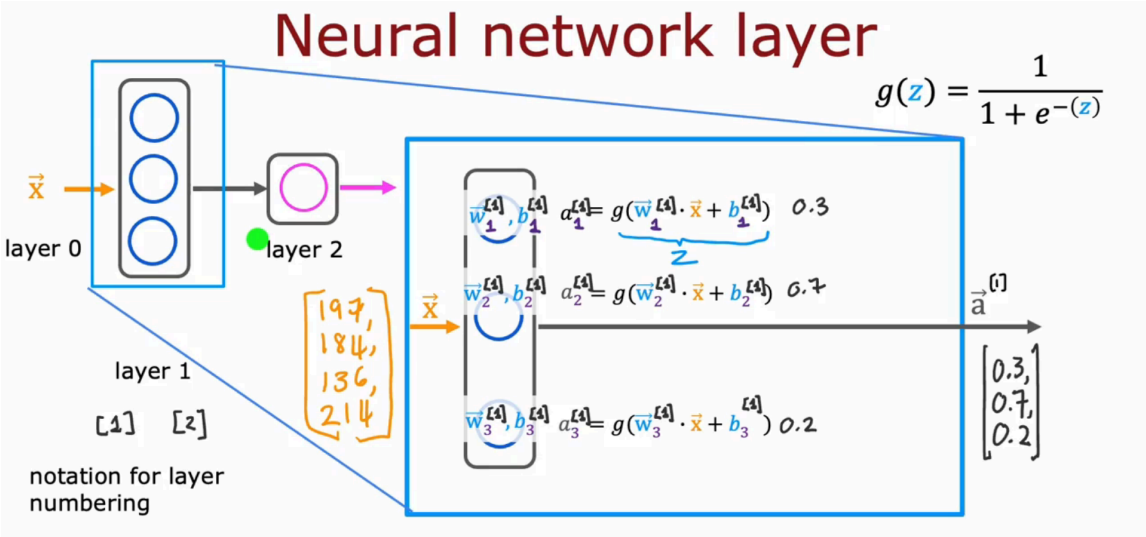
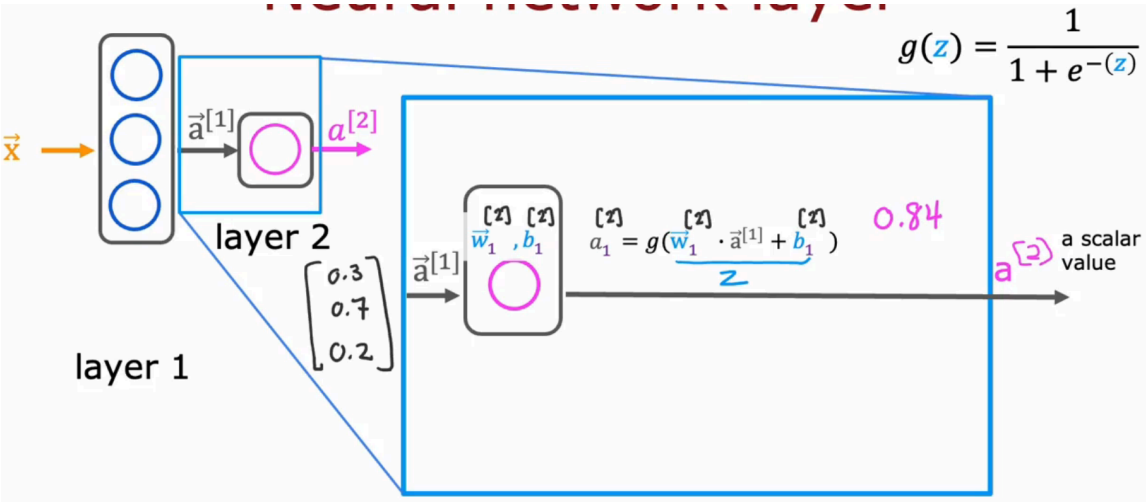
Neural Network Layer 1

The features vectors are given as input in Layer 1. The each neuron computes output using sigmoid function.

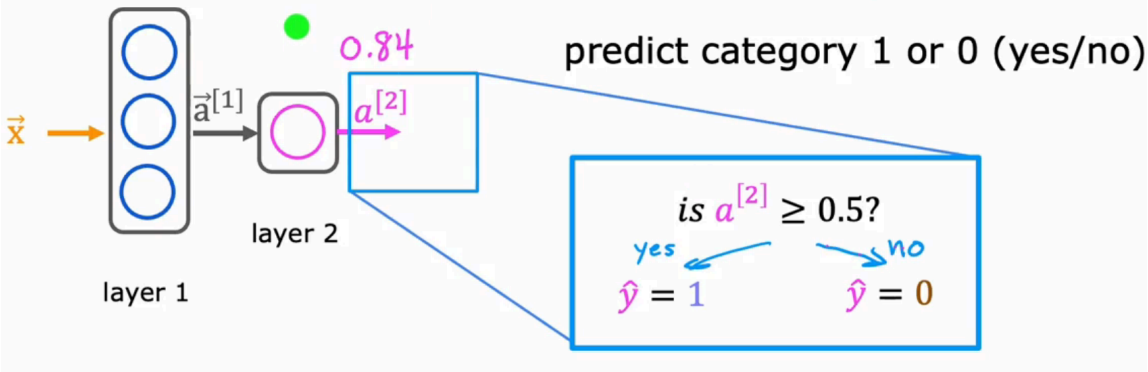


Neural Network Layer 2

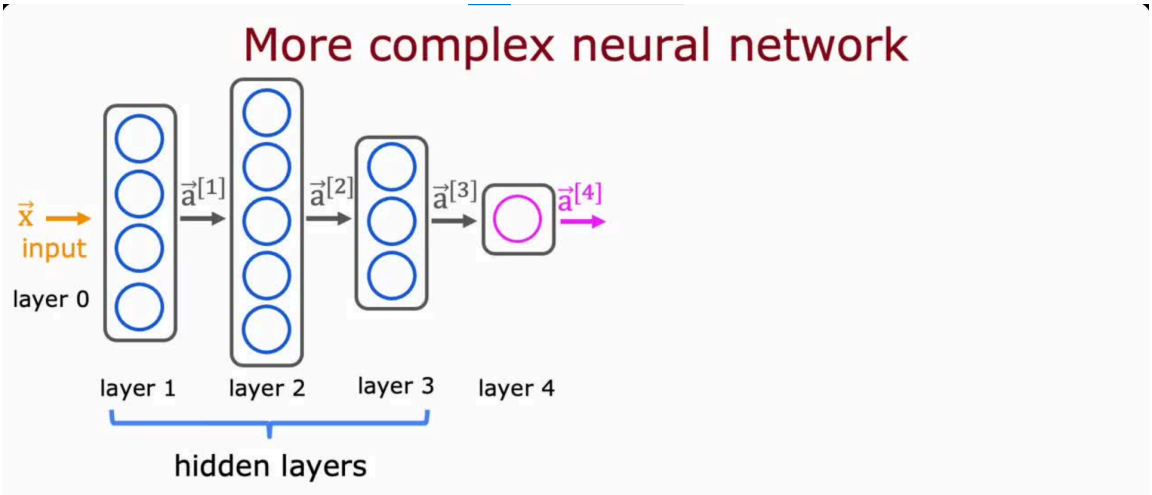


Output Layer of Neural Network

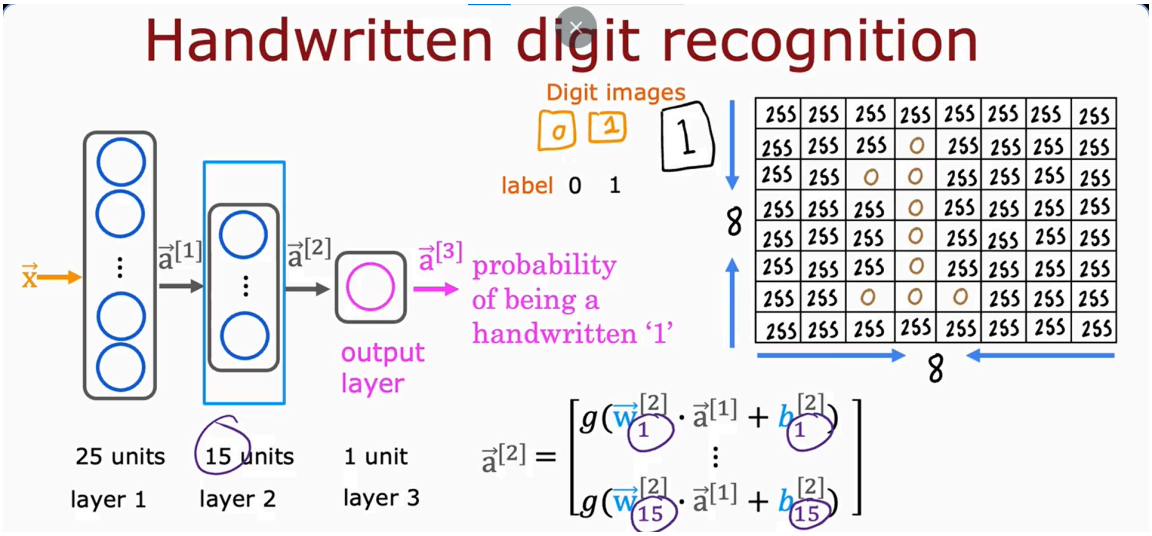
The final layer of the neural network uses decision boundary to classify the output and map them to different class.



