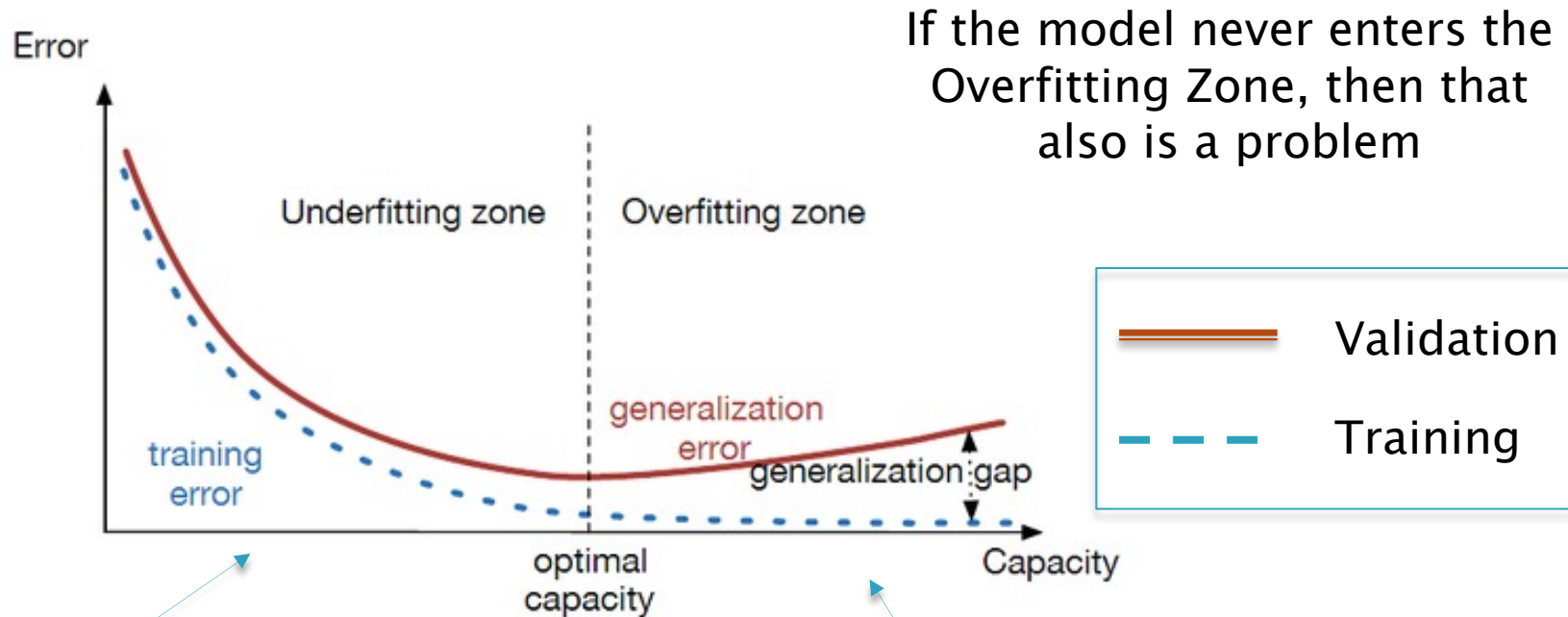


# Training Process Improvements: Part 3

Lecture 9  
Subir Varma

# Overfitting and Underfitting

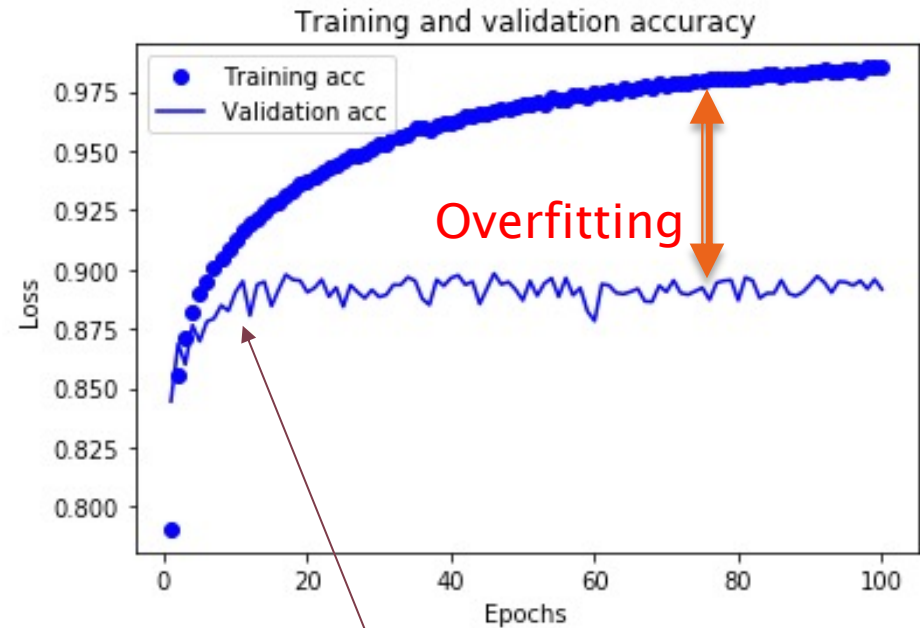
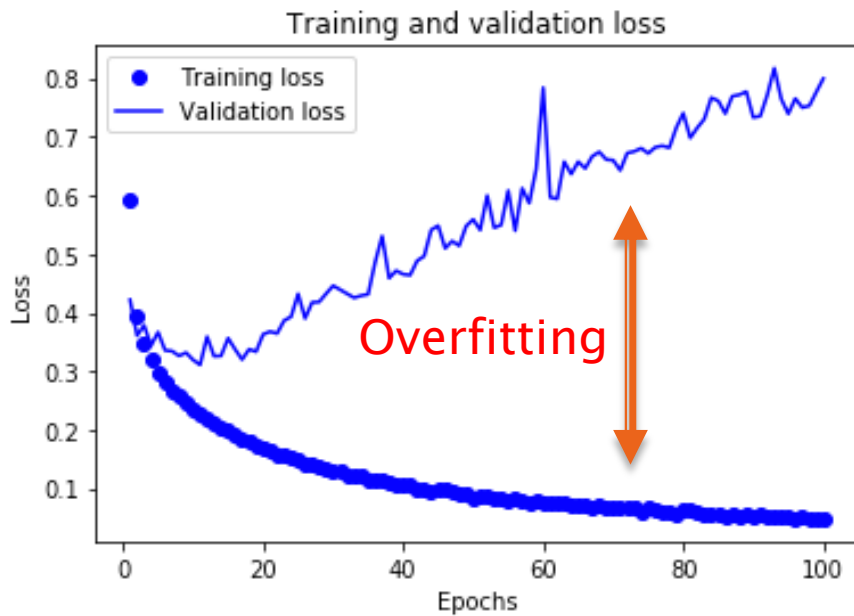


If the model never enters the Overfitting Zone, then that also is a problem

Model hasn't modeled all the relevant patterns in the Training Data

Model is learning patterns that are specific to the Training Data but irrelevant to the Test Data

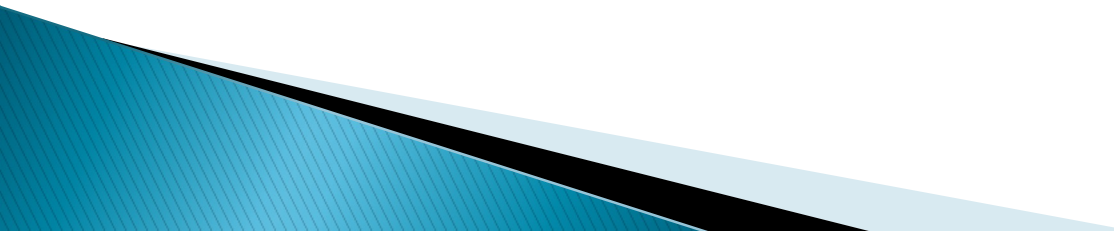
# Detecting Overfitting



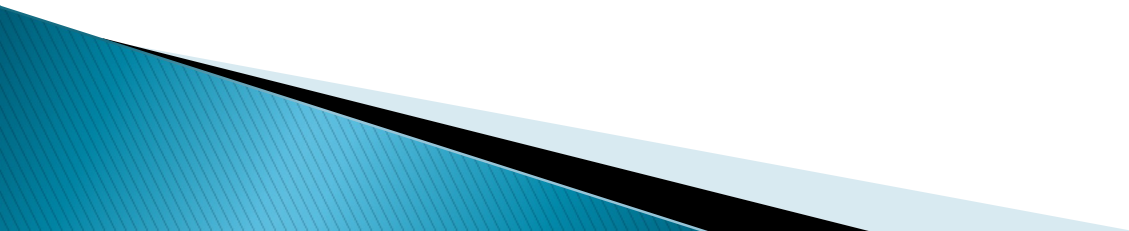
Fashion Dataset using a  
Dense Feed Forward Network  
Single Hidden Layer with 512 nodes

Improvement in Validation Accuracy stops when overfitting starts. Hence it is better to push out this threshold further out.

