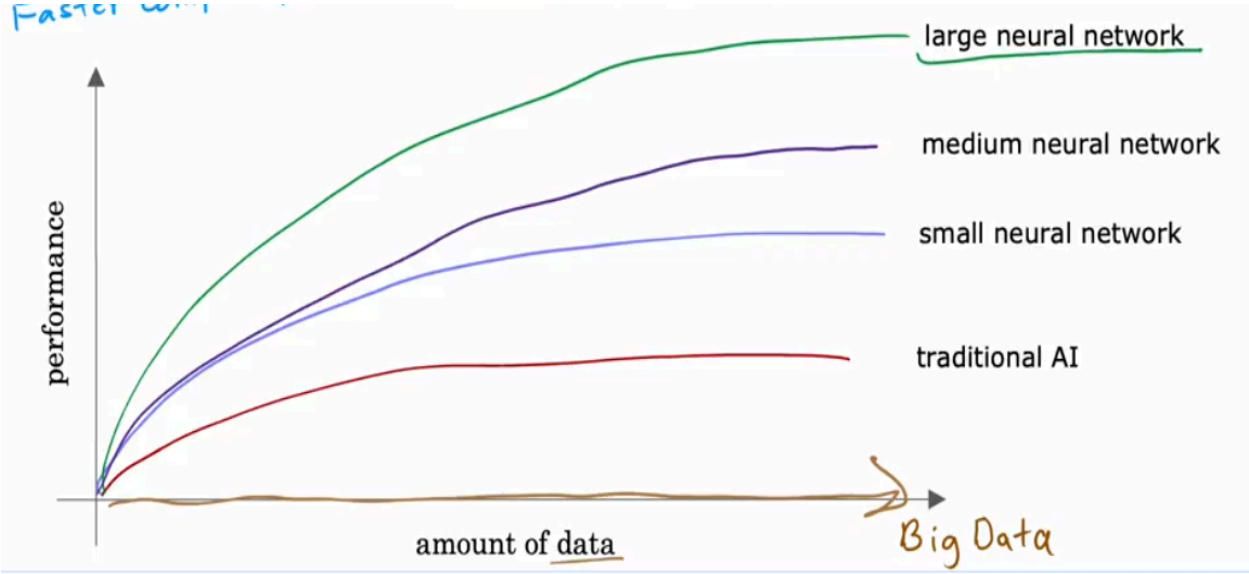
# Advanced Learning Algorithm

A **Neural Network** is a machine learning model inspired by the human brain, consisting of layers of interconnected neurons (or nodes). It's widely used in deep learning tasks such as image recognition, natural language processing, and forecasting.

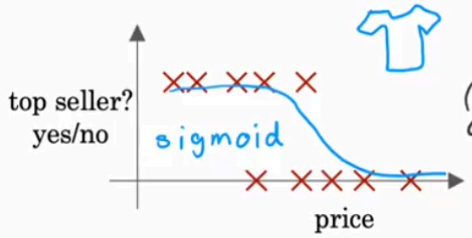



This image represents the relationship between **performance** and the **amount of data** for different types of AI models. The x-axis represents the **amount of data** (increasing towards "Big Data"), while the y-axis represents **performance** (e.g., accuracy, efficiency).

### Main Takeaways:

- **Larger neural networks benefit more from big data**, as they can capture more complex patterns.
- **Smaller networks plateau earlier**, meaning they cannot leverage additional data effectively.
- **Traditional AI does not benefit much from increased data**.
- **Deep learning is data-hungry**, meaning the more data available, the better it performs.

# Demand Prediction




$x = price$ input

$a = f(x) = \frac{1}{1+e^{-(wx+b)}}$ output

activation

# Demand Prediction


feature engineering

$x_1$ $x_2$

$\vec{x}$ $(x, y)$ "hidden layer"

layer ← can have multiple neurons

input layer

"activations"

price

shipping cost

marketing

material

affordability ←

awareness ←

perceived quality ←

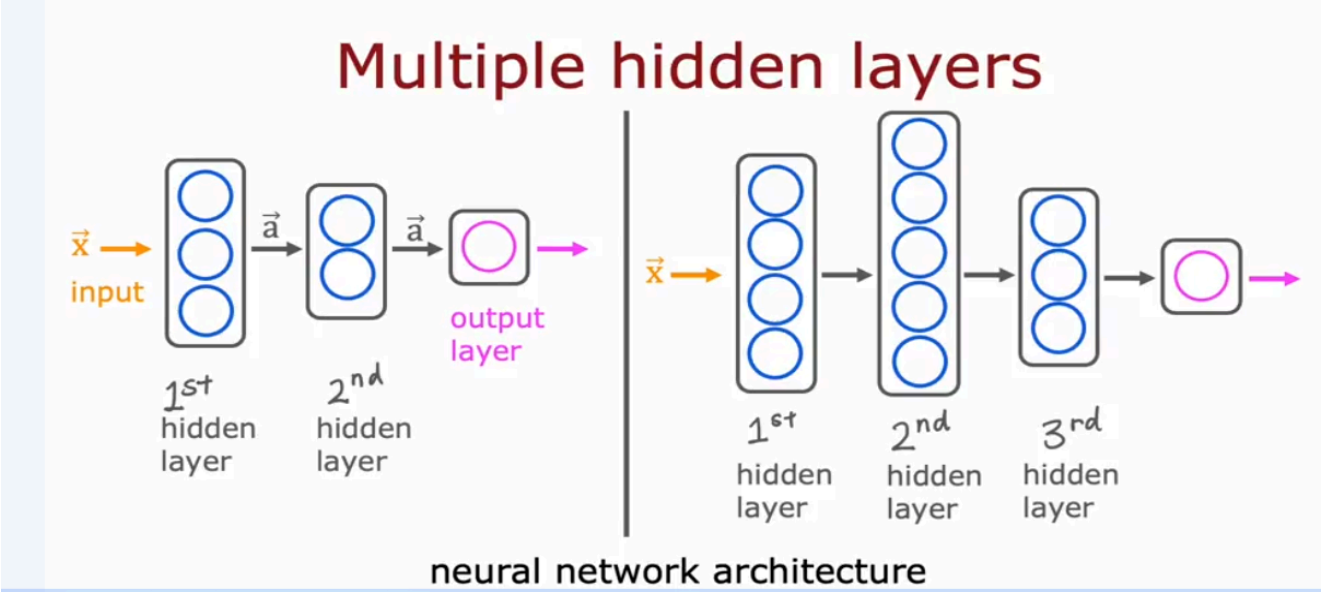
$\vec{a}$

layer ← output layer

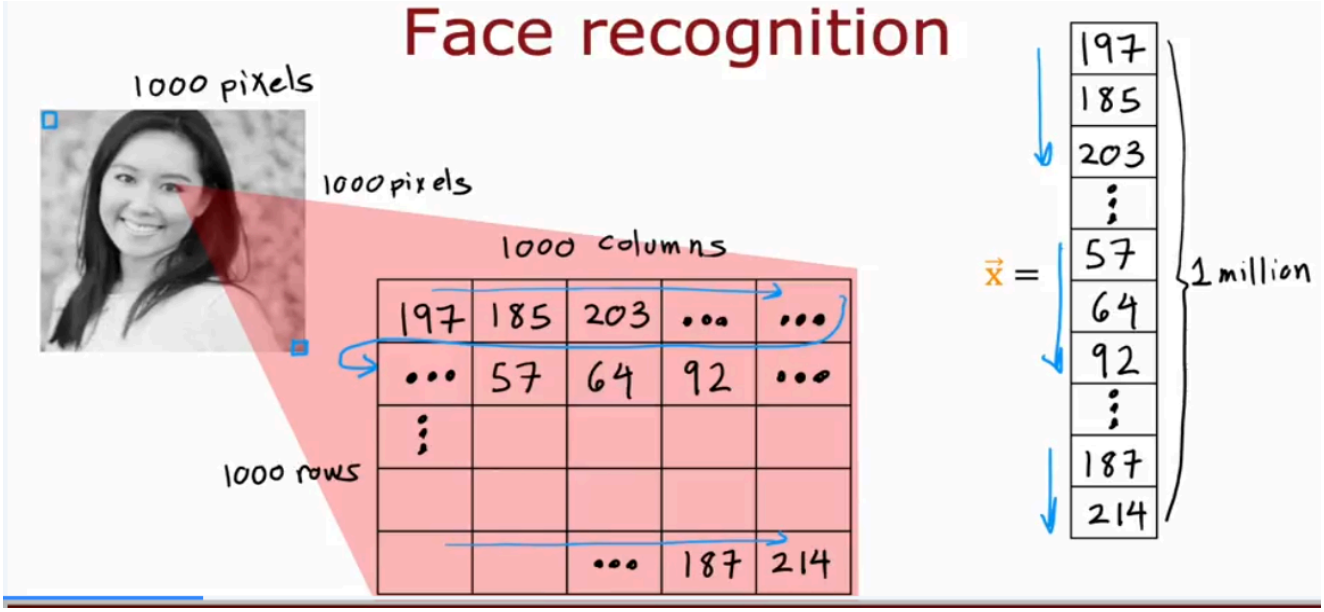
$a$ → probability of being a top seller

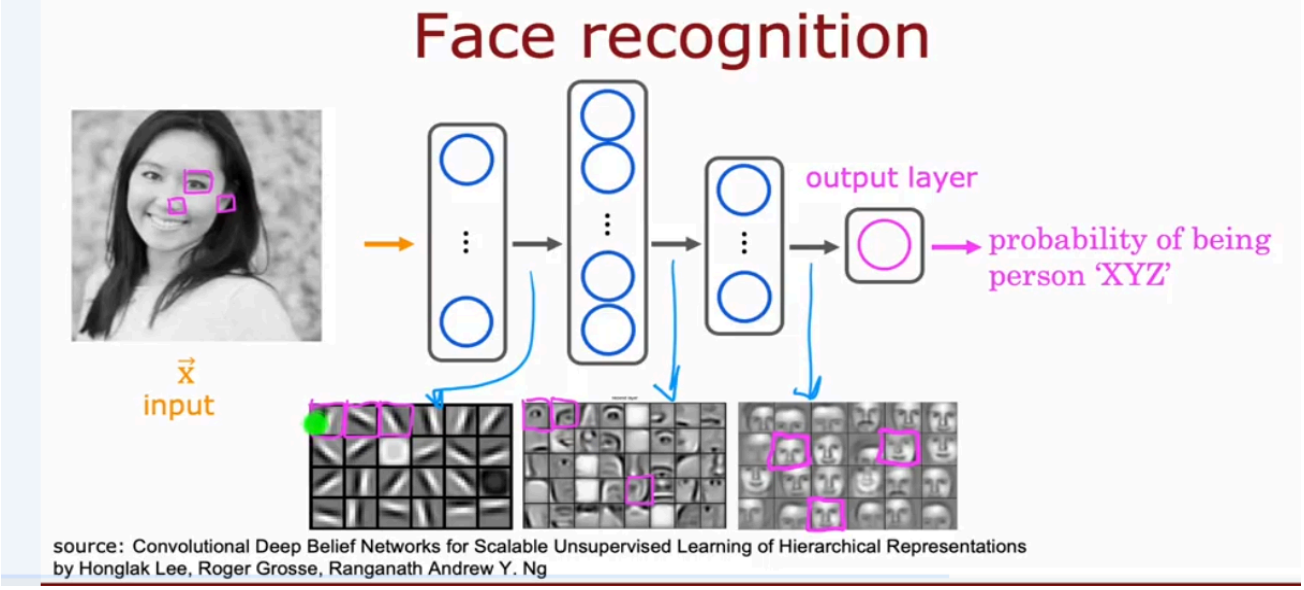
activation values

4 numbers 3 numbers 1 number




Recognizing image:

