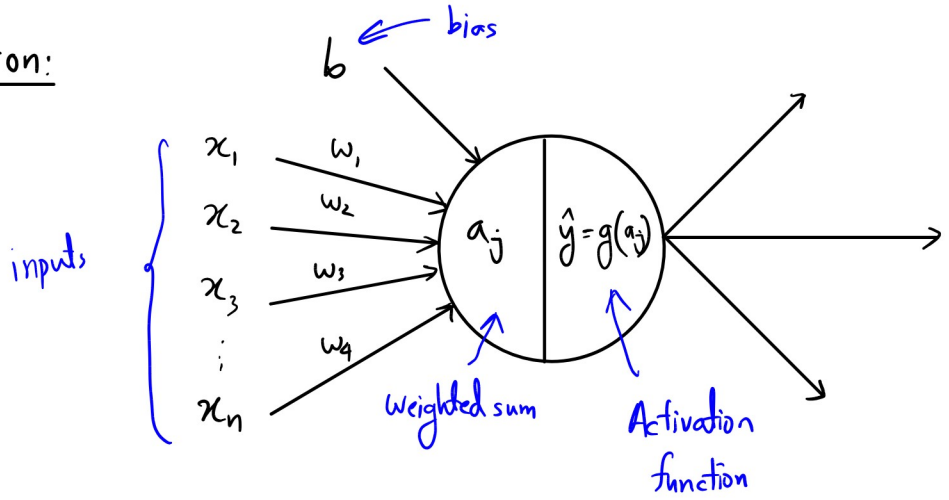
Neural Network

Book Ch 7, 7.1

Neuron:



jth cell

Weighted sum, $a_j = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b$

Let, $w_0 = b$ and $x_0 = 1$,

[Replacing the bias, book page 144]

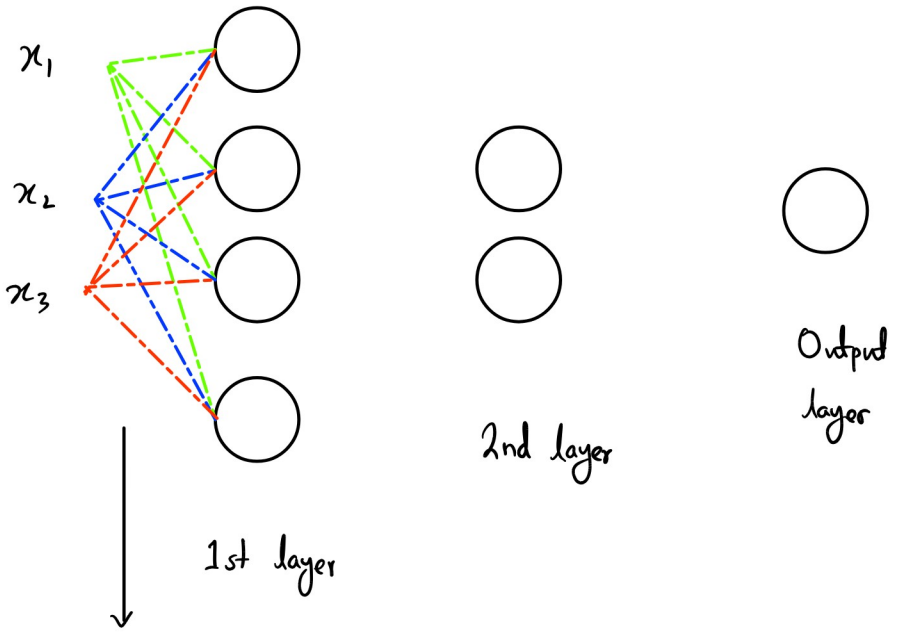
$$a_j = \underbrace{x_0 w_0}_{\text{bias}} + x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

∴ weighted sum of a single unit, $a_j = \sum_k x_k w_k$

∴ Output of jth neuron $= g\left(\begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} w_{1,j} \\ w_{2,j} \\ \vdots \\ w_{3,j} \end{bmatrix} \right)$