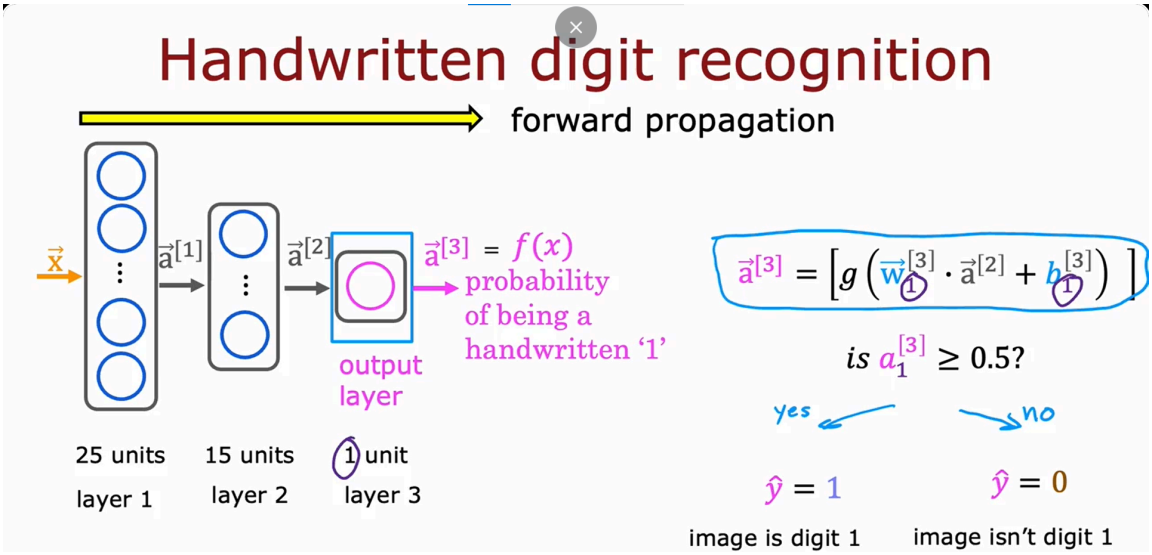
More Complex Neural Network



Here the g refers to the activation function which is sigmoid function.



The number of output from each layers depends on the number of neurons each layer has. In the above image we can see that, Layer 2 has 15 units of neuron which means that output of layer 2 ($\vec{a}^{[2]}$) has 15 units in the martix from.



The last layer of the neural network act as a decision layer to decide which class the input will fall. So, starting from the input layer up to the output layer input travels from left to right. This is known as froward propagation as the input propagates from left to right.

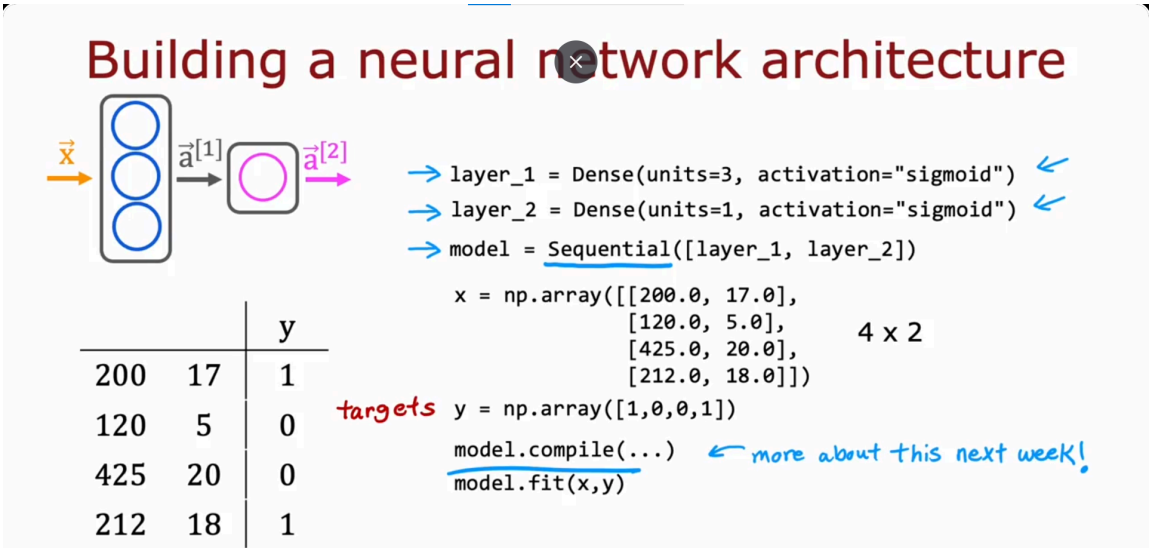
Layer Implementation in Tensorflow

```
#Layer 1
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
#Layer 2
layer_2 = Dense(units=1, activation='sigmoid')
a2(a1)
```

Conversion of numpy array and Tensorflow

```
x=a.tensor() #converted to tensor
y=x.numpy() #converted to numpy array
```

Building Neural Network in Tensorflow

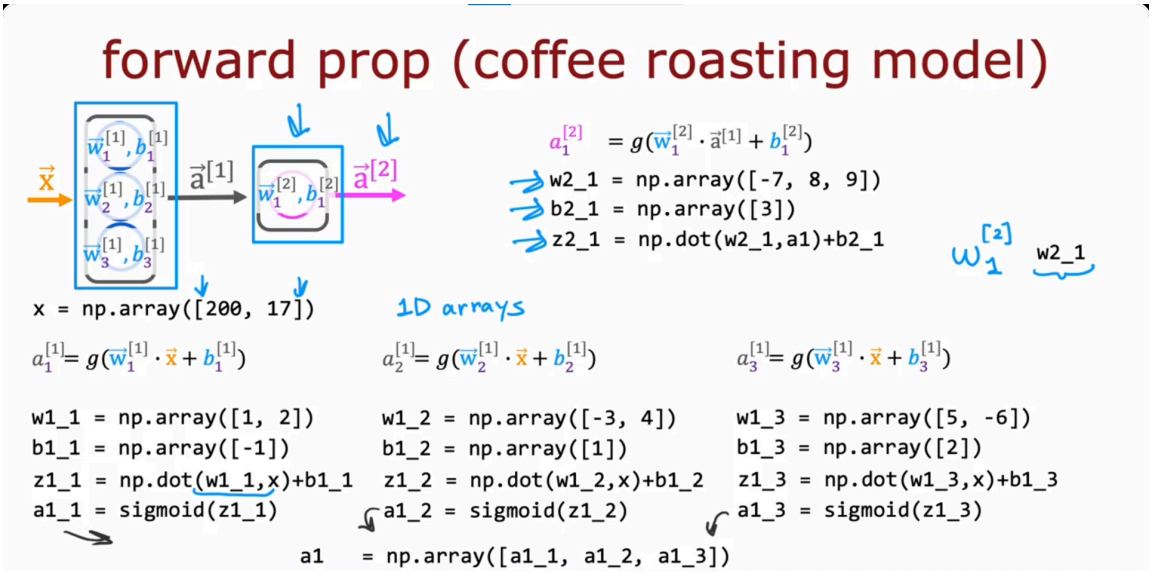



```
layer1 = Dense(units=3, activation='sigmoid')
layer1 = Dense(units=1, activation='sigmoid')
model = Sequential([layer1, layer2]) #connects 2 layers such that input flows from layer1 to layer2
x = np.array([[200.0, 17.0],
              [120.0, 5.0],
              [425.0, 20.0],
              [212.0,  18.0],
              ])
y = np.array([1, 0, 0, 1])
model.compile(...)
model.fit(x, y)
```

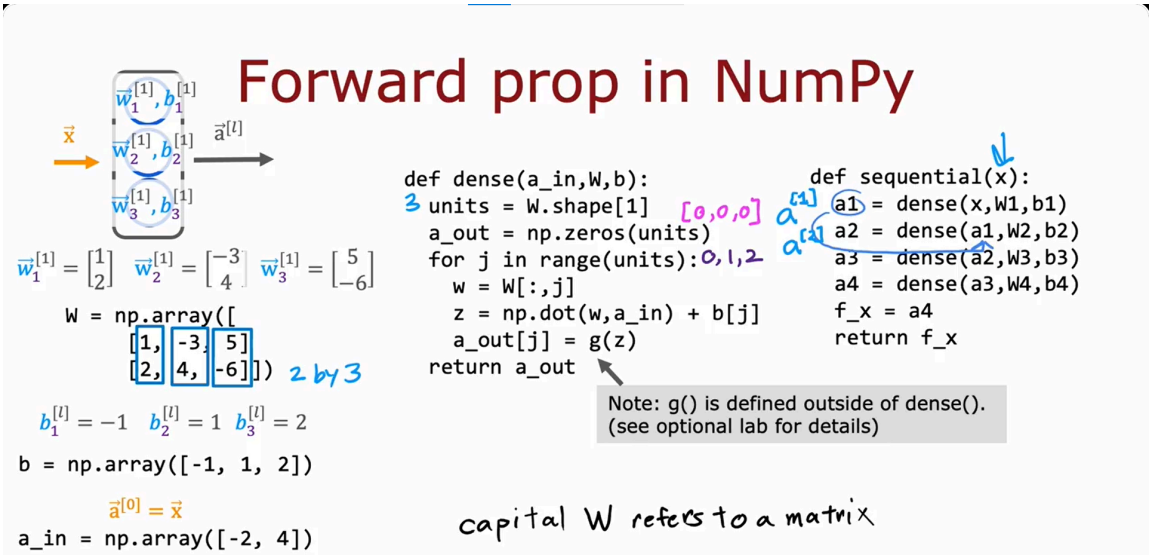
Alternative way to implement the same model architecture

```
model = Sequential([
  layer1 = Dense(units=3, activation='sigmoid'),
  layer1 = Dense(units=1, activation='sigmoid'),
])
```

Froward Propagation From Scratch



Implementing Froward Propagation Function Using Numpy



In [2]:
```
import numpy as np
def dense(a_in, W, b):
    units = W.shape[1] #getting number of cols from the matrix W (r x c)
    a_out = np.zeros(units) #creating an array of zeros with the same number of uni
    for j in range(units):
        w = W[:, j]
        z = np.dot(w, a_in) + b[j] #dot product of w and a_in plus the bias
        a_out[j] = g(z) #applying the activation function g to z
    return a_out #returning the output of the layer
```

Types of AI(Artificial Intelligence):

1. **ANI(Artificial Narrow Intelligence):** Refers to use of AI in particular field to narrow down its usecase such as Smart Speaker, Self-Driving Car, Web Search Bot etc.

2. **AGI(Artificial General Intelligence):** Refers to AI system that can do anything like human does.

