# 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.

# Layer-wise pre-training

We discussed this in the Autoencoder lecture.

In the absence of a "good" idea for initializing layer $l$'s weights

- train layer $l$ as an Autoencdoer
    - train it so that input $y_{(l-1)}$ has target $y_{(l-1)}$
    - this will result in weights $\mathbf{W}_{(l)}$ that have discovered some structure among the features $y_{(l-1)}$
    - Initialize the weights of layer $l$ to the ones produced by the Autoencoder
- 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 $\alpha$ 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

- $\mu_B, \sigma_B$ 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 $\epsilon$ 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 $\beta$ and standard deviation $\gamma$ ?

$\beta, \gamma$ which are **learned** parameters.

Why should $\beta, \gamma$ 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 $\beta$ 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 $\mu, \sigma$ 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 $\mathrm{BN}$ 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

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 al 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.

# Learning rate schedules

In addition to smarter optimzers, we can control learning rates by changing them across epochs of training.

This is very much of an art rather than a science.

We give a brief overview.

# Warm up

Bag of Tricks for Image Classification using CNNs (https://arxiv.org/abs/1812.01187)

- When training starts: initial values of $\mathbf{W}$ far optimal values
- At this point, losses (and gradients) are probably large
    - large updates to $\mathbf{W}$ might cause instability

So, we can start off "slow" with a low initial rate during a *warm-up period*.

- low learning rate compensates for high gradient

Post the warm-up, we can use a higher rate to speed training.

# Post warm-up

Typical strategy has been to decrease learning rate as the number of epochs increase.

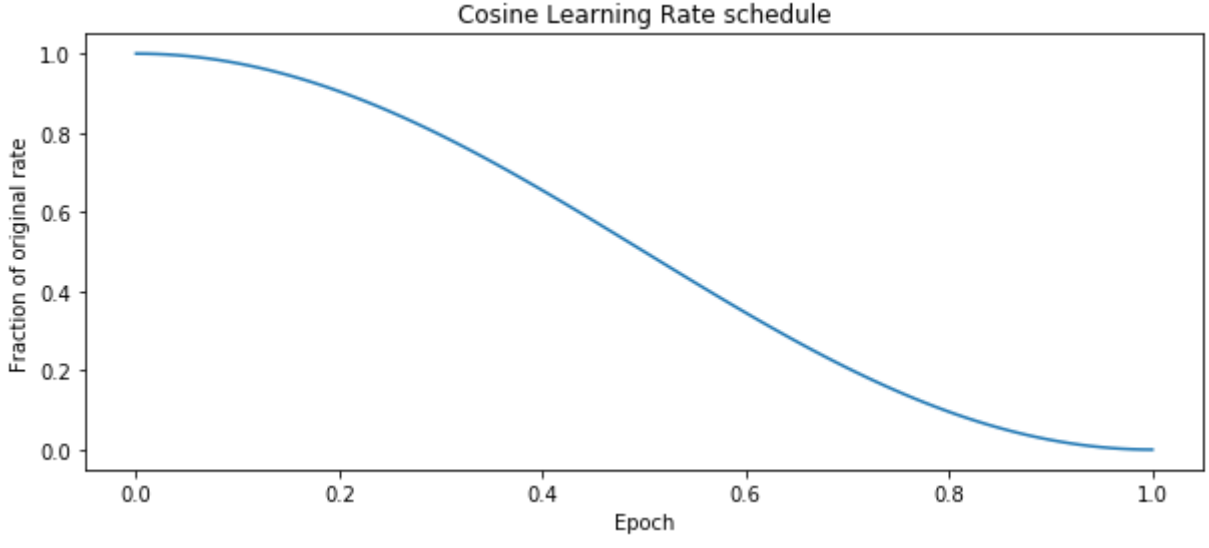
Idea is to take smaller steps as we approach the region of optimality

- don't want to overshoot

There are many ways to set a learning rate schedule (a function that maps epoch number to a rate)