# Ways to improve Model Generalization

- ▶ Add Regularization to the model
  - ▶ Increase the amount of Training Data
  - ▶ Decrease Model Capacity
- 

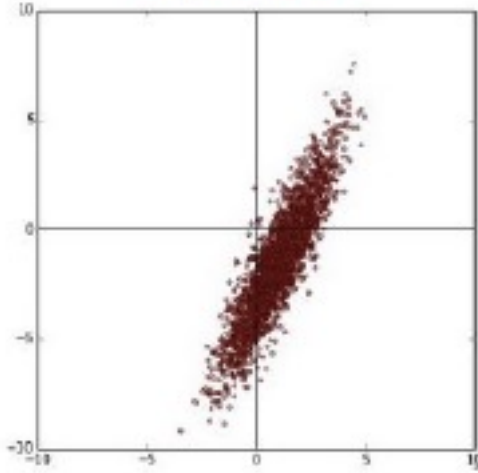
# Data Pre-Processing



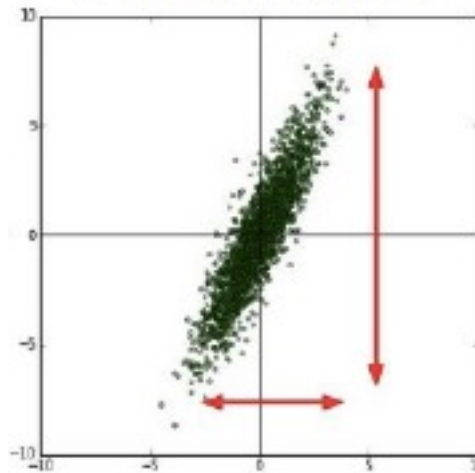
# Data Preprocessing

$$X(m) = (x_1(m), \dots, x_N(m)), m = 1, \dots, M,$$

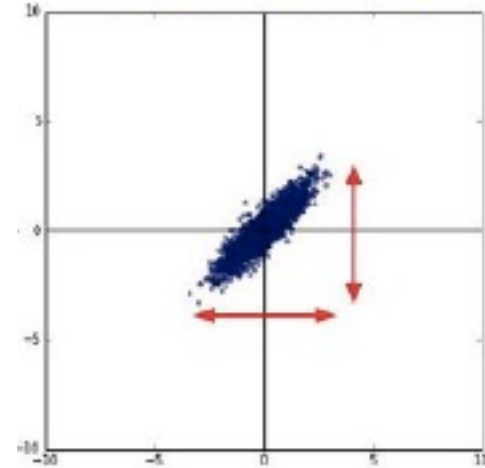
original data



zero-centered data



normalized data



Subtract the Mean

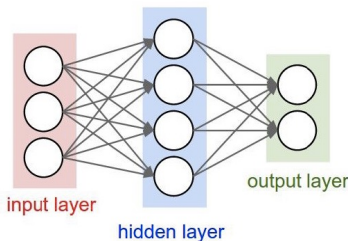
$$x_i(m) \leftarrow x_i(m) - \frac{\sum_{s=1}^M x_i(s)}{M}$$

`x -= x.mean(axis=0)`

Divide by Variance

$$x_i(m) \leftarrow \frac{x_i(m) - \frac{\sum_{s=1}^M x_i(s)}{M}}{\sigma_i}$$

`x /= x.std(axis=0)`



$$\frac{\partial \mathcal{L}}{\partial w} = x\delta$$

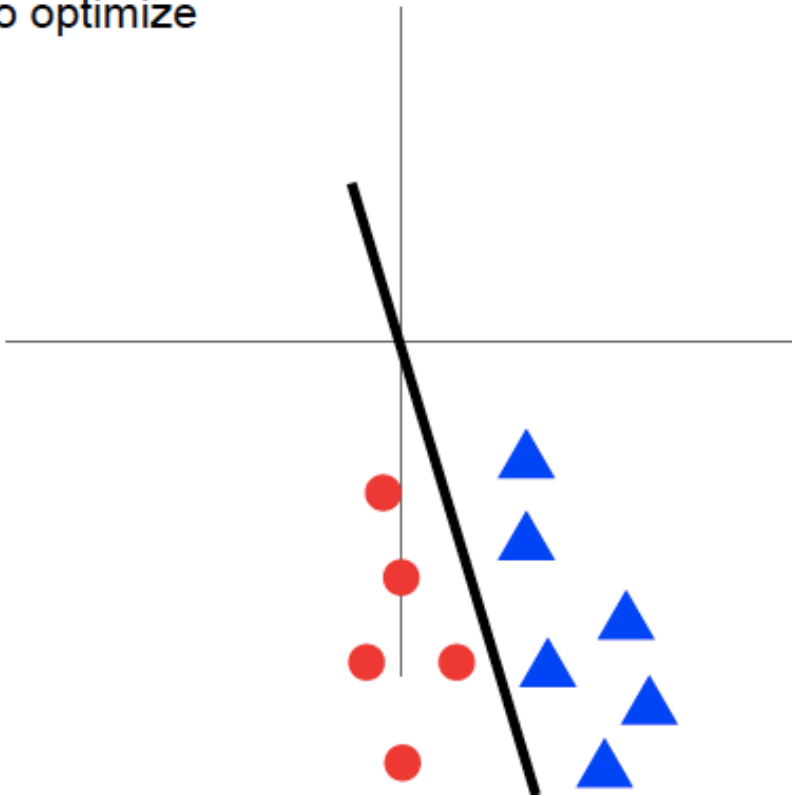
$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}}$$

Speeds up  
Convergence  
Of SGD

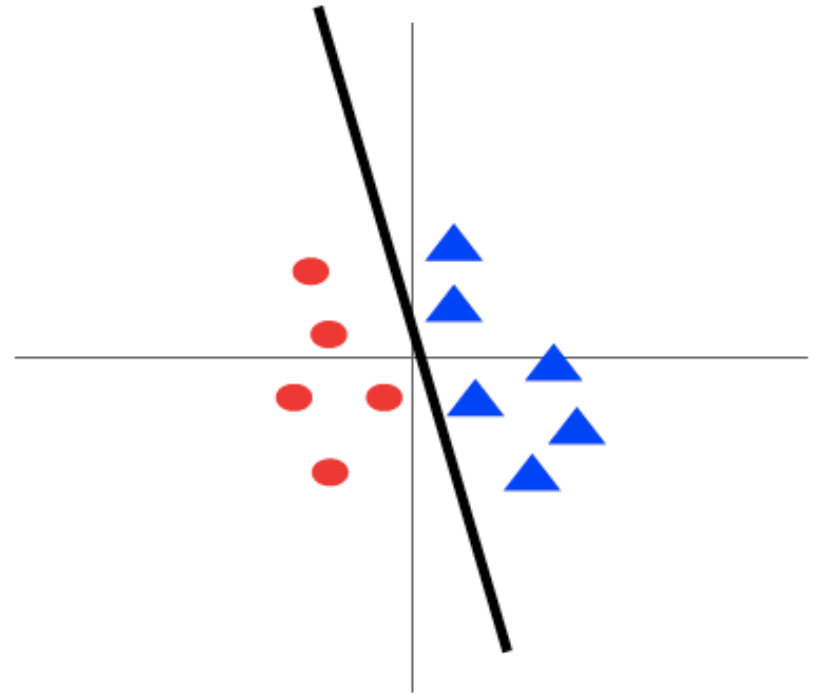
Balances out the rates  
at which weights learn

# How Does Data Normalization Help?

**Before normalization:** classification loss very sensitive to changes in weight matrix; hard to optimize



**After normalization:** less sensitive to small changes in weights; easier to optimize



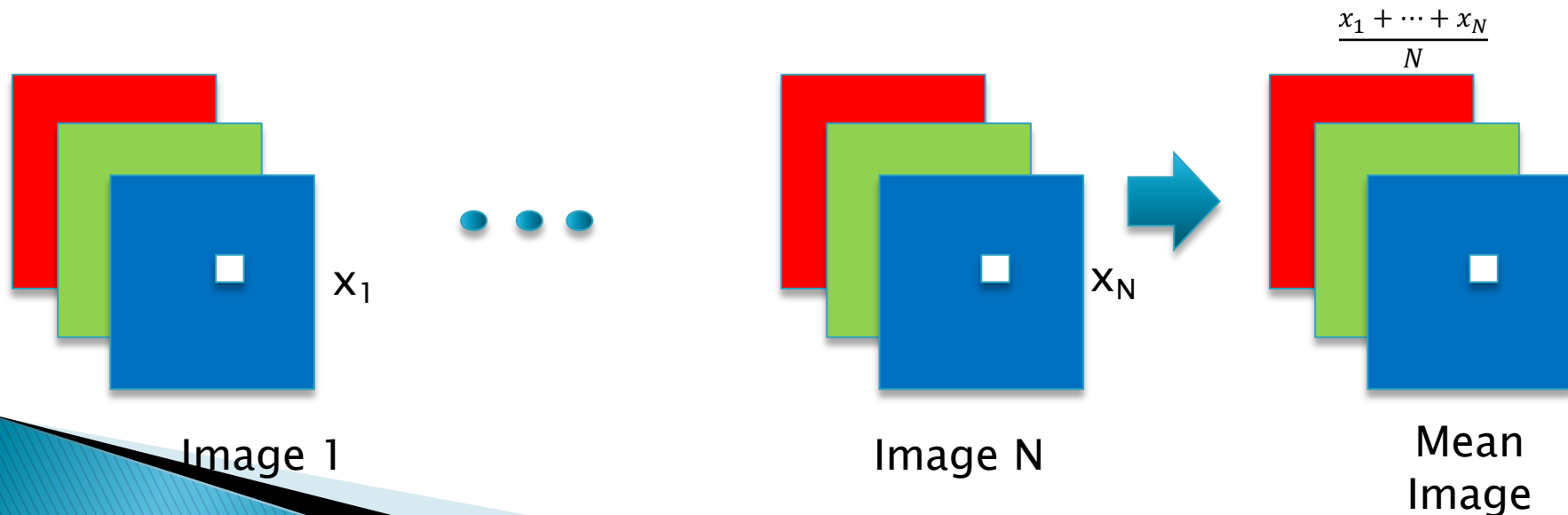
# Image Pre-Processing

Consider CIFAR-10 images with [3, 32, 32] pixels per image

Subtract the Mean Image: Mean Image forms a [3,32,32] Tensor

For CH = R, G, B,

$$x_{ij}^{CH}(m) \leftarrow x_{ij}^{CH}(m) - \frac{\sum_{s=1}^M x_{ij}^{CH}(s)}{M}, \quad 1 \leq i, j \leq N, 1 \leq m \leq M$$



