, \mathbf{W}_{(l)})} \\
f'_{(l)} &= \frac{\partial f_{(l)}(\mathbf{y}_{(l-1)}, W_{(l)})}{\partial \mathbf{y}_{(l-1)}}
\end{aligned}
$$

$a'_{(l)}$ is the derivative of activation function $a_{(l)}$.

We won't expicitly write it out other than to observe $a'_{(l)} \in [0, 1]$.

Substituting the value of the loss gradient into the backward update rule:

$$\begin{aligned} \mathcal{L}'_{(l-1)} &= \mathcal{L}'_{(l)} \frac{\partial \mathbf{y}_{(l)}}{\partial \mathbf{y}_{(l-1)}} \\ &= \mathcal{L}'_{(l)} a'_{(l)} f'_{(l)} \end{aligned}$$

Hopefuly, you can see that if iterate through single backward steps, we can derive an expression for the loss gradient at layer l in terms of the loss gradient of the final layer K:

Since

$$\mathcal{L}'_{(l)} = \mathcal{L}'_{(l+1)} \frac{\partial \mathbf{y}_{(l+1)}}{\partial \mathbf{y}_{(l)}}$$

we get

$$\mathcal{L}'_{(l)} = \mathcal{L}'_{(L+1)} \prod_{l'=i+1}^{L} a'_{(l')} f'_{(l')}$$

The issue is that, since

$$0 \leq a'_{(l)} \leq \max_z a'_{(l)}(z)$$

the product

$$\prod_{l'=i+1}^{K} a'_{(l')}$$

can be increasingly small as the number of layers K grows, if $\max_z a'_{(l)}(z) < 1$.

Note, for $a_{(l)} = \sigma$ (the sigmoid function), $\max_z a'_{(l)}(z) = 0.25$

Thus, unless offset by the $f'_{(l)}$ terms, $\mathcal{L}'_{(l)}$ will quickly diminish to 0 as K decreases, i.e., as we seek to compute $\mathcal{L}'_{(l)}$ for layers l closest to the input. This means

$$\frac{\partial \mathcal{L}}{\partial W_{(l)}} = \frac{\partial \mathcal{L}}{\partial y_{(l)}} \frac{\partial y_{(l)}}{\partial W_{(l)}} = \mathcal{L}'_{(l)} \frac{\partial y_{(l)}}{\partial W_{\mathbf{(i)}}}$$

will approach 0. Since this term is used in the update to $W_{\mathbf{(i)}}$, we won't learn weights for the earliest layers.

We can now diagnose one reason that training of early Deep Learning networks was difficult

- use of sigmoid activations were common (inspired by biology)
- if activations were very large/small, we are in a region where the sigmoid's derivatives are 0
- even when non-zero,the maximum of the derivative of the sigmoid is much smaller than 1
- the end result was that deep networks suffered from Vanishing Gradients

The ReLU function's derivative does not suffer from this problem and ReLU's now tend to be the standard activation (barring other considerations, such as the range of outputs)

What took so long ?

Why did it take so long for Deep Learning to achieve success.

An historical perspective:

- Perceptron invented 1957
- mid-1970's: First "AI Winter"
- Late 1980's: secibd "AI Winter"
- 2010: Re-emergence of AI

The promise of AI led to great expectations, that were ultimately unfulfilled. The difficulty was the inability to train networks.

We now spend some time investigating the causes, and solutions, to the difficulty of training networks.

Broadly speaking the issues are

- proper scaling of the inputs
- initialization of learnable weights
- making sure that the proper scaling of inputs continues to each layer, not just the input

Proper scaling of inputs

Importance of zero centered inputs (for each layer)

Efficient Backprop paper, LeCunn98 (http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf)

Zero centered means average (over the training set) value of each feature of examples is mean 0.

Gradient descent updates each element of a layer l's weights $\mathbf{W}_{(l)}$ by the per-example losses

$$\frac{\partial \mathcal{L}^{(\mathbf{i})}}{\partial W_{(l)}} = \frac{\partial \mathcal{L}^{(\mathbf{i})}}{\partial \mathbf{y}^{(\mathbf{i})}_{(l)}} \frac{\partial \mathbf{y}^{(\mathbf{i})}_{(l)}}{\partial \mathbf{W}_{(l)}}$$

summed over examples i.

