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 training 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.

Gradient Descent updates weights using the gradients.

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!

Neurons/units j and j' will never be able to differentiate and come to recognize *different* features.

This negates the advantage of multiple units in a layer.

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 assume each element $\mathbf{W}_{(l),j}$ is similarly distributed.

Consider the dot product in layer l

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

Recall that layer $(l-1)$ has $n_{(l-1)}$ outputs.

Thus, the dot product is the sum over $n_{(l-1)}$ pair-wise products

- $\mathbf{y}_{(l-1),j} * \mathbf{W}_{(l),j}$

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

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

So

$$\begin{aligned} \operatorname{Var}(\mathbf{y}_{(l-1),j} * \mathbf{W}_{(l),j}) &= 0^2 * 1 + 0^2 * 1 + 1 * 1 \\ &= 1 \qquad \text{Since } \mathbf{y}_{(l-1),j} \text{ and } \mathbf{W}_{(l),j} \text{are i} \end{aligned}$$

Thus

- The variance of the dot product involving $n_{(l-1)}$ pair-wise products
- Is $n_{(l-1)}$, not 1 as desired.

We can force the dot product to have unit variance

- By scaling each $\mathbf{W}_{(l),j}$ by $\frac{1}{\sqrt{n_{l-1}}}$

This is the basis for *Glorot/Xavier Initialization*

- Sets the initial weights to a number drawn from a mean 0, unit variance distribution (either normal or uniform)
- Multiplied by $\frac{1}{\sqrt{n_{l-1}}}$.

Note that we don't strictly need the requirement of *unit* variance

- It suffices that the input and output variances are *equal*

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)](http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf) justifies this

- By assuming either a $\tanh$ or sigmoid activation function
- Which are approximately linear in the active region.
- So the **output** of the activation function is equal to the input in this region
- And is therefore unit variance as desired

Thus far, we have achieved unit variance during the forward pass.

During back propagation

- It can be shown that the scaling factor
- Depends on the number of outputs $n_{(l)}$ of layer l, rather than the number of inputs $n_{(l-1)}$
- Thus, the scaling factor needes to be $\frac{1}{\sqrt{n_{(l)}}}$ rather than $\frac{1}{\sqrt{n_{l-1}}}$

Taking the average of the two scaling factors gives a final factor of

$$\frac{1}{\sqrt{\frac{n_{l-1}+n_l}{2}}} = \sqrt{\frac{2}{n_{l-1}+n_l}}$$

which is what you often see in papers using this form of initialization.

Kaiming/He initialization

Glorot/Xavier initialization was tailored to two particular activation functions ($\tanh$ or sigmoid).

Kaiming et al (https://arxiv.org/pdf/1502.01852.pdf) extended the results 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.

Layer-wise pre-training

In the early days of Deep Learning

- Before good weight initialization techniques were discovered
- A technique called *Layer-wise pre-training* was very popular

This technique is best discussed after we introduce Autoencoders but, for now:

Suppose we want to initialize the weights of layer l

- We *temporarilty* create a two layer sub-network consisting of layers l and a "Decoder" layer
- Let $\mathbf{y}'_{(l+1)}$ denote the output of this two layer network
- This two layer network has input $\mathbf{y}_{(l-1)}$ and output $\mathbf{y}'_{(l+1)}$

We train this two layer network with training examples

$$\langle \mathbf{y}_{(l-1)}, \mathbf{y}_{(l-1)} \rangle = [\mathbf{y}^{(\mathbf{i})}_{(l-1)}, \mathbf{y}^{(\mathbf{i})}_{(l-1)} \mid 1 \leq i \leq m]$$

That is: we are asking the two layer network to implement the identity function

To avoid making this trivial

- The Decoder creates a "bottle-neck": a layer with a small number of units
- The small number prevents the Decoder from memorizing the inputs
- This is a form of *dimensionality reduction* (like Principal Components Analysis)

Like Principal Components Analysis

- We are asking the two layer network
- To discover a small number of synthetic features that summarize the diversity of $\mathbf{y}_{(l-1)}$

The weights discovered by this procedure become the initial weights of layer l

- The Decoder is then dropped
- And we proceed to find initial weights for layer $(l+1)$.

These weights *may or may not* be useful in predicting $\hat{\mathbf{y}} = \mathbf{y}_{(L)}$

- But they are probably better than *random* weights

Normalization

- We addressed the importance of normalization of the inputs to layer $l = 1$.
- The same argument applies to *all* layers $l > 0$

We discuss some Normalization methods that attempt to keep the distribution of $\mathbf{y}_{(l),j}$ normalized through all layers l.