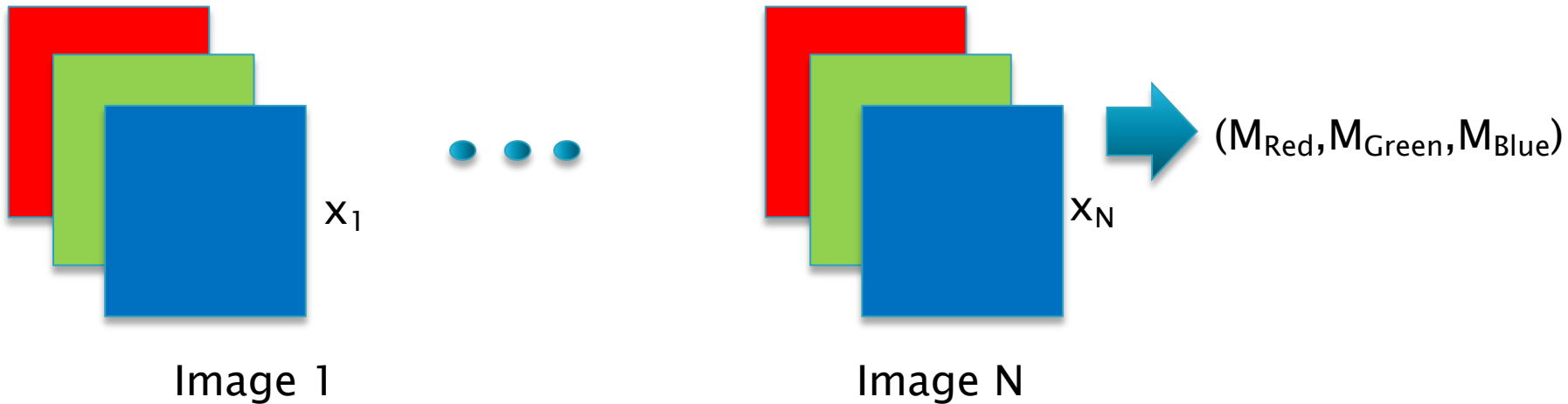


# Image Pre-Processing

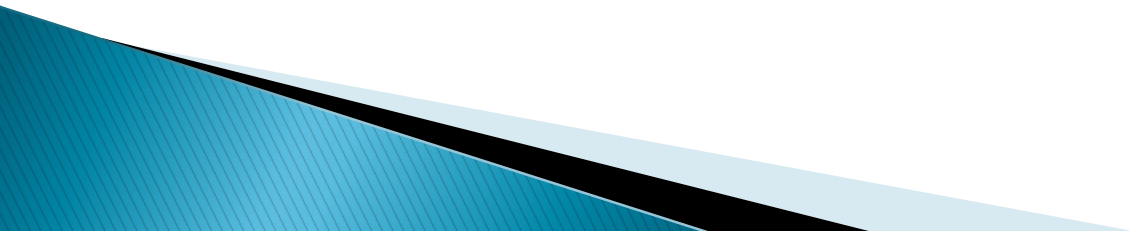
Consider CIFAR-10 images with [3, 32, 32] pixels per image

Subtract the Per-Channel Mean: 3 Numbers

$$x_{ij}^{CH}(m) \leftarrow x_{ij}^{CH}(m) - \frac{\sum_{s=1}^M \sum_{i=1}^N \sum_{k=1}^N x_{ij}^{CH}(s)}{M}, \quad 1 \leq i, j \leq N, 1 \leq m \leq M$$



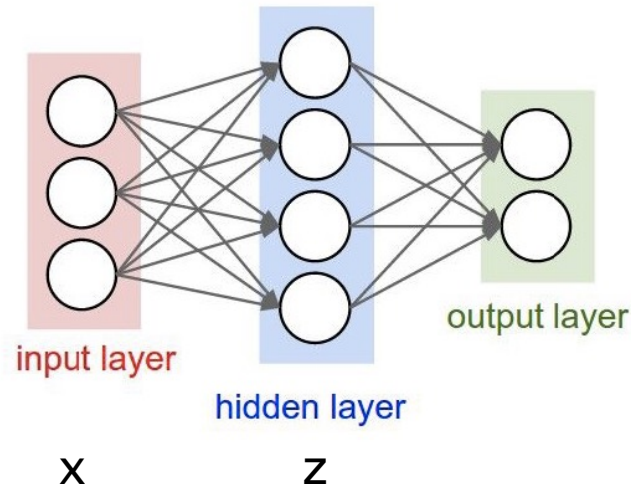
# Batch Normalization



# Batch Normalization

Problem: The Activations in the interior of the network may become unbalanced as the training progresses

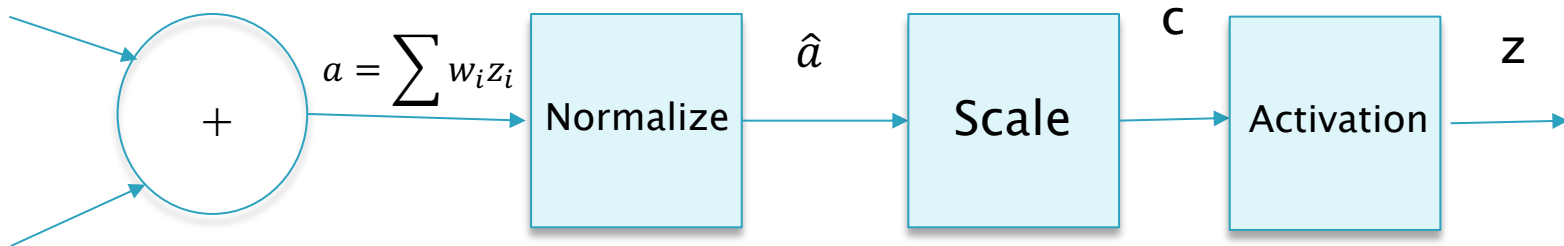
Solution: Instead of Normalizing just the input, why not normalize the per-layer activations in as well.



# Batch Normalization

Problem: Unlike the input, the activations change as the training progresses

Solution: Do the normalization in batches, such that during each batch, the weights remain fixed