Over-simplifying:

- the local derivative is proportional to the input:
$$\frac{\partial \mathbf{y}_{(l)}^{(\mathbf{i})}}{\partial \mathbf{W}_{(l)}} = a'_{(l)} \mathbf{y}_{(l-1)}^{(\mathbf{i})}$$
for FC $y_{(l)} = a_{(l)}(\mathbf{y}_{(l-1)} \mathbf{W}_{(l)})$.
- thus the updates of $\mathbf{W}_{(l),j}$ will be biased by $\bar{\mathbf{y}}_{(l-1),j}$ = the average (over examples i) of $\mathbf{y}_{(l-1),j}^{(\mathbf{i})}$
- for $l = 1$, this is the average of the input feature $\mathbf{x}_j$

In the particular case that each feature j's average $\bar{\mathbf{x}}_j$ has the same sign:

- updates in all dimensions will have the same sign
- this can result in an indirect "zig-zag" toward the optimum
    - Exampe: two dimensions:
        - We can navigate the loss surface north-east or south-west only !
        - To get to a point north-west from the current, we have to zig-zag.

- Note that this is an issue for *all* layers, not just layer $l = 1$.
- Also note: the problem is compounded by activations whose outputs are not zero-centered (e.g., ReLU, sigmoid)

Importance of unit variance inputs (weight initialization)

The same argument we made for zero-centering a feature can be extended to it's variance:

- the variance of feature j over all trainng examples i is the varaince of $\mathbf{y}_{(l-1),j}$

If the variance of features j and j' are different, their updates will happen at different rates.

We will examine this in greater depth during our discussion of weight initialization.

For now: it is desirable that the input to *each* layer have it's features somewhat normalized.

Initialization

Training is all about discovering good weights.

As prosaic as it sounds: how do we *initialize* the weights before training ? Does it matter ?

It turns out that the choice of initial weights does matter.

Let's start with some *bad* choices

Bad choices

Too big/small

Layers usually consist of linear operations (e.g., matrix multiplication and addition of bias) followed by a non-linear activation. The range of many activation functions includes large regions where the derivatives are near zero, usually corresponding to very large/small activations.

The SGD update rule uses the magnitude of the gradient to update weights. Obviously, if the gradients are all near-0, learning cannot occur.

So one bad choice is any set of weights that tends to push activations to regions of the non-linear activation with zero gradient.

Identical weights

Consider layer l with n_l units (neurons) implementing identical operations (e.g. FC + ReLu).

Let $\mathbf{W}_{(l),k}$ denote the weights of unit k.

Suppose we initialized the weights (and biases) of all units to the *same* vector.

$$\mathbf{W}_{(l),k} = \mathbf{w}_{(l)}\,,\ 1 \leq k \leq n_l$$

Consider two neuron j, j' in the same layer l

$$\begin{aligned} \mathbf{y}_{(l),j} &= a_{(l)}(\mathbf{w}_{(l)}\mathbf{y}_{(l-1)} + \mathbf{b}_{(l)}) \\ \mathbf{y}_{(l),j'} &= a_{(l)}(\mathbf{w}_{(l)}\mathbf{y}_{(l-1)} + \mathbf{b}_{(l)}) \end{aligned}$$

- Both neuron will compute the same activation
- Both neurons will have the same gradient
- Both neurons will have the same weight update

Thus, the weights in layer i will start off identical and will remain identical due to identical updates.

So identical initialization will lead to a failure for individual neurons to learn different features.

Many approaches use some for of random initialization to break the symmetry we just described.

Glorot initialization

We have previousy shown that each element j of the first input layer ($\mathbf{x}_{(0),j}$) should have unit variance across the training set.

This was meant to ensure that the first layer's weights updated at the same rate and that the activations of the first layer fell into regions of the activation function that had non-zero gradients.

But this is not enough.

Let's assume for the moment that each element j of the input vector $\mathbf{y}_{(l-1)}$ is mean 0, unit variance and mutually independent.

So view each $\mathbf{y}_{(l-1),j}$ as an independent random variable with mean 0 and unit variance. Furthermore, let's assome each element $\mathbf{W}_{(l),j}$ is similarly distributed.

Consider the matrix multiplication in layer l involving the n_{l-1} output of layer l.

$$f_{(l)}(\mathbf{y}_{(l-1)}, W_{(l)}) = \mathbf{y}_{(l-1)} \cdot W_{(l)}$$

This expression is the weighted sum over j of the product of

- a mean 0, unit variance random variable $\mathbf{y}_{(l-1),j}$
- a mean 0, unit variancerandom variable $\mathbf{W}_{(l),j}$

For two random variables X, Y, the variance of the product [is (https://en.wikipedia.org/wiki/Variance#Product_of_independent_variables)]

$$\operatorname{Var}(XY) = \mathbb{E}(X)^2 \operatorname{Var}(Y) + \mathbb{E}(Y)^2 \operatorname{Var}(X) + \operatorname{Var}(X)\operatorname{Var}(Y)$$

So

$$\operatorname{Var}(\mathbf{y}_{(l-1),j}\mathbf{W}_{(l),j}) = 0^2 * 1 + 0^2 * 1 + 1 * 1 = 1$$

Thus the variance of the dot product of n_{l-1} products is n_{l-1}, not 1 as desired.