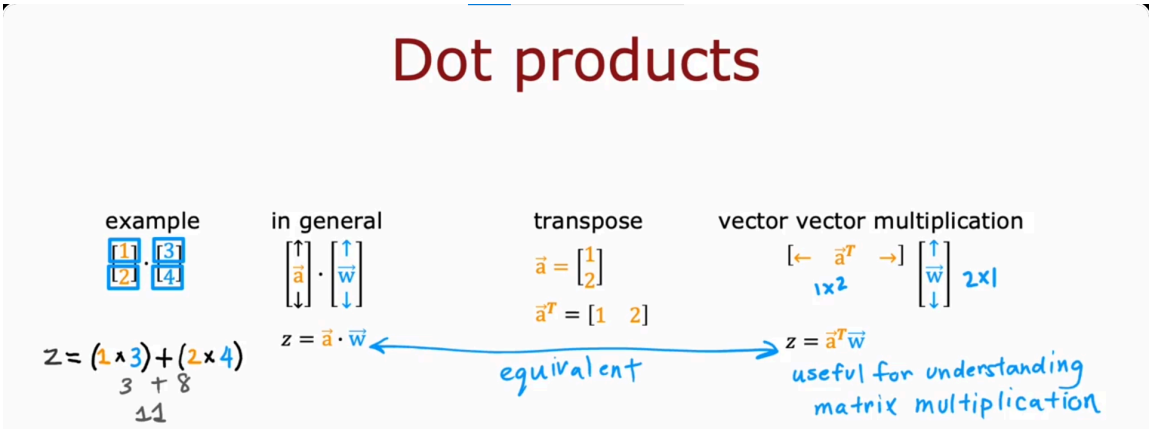
Implementing Froward Propagation Function Using Vector Multiplication

Matrix multiplication is quite efficient compared to numpy arrays dot product. It can use efficient computation utilizing parallel processing of GPU.

In [3]:
```
X = np.array([[200, 17]])
W = np.array([[1, -3, 5],
              [-2, 4, -6],
              ])
B = np.array([[1, 2, 3],
              ])
def dense(A_in, W, B):
    Z = np.matmul(A_in, W) + B
    A_out = g(Z)
    return A_out
```

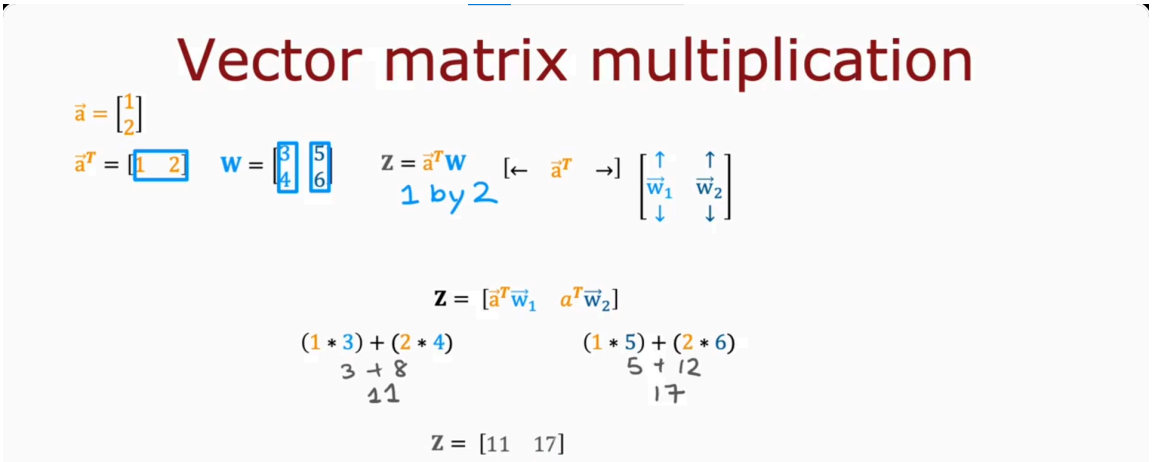
Dot Product

Dot product between 2 vectors can be can calculated as following. But the same calculation can be performed efficiently using matrix multiplication. For, matrix multiplication we need to transpose one of the matrices to multiply it with another as due to natural rule of matrix multiplication.



Vector Matrix Multiplication

To perform vector matrix multiplication we need to transpose matrix a so that, the rows of matrix a equals to become the matrix w. So, before transpose $\vec{a}_{(2*1)}$ has 2 rows and 1 column. But, W has 2 rows and 2 columns. In order, to multiply them the column of $\vec{a}$ needs to similar to the row of matrix w. So, we transpose $\vec{a}$ and then its dimension become (1*2). Now, the column of $\vec{a}$ is similar to the row of matrix W.

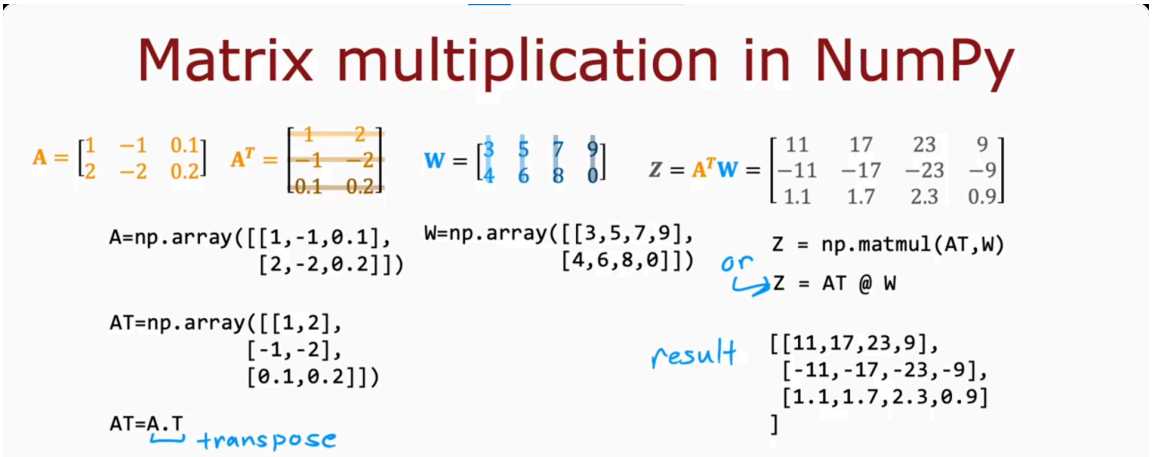


Matrix Multiplication in numpy

```
A = np.array([[1, -1, 0.1],
              [2, -2, 0.2],
              ])
AT = np.array([[1, 2],
               [-1, -2],
               [0.1, 0.2],
               ])
W = np.array([[3, 5 7, 9],
              [4, 6, 8, 0],
             ])
Z = np.matmul(AT, W) alternative Z = AT @ W
```

Dense Layer Function using Vectorized Form

```
def dense(AT, W, b):
    z = np.matmul(W, AT) + b
    a_out = g(z)
    return a_out
```



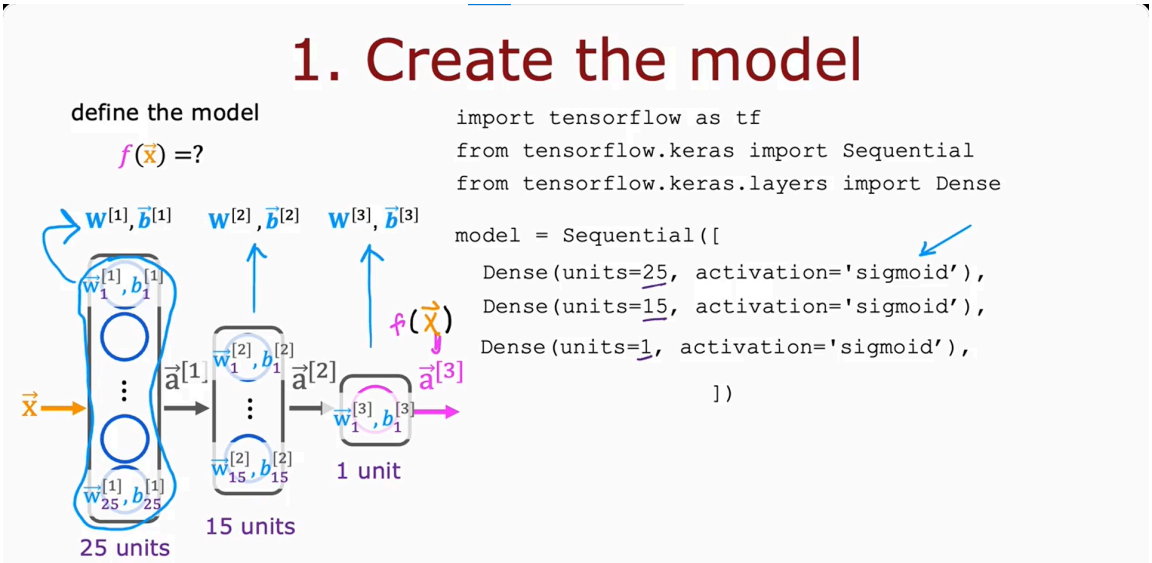
Model Training Steps Using Tensorflow

1. Create the model
2. Setting up the Loss and Cost Functions

Creating Model Using Tensorflow

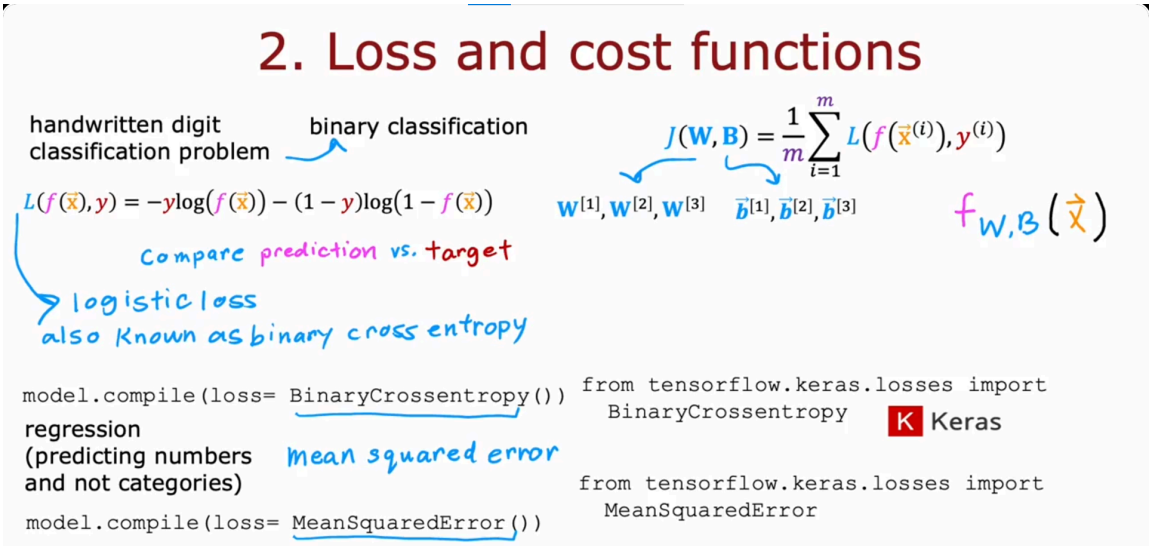
```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(units=25, activation='sigmoid'),
    Dense(units=15, activation='sigmoid'),
    Dense(units=1, activation='sigmoid'),
])
```