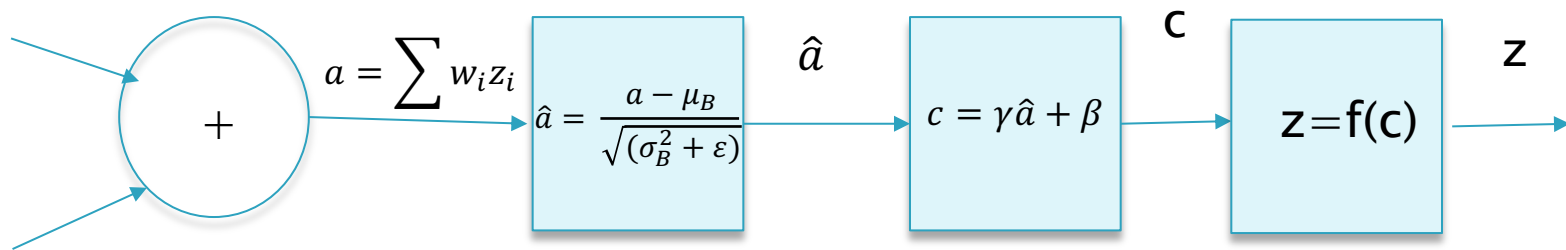


$$\hat{a} = \frac{a - \mu_B}{\sqrt{(\sigma_B^2 + \epsilon)}}$$

$$\sigma_B^2 = \frac{\sum_{s=1}^B (a(s) - \mu_B)^2}{B}$$

$$\mu_B = \frac{\sum_{s=1}^B a(s)}{B}$$

# Backprop Gradient Calculations with Batch Normalization



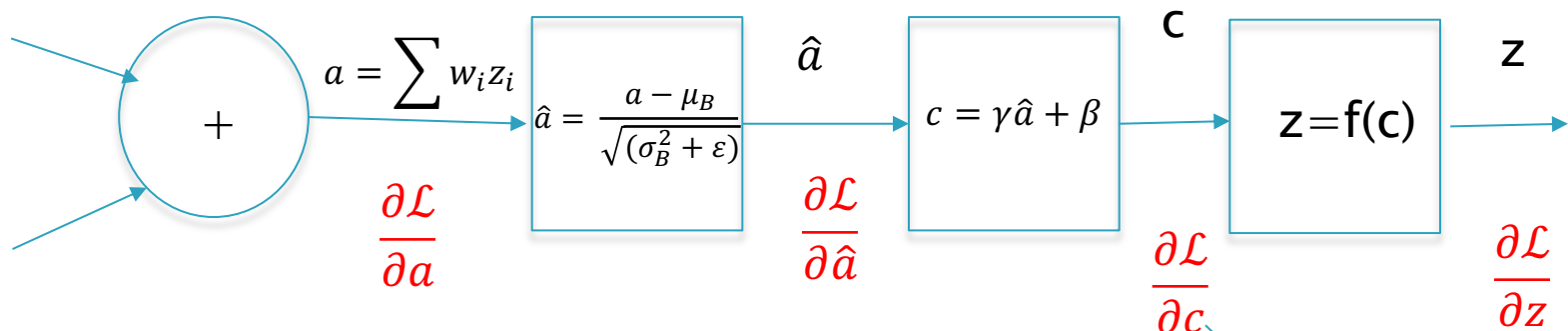
$(\gamma, \beta)$  learnt using Backprop

New Variance

New Mean

Activations no longer a function of a single training sample!

# Backprop Gradient Calculations with Batch Normalization



$$\sigma_B^2 = \frac{\sum_{s=1}^m (a(s) - \mu_B)^2}{m} \quad \mu_B = \frac{\sum_{s=1}^m a(s)}{m}$$

$$\beta \leftarrow \beta - \eta \frac{\partial \mathcal{L}}{\partial \beta}$$

$$\gamma \leftarrow \gamma - \eta \frac{\partial \mathcal{L}}{\partial \gamma}$$

# Batch Normalization: Forward Pass

1. Take a batch of size  $s$ , run it through the system, and compute the following:

$$\mu_B = \frac{1}{s} \sum_{m=1}^s a(m)$$

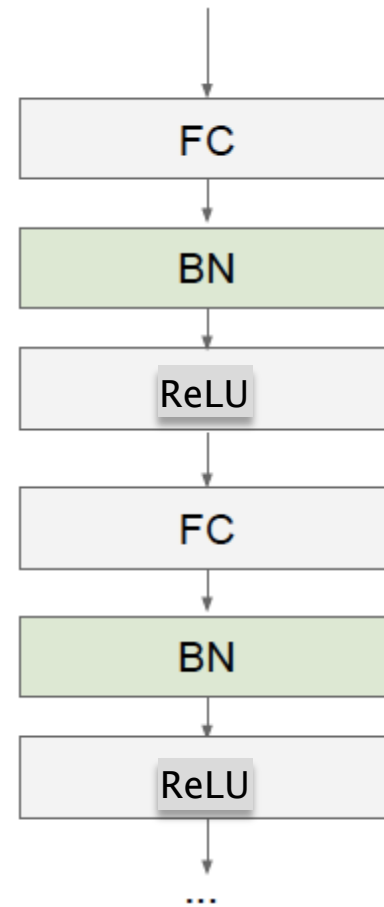
$$\sigma_B^2 = \frac{1}{s} \sum_{m=1}^s (a(m) - \mu_B)^2$$

2. Normalize all the pre-activations using

$$\hat{a}(m) = \frac{a(m) - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$c(m) = \gamma \hat{a}(m) + \beta$$

This is done on a layer by layer basis so that the normalized output from layer  $r$  is fed into layer  $(r+1)$ , which is then normalized.



# Batch Normalization: Backward Pass

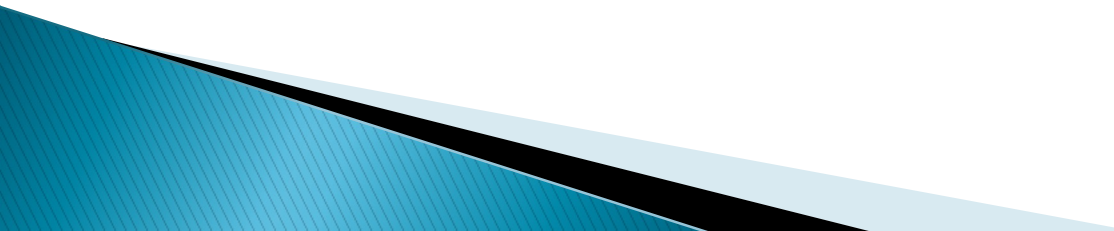
3. Run backprop through each of the samples in the batch, using the normalized activations. In addition to the weights, also calculate the gradients for the batch normalization parameters  $(\gamma, \beta)$  for each node.
4. Average the gradients across the batch, and use it to calculate the new weights and batch normalization parameters.
5. Go back to step 1, and process the next batch.

During Testing:

The pre-activations are normalized using ALL the test data, rather than just the batch



# Benefits of Batch Normalization

- ▶ Enables higher Learning Rates: Higher rates speed up convergence but can also lead to problems such as non-convergence of the SGD algorithm
  - ▶ Enables better gradient propagation through the network: Leads to effective training of larger networks
  - ▶ Helps to reduce strong dependencies on the parameter initialization scheme
  - ▶ Helps to regularize the model
- 

# Hyper-Parameter Optimization



# Hyper-Parameters

The following parameters have to be chosen:

- ▶ The number of Hidden Layers and the Number of Nodes per Hidden layer
- ▶ The Learning Rate Parameter  $\eta$
- ▶ The Regularization Parameter  $\lambda$
- ▶ The mini-batch size B
- ▶ The Dropout Rate p

How to find best value?



All these parameters influence Model Capacity

Model Capacity = Data Complexity

# Manual Tuning

The following strategy can be used to do Manual Tuning, for all parameters other than the Learning rate:

Monitor both the training and validation Loss to figure out if the model is overfitting or underfitting

High Training Loss, Valid Loss plateaus

Change hyper-parameters to increase Model Capacity

Low Training Loss, but big gap with Validation Error

Change hyper-parameters to reduce Model Capacity

The primary goal of the manual tuning is to try to match the effective capacity of the model with the complexity of the data

# Effect of Parameter Values on Model Capacity

Number of Hidden Layers  
Nodes per Hidden Layer



An increase causes the  
Model Capacity to increase

Regularization Parameter  
 $\lambda$



An increase causes the  
Model Capacity to decrease

Dropout Parameter  $p$



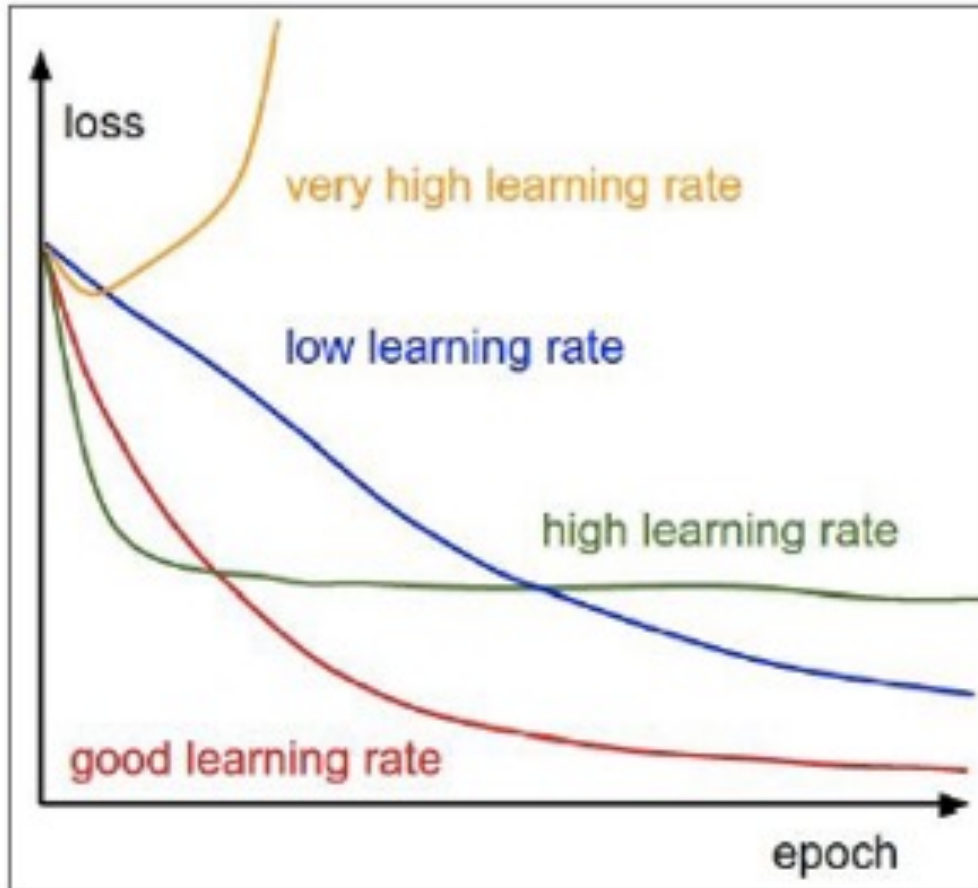
An increase causes the  
Model Capacity to increase

Learning Rate  
 $\eta$



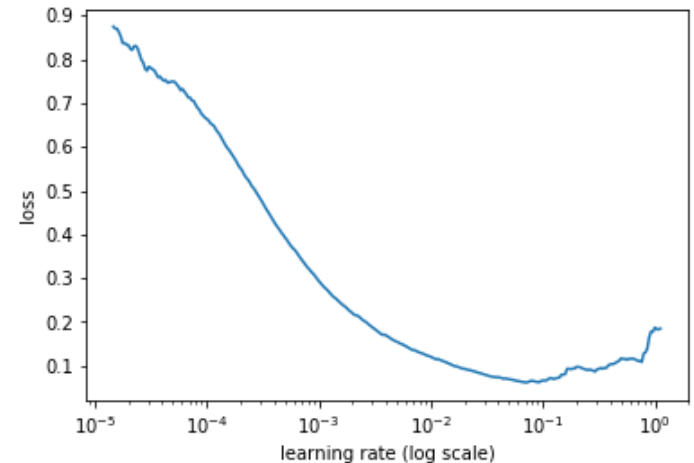
?