- step schedule
    - vary rate by epoch
    - rate decreases as epoch increases
- cosine decay
    - decrease rate according to a cosine function
        - $\text{learning\_rate}_t = \frac{1}{2}\left(1 + \cos(\pi \frac{t}{T})\right) * \text{learning\_rate}_0$
            - where $\text{learning\_rate}_0$ is initial learning rate, $T$ is number of batches
        - slow decrease in rate at start
        - near-linear decrease in middle
        - slow decrease near end

In [4]:
```
_= nnh.plot_cosine_lr()
```



# Regularization

The ultimate goal of Machine Learning is out of sample prediction.

Because Neural Networks often learn a large number of parameters (weights), overfitting is a concern.

We will briefly review several methods to combat overfitting

# Cost function: add regularization penalty

The same methods that were applicable in Classical Machine Learning apply to Deep Learning as well.

These include regularization penalties that aim to reduce the number of parameters.

- L2 regularization
- L1 regularization

# Dropout

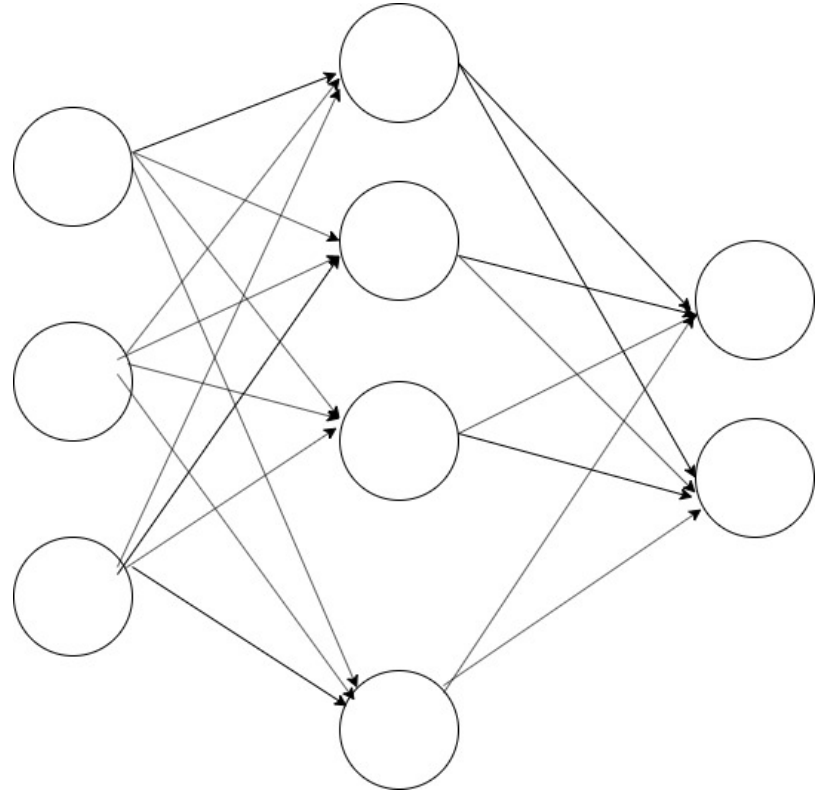
Droput paper (http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)

Overfitting can occur because some weights in the NN adapt so as to memorize "noisy" features.

*Dropout* is a method that *randomly drops a unit in the NN*

- For each training example $\mathbf{x}^{(\mathbf{i})}$
- Each unit gets dropped with probability $p$

**NN, no dropout**



A Neural Network with $N$ units contains $2^N$ possible sub-networks.

Dropout can be viewed as training many of these sub-networks (with weigh
sub-networks.)

If a feature is truly important, the NN must adapt to robustly recognize the

If it is not important, the goal is to prevent a unit from memorizing it.

In Keras, Dropout is implemented by a layer:

```
Dropout(rate)
```

where `rate` is the probability of dropping a unit.