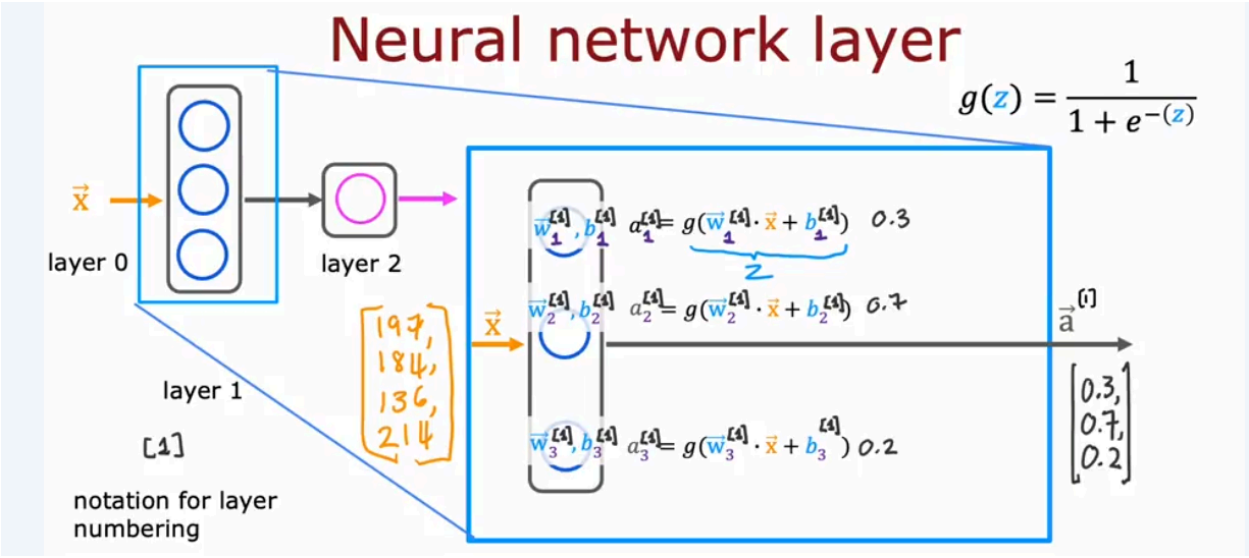




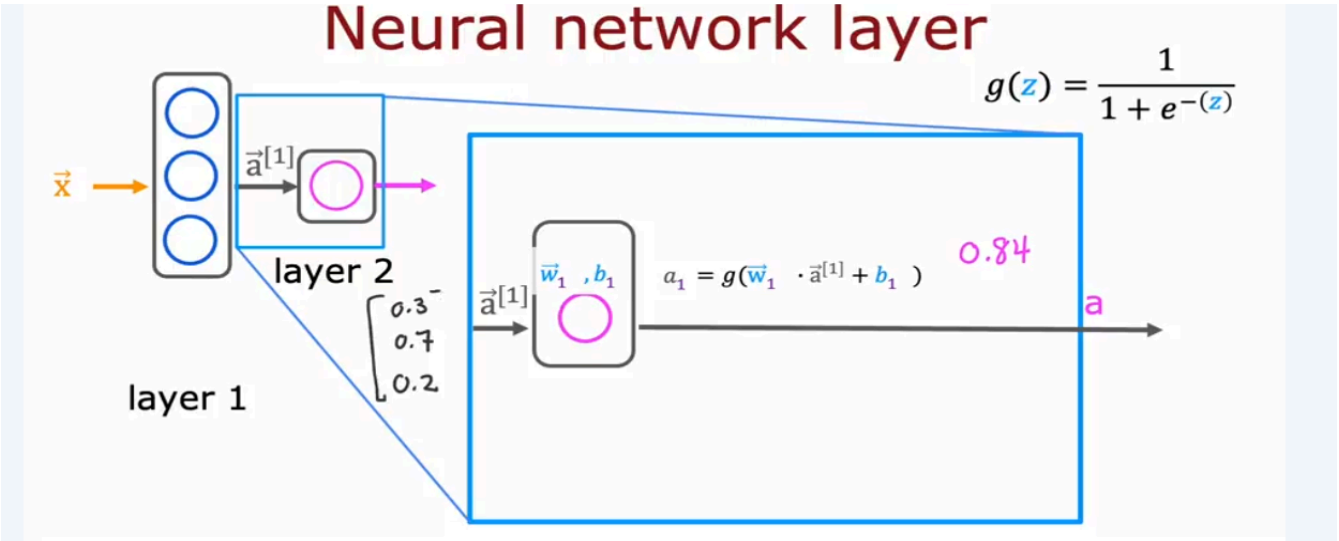

- **Hierarchical Feature Learning**: Lower layers detect simple patterns, while deeper layers recognize full faces.
- **Probabilistic Classification**: The model outputs a probability score for different identities.
- **Deep Learning-Based Approach**: Uses convolutional deep belief networks to process images efficiently.

Would you like a more detailed breakdown of how convolutional layers work in this context? 🚀

## Neural Network Layer:




This image explains the **neural network layers** and how a single layer processes inputs using weights, biases, and an activation function.



## Key Components in the Image

1. **Layers in the Neural Network**
   - **Layer 0**: Input layer (denoted as $\mathbf{x}$).
   - **Layer 1**: Hidden layer (blue box) that performs computations.
   - **Layer 2**: Output layer, providing the final prediction.
2. **Mathematical Representation of a Neuron**
   - Each neuron in **Layer 1** receives input $\mathbf{x}$.
   - Each neuron applies a weighted sum of the inputs:

$$z_i = w_i \cdot \mathbf{x} + b_i$$

   - The activation function **sigmoid** (shown at the top right) is applied:

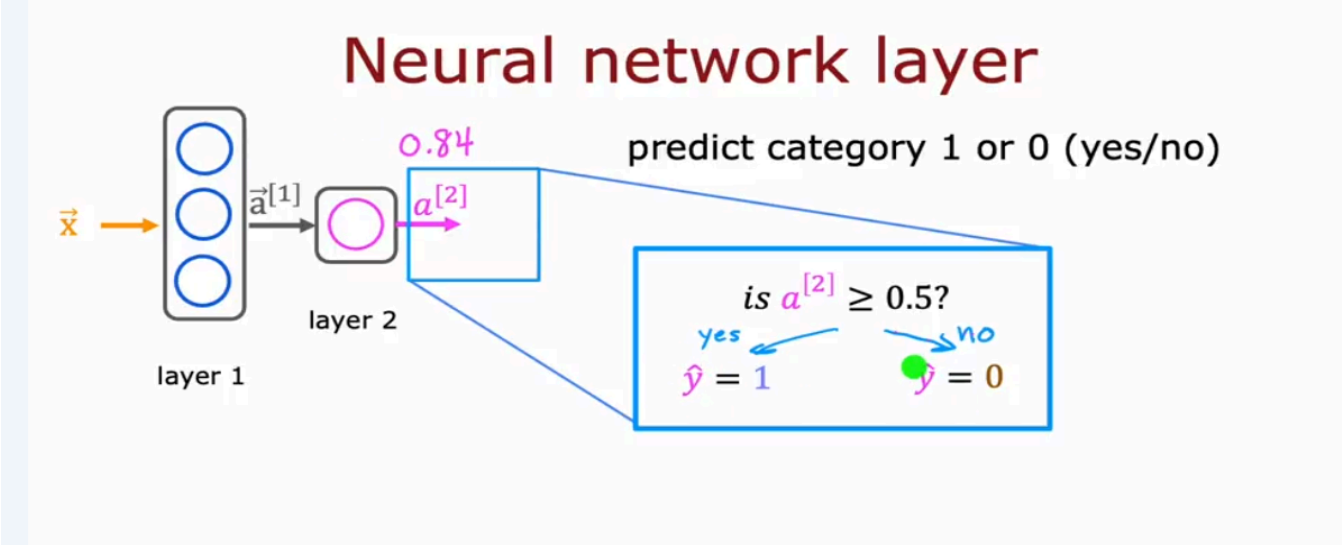
$$g(z) = \frac{1}{1 + e^{-z}}$$

   - The output of each neuron in Layer 1 is represented as $a_i^{(1)}$.

3. **Notations in the Zoomed-In Box**
   - $w_1^{[4]}$, $w_2^{[4]}$, $w_3^{[4]}$: Weights associated with different neurons.
   - $b_1^{[4]}$, $b_2^{[4]}$, $b_3^{[4]}$: Biases added to the weighted sum.
   - $a_1^{(4)}$, $a_2^{(4)}$, $a_3^{(4)}$: Outputs of neurons in Layer 1.
   - The final output is a **vector** $[0.3, 0.7, 0.2]$, which will be passed to the next layer.