# Tuning the Learning Rate



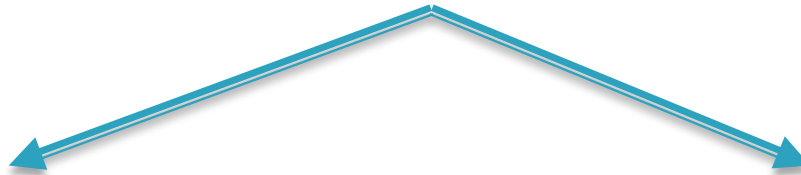
For Learning Rate and Regularization Parameter, the search is done in the logarithmic space

Example:  $10^{-6}$  to  $10^{-2}$  translates to  $(-6, -2)$



# Automated Tuning

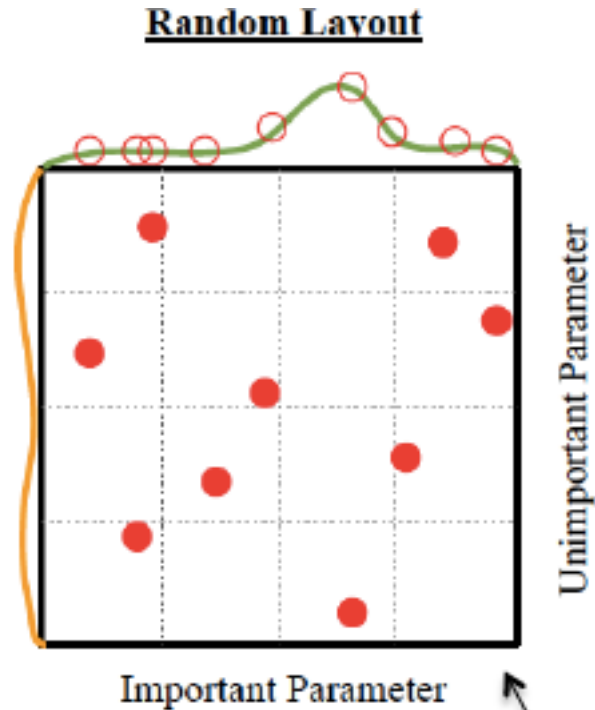
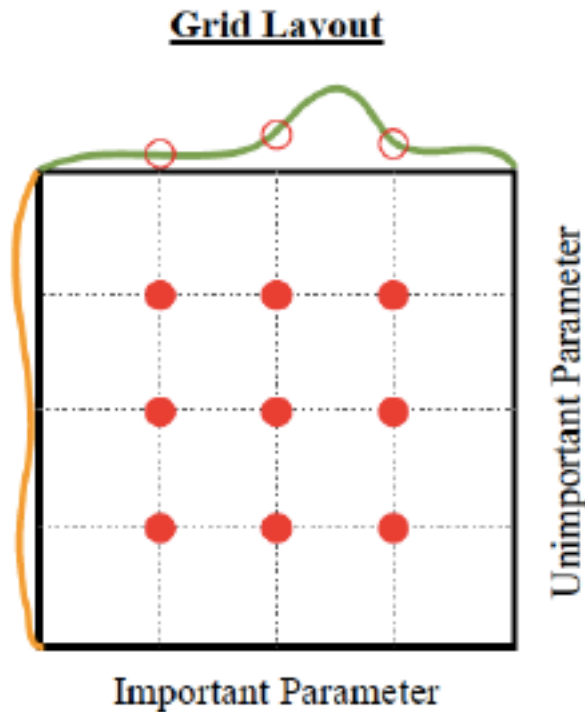
“brute-force” search in the hyper-parameter space



Grid Search: This is an exhaustive search in the parameter space in which all possible combinations are tested

Random Search: Instead of doing an exhaustive evaluation all possible hyper-parameter values, choose a few random points

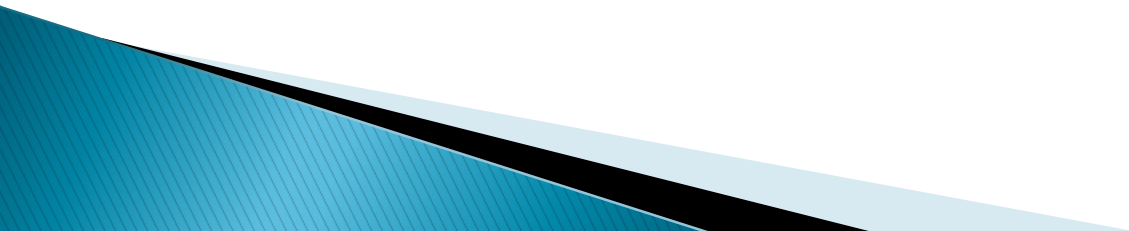
# Grid Search vs Random Search



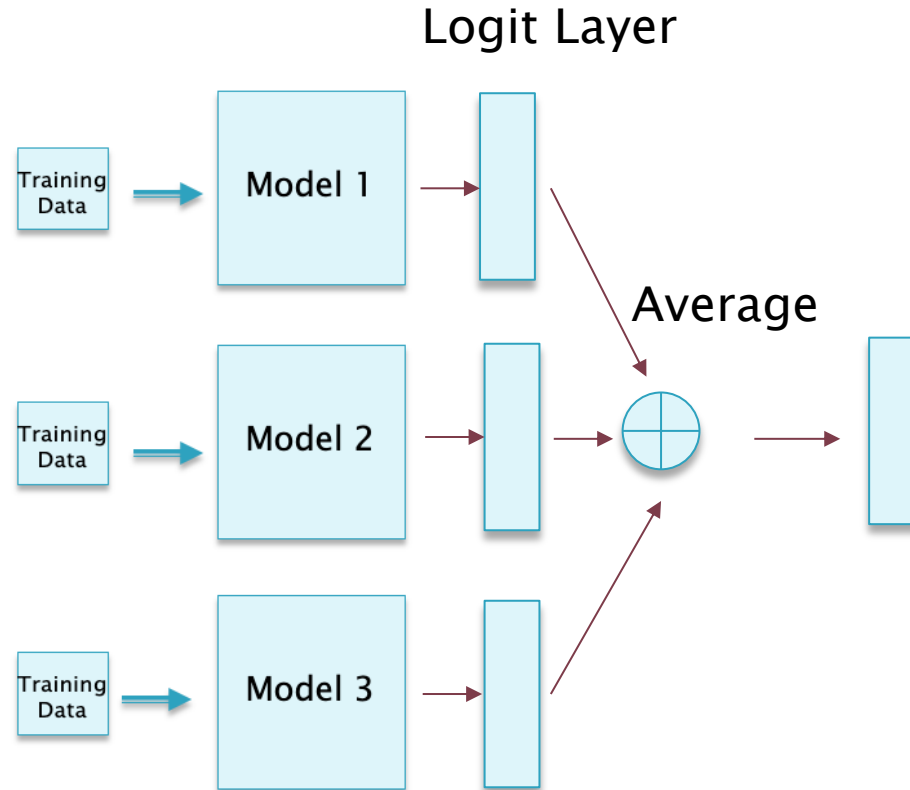
Works much better!