Dropout has been supplanted by Batch Normalization, but is worth studyin

- for its simplicity and ease of use
- inspiration it offers.

# Data Augmentation

It is sometimes possible to expand the training set in such a way as to dis
overfitting.

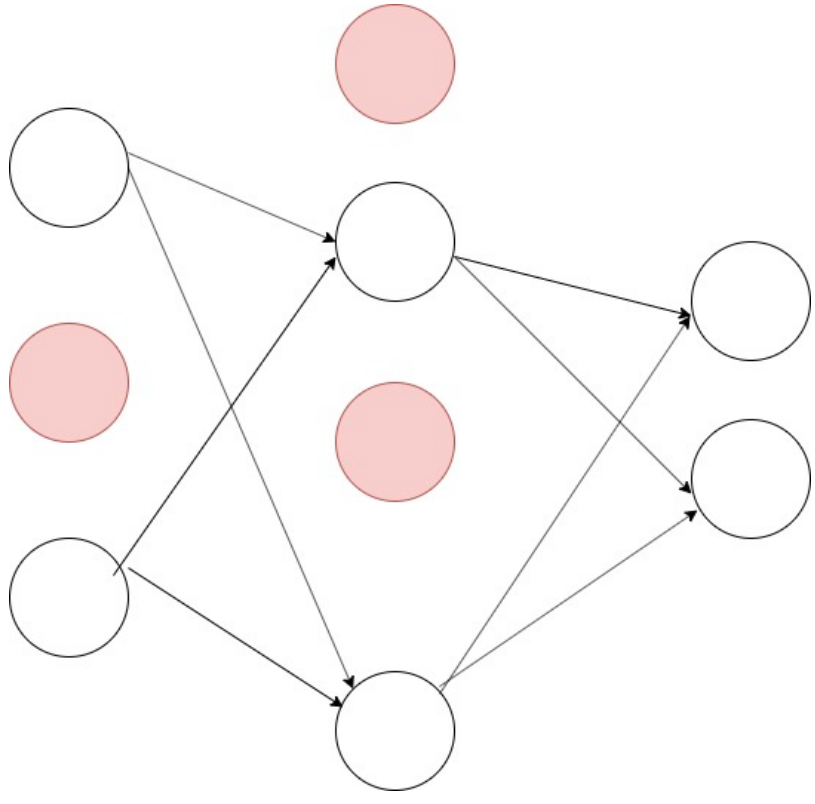
This usually involves creating variants of training examples

- make it hard to memorize them all.

## Input Transformation

Alter the image while preserving its label.

- Image transformation
    - rotate, crop, flip



| Example | Smoothed label |
|---|---|
| $(\mathbf{x^{(i)}}, 0)$ | $(\mathbf{x^{(i)}}, 0+\epsilon)$ |
| $(\mathbf{x^{(i)}}, 1)$ | $(\mathbf{x^{(i)}}, 1-\epsilon)$ |

So rather than using One Hot Encoding (OHE), we use "$\epsilon$ Hot Encoding"

# Mixup training

Mixup training paper (https://arxiv.org/abs/1905.11001)

*Mixup training* is a second solution to prevent an NN from seeking absolute confidence.

It creates additional training examples that are *mixtures* of existing examples:

| Training example | Mixup ? |
|---|---|
| $(\mathbf{x^{(i)}}, \mathbf{y^{(i)}})$ | original |
| $(\mathbf{x}^{(i')}, \mathbf{y}^{(i')})$ | original |
| $(\mathbf{x^{(i)}} + \lambda\mathbf{x}^{(i')}, \mathbf{y^{(i)}}) + \lambda\mathbf{y}^{(i')})$ | Mixup |

The mixing parameter $\lambda$ is best when it is close to $0$ or $1$

- $(0 \quad$ or $(1 \quad .$
  $+\epsilon \qquad -\epsilon$
  $) \qquad )$

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

```
Done
```