$= g(x \quad w_{*,j})$

Now, considering the following neural network



$$W^{(1)} = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \end{bmatrix}$$

3×4 weight matrix

So, $a^{(1)} = x \times W^{(1)}$

$= [1 \times 3] \times [3 \times 4]$ ← matrix shape

$= [1 \times 4]$ ← " "

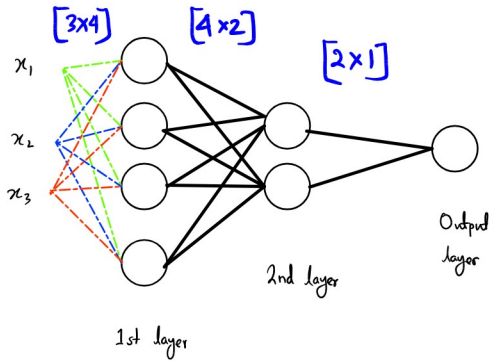
After using activation function on each element,

$$g(a^{(1)}) = g([1 \times 4])$$

$$= [1 \times 4]$$

This $[1 \times 4]$ contains the output of 4 neurons in the 1st layer. These will be the input of 2nd layer.

Now we can easily find out the required weight matrix shape.



∴ Output of this NN,

$$O = g(g(g(x \times w^{(1)}) \times w^{(2)}) \times w^{(3)})$$

$$= g(g(g([1\times3][3\times4]) \times [4\times2]) \times [2\times1])$$

$$= g(g([1\times4] \times [4\times2]) \times [2\times1])$$

$$= g([1\times2] \times [2\times1])$$

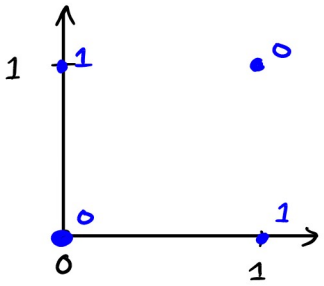
$$= g([1\times1])$$

$$\text{Output} = [1\times1]$$

Determining the size of matrix are very important.

✱ Using the NN to solve the XOR problem

Ch 7.2



✱ We can not find a boundary line in this case using LR or a single neuron.

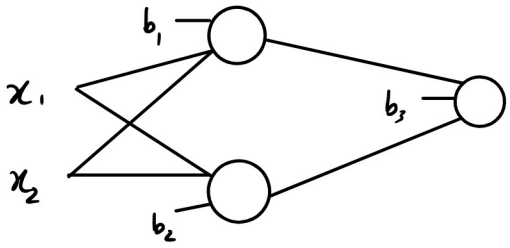
$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$y = XW + b$$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} ? \\ ? \end{bmatrix} + ?$$

✱ No such w exists.

But using a NN,



$$g(x) = \text{Relu}(x) = \max(0, x)$$

$$\Rightarrow \max\left(0, XW^{(1)} + b^{(1)}\right) w^{(2)} b^{(2)} = y$$

$$= \max\left(0, \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} + [? \;\; ?]\right) \begin{bmatrix} ? \\ ? \end{bmatrix} + ? = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