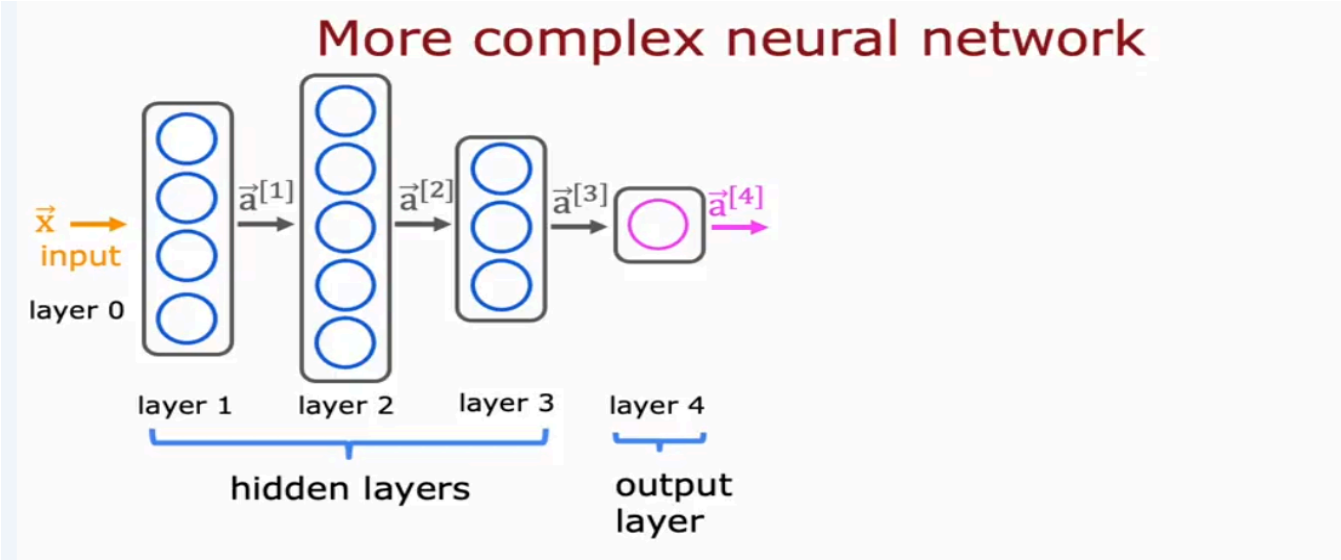
## Key Takeaways

- Each neuron takes inputs, applies a weighted sum, adds a bias, and passes it through an activation function.
- The **sigmoid function** helps in normalizing outputs between 0 and 1.
- The computed values $a^{(1)}$ form the activation outputs for Layer 1 and are passed forward to the next layer.

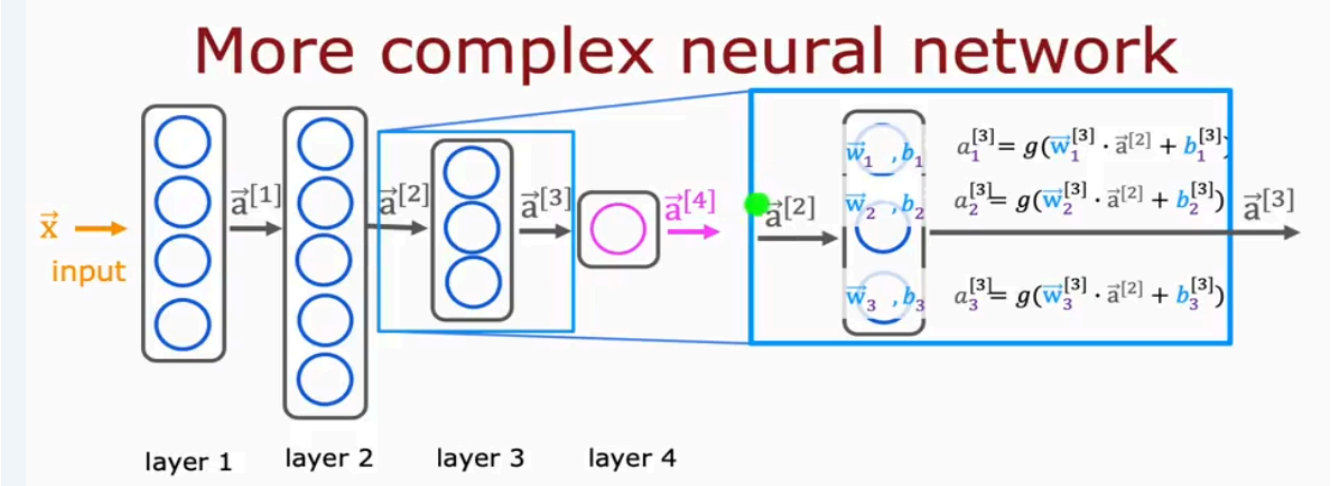



## More Complex Neural Network :

Neural Network with 4 layers.



Zoom in Layer 3




1. **Layers in the Neural Network**
   - **Layer 1**: Input layer where the data $\mathbf{x}$ enters.
   - **Layers 2 & 3**: Hidden layers where computations occur.
   - **Layer 4**: Output layer providing the final prediction.
2. **Notations and Data Flow**
   - $\mathbf{x}$ (input) is processed in **Layer 1**, producing activations $\mathbf{a}^{[1]}$.
   - The activations from **Layer 1** become the input to **Layer 2**, producing $\mathbf{a}^{[2]}$.
   - Similarly, activations propagate through **Layer 3** ($\mathbf{a}^{[3]}$) and finally reach the **output layer** ($\mathbf{a}^{[4]}$).

3. **Mathematical Representation of Layer Computation**
    - Each neuron in **Layer 3** computes:

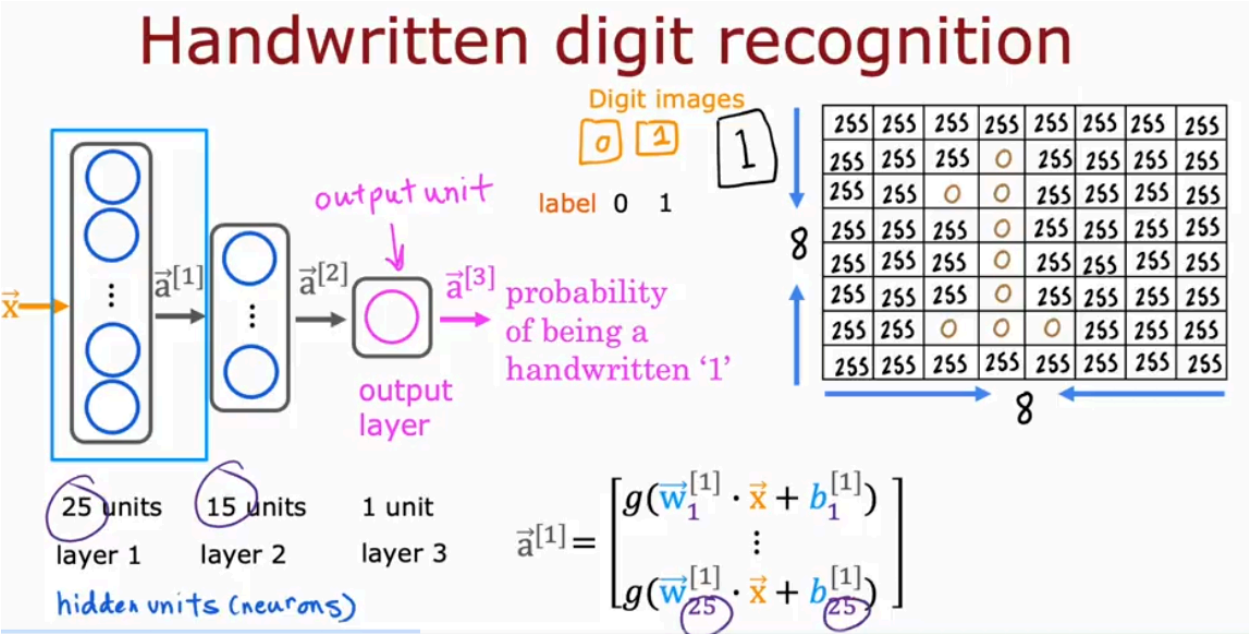
$$a_i^{[3]} = g\left(w_i^{[3]} \cdot a^{[2]} + b_i^{[3]}\right)$$

    - Here, $w$ represents **weights**, $b$ represents **biases**, and $g(\cdot)$ is the **activation function** (such as ReLU or Sigmoid).
    - This process continues for all neurons in the layer.
4. **Key Observations**
    - This network is **deeper**, meaning it has more layers, which allows for learning more complex patterns.
    - Each layer refines the features extracted from the previous layer.
    - The structure follows **forward propagation**, where data flows from the input to the output.

## Takeaways

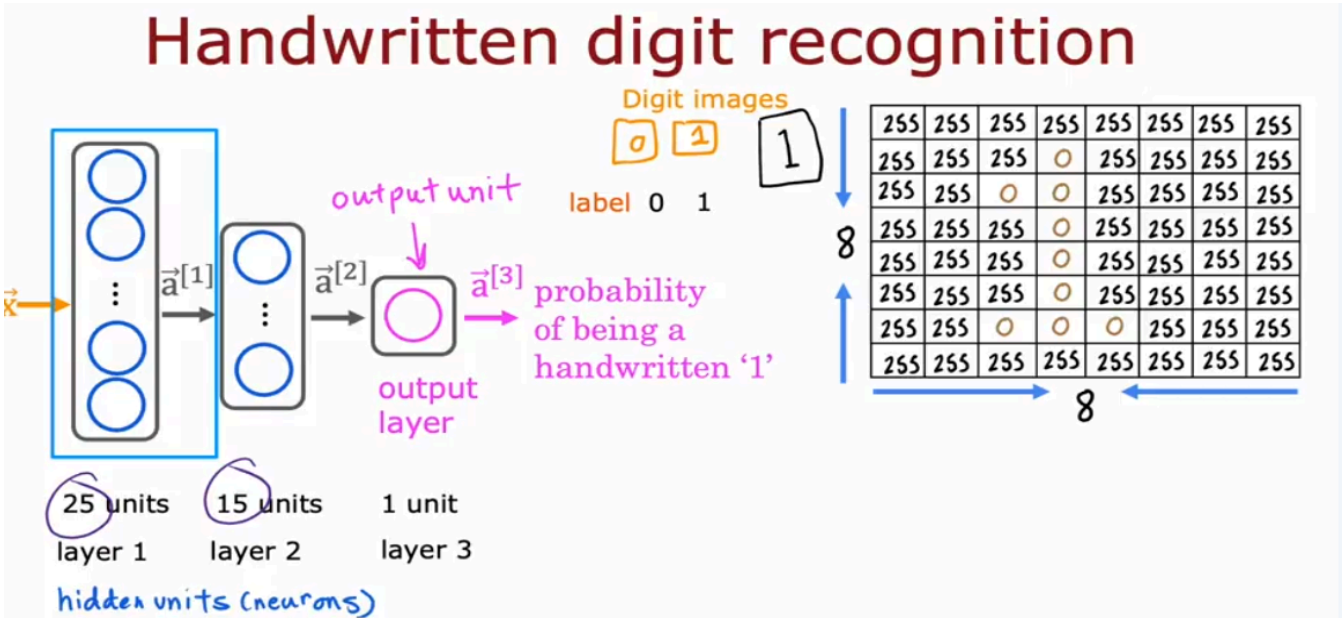
- The deeper the network, the more complex functions it can learn.
- The **activation function** $g(\cdot)$ plays a crucial role in transforming inputs at each layer.
- The **weights and biases** are the learnable parameters that adjust during training.