# Model Ensembles



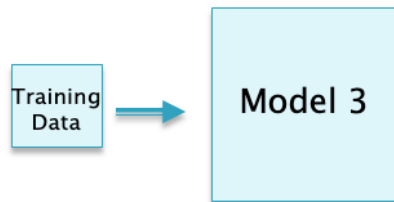
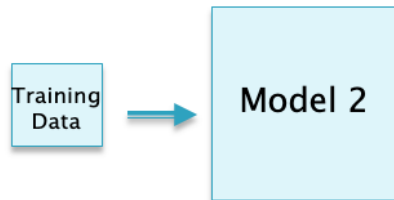
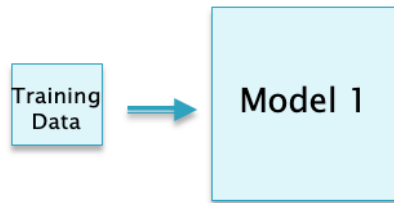
# Averaging



Models are Different  
Because they are  
initialized differently

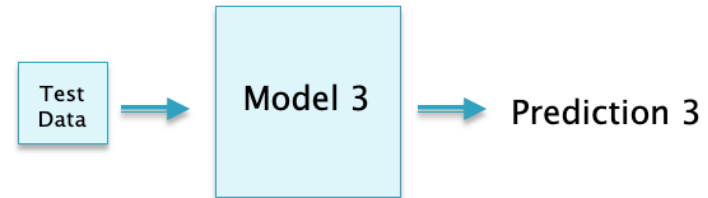
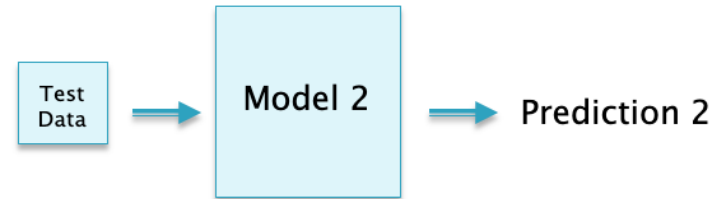
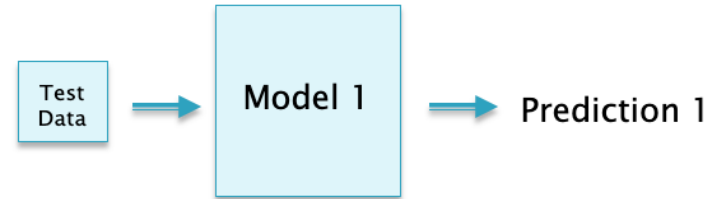
Training

# Majority Vote



Models are Different  
Because they are  
initialized differently

Training

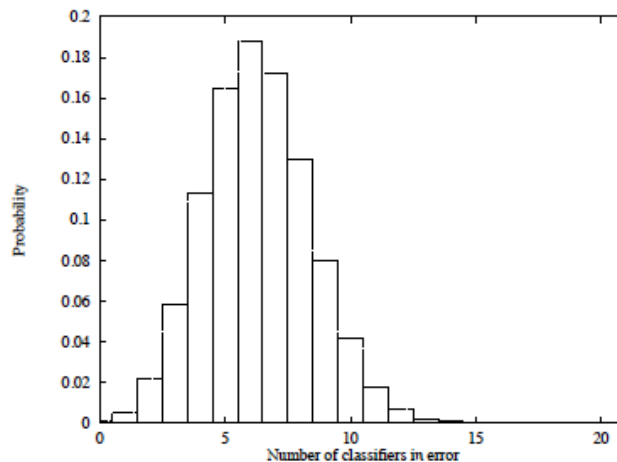


Final  
Prediction  
Majority Vote

Testing

# When do Ensembles Work?

- ▶ A necessary and sufficient condition for an ensemble of classifiers to be more accurate than any of its individual members is if the classifiers are accurate and **un-correlated**
  - Accurate: An accurate classifier is one that has an error rate of better than random guessing
  - Un-Correlated: Two classifiers are un-corelated if they make different errors on new data points

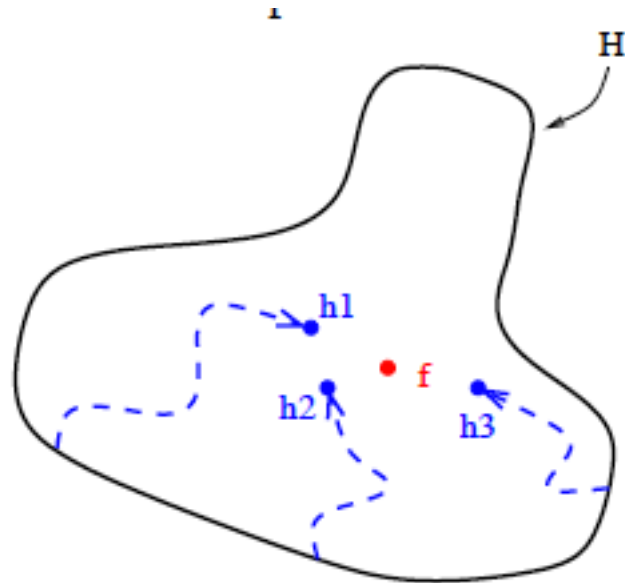


Biased Coin  
 $P(E) = 0.3$ ,  $P(C) = 0.7$   
If the coin is tossed 3 times, what is the prob of getting 2 or more E's?

Example: 3 classifiers,  $P(\text{error}) = 0.3$ . Using majority voting, if classification is done 21 Times, then the probability that 11 or more results will be in error is 0.026

CCC, CCE, CEC, ECC → Ensemble Prediction Correct  
EEE, EEC, ECE, CEE → Ensemble Prediction Error

# Why do Ensembles Work?

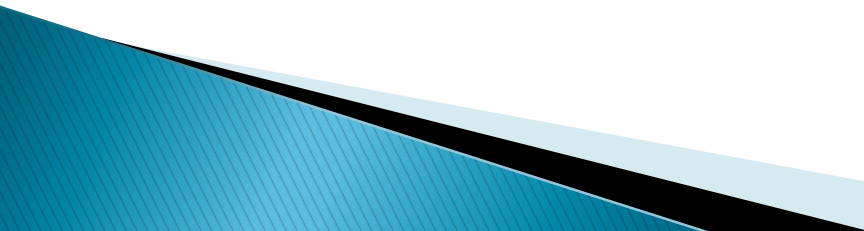


Random Initializations result in SGD based optimizations that end up in different minima → Averaging smoothens out the final result

From “Ensemble Methods in Machine Learning” by Thomas Dietterich

# A Training Example

# Universal Workflow for ML/DL Problems

- ▶ Define the Problem:
    - What is the input data? What are you trying to predict?
    - What type of problem are you solving?  
Binary Classification, Multiclass Classification, Multiclass/Multi-object Classification, Scalar Regression, Vector Regression
  - ▶ Assumptions at this stage:
    - Outputs can be predicted given the inputs
    - Available data is sufficiently informative to learn the relationship between inputs and outputs
- 

# Collect the Data Set

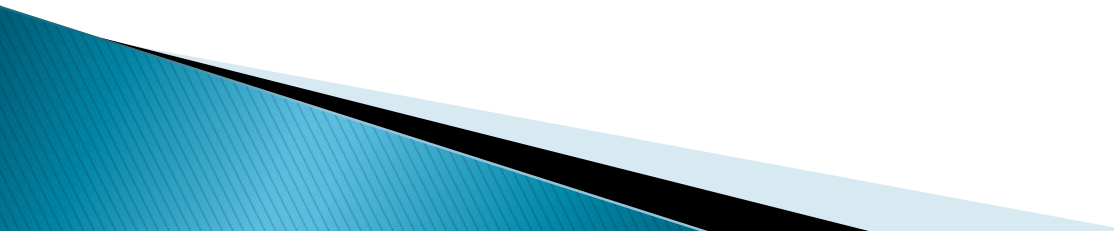
How many Training Examples Needed?

- ▶ Not enough data → Overfitting
- ▶ In general: More parameters in the model (i.e., Bigger the Model) → Larger the number of training examples needed
- ▶ For image processing problems: About 1000 samples per category usually work.
- ▶ Annotate the Data
  - Manually
  - Use a crowd sourcing platform such as Mechanical Turk
  - Use the services of a data-labeling company



# Visualize the Data Set

## Understand the Data

- Look at a few samples of the data and their labels
  - If the data is a Table, plot a histogram of features, look at range and the frequencies
  - For classification compute the number of samples from each class
- 

# Common Pitfalls

Common Cause of Model Failure: Non Stationary Problems!  
The Mapping between the Input and Output changes with time

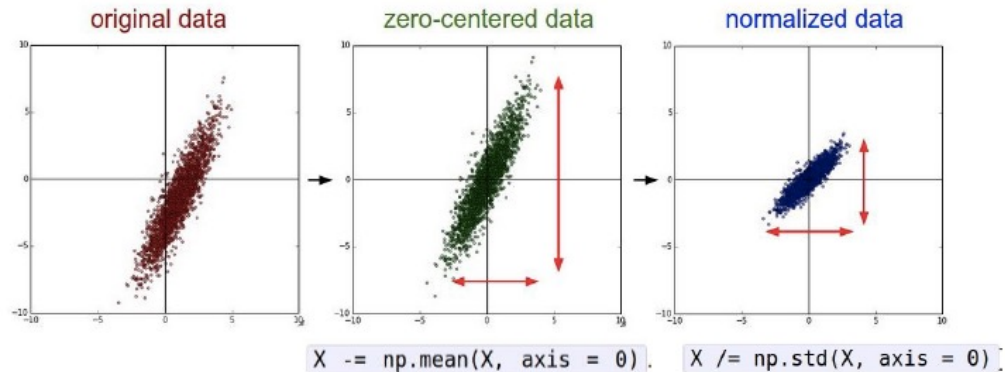
Number of samples in each category should be close in number

- Both the Training Set and the Test Set should be representative of the data
- If the data samples are ordered by class, then make sure to randomly shuffle the data before splitting it into training and test
- If you are trying to predict the future given the past, don't randomly shuffle the data
- If data samples repeat, then make sure that the training and validation/test sets are disjoint



Target  
Leaking!