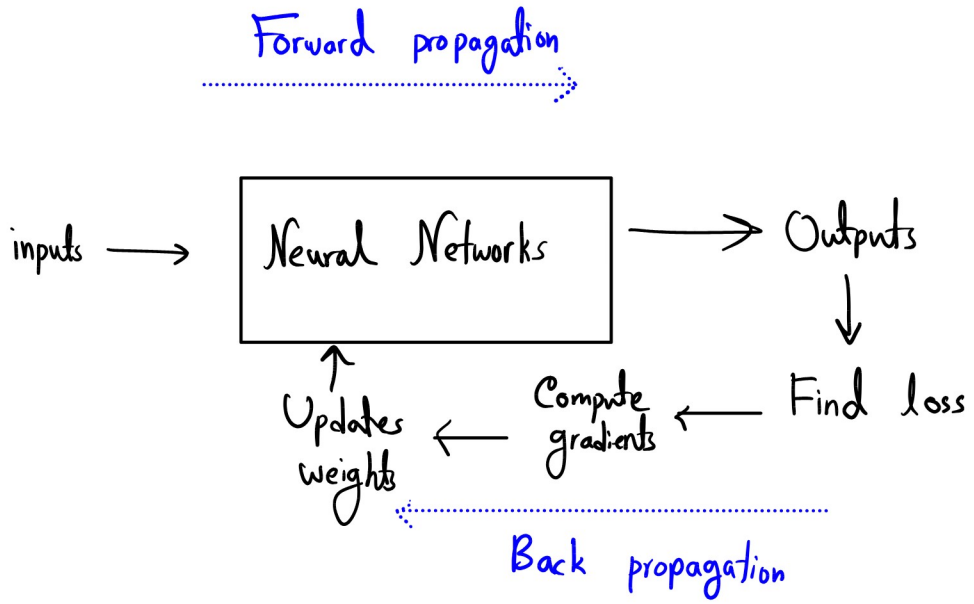
If we use the following weights & Bias,

$$= \max\left(0, \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + [0 \;\; -1]\right) \begin{bmatrix} 1 \\ -2 \end{bmatrix} + 0$$

$$= \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = y$$

Note: There are multiple solution for this XOR problem. You can use other weights & other activation functions too.



Component of a Neural Networks:

Activation functions: [Ch 7.3.1]

In neural network, we denote activation function by $g(x)$ which can be any function.

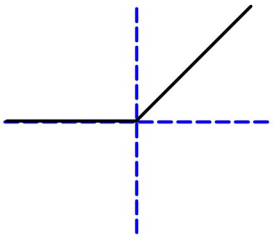
$g(x)$ can be linear function. But a non-linear function is required. Why?

Because, with linear activation functions, the neural network becomes just a linear transformation, no matter how many layer it has. So, it can not do complex mapping like the XOR problem.

Also, in a linear activation function, we can not use back propagation as the derivative of a linear function is constant.

Common Activation function:

ReLU: $h = \max(0, z)$



Active only when input is positive

Gradient:

1 when z = +ve

0 when z = -ve

But, ReLU is non differentiable when $z = 0$.

in this case, left derivative = 0

right " = 0

A few non differentiable point is not a problem as

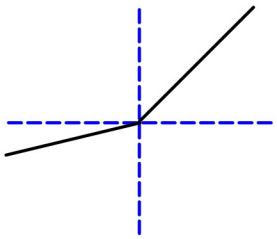
① Rarely we get a point with gradient 0 anyway
② Software simply return either left or right derivative.

ReLU has a dead neuron problem. As ReLU outputs 0 for all negative inputs, which can make some neurons stop activating entirely.

When this happends, their gradient becomes 0, and their weights never updates. Thus those neurons are dead.

Leaky ReLU:

$$h = \max(0.1z, z)$$



- Strong activation when z is positive
- weak negative activation, $\because z$ is negative

Gradient:

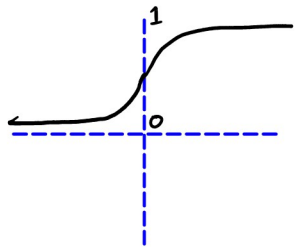
1 when positive
0.1 when negative

Sigmoid:

$$f(z) = \frac{1}{1+e^{-z}}$$

→ Maps z into probability

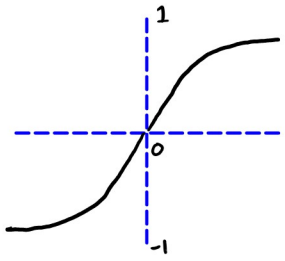
→ 0 / 1 switch



Gradient:

Saturates accross most of its domains.