Inference making predictions ( Forward propagation) :




1. **Input Representation (Right Side)**
   - The rightmost part shows an **8×8 pixel grayscale image** where each pixel has an intensity (ranging from **0 to 255**).
   - The goal is to determine whether the digit is **0 or 1**.
2. **Neural Network Architecture (Left Side)**
   - The neural network consists of **three layers**:
     - **Layer 1 (Input Layer + First Hidden Layer)**:
       - Takes the **flattened 8×8 image** as input.
       - Has **25 neurons (units)** to learn patterns from the image.
     - **Layer 2 (Second Hidden Layer)**:
       - Has **15 neurons**, further refining the extracted features.
     - **Layer 3 (Output Layer)**:
       - Has **1 neuron** that outputs the **probability of the image being a '1'**.

Inference : Making Predictions(Forward Propagation) :




## Understanding the Image

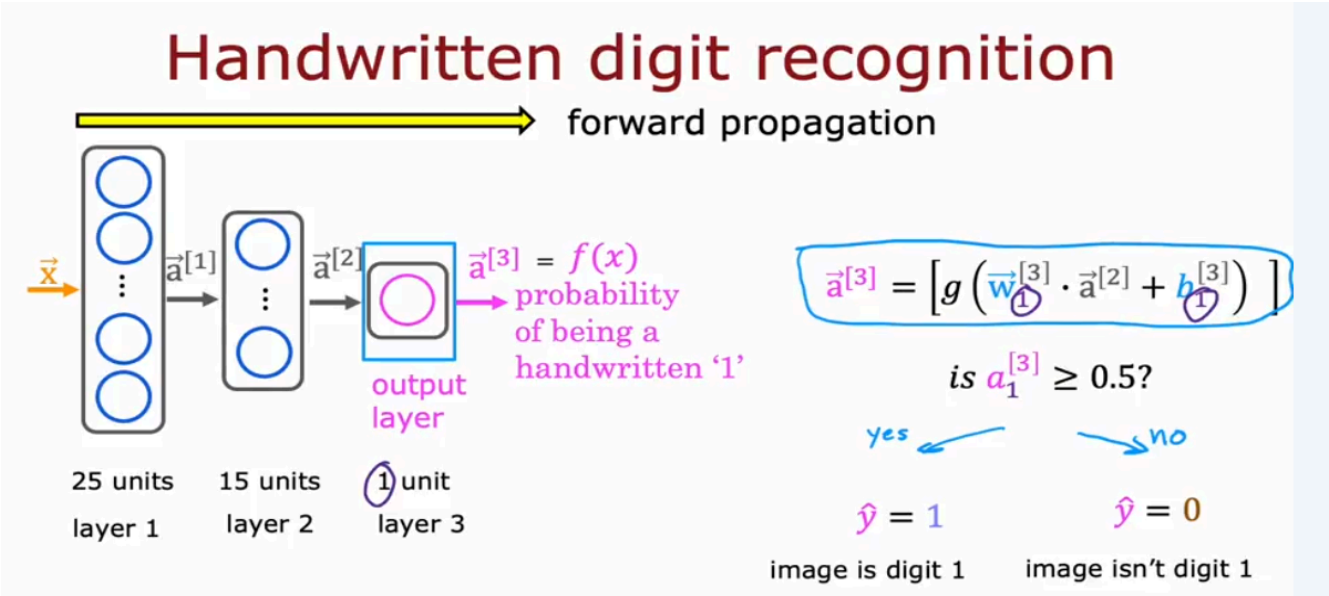
1. **Input Representation (Right Side)**
   - The rightmost part shows an **8×8 pixel grayscale image** where each pixel has an intensity value (ranging from **0 to 255**).
   - The goal is to determine whether the digit is **0 or 1**.
2. **Neural Network Architecture (Left Side)**
   - The neural network consists of **three layers**:
     - **Layer 1 (Input Layer + First Hidden Layer)**:
       - Takes the **flattened 8×8 image** as input.
       - Has **25 neurons (units)** to learn patterns from the image.
     - **Layer 2 (Second Hidden Layer)**:
       - Has **15 neurons**, further refining the extracted features.
     - **Layer 3 (Output Layer)**:
       - Has **1 neuron** that outputs the **pr ↓ bility of the image being a '1'**.

3. **Forward Propagation**
    - The image pixels (flattened into a vector $\mathbf{x}$) pass through multiple layers.
    - Each neuron applies a **weighted sum** of inputs, adds a bias, and applies an **activation function** (like Sigmoid or ReLU).
    - The final neuron gives $a^{[3]}$, which is the probability that the digit is '**1**'.
4. **Output & Classification**
    - If $a^{[3]}$ is **close to 1**, the network predicts **digit 1**.
    - If $a^{[3]}$ is **close to 0**, the network predicts **digit 0**.

---

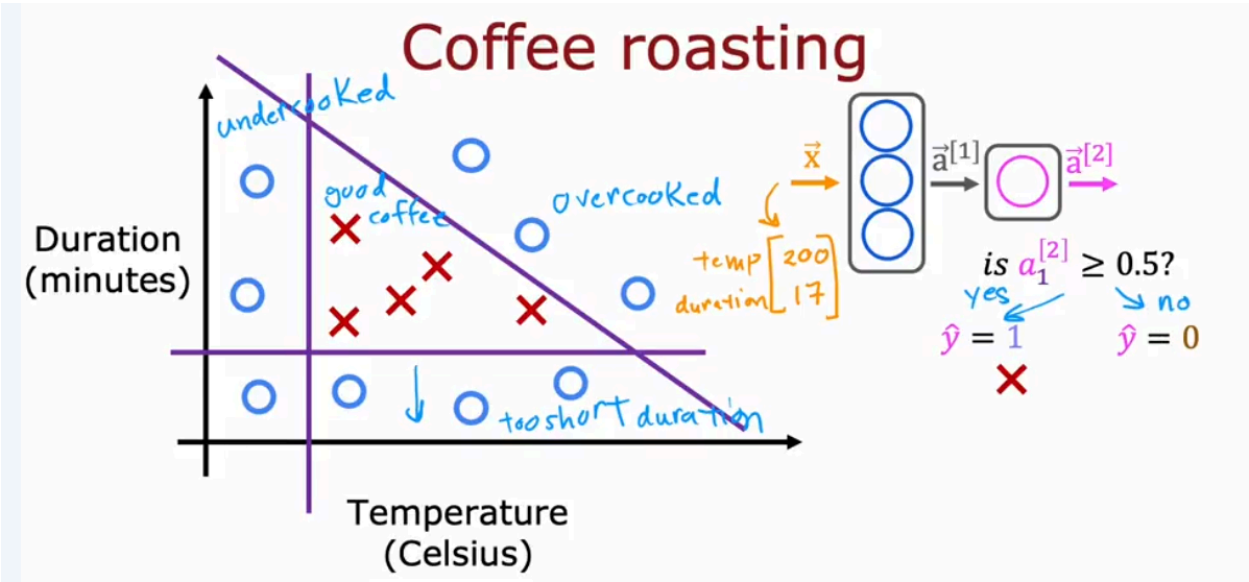
## Key Takeaways

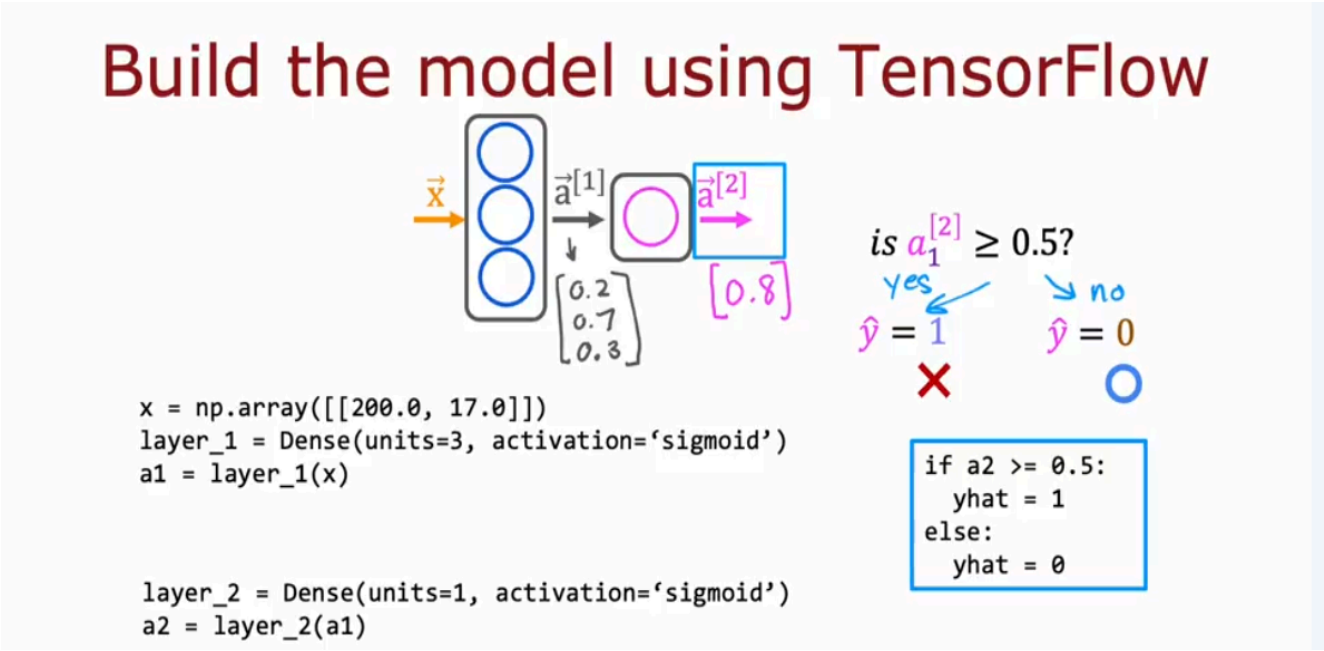
- The model learns features through hidden layers.
- The output layer uses **probabilities**, making it suitable for binary classification (0 vs. 1).
- A more complex neural network (with more layers and neurons) could handle all **digits (0-9)** instead of just 0 and 1.




Tensorflow implementation:

Good coffee or not classification  using neural network , forward inference






This image demonstrates how to build a simple binary classification model using TensorFlow. The model follows a feedforward neural network structure with the sigmoid activation function.

## Understanding the Model

1. **Input $\mathbf{x}$**:
   - The input has **two features**:

$$\mathbf{x} = [200.0, 17.0]$$

   - This input is passed through the first layer.
2. **First Layer (Hidden Layer)**

```python
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
```

   - This is a **fully connected (Dense) layer**.
   - It has **3 neurons (units)**.
   - The activation function is **sigmoid**, meanin utputs are in the range (0,1).