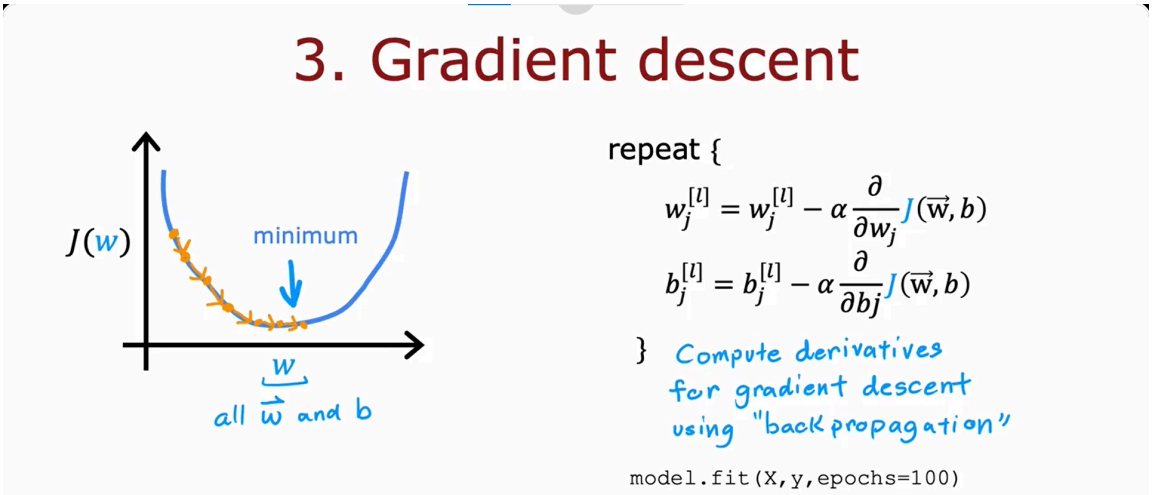
Setting up the Loss Function

```
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.losses import MeanSquaredError
model.compile(loss=BinaryCrossentropy()) #for binary
classification/Logistic Regression
model.compile(loss=MeanSquaredError()) #for Linear Regression
```



Cost Function and Iteration

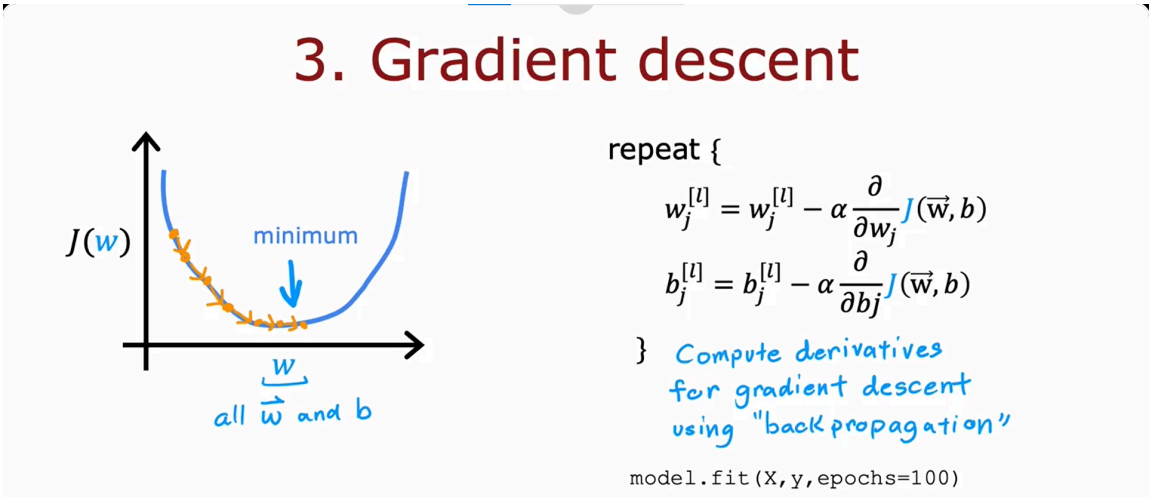
```
model.fit(X, y, epochs=100)
```



Alternative to Sigmoid for Activation

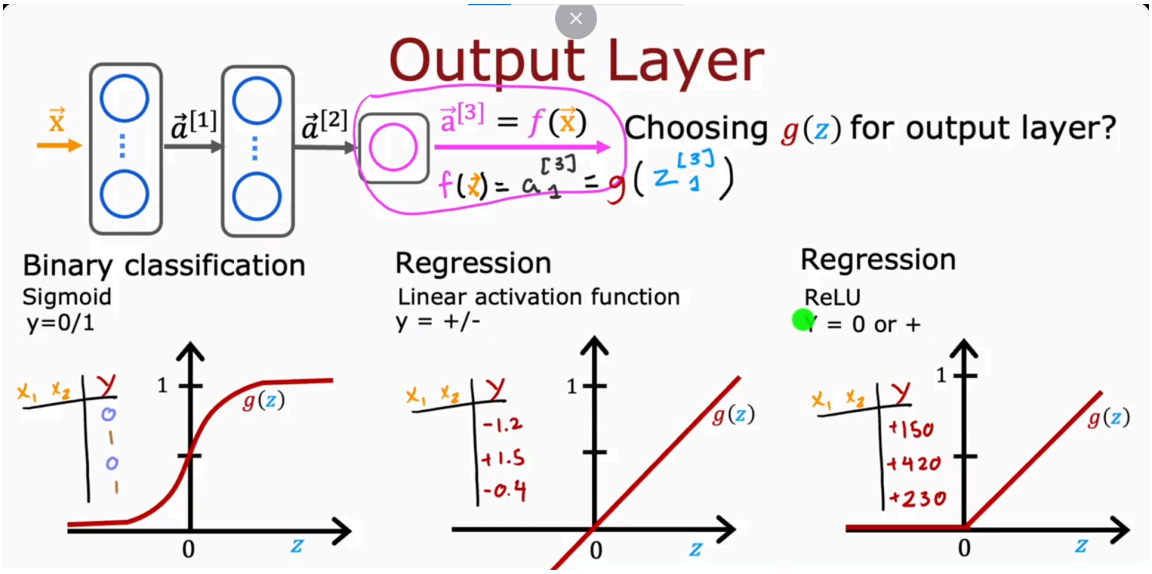
In Deep Learning or Neural Network most the activation function that is widely used is ReLu(Linear Rectified Unit). ReLu takes input and gives output 0 or input. ReLu = g(z) = max(0, z) Behavior of ReLu:

- If $z < 0$, then g(z) = 0
- If $z \geq 0$ then g(z) = z



Choice of Activation Function For Output Layer:

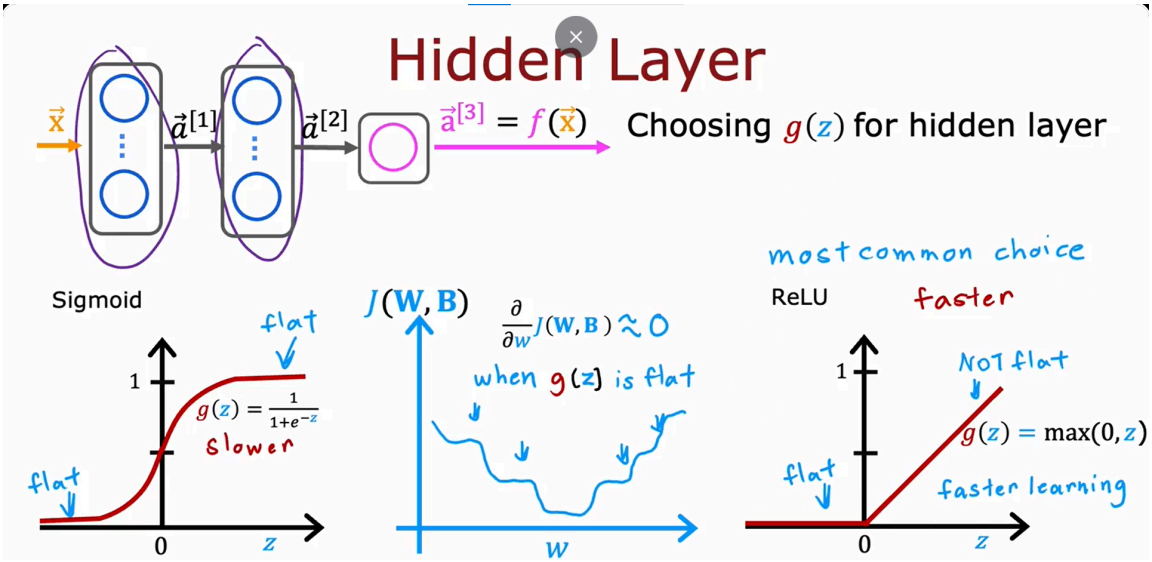
- For Binary Classification the output will be 0/1. So, we will use Sigmoid Function which is $\frac{1}{1+e^{-x}}$
- For regression problem that predict both (+/-) values, we will use, Linear Activation which is $y = wx + b$
- For regression problem that can only predict (+) values, we will use, ReLu which is $f(x) = max(0, x)$



Choice of Activation Function For Hidden Layer

- Sigmoid: As we have seen sigmoid in commonly used activation function.
- ReLu: It is mostly used as the activation function for hidden layer.
  However, ReLu is a bit faster in terms of computation. As it has no logarithmic calculation like sigmoid. Also, sigmoid function is flat 2 places at the very left of x axis and upper corner of y axis. So, it is difficult for gradient descent algorithm to find the

