Impediments to learning

The updating of weights, used by Gradient Descent to minimize the loss, can be inhibited in less-than-obvious manners.

In this module, we explore these impediments.

This will motivate the creation of a new class of Layer-types: Normalization.

Proper scaling of inputs

We briefly explore the statistical properties of the outputs of a layer.

- We show how some of these properties can inhibit learning (weight update)
- Will motivate the Normalization Layer-type, which will maintain good properties of layer outputs

Importance of zero centered inputs (for each layer)

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

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

summed over examples i.

Let's look into the per example loss in more detail.

Since $\mathbf{W}_{(l)}$ is a vector, the derivative wrt $\mathbf{W}_{(l)}$ is a vector of derivatives:

$$\frac{\partial \mathbf{y}_{(l)}^{(\mathbf{i})}}{\partial \mathbf{W}_{(l)}} = \left(\cdots, \frac{\partial \mathbf{y}_{(l)}^{(\mathbf{i})}}{\partial \mathbf{W}_{(l),}}, \cdots, \right)$$

Examining the jth element of the derivative vector:

$$\frac{\partial \mathbf{y}_{(l)}^{(\mathbf{i})}}{\partial \mathbf{W}_{(l),j}} = \frac{\partial a_{(l)}(\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l)})}{\partial \mathbf{W}_{(l),j}} \qquad \text{when layer } l \text{ is Dense since } y_{(l)} = a_{(l)}(\mathbf{y}$$

$$= \frac{\partial a_{(l)}(\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l)})}{\partial (\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l)})} \frac{\partial (\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l)})}{\mathbf{W}_{(l),j}} \qquad \text{Chain rule}$$

$$= a'_{(l)} \mathbf{y}_{(l-1),j}^{(\mathbf{i})} \qquad \text{since } \mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l)} = \sum_j \left(\mathbf{y}_{(l-1),j} * \mathbf{W}_{(l)} \right.$$

$$\text{where } a' = \frac{\partial a_{(l)}(\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l)})}{\partial (\mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l)})}$$

This is for $\mathcal{L}^{(\mathbf{i})}$, the per-example loss for example i.

The (total) Loss $\mathcal{L}$ is averaged across all m' examples in the mini-batch

So the derivative of the Loss (with respect to the jth weight) $\frac{\partial \mathcal{L}}{\partial W_{(l),j}}$ will have the term

$$\sum_{i=0}^{m'} \mathbf{y}_{(l-1),j}^{(\mathbf{i})}$$

Thus, the update to $\mathbf{W}_{(l),j}$ will be proportional to the average (across the m' examples) of the jth input to layer l.

To be concrete, let's focus on layer 1, where

$$\mathbf{y}_{(l-1),j} = \mathbf{x}_j$$

so that

$$\sum_{i=0}^{m} \mathbf{y}_{(l-1),j}^{(\mathbf{i})} = \bar{\mathbf{x}}_j$$

i.e., the average (across examples) value of input feature j.

In the particular case that the average $\bar{\mathbf{x}}_j$ of *every* feature j has the same sign:

- updates in all weight dimensions will have the same sign (the sign of a')
- Example: two dimensions. The weight space is $\mathbf{W}_{(l),0} \times \mathbf{W}_{(l),1}$
    - We can navigate the the loss surface by moving in the weight space north-east or south-west only !
    - this can result in an indirect "zig-zag" toward the optimum
        - To get to a point south-east from the current, we have to zig-zag.

Although we have illustrated this issue using layer 1, the issue applies to each layer.

In fact, the issue may be more likely in deeper layers

- when the activation of layer $(l-1)$ is *not* zero-centered, e.g., the ReLU and sigmoid

This will motivate the creation of a new layer type whose purpose will be to keep the inputs to successive layers zero-centered.

Note

Although we zero center the m examples in the training set, the $m' < m$ examples in any mini-batch will not necessarily be zero mean in all features.

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 variance 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

$$\mathbf{y}_{(l),j} = a_{(l)}\big(\mathbf{w}_{(l)}\,\mathbf{y}_{(l-1)} + \mathbf{b}_{(l)}\big)$$

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

- 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 previously argued 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 <u>is</u>
<u>(https://en.wikipedia.org/wiki/Variance#Product_of_independent_variables)</u>

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

So

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

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 needs 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)](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

We can motivate this technique by briefly introducing an Autoencoder network.