Tanh:



$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

→ -1/+1 switch

Gradient:

Saturates accross most of its domains.

Optimizes slightly better as around $z = 0$, the gradient is closer to 1.

NN weights aka parametors are learnt through out the training phase.

Hyperparameters:

→ The parameters that we can control to facilitate the learning are called hyperparameter.

- Model Architecture Related
  - (I) Number of layers
  - (II) Number of neuron per layer
  - (III) Activation functions in each layer

- Training Related:
  - (I) Optimizer
  - (II) Number of epoch
  - (III) Learning Rate
  - (IV) Batch Size
  - (V) Regularization & Normalization

& many more.

Optimizer:

These are algorithms that are used to update weights in order to reduce loss.

Common optimizers:

First Order Methods:
- Gradient Descent
- Stochastic Gradient Descent
- Mini-batch Gradient Descent
- Momentum

- Use first order derivatives only
- Maintain a single learning rate

Adaptive methods:
- Adagrad
- RMS prop
- Adadelta
- Adam

- Adagrad: → provide learning rate for each parameter.
  → improves learnability but computationally expensive.

- RMS prop: → provide learning rate for each parameter.
  → adapts based on the mean of the recent magnitudes of the gradient of each weight

- Adadelta: → Similar as RMS prop but works with second order derivatives.

✯ Adam: Adaptive Moment Estimation

→ Most popular and mostly used
→ Straight forward to implement
→ Little memory requirements
→ Need little to no manual tuning.

✯⇒ It combines Momentum & RMSProp for efficient & adaptive update.

Regularization : [Ch 5.7]

→ A set of strategies used in ML to reduce Generalization Error so that model can perform well on unseen data, not just traing data.

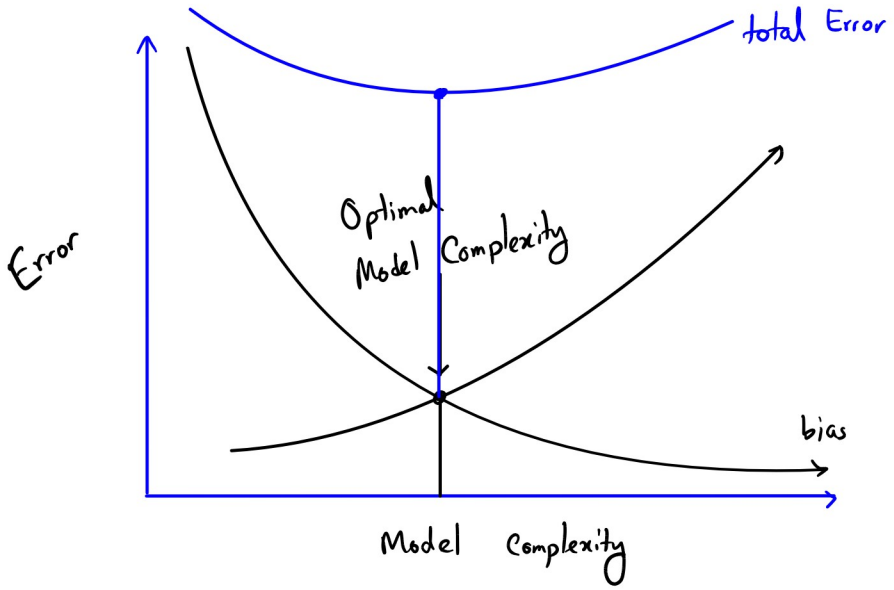
✱ → Bias Varience Tradeoff

Bias : • Error from overly simple assumption by simple models.

- High bias means the model makes strong assumptions and often results in Underfitting.
- Observation doesn't matter.