3. **Second Layer (Output Layer)**

```python
layer_2 = Dense(units=1, activation='sigmoid')
a2 = layer_2(a1)
```

- This layer has **1 neuron** (for binary classification).
- It applies the **sigmoid activation** to produce $a^{[2]}$, which is a probability score.

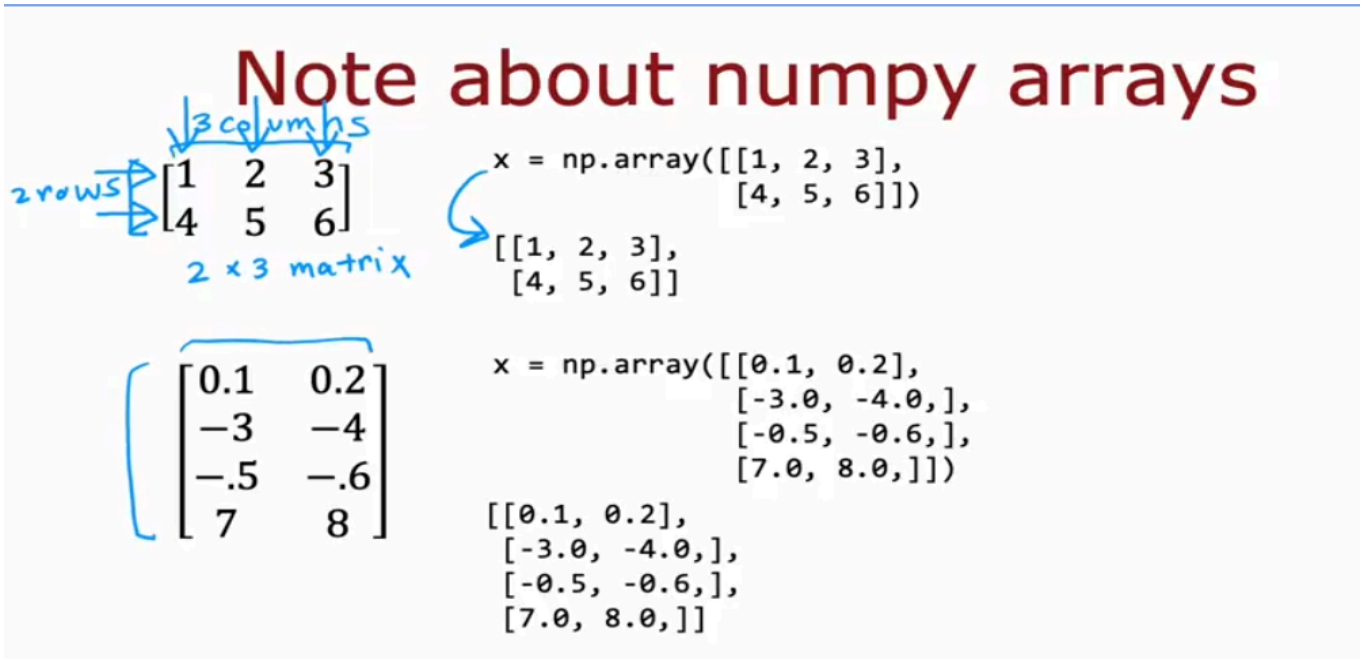
4. **Decision Rule (Classification)**

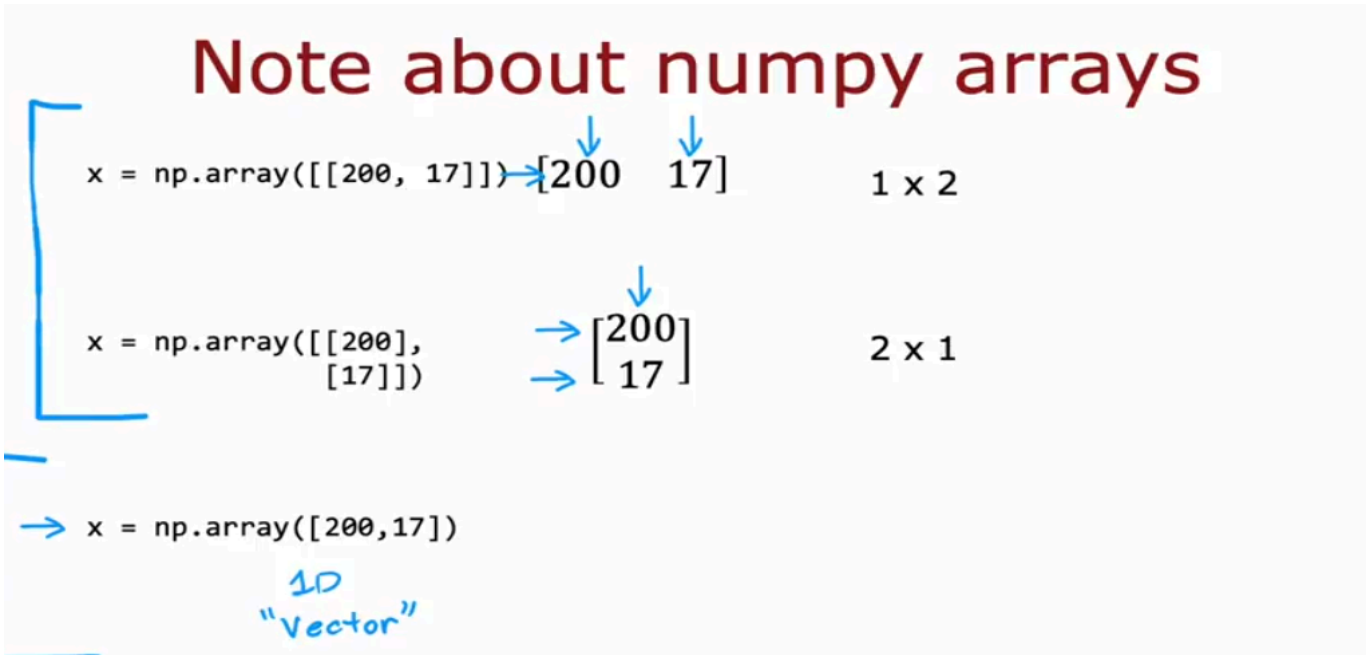
- The final output $a^{[2]}$ is compared to **0.5**:
  - If $a^{[2]} \geq 0.5$, predict **1**.
  - Otherwise, predict **0**.

```python
if a2 >= 0.5:
    yhat = 1
else:
    yhat = 0
```

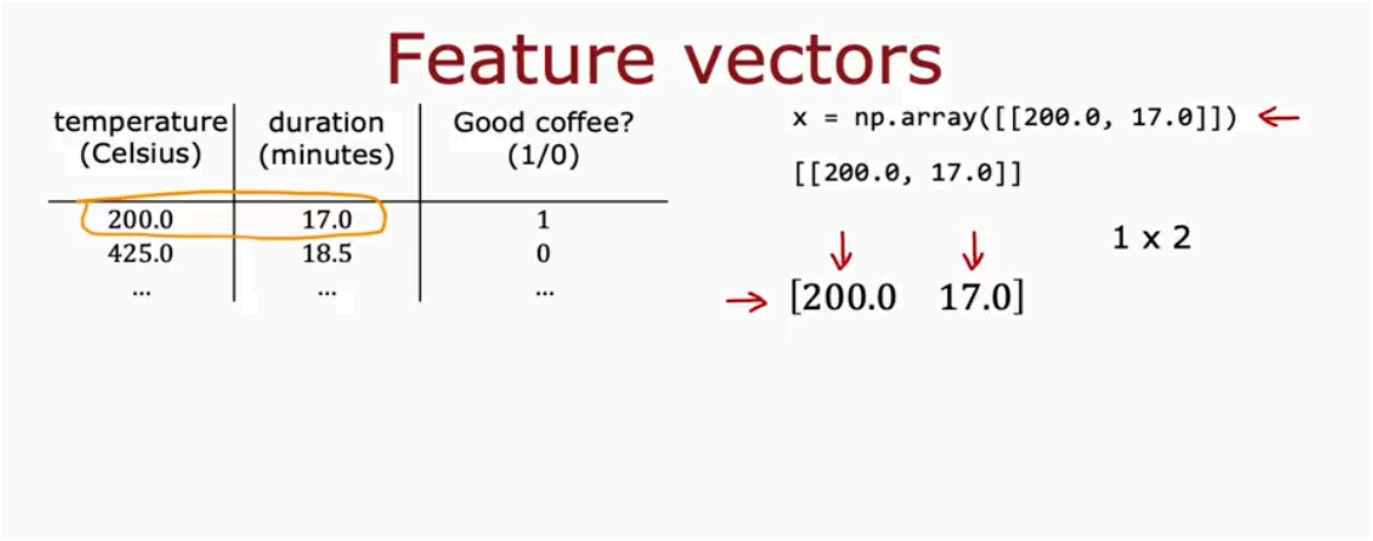
- This follows the standard binary classification approach.

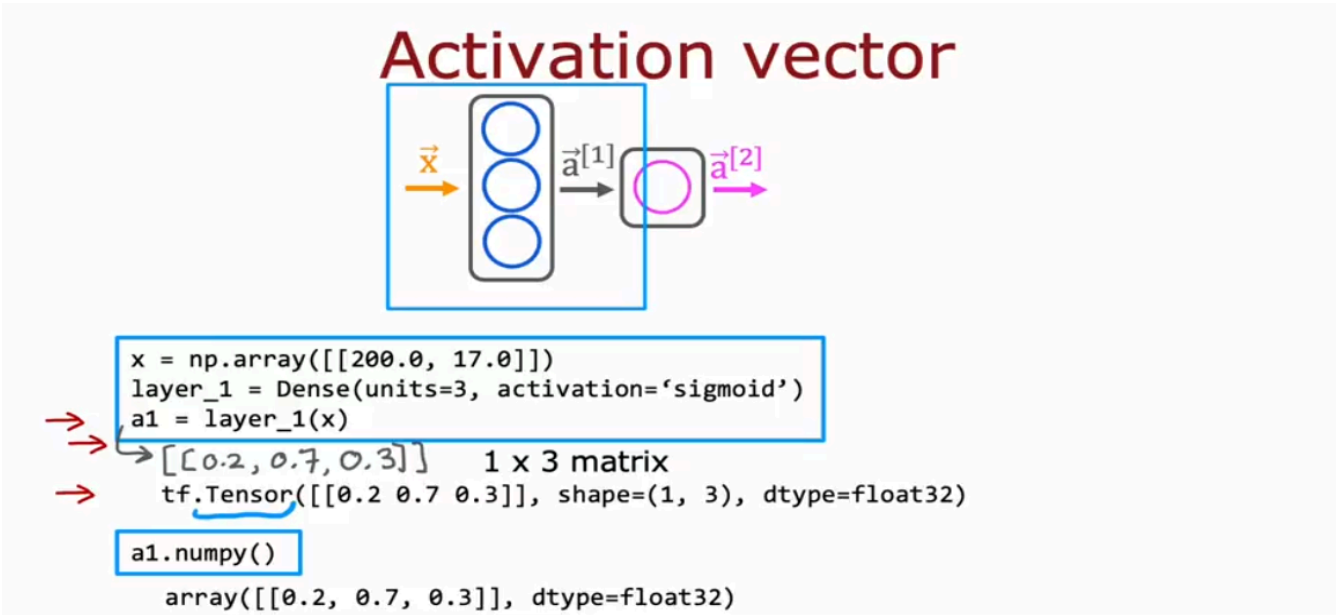
Data in Tensorflow:

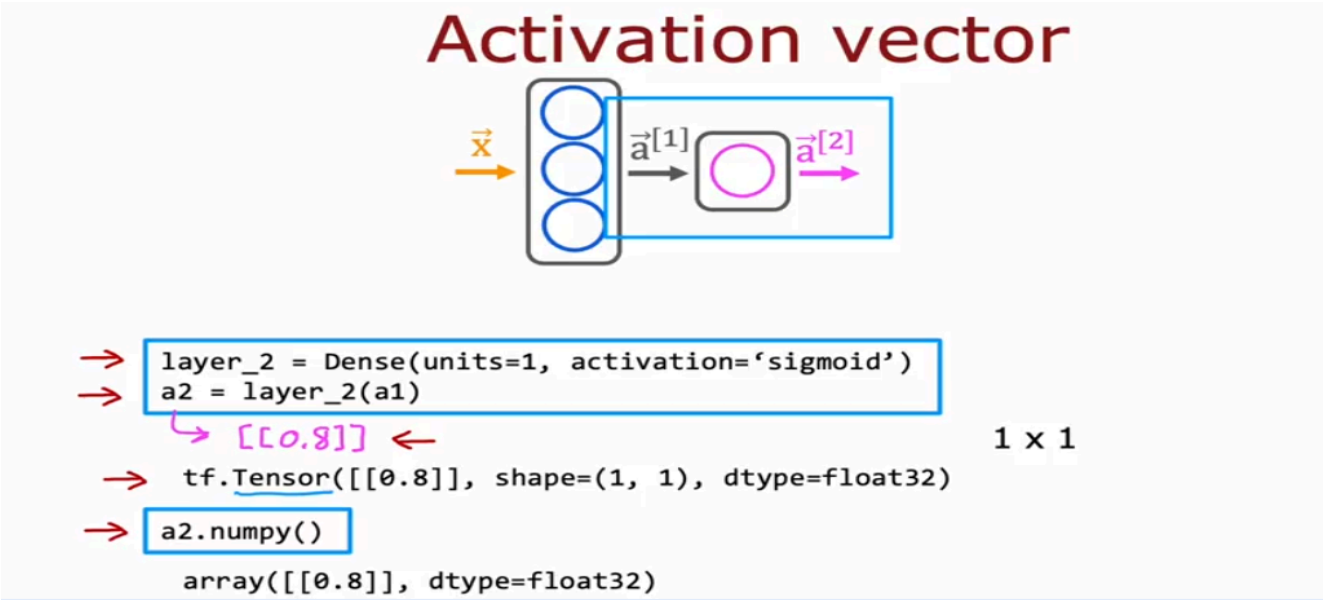


Feature vector :






These images illustrate the process of building a simple neural network using TensorFlow.

## . Feature Vectors

- The first image shows how raw input data is represented as a **feature vector**.
- The dataset consists of two input features:
    - **Temperature (Celsius)** (e.g., 200.0)
    - **Duration (minutes)** (e.g., 17.0)
- The corresponding label (Good coffee? 1 or 0) is the target.
- The input feature vector is stored as a **NumPy array**:

```python
x = np.array([[200.0, 17.0]])
```

This vector has a shape of **(1, 2)**, meaning 1 sample with 2 features.

## 2. First Dense Layer (Hidden Layer)

- The second image introduces a **Dense (fully connected) layer** with:
  - **3 neurons**
  - **Sigmoid activation function**
- This layer transforms the input vector into an **activation vector**:

```python
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)
```

- The output is a **TensorFlow tensor**:

```lua
tf.Tensor([[0.2  0.7  0.3]], shape=(1, 3), dtype=float32)
```

- Converting this tensor to a NumPy array:

```python
a1.numpy()
```

Produces:

```lua
array([[0.2, 0.7, 0.3]], dtype=float32)
```

### 3. Second Dense Layer (Output Layer)

- The third image shows the **output layer**:
  - **1 neuron**
  - **Sigmoid activation function**
- This layer takes the activation values from the previous layer and produces the final prediction:

```python
layer_2 = Dense(units=1, activation='sigmoid')
a2 = layer_2(a1)
```

- The output is another tensor:

```lua
tf.Tensor([[0.8]], shape=(1, 1), dtype=float32)
```

- Converting this to a NumPy array:

```python
a2.numpy()
```

Results in:

```lua
array([[0.8]], dtype=float32)
```

## 4. Final Prediction

- The final value (0.8) is compared with **0.5** (threshold) to classify the output:

```python
if a2.numpy() >= 0.5:
    yhat = 1  # Positive class
else:
    yhat = 0  # Negative class
```

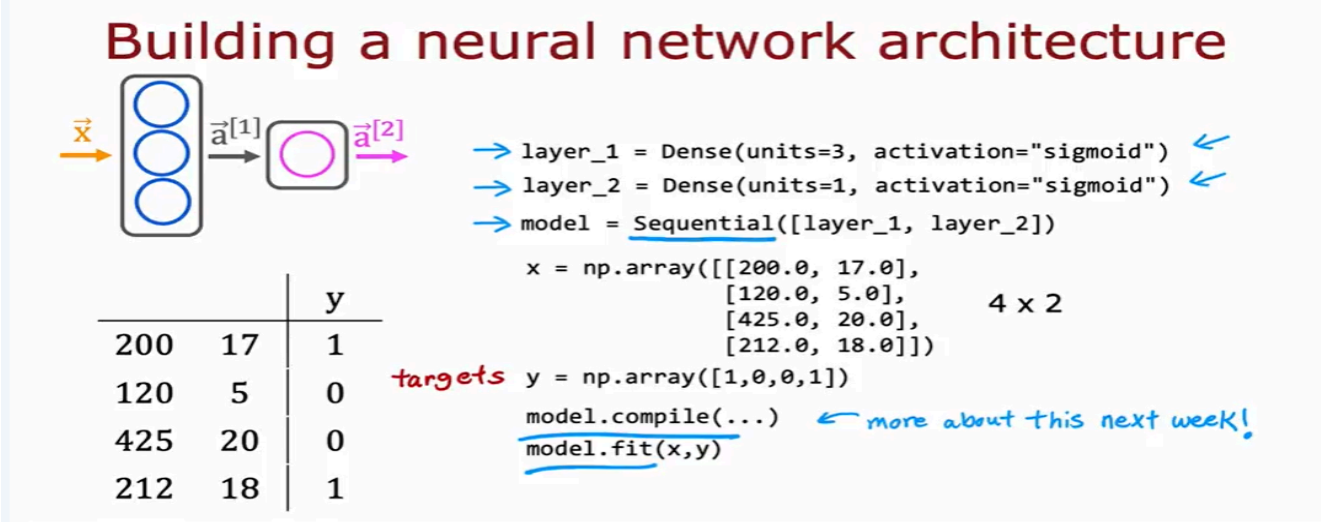
## Summary

- **Step 1**: The input feature vector is prepared using NumPy.
- **Step 2**: It is passed through a **hidden layer** with 3 neurons and a sigmoid activation function.
- **Step 3**: The activations from the first layer are passed to an **output layer** with 1 neuron and a sigmoid activation function.
- **Step 4**: The output is converted into a final class label (0 or 1).

## Why Use TensorFlow Instead of NumPy?

- NumPy is useful for basic computations but lacks features like automatic differentiation, GPU acceleration, and model training.
- TensorFlow provides:
  - **Automatic differentiation** for gradient computation.
  - **Optimized execution** for training deep models.
  - **Support for GPU/TPU acceleration**.
  - **Scalability** for large datasets and distributed training.

Building a neural network:




### 🔷 What each layer does:

- `Dense(units=3)`: A **fully connected layer** with 3 neurons.
- `activation="sigmoid"`: Squashes values between 0 and 1, good for binary outputs.
- Second layer (output layer): 1 neuron, since we want a single output (0 or 1).