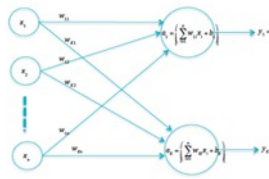
# Prepare the Data



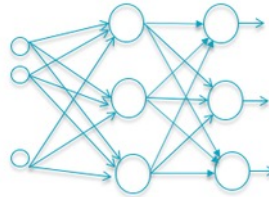
Pre-process the data so that it can be fed into the model, usually Domain Specific

- Convert into tensors
- Normalize: Zero center and/or scale
- For NLP Data: Convert into vectors

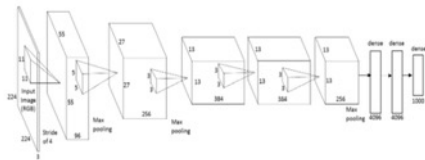
# Choose the Model



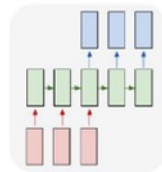
(a) Linear Model



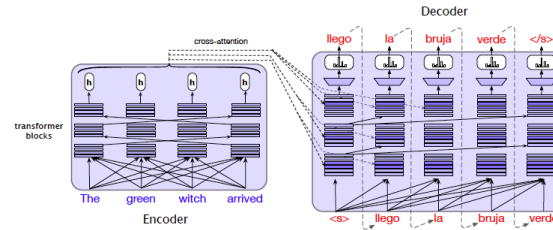
(b) Deep Feed Forward Network



(c) Convolutional Neural Network



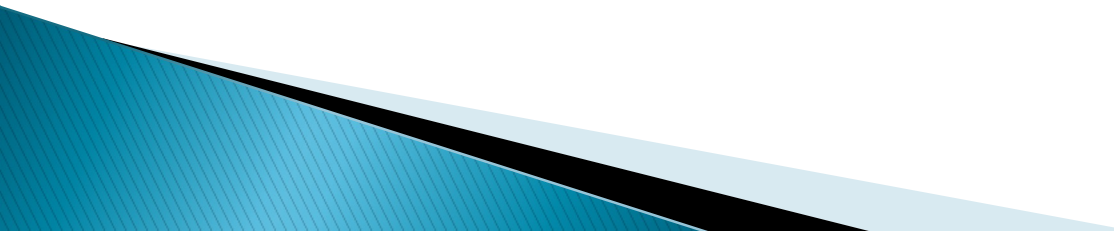
(d) Recurrent Neural Network



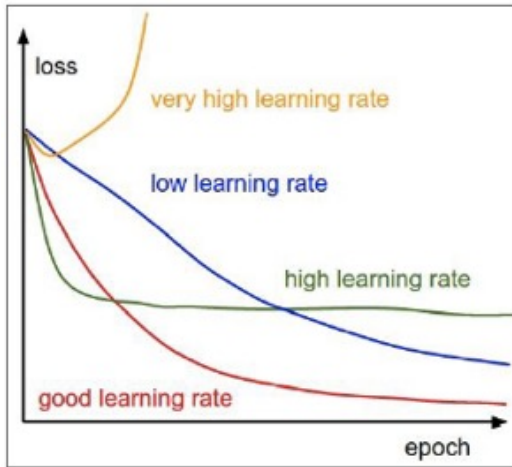
(e) Transformers

- If the input data is such that the various classes are approximately linearly separated then a linear model will work
- Deep Learning Networks are needed for more complex datasets with non-linear boundaries between classes. If the input data has a 1-D structure, then a Dense Feed Forward Network will suffice
- If the input data has a 2-D structure (such as black and white images), or a 3-D structure (such as color images), then a Convolutional Neural Network or ConvNet is called for. ConvNets excel at object detection and recognition in images
- If the input data forms a sequence with dependencies between the elements of the sequence, then a Recurrent Neural Network or a Transformer is required.  
Typical examples of this kind of data include:  
Speech waveforms, natural language sentences, stock prices etc.  
RNNs/Transformers are ideal for tasks such as speech recognition, machine translation, captioning

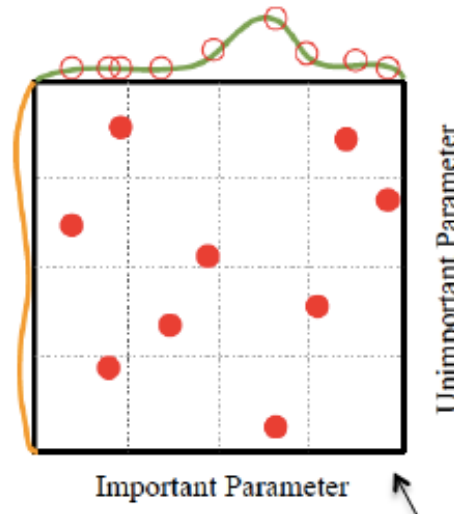
# Model (cont)

- Choose Last Layer Activation: Sigmoid, Softmax, tanh, None
  - Choose Loss Function:  
Binary Classification → Binary Crossentropy,  
Multiclass, Single Label Classification → Categorical Crossentropy  
Multiclass, Multi Label Classification → Binary Crossentropy  
Regression → MSE
  - Choose Optimizers: rmsprop, Adam
- 

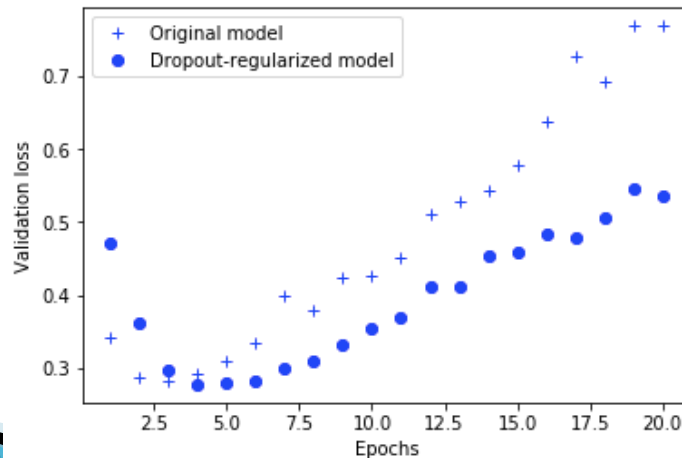
# Train the Model



Random Layout



Choose Learning Rate: The most important hyper parameter (keep regularization low while doing this)

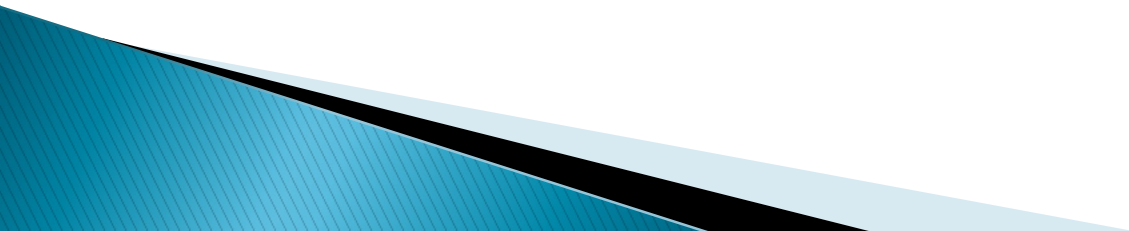


Choose other hyper-parameters by checking for overfit/underfit - change model capacity to improve generalization

# Deploy the Model

- ▶ Models are typically deployed as part of Web Servers, Mobile Apps, Web Pages, Embedded Devices etc
  - Keras and Tensorflow have support for deploying models in these environments → Example: TensorFlow.js is a JavaScript library that implements most of the Keras API, TensorFlow Lite for embedded devices
- ▶ Monitor performance after deployment
- ▶ Collect data for next generation model: Some models need to be updated frequently
  - Data changes over time: Example Credit Card fraud detection model, has to be updated every few days

# Expanding the Dataset Artificially

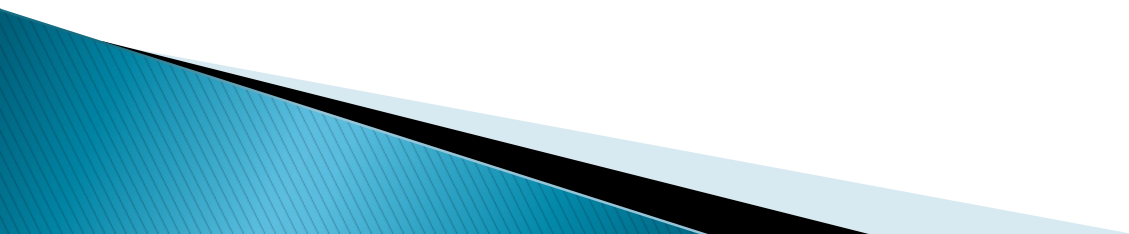




# Data Augmentation

Model Overfitting can be caused due to  
insufficient Training Data

Symptom: Model Overfits even after application  
of Regularization



# Artificially Expanding Training Data (For Images)

Main Idea: Expand the training data by applying operations that reflect real world variation, such that  
The Training Label does not change.