Autoencoder

An Autoencoder network has two parts

- An Encoder, which takes input $\mathbf{x}$ and "encodes" it into $\mathbf{z}$
- A Decoder, which takes the encoding $\mathbf{z}$ and tries to reproduce $\mathbf{x}$

Each part has its own weights, which can be discovered through training, with examples

- $\langle \mathbf{X}, \mathbf{y} \rangle = \langle \mathbf{X}, \mathbf{X} \rangle$

That is: we are asking the output to be identical to the input.

This will not be possible when the dimension of $\mathbf{z}$ is less than the dimension of $\mathbf{x}$.

- $\mathbf{z}$ is a *bottle-neck*

$\mathbf{z}$ becomes a *reduced-dimensionality* approximation of $\mathbf{x}$.

This is quite similar to discovering Principal Components.

- We discover a small number of synthetic features $\mathbf{z}$ that summarize the diversity of $\mathbf{y}_{(l-1)}$

What does this have to do with layer-wise initialization of weights ?

Suppose we want to initialize the weights of layer l

- We *temporarily* create a two layer Autoencoder network with layer l serving the role of Encoder
- We train this temporary Autoencoder
- This initializes the weights of layer l
- We discard the Decoder

The weights we create

- Are not random, they meet the Autoencoder task objective
- Perhaps non-random weights are better initializers because they discover some structure of the input

Transfer Learning (the subject of another module) works in a similar manner

- Use the weights obtained from training on a Source task
- To use as initial weights for a second Target task

Normalization

We addressed the importance of normalization of the inputs to layer $l = 1$.

The same argument applies to *all* layers $l > 0$

This motivates the introduction of a new class of layer-types: Normalization layers

- These layer types attempt to keep the distribution of $\mathbf{y}_{(l),j}$ normalized through all layers l.
- They become necessary for *very deep* (large number of layers) networks

Normalization layers were one of the innovations that advanced Deep Learning by enabling learning in networks of extreme depth.

Batch normalization

The idea behind batch normalization:

- perform standardization (mean 0, standard deviation 1) at each layer, using the mean and standard deviation of each mini batch.

- facilitates higher learning rate

    - controlling the size of the derivative allows higher α without increasing product

Experimental results show that the technique:

- 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 $\mathbf{y}_{(l)} = a_{(l)}\left(\mathbf{W}_{(l)}\mathbf{y}_{(l-1)}\right)$

- each output feature j: $\mathbf{y}_{(l),j} = a_{(l),j}\left(\mathbf{W}_{(l),j}\mathbf{y}_{(l-1)}\right)$

- Denote the dot product for output feature j by $\mathbf{x}_{(l),j} = \mathbf{W}_{(l),j}\mathbf{y}_{(l-1)}$

- We will replace $\mathbf{x}_{(l),j}$ by a "standardized" $\mathbf{z}_{(l),j}$ to be described

Rather than carrying along subscript 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. $\mu_B = \dfrac{1}{m_B} \sum\limits_{i=1}^{m_B} \mathbf{x}^{(\mathbf{i})}$ Batch mean

2. $\sigma_B{}^2 = \dfrac{1}{m_B} \sum\limits_{i=1}^{m_B} (\mathbf{x}^{(\mathbf{i})} - \mu_B)^2$ Batch variance

3. $\hat{\mathbf{x}}^{(\mathbf{i})} = \dfrac{\mathbf{x}^{(\mathbf{i})} - \mu_B}{\sqrt{\sigma_B{}^2 + \epsilon}}$ Standardize $\mathbf{x}^{(\mathbf{i})}$

4. $\mathbf{z}^{(\mathbf{i})} = \gamma \hat{\mathbf{x}}^{(\mathbf{i})} + \beta$ De-Standardize $\hat{\mathbf{x}}^{(\mathbf{i})}$ with learned mean and

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 there solely to prevent "divide by 0" errors

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

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 participates 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)](https://arxiv.org/abs/1901.09321)

- good initialization means you don't need normalization layers

But good initialization can help too.

Conclusion

Maintaining good properties of layer inputs throughout the depth of a multi-layer network is like priming a pump.

Proper priming helps our learning to flow smoothly.

We explored some of the stumbling blocks to learning (weight update) along with their solutions.

```python
In [4]: print("Done")
```

Done