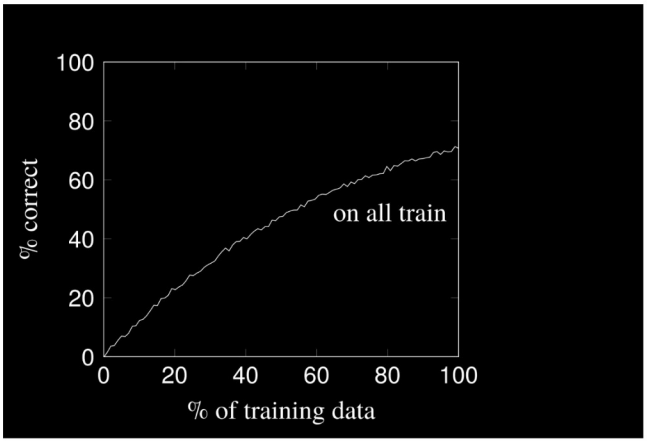
Variance: • Error from sensitivity to small flactuations in the training data.

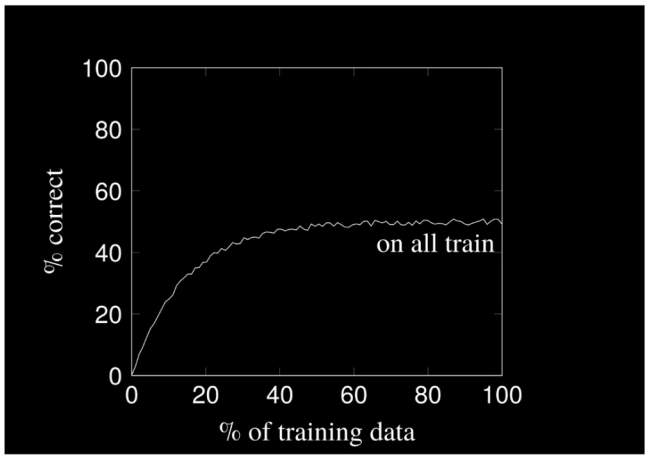
- High variance may result in modeling the noise in training data, causes Overfitting
- focuses too much in observation



Underfitting: Low training & test accuracy.