# Data Augmentation

## Horizontal Flips



# Data Augmentation

## Random crops and scales

**Training:** sample random crops / scales

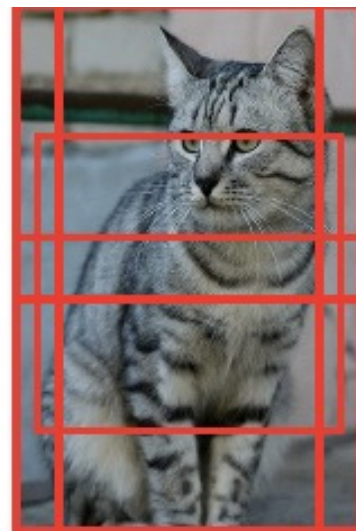
ResNet:

1. Pick random  $L$  in range  $[256, 480]$
2. Resize training image, short side =  $L$
3. Sample random  $224 \times 224$  patch

**Testing:** average a fixed set of crops

ResNet:

1. Resize image at 5 scales:  $\{224, 256, 384, 480, 640\}$
2. For each size, use 10  $224 \times 224$  crops: 4 corners + center, + flips



# Data Augmentation

## Color Jitter

Simple: Randomize  
contrast and brightness



# Data Augmentation

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, ...

Is Data Augmentation also a type of Regularization?

# Data Augmentation Using Keras

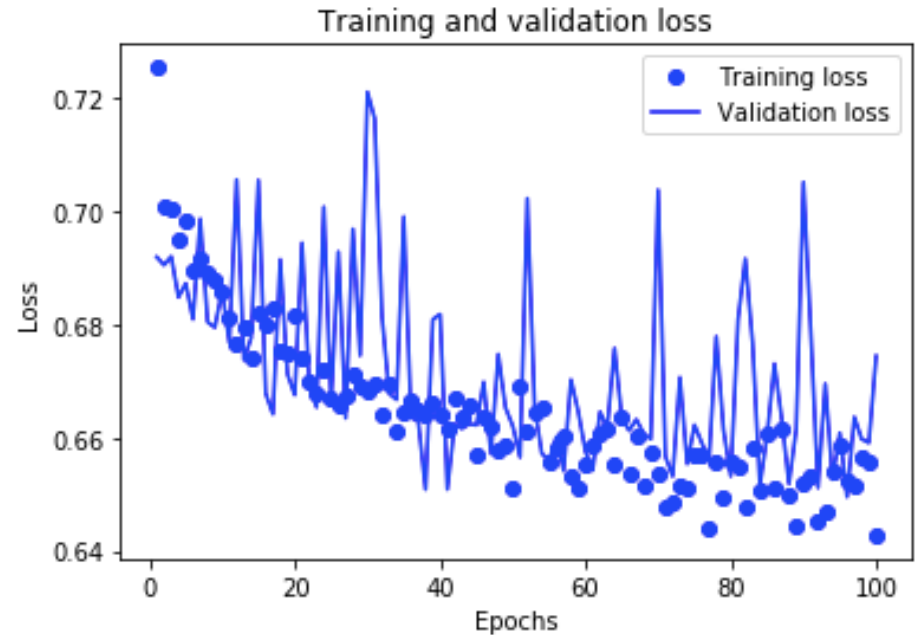
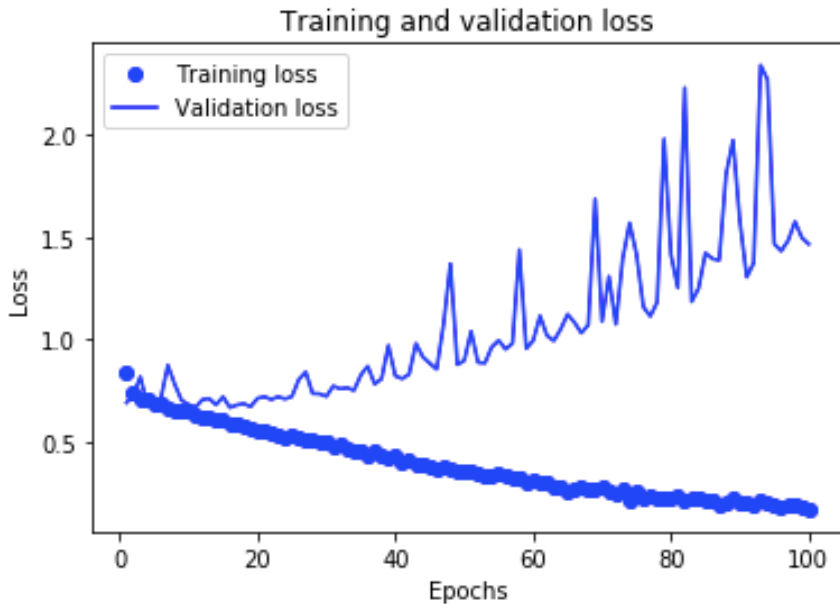
```
datagen = ImageDataGenerator(  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```

These are just a few of the options available (for more, see the Keras documentation). Let's quickly go over what we just wrote:

- `rotation_range` is a value in degrees (0-180), a range within which to randomly rotate pictures.
- `width_shift` and `height_shift` are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
- `shear_range` is for randomly applying shearing transformations.
- `zoom_range` is for randomly zooming inside pictures.
- `horizontal_flip` is for randomly flipping half of the images horizontally -- relevant when there are no assumptions of horizontal asymmetry (e.g. real-world pictures).
- `fill_mode` is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

# Loss Functions

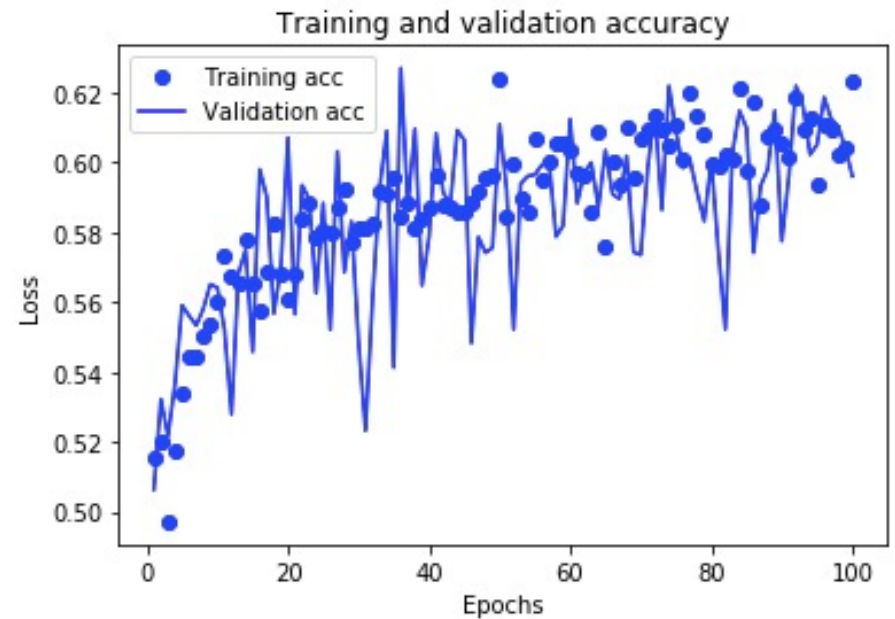
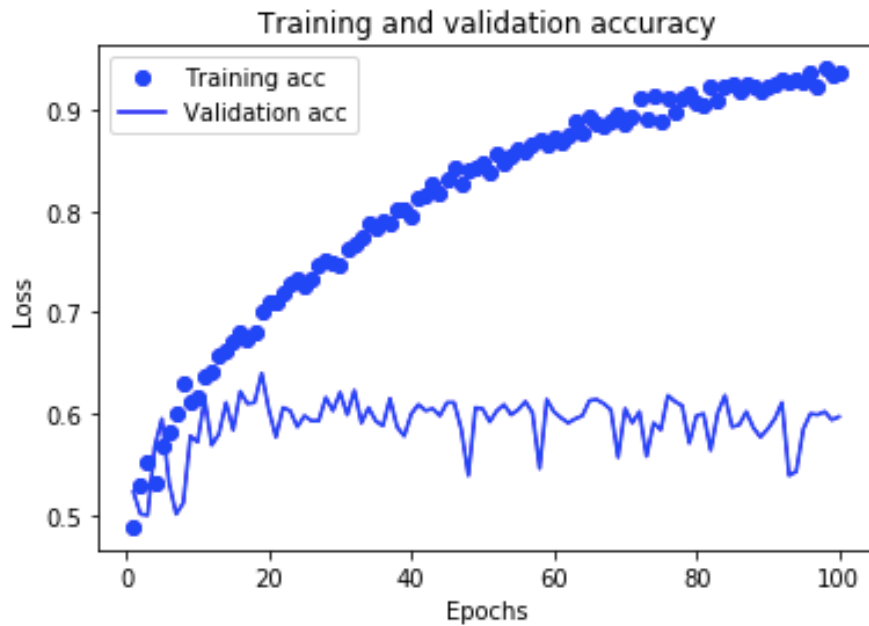
## With and Without Data Augmentation





# Accuracy

## With and Without Data Augmentation



# Further Reading

- ▶ Chapter: GradientDescentTechniques
  - ▶ Chapter: HyperParameterSelection
  - ▶ Chollet Chapter 6
- 