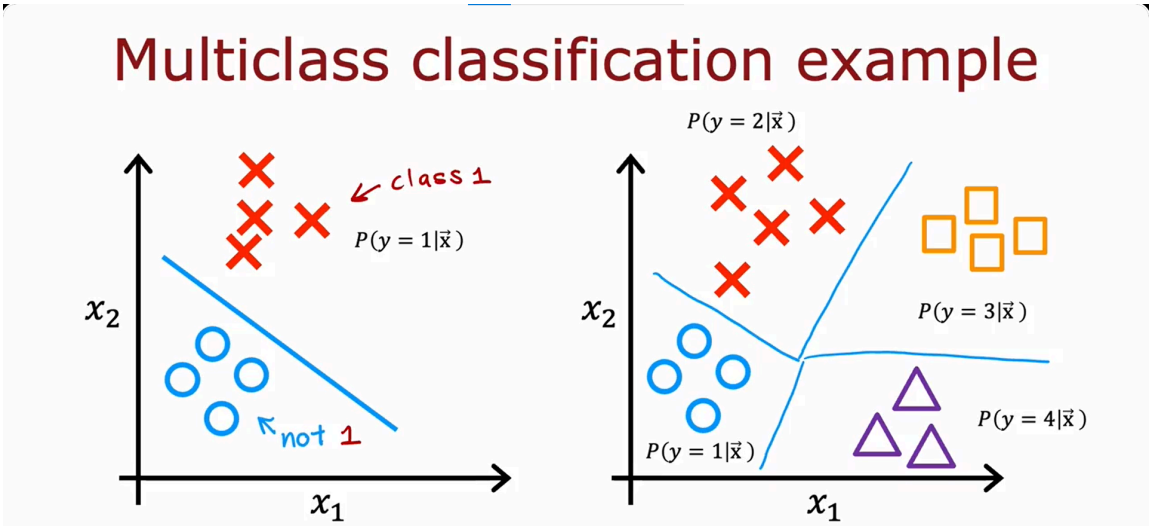
minima and the process of convergence become slow. So, ReLu becomes the most common choice as an activation function for hidden layer.



Why Do We Need activation

Without activation function neural network simply works as a regression model which fits a line. So, the main idea behind introducing activation function is to introduce the non-linearity to our model. Thus, activation function is a most essential part for a neural network to learn the non-linear trend or pattern in the dataset.

Multiclass Classification using Neural Network



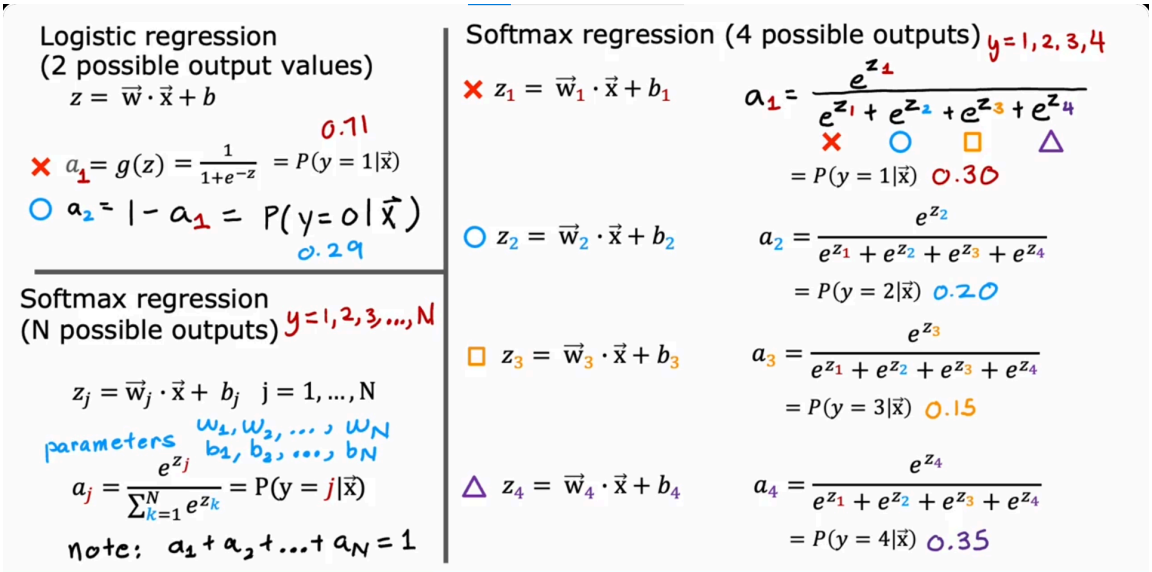
Softmax For Multiclass Classification

Softmax Function can be expressed as following:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, K$$

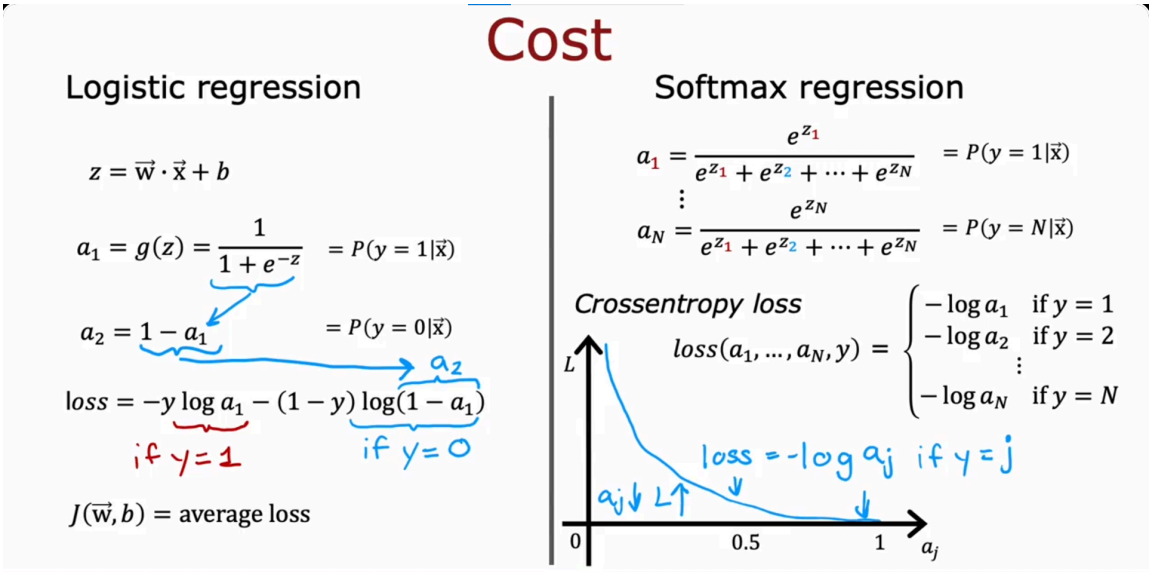
Advantage of Softmax:

- The output of Softmax function is always positive
- Sum of the output probabilities for all classes are sum up to 1.
- It is differentiable which makes easy for Gradient Descent algorithm to converge faster.
- The output of the softmax function can be interpreted as probability that provides a clear way to measure the confidence of the model in it's prediction.



Cost Function for Multiclass Classification

For multiclass classification problem we will use Sparse Categorical Cross Entropy as our loss function.



Logits in Neural Network

Logits refers to the raw output before passing it to the activation function. Passing the logits value to the activation function give us the output in terms of probability. Activation function converts the logits to meaningful probabilities. Each logits correspond to a class score.

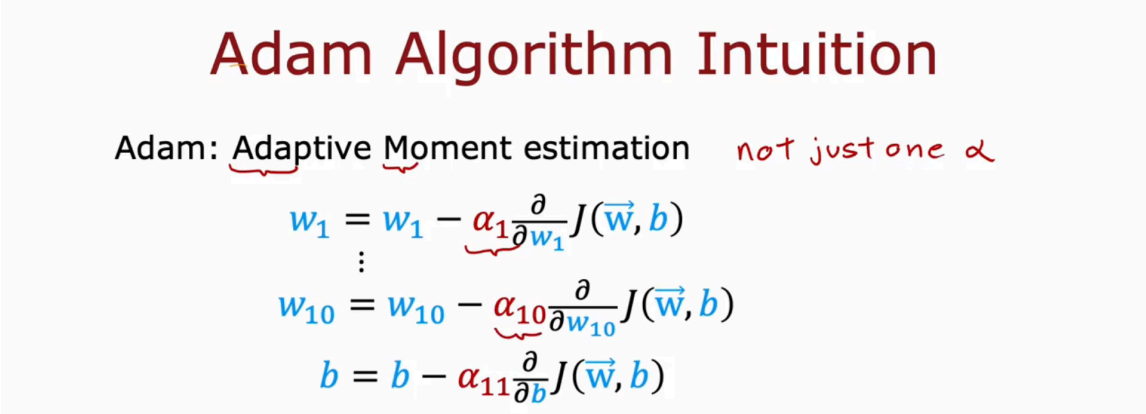
Why use Logits?
Applying softmax directly to the model output can lead to numerical issue or instability, especially when working with low and high values. But only getting logits and applying softmax when required ensure stable and accurate learning and inference.