### 🛠️ Model Compilation & Training

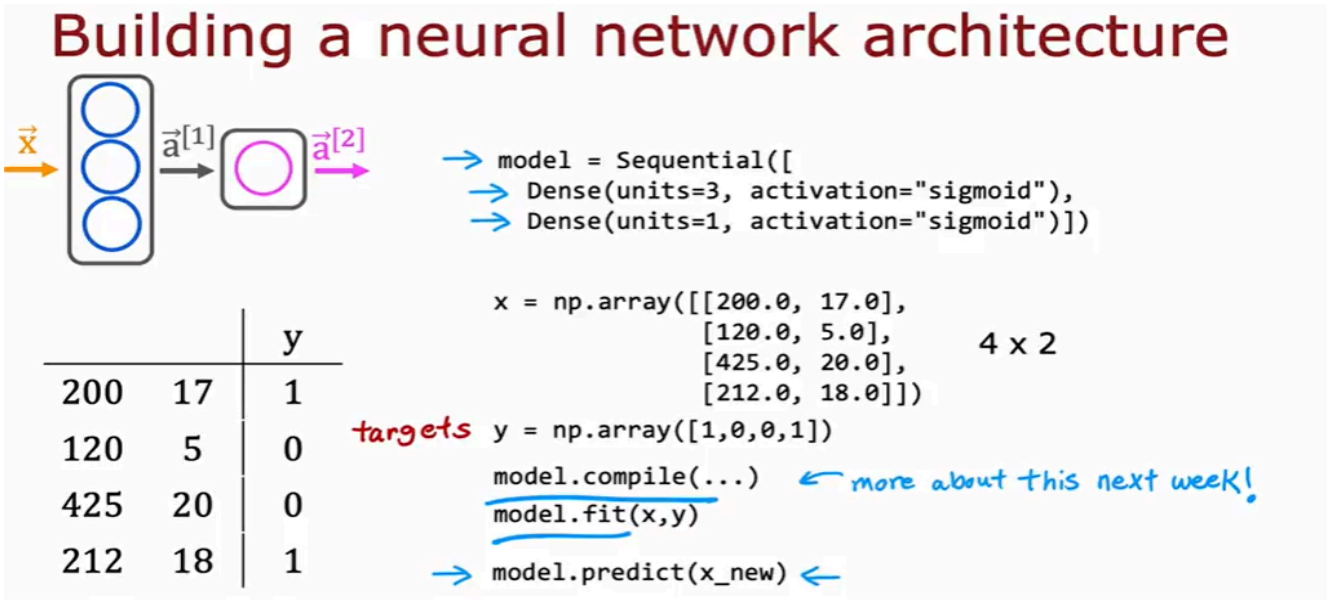
python Copy Edit

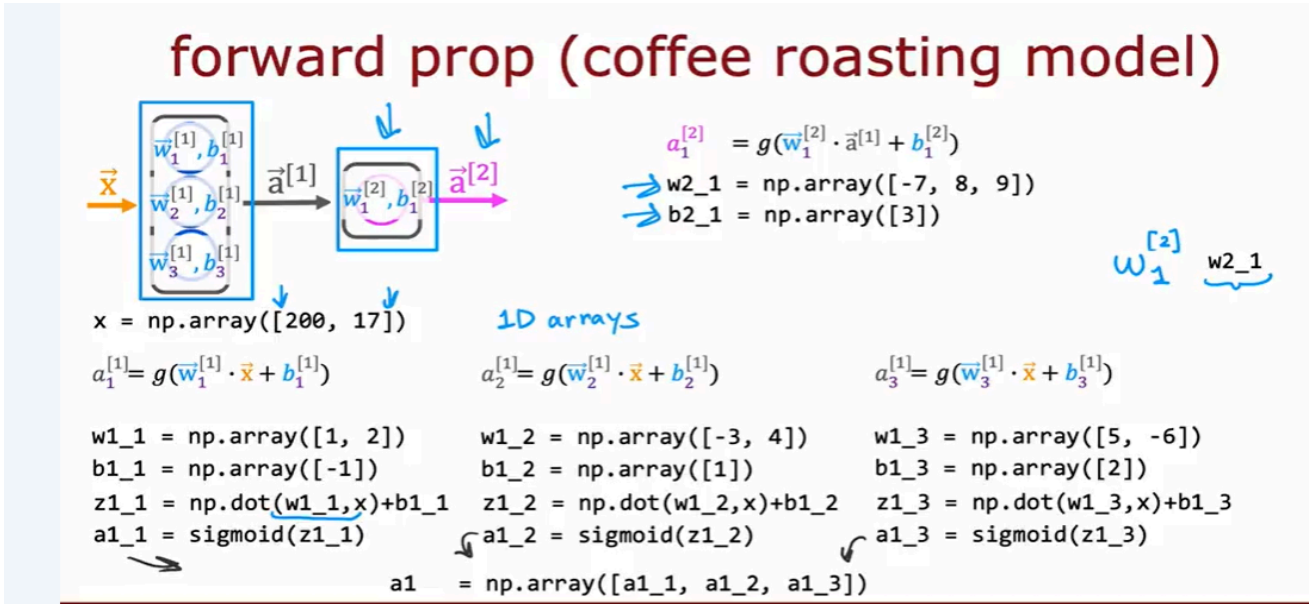
```
model.compile(...)  # This sets loss function, optimizer, metrics — explained later
model.fit(x, y)     # Trains the model using input x and target y
```

- `model.compile(...)`: This typically includes things like:

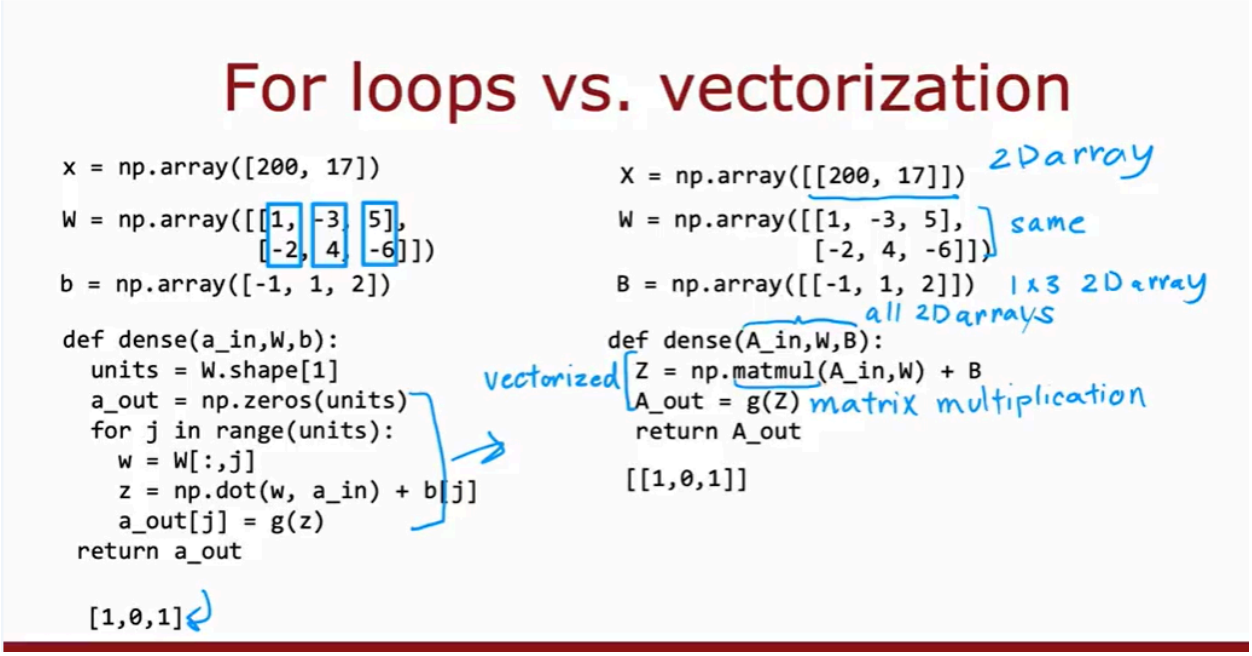
```python
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

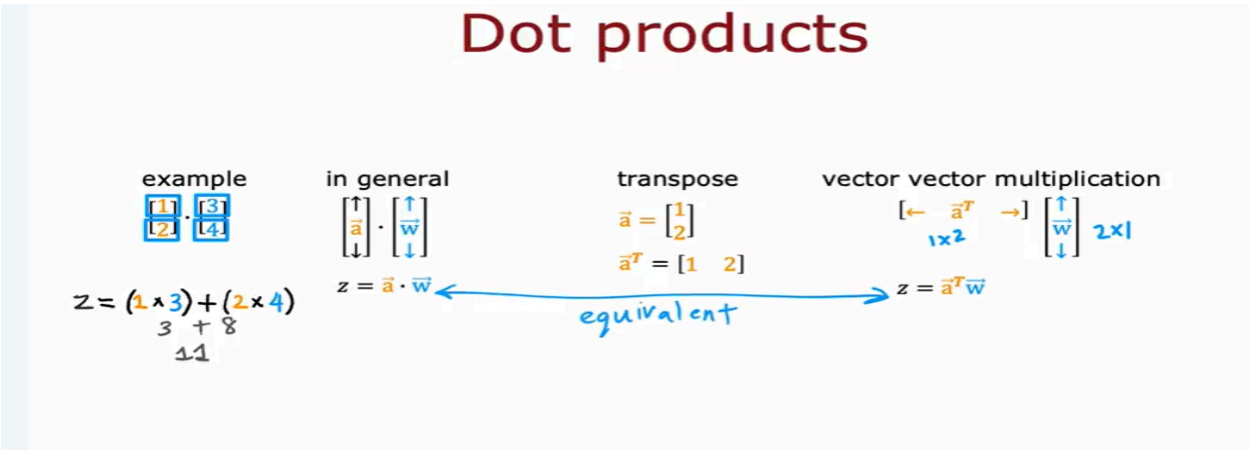
- `model.fit(x, y)`: Fits the model to the training data.



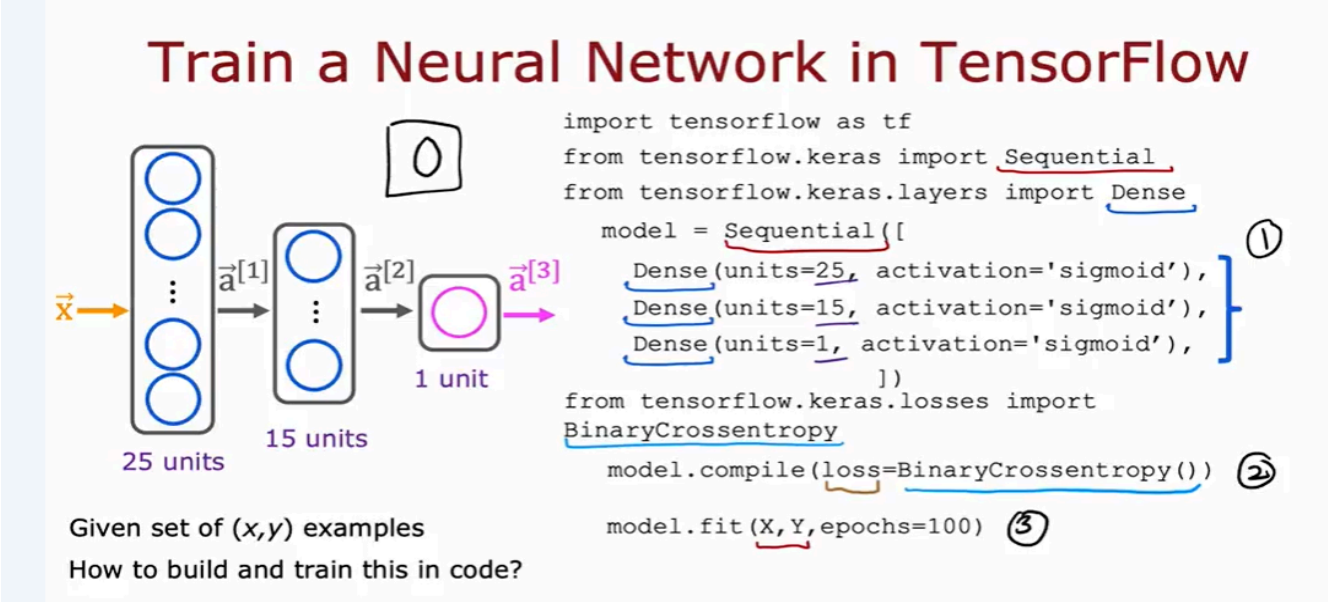



Vectorized implementation of Forward Propagation:

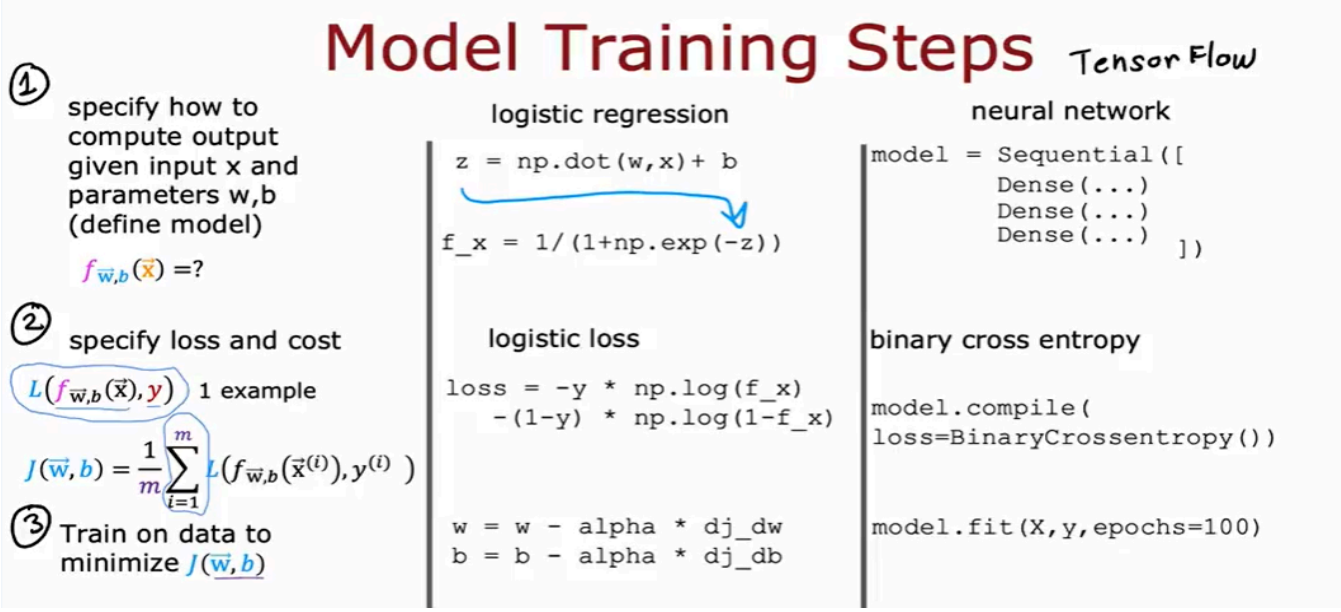


Neural Network Training:




How neural network training maps with regression

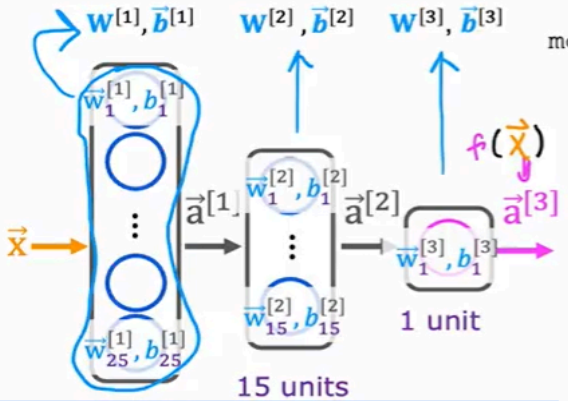



This image summarizes the **model training steps** for both **logistic regression** and a **neural network**, highlighting the key similarities and differences — especially in **how output is calculated, loss is computed, and the model is trained**.

# 1. Create the model

define the model

$f(\vec{x})$ =?




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

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

# 2. Loss and cost functions

handwritten digit classification problem — binary classification

$$L(f(\vec{x}), y) = -y\log(f(\vec{x})) - (1-y)\log(1-f(\vec{x}))$$

Compare prediction vs. target

logistic loss

also known as binary cross entropy

$$J(\mathbf{W}, \mathbf{B}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\vec{x}^{(i)}), y^{(i)})$$

$\mathbf{W}^{[1]}, \mathbf{W}^{[2]}, \mathbf{W}^{[3]}$ $\quad \vec{b}^{[1]}, \vec{b}^{[2]}, \vec{b}^{[3]}$

$f_{W,B}(\vec{x})$