Batch normalization

Batch Normalization paper (https://arxiv.org/abs/1502.03167)

The idea behind batch normalization:

- perform standardization (mean 0, standard deviation 1) at each layer, using the mean and standard deviation of each minibatch.
- faciliates higher learning rate
    - controlling the size of the derivative allows higher α without increasing product

Experimental results show that the technqiue:

- facilitates the use of much higher learning rates, thus speeding training. Accuracy is not lost.
- facilitates the use of saturating activations functions (e.g., $\tanh$ and sigmoid) which otherwise are subject to vanishing/exploding gradients.
- acts as a regularizer; reduces the need for Dropout
    - L2 regularization (weight decay) has *no* regularizing effect when used with Batch Normalization !
        - see (https://arxiv.org/abs/1706.05350)
        - L2 regularization affects scale of weights, and thereby learning rate

Details

Consider a FC layer l with n_l outputs and a mini-batch of size m_B.

Each of the n_l outputs is the result of

- passing a linear combination of $\mathbf{y}_{(l-1)}$ (*activation inputs*)
- through an activation $a_{(l),j}$ (*activation outputs*)

We could choose to standardize either the activation inputs or the activation outputs.

This algorithm standardizes the **activation inputs**.

Standardization is performed relative to the mean and standard deviation of each batch.

Summary for layer l with equation $\mathbf{y}_{(l)} = a_{(l)}(\mathbf{W}_{(l)}\mathbf{y}_{(l-1)})$

- each output feature j: $\mathbf{y}_{(l),j} = a_{(l),j}(\mathbf{W}_{(l),j}\mathbf{y}_{(l-1)})$
- Denote the dot product for output feature j by $\mathbf{x}_{(l),j} = \mathbf{W}_{(l),}\mathbf{y}_{(l-1)}$
- We will replace $\mathbf{x}_{(l),j}$ by a "standardized" $\mathbf{z}_{(l),j}$ to be described

Rather than carrying along supscript j we write all operations on the collection $\mathbf{x}_{(l),j}$ as a vector operation on $\mathbf{x}_{(l)}$ for ease of notation.

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. ${\sigma_B}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$

So

- μ_B, σ_B are vectors (of length n_l) of
  - the element-wise means and standard deviations (computed across the batch of m_B examples)
- $\hat{\mathbf{x}}^{(\mathbf{i})}$ is standardized $\mathbf{x}^{(i)}$

** Note the ϵ in the denominator is their solely to prevent "divide by 0" errors

What is going on with $\mathbf{z}^{(\mathbf{i})}$?

Why are we constructing it with mean β and standard deviation γ ?

β, γ which are **learned** parameters.

Why should β, γ be learned ?

At a minimum: it can't hurt:

- it admits the possibility of the identity transformation
    - which would be the simple standardization
- but allows the unit to be non-linear when there is a benefit

Moreover, depending on the activation $a_{(l),j}$

- $\hat{\mathbf{x}}_{(l),j}$ can wind up *within the active region* of the activation function

This effectively makes our transformations linear, rather than non-linear, which are more powerful.

By shifting the mean by β we gain the *option* to avoid this should it be beneficial.

The final question is: what do we do at inference/test time, when all "batches" are of size 1 ?

The answer is

- compute a single μ, σ from the sequence of such values across all batches.
- "population" statistics (over full training set
- rather than "sample" statistics (from a single training batch).

Typically a moving average is used. We refer readers to the paper.

We create a new layer type BN to perform Batch Normalization to the inputs of any layer.

Thus, it particpates in both the forward (i.e., normalization) and backward (gradient computation) steps.

Unbelievably good initialization

We have seen several methods that attempt to create "good" weights Glorot and Kaiming weight initialization

- ensures "good" distribution of outputs of a layer, given a good distribution of inputs to the layer

Normalization (e.g., Batch Normalization)

- tries to ensure good distribution of inputs across all layers

There are some initialization methods that attempt to create weights that are so good, that Normalization during training is no longer necessary.

Fixup initialization paper (https://arxiv.org/abs/1901.09321)

- good initialization means you don't need normalization layers

But good initialization can help too.

Conclusion

Initialization is like priming a pump: we want our gradients and learning to flow smoothly.

Proper initialization not only facilitates learning, but may actually speed up training (e.g. Batch Normalization).

In [5]:
```
print("Done")
```

```
Done
```