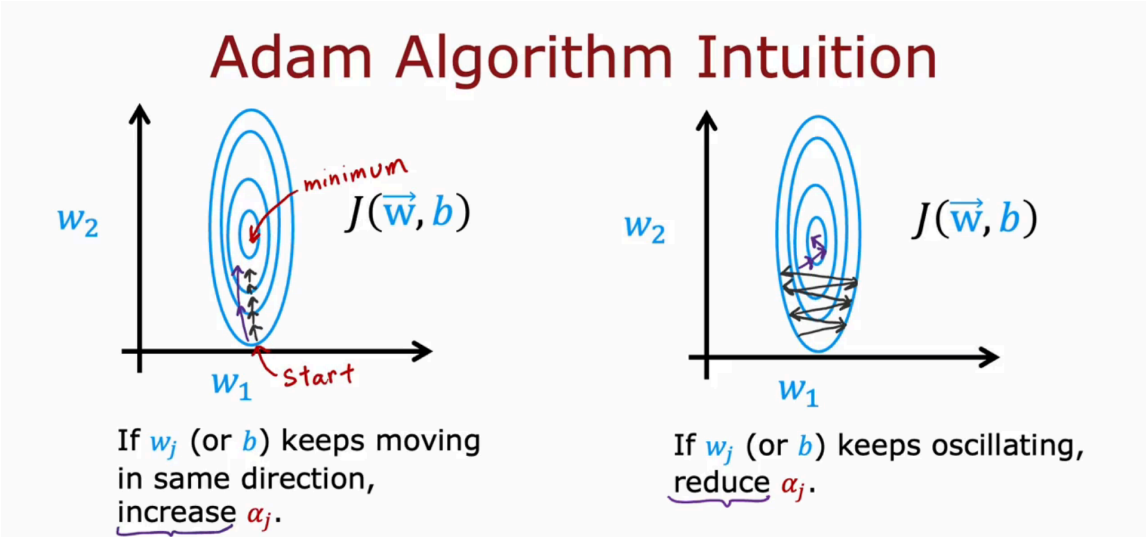
Advance Optimization over Gradient Descent

As we know gradient descent algorithm works with fixed learning rate. So, if the algorithm find the right tracks that minimizes the cost function or in other words it walks in the way of convergence a small learning rate will take a long time to converge.

So, Adam (Adaptive Moment Estimation) can solve this problem. Based on the context of how the cost function is minimizing it can adjust it's learning rate. Instead of using a constant learning rate, it uses different learning rate for all the parameters of the cost function. For example, if we have 10 weights and 1 bias. Then adam will have 11 different learning rates for all these variables.

If w_j keeps moving in the same direction increase α_j
If w_j keeps oscillating, then reduce α_j





Tensorflow Implementation of Adam Optimizer

model

```
model = Sequential([
        tf.keras.layers.Dense(units=25, activation='sigmoid'),
        tf.keras.layers.Dense(units=15, activation='sigmoid'),
        tf.keras.layers.Dense(units=10, activation='linear')
])
```

compile

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

fit

```
model.fit(X,Y,epochs=100)
```

Evaluating Model Performance

Linear Regression: To evaluate the performance of a regression model. we can split the data into train(70%) and test(30%) set. We will train our model on train set and evaluate the performance of the model on test. So, from the test score we will be able to identify the model performance on unseen dataset.

Classification: To evaluate performance of classification model we have 2 approach

- Logistic Loss: For binary classification problem we can get the performance of the model from average logistic loss on train and test set.
- Misclassification Rate: prediction can be written as, $\hat{y} = \begin{cases} 1, & \text{if } f(x) \geq 0.5 \\ 0, & \text{if } f(x) < 0.5 \end{cases}$

Now, from J_test and J_train we can get the fraction of test examples that was misclassified.

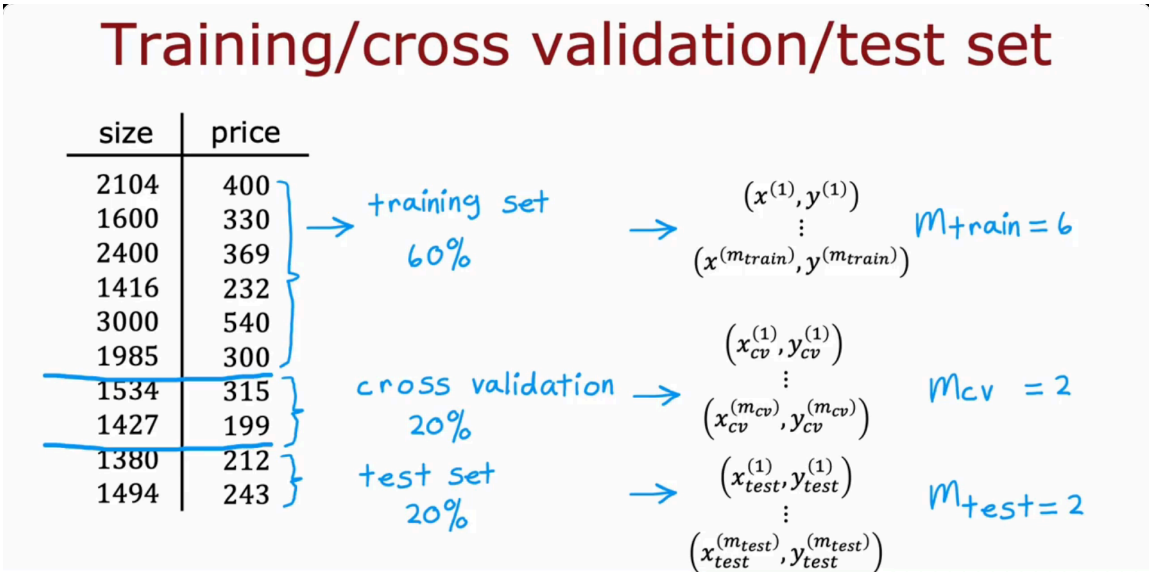
Model Selection Using Cross-Validation

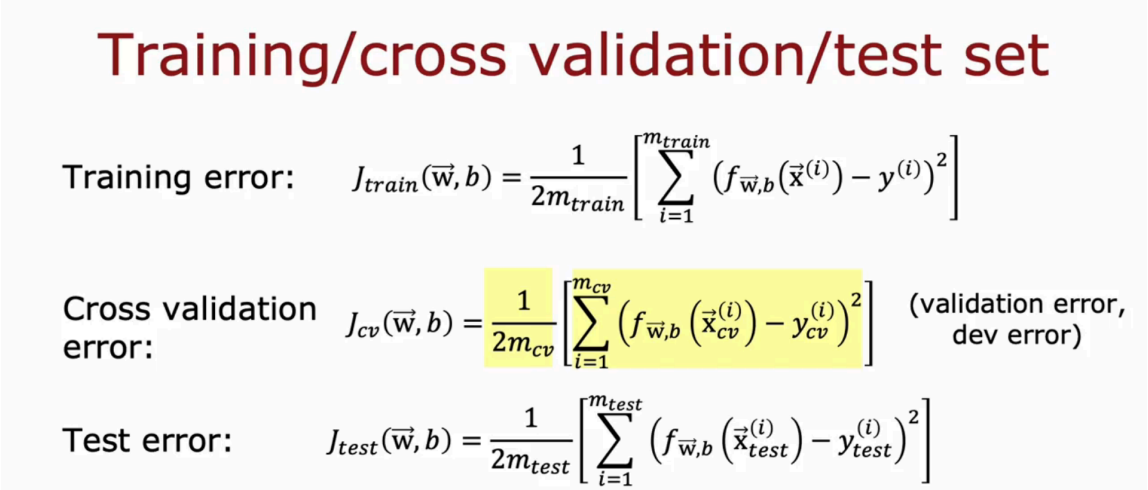
So, while selecting model we have many options. For example, if we have 10 different models for predicting house price, then we need to test each of them to see which one of them performs well. So, here we can use cross-validation score to see which model performs better on our dataset.

For cross-validation we need to split the data into 3 sets

- Training Set
- Cross-Validation Set
- Test

Selected model is trained on training and tested on the cv set. So, depending on the cv errors we can select the best model among the available options. After that, we can use the test set that was never given to the model. From the test set score, we will come to a conclusion how the model generalizes on unseen data.





Bias-Variance Problem

High Bias (Underfitting)

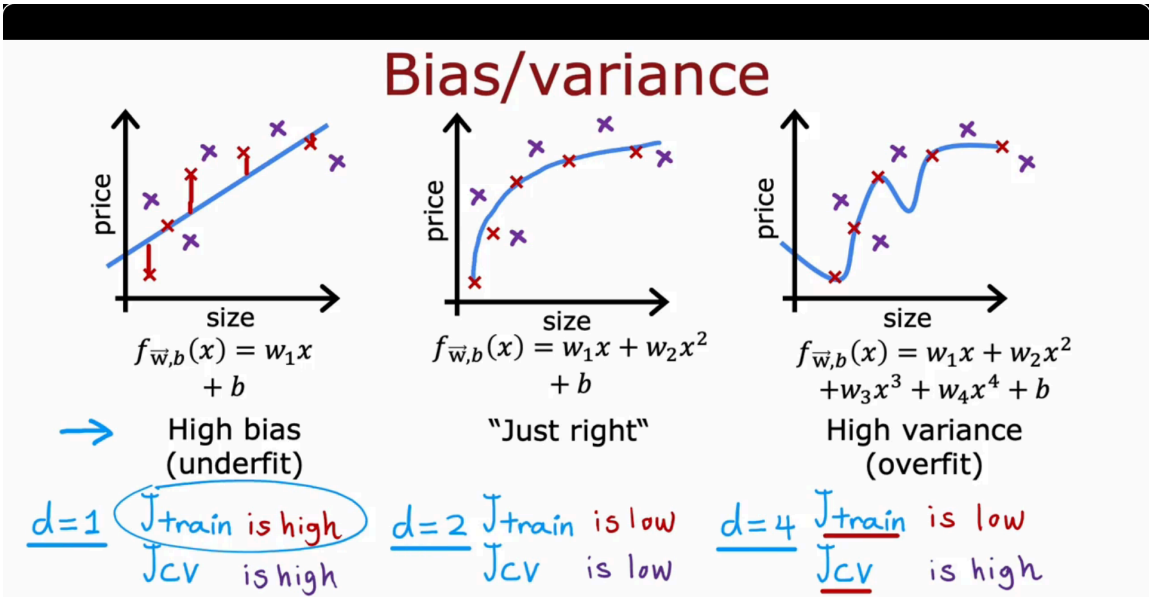
- performs poorly both training and test set
- both J_train and J_cv is high
- happens when model is too simple to catch the non-linear trend

High Variance (Overfitting)

- model performs very well on the training set but poorly on unseen data.
- J_train is low, but J_cv is much higher.
- Happens when model is too complex and memorizes the training data.