- Insufficient training data
- Too simple model



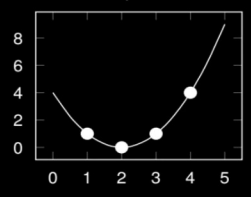


Addressing underfitting

More complex model

x_1	$f(x)$	$h(x)$
1	1	1
2	0	0
3	1	1
4	4	4

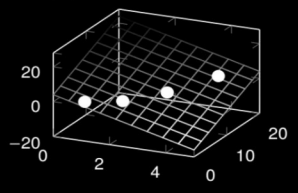
$h(x) = x_1^2 - 4x_1 + 4$



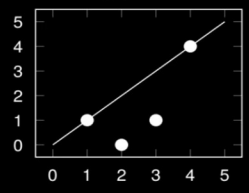
More features

x_1	x_2	$f(x)$	$h(x)$
1	1	1	1
2	4	0	0
3	9	1	1
4	16	4	4

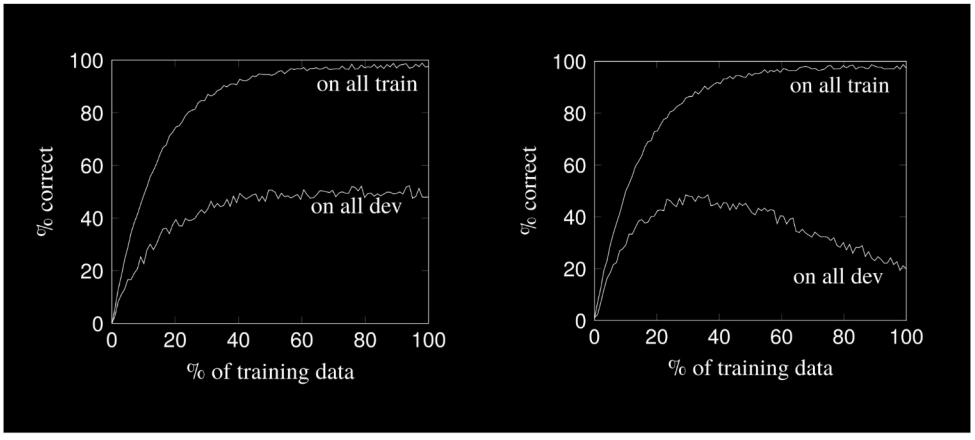
$h(x) = -4x_1 + x_2 + 4$



$h(x) = x_1$



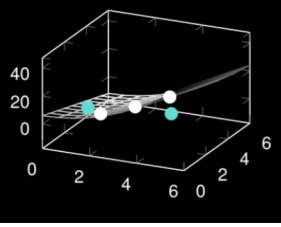
Overfitting: High training accuracy but low dev accuracy.



Addressing overfitting

x_1	x_2	$f(x)$	$h(x)$
2	1	7	7
3	2	9	9
4	3	13	13
1	2	3	1
5	1	15	28

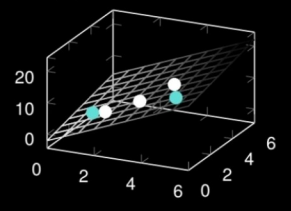
$h(x) = x_1^2 - 3x_2 + 6$



Simpler model

x_1	x_2	$f(x)$	$h(x)$
2	1	7	5
3	2	9	9
4	3	13	13
1	2	3	3
5	1	15	14

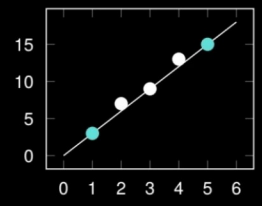
$h(x) = 3x_1 + x_2 - 2$



Fewer features

x_1	$f(x)$	$h(x)$
2	7	6
3	9	9
4	13	12
1	3	3
5	15	15

$h(x) = 3x_1$



What can be done to get the balance between bias & variance?

① Introduce a regularization term in the loss function

→ Adds a small bias to reduce variance → Reduces overfitting

$$\text{Loss} = \underbrace{\text{Original Loss}}_{\text{Cross entropy or others}} + \left(\lambda * \underbrace{\text{Regularization Term}}_{\text{L1 or L2}}\right)$$

λ controls how strong the penalty is.

Common Regularization methods:

L2: Ridge Regression

$$\text{Penalty} = \lambda \sum w_i^2$$

- Adds the squared magnitude of weights to the loss
- Penalize the large weights more strongly.

- Keeps all features but shrinks their influence
- Higher λ → smaller weights → smoother model
  ↳ less complex

L1: Lasso Regression

$$\text{Penalty} = \lambda \sum |w_i|$$

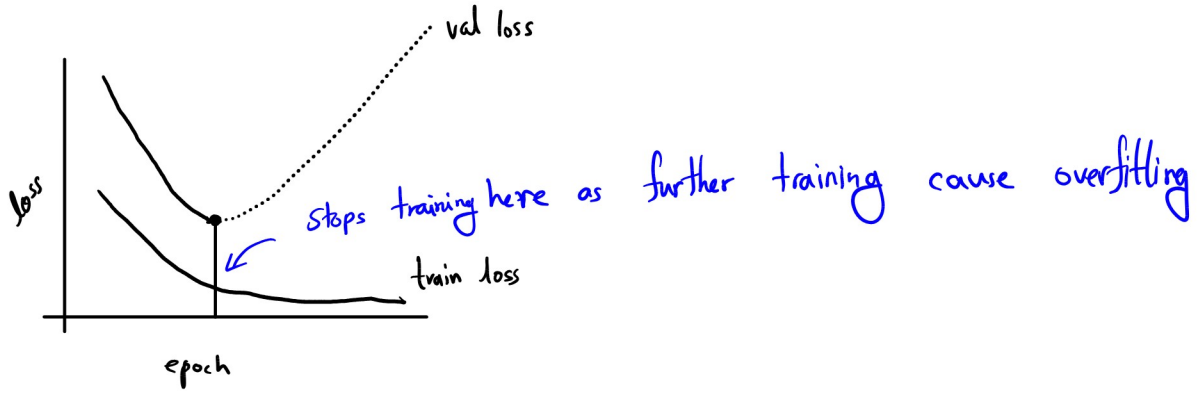
- Adds the absolute value of weights to the loss
- Higher λ → more weights becomes 0
- If a weights becomes 0, the feature doesn't affect the model.
  reduces complexity