However, by scaling each $\mathbf{W}_{(l),j}$ by

$$\frac{1}{\sqrt{n_{l-1}}}$$

the variance becomes 1.

Glorot (also called *Xavier*) initialization sets the initial weights to a number drawn from mean 0, unit variance distribution (either normal or uniform) $\frac{1}{\sqrt{n_{l-1}}}$.

Note that we don't strictly need the requirement of unit variance -- it works equally well as long as the input and output variances are the same, which is the goal.

This only partially solves the problem as it only ensures unit variance of the input to the activation function.

The original Glorot paper (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf) justifies this by assuming either a $\tanh$ or sigmoid activation functions and these functions can be approximated as linear in the active region.

Thus far, we have achieved unit variance during the forward pass. During back propagation, it was shown that the scaling factor depends on the number of outputs of layer l, rather than number of inputs, so the scaling factor needs to be $\frac{1}{\sqrt{n_l}}$

Taking the average of the two scaling factors gives a final factor of
$\frac{1}{\sqrt{\frac{n_{l-1}+n_l}{2}}} = \sqrt{\frac{2}{n_{l-1}+n_l}}$

Kaiming/He initialization

Kaiming et al (https://arxiv.org/pdf/1502.01852.pdf) extended the results of Glorot et. al to the ReLU activation.

The ReLU activation has two distinct regions: one linear (for inputs greater than 0) and one all zero.

The linear region of the activation corresponds to the assumption of the Glorot method.

So if inputs to the ReLU are equally distributed around 0, this is approximately the same as the Glorot method with half the number of inputs.

- that is: half of the ReLU's will be in the active region and half will be in the inactive region.

The Kaiming scaling factor is thus:

$$\sqrt{\frac{2}{n_{(l-1)}}}$$

in order to preserve unit variance.

How big should my NN be ?

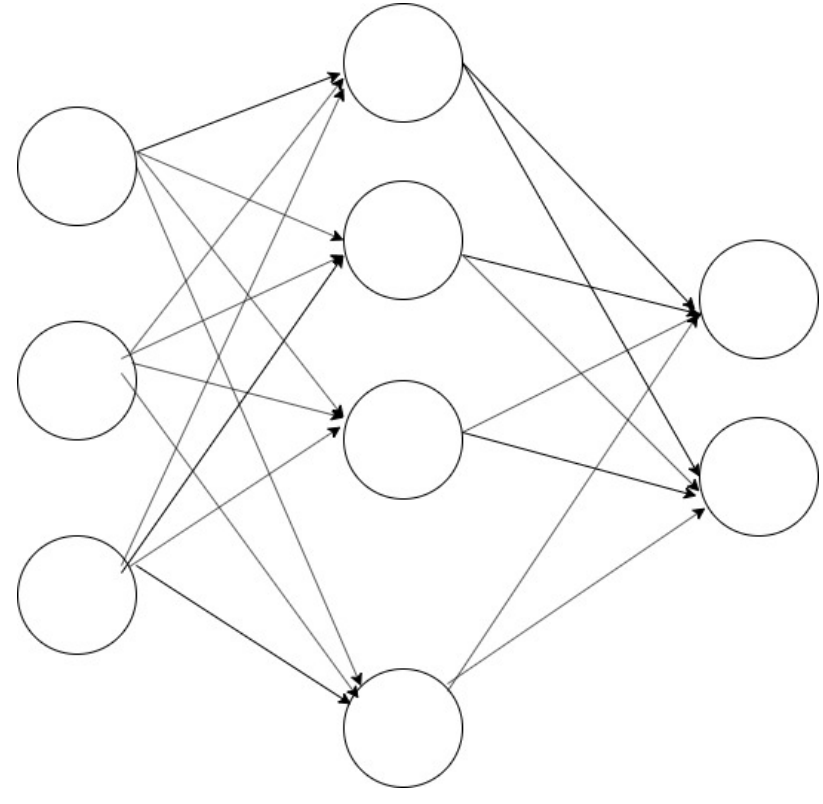
There is a paradox in building Neural Networks:

- Start off training an overly large NN (many units)
- Many units turn out to be "dead": near zero weights
- Reduce the number of units
- Can't train !

Given a fixed number of layers: it is easier to train a big NN than a small one.

"Somewhere in this big mess must be something valuable"

"Big" NN with dead nodes



The Lottery Ticket Hypothesis (https://arxiv.org/abs/1803.03635) is an inte
that addresses this issue.

For now:

- Use bigger than necessary NN's
- With regularization to "prune"

In [4]:

```
Done
```

"Big" NN after dead nodes have been pruned