Generalizes Well

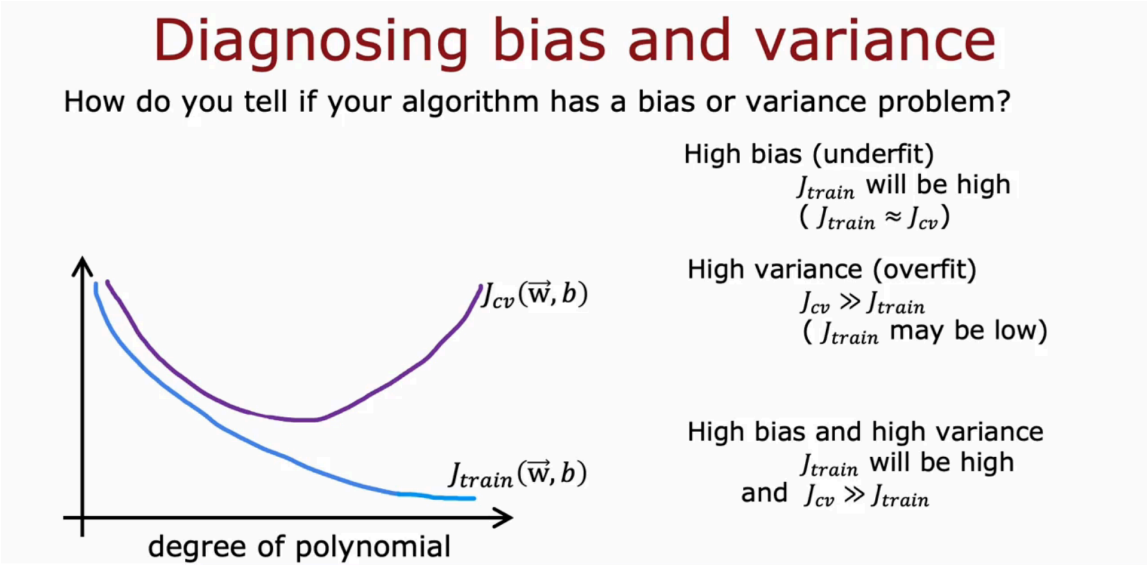
- Both J_train and J_cv are low and close in value.
- Indicates that the model generalizes well.



Diagnosing Bias & Variance

From J_train and J_cv we can identify the bias-variance problem of a model as following:

- If J_train is high → High Bias.
- If J_train is low but J_cv >> J_train → High Variance.
- If both are low and close → Good generalization.
- both J_train & J_cv is very much high → High Bias + High Variance

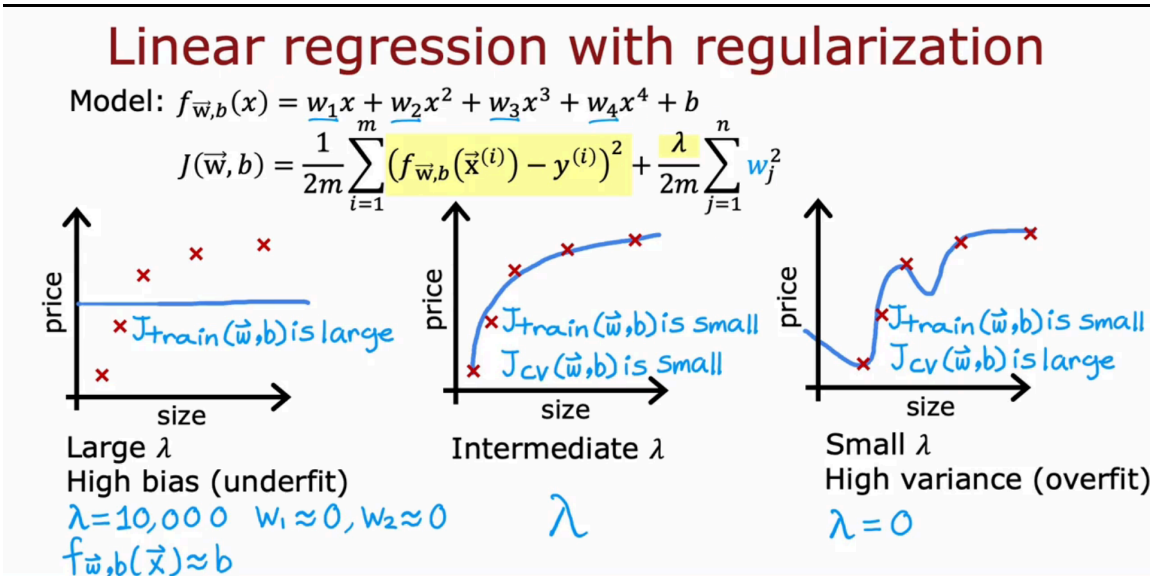


Regularization and Bias-Variance

Regularization solves the problem of bias-variance. The value of λ decide the strength of regularization.

Cost Function with regularization, $j(w)$ = Training Error + λ · Regularization Term

- If λ is too high model tends to under fit and suffers from high bias.
- If λ is too low model tends to overfit and suffers from high variance.
- If λ value is something in between low and high model generalizes well.



Behavior at Different λ Values

🔴 λ = Very Large (e.g., 10,000):

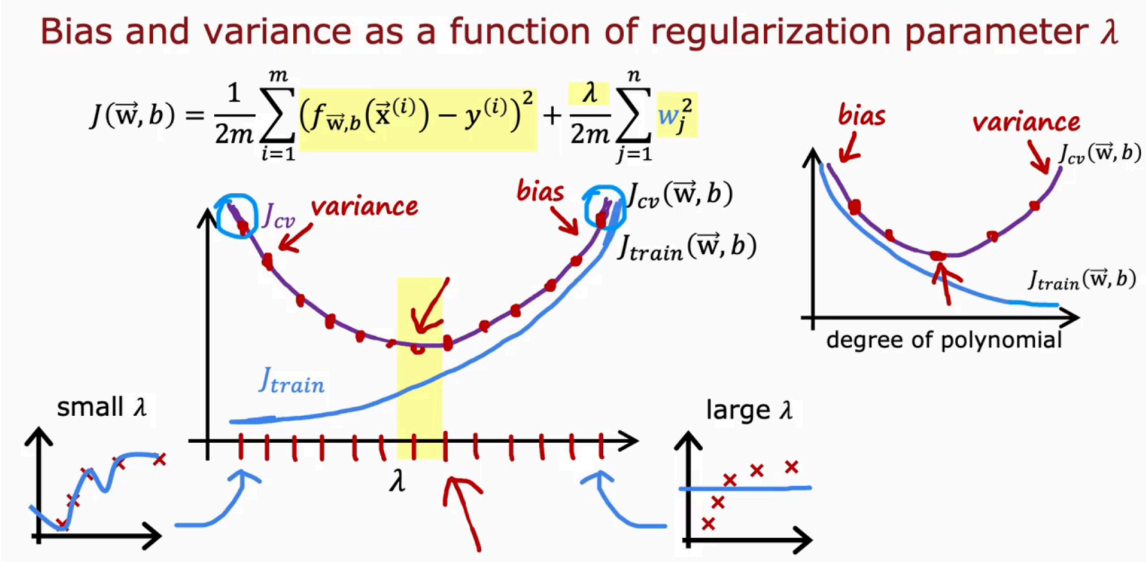
- Model heavily penalizes large weights → pushes weights close to zero.
- Output becomes a flat line (almost a constant).
- High bias → Underfits training data.
- Both training error (J_train) and cross-validation error (J_cv) are high.

🟢 λ = 0 (No regularization):

- Model tries to fit training data perfectly → results in overfitting.
- Low J_train, but high J_cv (bad generalization).
- High variance.

🟡 λ = Moderate (Just Right):

- Achieves a balance between underfitting and overfitting.
- Both J_train and J_cv are low.
- This is the desired sweet spot → the model generalizes well.



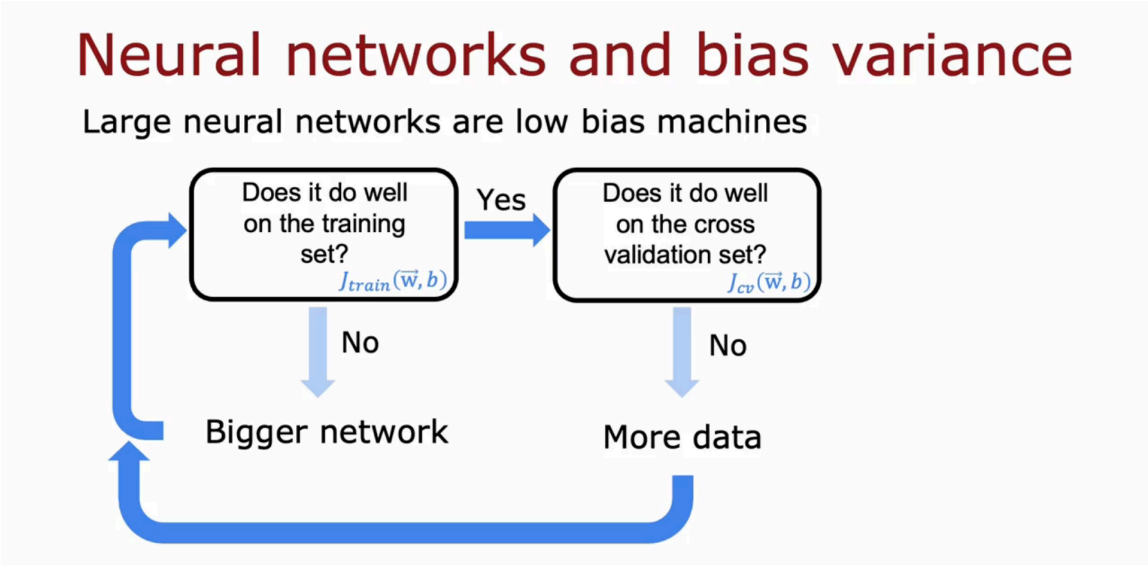
Choosing the Best λ Using Cross-Validation

- Try several values of λ (e.g. 0.01, 0.002, ... 10)
- For each λ:
  - Train model and get weights (w, b)
  - Compute cross-validation error J_cv (w, b)
- Choose λ that gives lowest J_cv

Debugging a Learning Algorithm

Technique	Problem	How it improves
Get more training example	High Variance	Helps reducing overfitting by exposing model to more example
Try small sets of feature	High Variance	Reducees model complexity and limits overfitting
Add additional features	High Bias	Gives model more information to better captures complex patterns
Add polynomial features	High Bias	Increase model flexibility to fit more complex functions
Decrease Regularizations	High Bias	Reduces penality on model complexity to better fit training data
Increase Regularizations	High Variance	Increase penalty on complexity to prevents overfitting

Bias-Variance Tradeoff in Neural Network



Traditional Bias-Variance Tradeoff

- High Bias: Simple models (linear regression) fail to capture data complexity → underfitting.
- High Variance: Complex models (high-degree polynomials) capture noise → overfitting.
- Traditional ML focused on balancing bias and variance, often using:
  - Model complexity (degree of polynomial)
  - Regularization parameters (lambda(λ))

Practical Recipe for Training Neural Networks

1. Train the model and evaluate on the training set:
    - If training error is high → High Bias
        - Increase model size (more layers or units)
        - Train longer
2. Once training error is low, check cross-validation (CV) error:
    - If CV error is high → High Variance
        - Collect more data
        - Use regularization (L2, dropout, etc.)