② Dropout:

Randomly drop outs (ignore) some neuron outputs during training with a probability of p.
↳ depends on architecture
↳ Usually (0.2 to 0.5)

③ Early Stopping:

- Stops when validation performance stops improving.
- Returns the best current model.



④ Data Augmentation:

→ Introduce new training data

→ Injects noise

Other Hyperparameters:

① Learning Rate:

Lower LR = Slows down Learning but smooth converges.
Higher LR = speeds up learning but may not converge.

Decaying LR is preferred.

↳ Initially high LR, then low LR.

② Epochs:

how many time the entire training dataset has passed through the model.

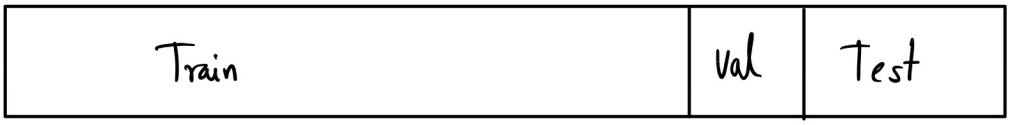
③ Batch size:

a subset of training data. Weights are updated after each mini batch of training.

Smaller mini batch → too many update
Larger mini batch → too few updates, too many epoch to converge.

ML Best Practices:

Best practice:



- Never peak the test data

Bad idea:

1. Train on training data
2. Evaluate on test data
3. Tune / Update parameters

Better idea

1. Train on training data
2. Tune on validation data
3. Evaluate on test data

Why? Indirectly using test data to guide learning. Causes overfitting. Model has good test performance but performs poorly in real world data.

Performance Metrics

[Ch 4.7, 4.8]

$$\text{Accuracy} = \frac{\text{Correct}}{\text{Total}}$$

- A bad metrics, specially for
- If a dataset contains 95% not spam and 5% spam and the model always predict not spam always, then the accuracy is 95% which is useless

Confusion Metrics:

Actual \ Predicted	0	1
0	TN True Negative	FP False Positive
1	FN False Negative	TP True Positive

Alternative of Accuracy:

① **Precision** $= \dfrac{TP}{TP + FP}$

- Measures how many predicted positives were actually correct.
- high precision = less false positive
- Used when false alarm is costly. (Spam detection)

② **Recall** $= \dfrac{TP}{TP + FN}$

- Measures how many actual positives were correctly classified.
- High Recall → few false negative
- Use when missing a positive case is risky (Diseases Detection)

③ **F_1 Score:**

$$F_1 = 2 * \frac{(\text{Precision} * \text{Recall})}{(\text{Precision} + \text{Recall})}$$

- Harmonic mean of precision and recall
- Balances both P and R
- Useful when dataset is imbalanced and both FP and FN matters

Other task specific Metrics:

- BLEU – Machine Translation
- WER – Automatic Speech Recognition
- Cosine Similarity – Semantic Similarity
- Euclidean distance – Spell checking.
- F-latency – Early Detection

Statistical Significance:

[Ch 4.9]

- Two models A and B has tiny score differences. (example: A = 0.992 F_1, B = 0.991 F_1)

  is A actually better than B?
- We can't just trust one test set.
- So, we frame two hypothesis

  → Null hypothesis → H_0 : A is not truly better than B (difference is luck)

  → Alternative → A is actually better

- Now, we gather multiple evaluation samples (via cross validation)
- Then we run significance test (Student's t test) to get a p-value.
- If pvalue is small, we reject H_0 and say, 'A is significantly better than B'