Applying Regularization on Neural Network Models

Neural network regularization

$$J(\mathbf{W},\mathbf{B}) = \frac{1}{m}\sum_{i=1}^{m} L\left(f\left(\vec{\mathbf{x}}^{(i)}\right), y^{(i)}\right) + \frac{\lambda}{2m}\sum_{\text{all weights } \mathbf{W}} \left(w^2\right)$$

Unregularized MNIST model

```
layer_1 = Dense(units=25, activation="relu")
layer_2 = Dense(units=15, activation="relu")
layer_3 = Dense(units=1, activation="sigmoid")
model = Sequential([layer_1, layer_2, layer_3])
```

Regularized MNIST model

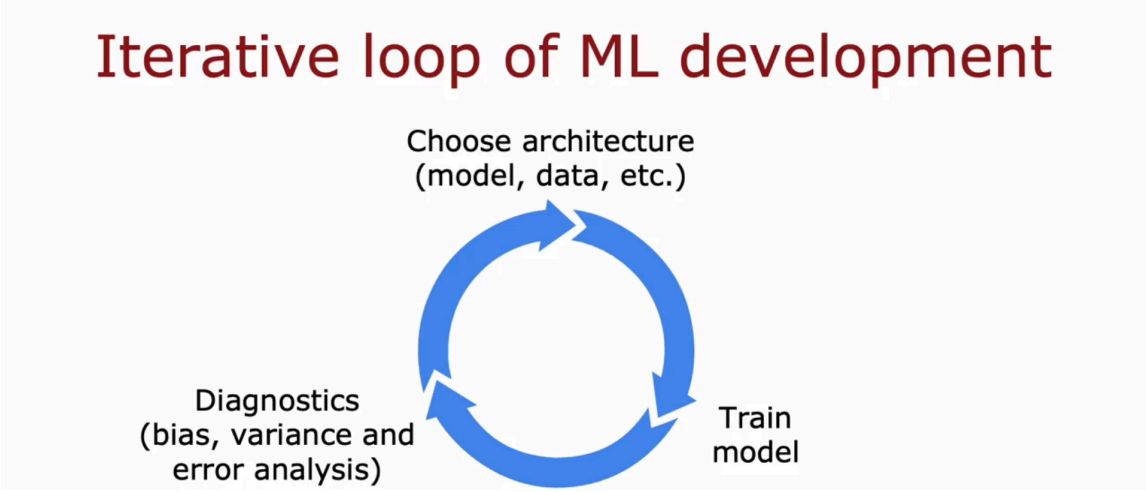
```
layer_1 = Dense(units=25, activation="relu", kernel_regularizer=L2(0.01))
layer_2 = Dense(units=15, activation="relu", kernel_regularizer=L2(0.01))
layer_3 = Dense(units=1, activation="sigmoid", kernel_regularizer=L2(0.01))
model = Sequential([layer_1, layer_2, layer_3])
```

Iterative Loop of ML Development

1. Initial Architecture Decision:
    - Choose the ML model (e.g., logistic regression, neural network).
    - Decide on input features and hyperparameter.
2. Model Training:
    - Train the model on labeled data.

- The first trained model rarely performs optimally.

3. Diagnostics:
    - Analyze errors using bias/variance and error analysis (explained in the next video).
    - Use these insights to guide next steps.
4. Iteration:
    - Modify the architecture or data (add features, adjust regularization, gather more data).
    - Repeat the loop to improve model performance.



Error Analysis

Error analysis is the 2nd most important thing after vias-variance trade off. It can help to identify promising ares that needs improvement.

Benefits of Error Analysis:

- Helps you prioritize what to improve based on:
    - Frequency of error type.
    - Potential impact of fixing that category.
- May inspire:
    - Feature engineering (e.g., drug names, suspicious URLs).
    - Targeted data collection (e.g., more pharmaceutical spam or phishing emails).
- Efficient even when dataset is large:

    - If misclassified examples are many (e.g., 1000 out of 5000), sample and analyze a subset (e.g., 100–200).

Adding Data

Data Augmentation

- Create new training sample bby applying transformation to existing examples

For Images:

- Rotate, scale, warp, mirror(if appropriate)

For Audio:

- Add background, noise (crowd, car)
- Simulate poor recording condition

However, we need to remember that augmentation should mimic real-world situation in the test set. Avoid unrealistic conditions.

Transfer Learning

It a technique where a model trained on one task or dataset is reused on a second task. It is useful when we don't have much data for any specific problem.

How it Works

Training on Large Dataset

- Model trained on large dataset learn useful features like edges, corners, curve etc
- This type of common feature improves model performance that can be utilized on other tasks

Fine Tuning

- Replace the last layer of the model with a new one based on the no of classes for our problem
    - Option 1: Freeze the earlier layer except the classification layer and only train the final layer
    - Option 2: Fine tune the entire model starting from the pre-trained weights

Limitations

- Input data type must match(like image size)

- Need domain specific model for each task

Performance Metrics in Imbalanced/Skewed Dataset

Accuracy is misleading in imbalanced dataset as there is class imbalance. So, to handle such situation we need new metrics such as Precision and Recall.

To calculate precision and recall across different class we need to plot the confusion matrix of the classification dataset.

Notation & Details:

True Positive(TP): The model correctly predicted the positive class as positive. Example: Model predict spam and the mail is actually spam.

False Positive(FP): The model incorrectly predicted the negative class as positive class. Example: Model predicts spam but the mail is not spam.

False Negative(FN): The model incorrectly predicted negative class. Example: Model predicted not spam but it was actually spam.

True Negative(TN): The model correctly predicted negative class. Example: Model says not spam and the mail is not spam.

		Predicted condition	
	Total population = P + N	Positive (PP)	Negative (PN)
Actual condition	Positive (P)	True positive (TP)	False negative (FN)
	Negative (N)	False positive (FP)	True negative (TN)

Metrics for Imbalanced Dataset

		Predicted Class	
		Spam	Not Spam
Actual Class	Spam (P)	TP = 80	FN = 20
	Not Spam (N)	FP = 10	TN = 90

Accuracy

The number of correct predictions that was correct and predicted as correct by the model. This indicates the number of class that was classified correctly.

Example: Accuracy = $\frac{TP+TN}{TP+TN+FP+FN}$ = $\frac{80+90}{80+10+20+90}$ = 85%

Precision

It indicates among all emails predicted as spam, how many of them were actually spam. It is also known as positive predicted value.

Example: Precision = $\frac{TP}{TP+FP}$ = $\frac{80}{80+10}$ = 88.9%

Out of 90 emails predicted as spam, 80 of them were actually spam.

So, precision tells how well the model predicts the positive class. Higher precision means model has few false positive rates. If classifying the positive class is crucial then, precision score is a critical for understanding model performance.

Precision measures how accurate your model's spam predictions are.

- High precision means few non-spam emails are wrongly flagged as spam (low false positives).
- If avoiding false alarms (e.g., not sending real emails to spam) is important, then precision becomes crucial.

Recall

It indicates of all actual spam emails, how many of them were correctly classified as spam. It is also known a true positive rate.

Example: Recall = $\frac{TP}{TP+FN}$ = $\frac{80}{80+20}$ = 80%

Out of 100 actual spam emails, the model only caught 80.

Recall measures how well your model captures all actual spam emails

- High recall means the model misses few spam emails (low false negatives).
- If detecting all spam is very important (e.g., phishing or scams), then recall is a critical metric.

F1 Score

It refers as the harmonic mean of precision and recall. It provides a single, balanced metric when we can't afford to optimize one of the metrics. It helps to find a average model that performs well both in terms of precision and recall.

Trade of between Precision & Recall

If predicting positive is costly (e.g., invasive treatment):

- Raise the threshold (e.g., to 0.7 or 0.9).
- Predict 1 only if very confident.
- increase 🔼 precision, decreases 🔽 Recall.

If missing positives is dangerous (e.g., untreated serious illness):

- Lower the threshold (e.g., to 0.3 or 0.1).
- Predict 1 even with mild suspicion.
- increase 🔼 Recall ,decrease 🔽 precision.

Decision Tree Algorithm

Decision 1: Which Feature to Split on?

- Goal is to maximize purity of resulting subset
- Choose feature that best separate classes

Decision 2: When to stop splitting?

Stop if:

- Node is pure
- Max depth reach
- Gain in purity is too small
- Too few examples at a node

Entropy in Decision Tree

What is Entropy?

- Entropy is a measure of the impurity (or disorder) in a set of labeled examples.
- It quantifies how mixed the examples are with respect to their class labels (e.g., cats vs. dogs).

Key Concepts:

- If a set contains only one class (e.g., all cats or all dogs), it's pure, and entropy = 0.
- If the set is evenly mixed (e.g., 50% cats, 50% dogs), it's most impure, and entropy = 1.

Formula: Entropy = $-p_1 \log_2(p_1) - p_0 \log_2(p_0)$ = $\sum p_i \log 2p_i$

Information Gain = Entropy of a root -

One Hot Encoding

When one single features have more than one value, then it we need one hot encoding to convert those into multi-label categorical values into binary features.

If categorical feature can take k values then we need to create k binary features

Each binary features = 1 if the original value matches otherwise 0

One hot encoding works with decision tree algorithm, Neural network, Logistic Regression

Color
Red
Green
Blue

→

Color	Red	Green	Blue
Red	1	0	0
Green	0	1	0
Blue	0	0	1

In [ ]:
```
!jupyter nbconvert Documentation-Advanced-Learning-Algorithm-Imtiaz-Ahammed.ipynb -
```

In [ ]:
```

```