```
model.compile(loss= BinaryCrossentropy())
```

```
from tensorflow.keras.losses import
  BinaryCrossentropy
```

K Keras

regression (predicting numbers and not categories)

mean squared error

```
model.compile(loss= MeanSquaredError())
```

```
from tensorflow.keras.losses import
  MeanSquaredError
```

## 📌 What are Loss and Cost Functions?

**Loss Function**:

- Measures how far the **predicted output** is from the **actual target** for a **single example**.
- Denoted here as:

$$L(f(\vec{x}), y) = -y \log(f(\vec{x})) - (1 - y) \log(1 - f(\vec{x}))$$

- This is **logistic loss**, also called **binary cross-entropy**, and it's used for **binary classification** problems.

**Cost Function**:

- The **average of the loss** over all training examples (i.e., dataset-level).
- Denoted here as:

$$J(W, B) = \frac{1}{m} \sum_{i=1}^{m} L(f(\vec{x}^{(i)}), y^{(i)})$$

- It depends on all weights and biases in the model: `W[1], W[2], ..., B[1], B[2], ...`

---

## 💡 When to Use Which Loss?

🟦 **Binary Classification** (e.g. yes/no, 0/1):

- Use **Binary Crossentropy**:

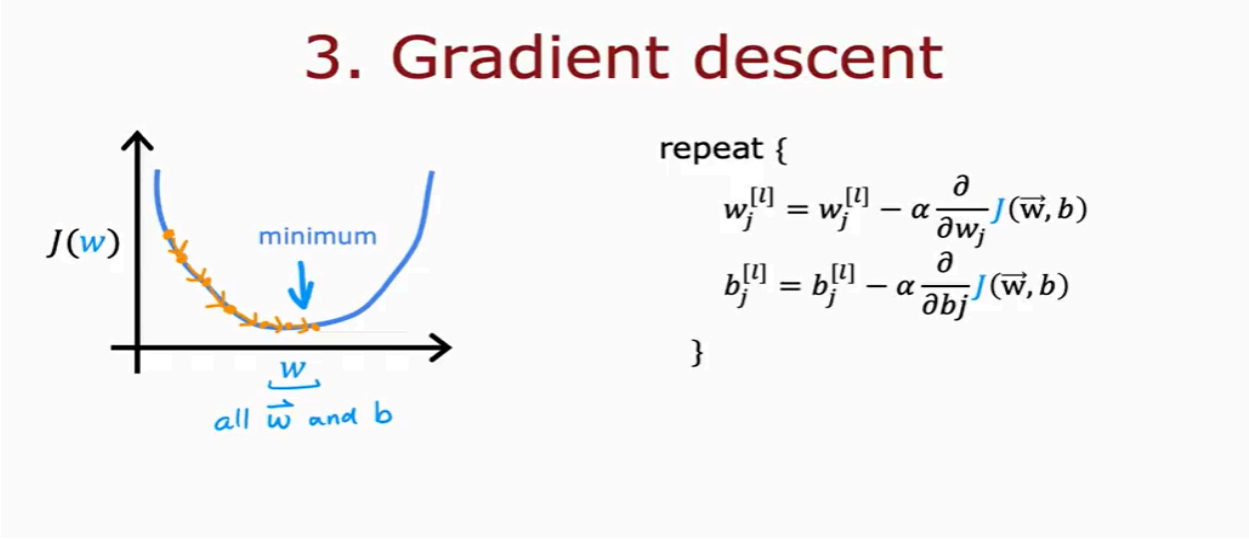
```python
from tensorflow.keras.losses import BinaryCrossentropy
model.compile(loss=BinaryCrossentropy())
```

### 🧠 Key Concept:

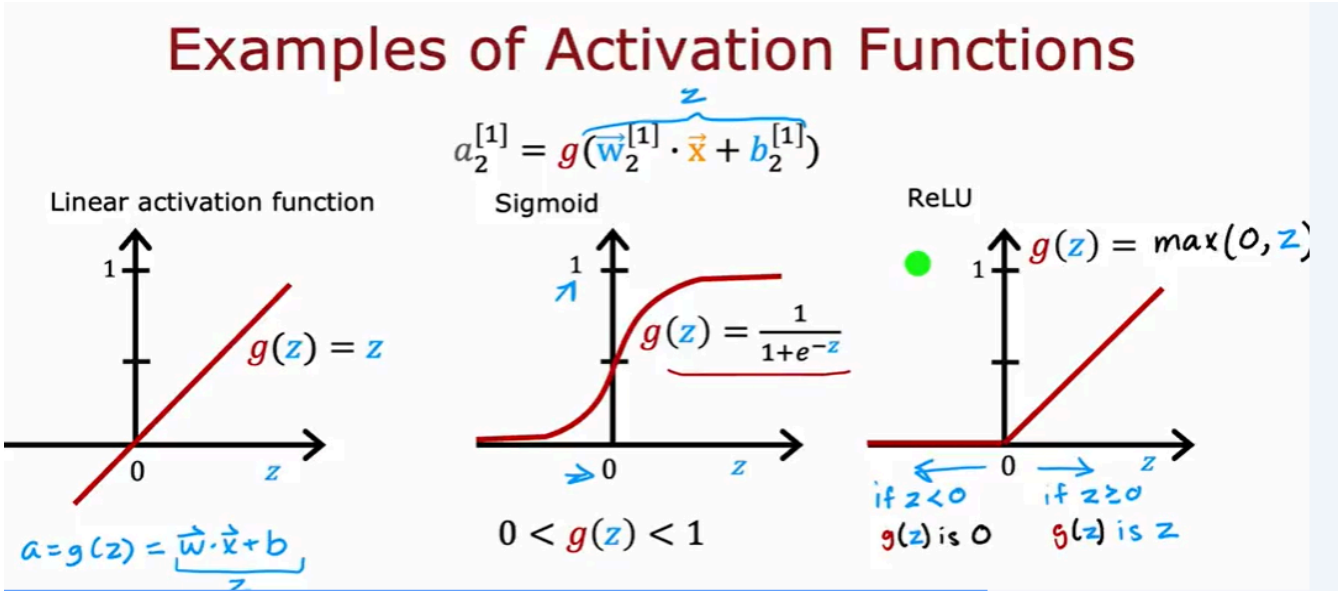
> The loss function helps the model learn by quantifying how "bad" a prediction is — and we train the model by adjusting weights/biases to minimize the cost function using algorithms like gradient descent.

---

Now compute derivatives for back propagation and updation of weights have been done

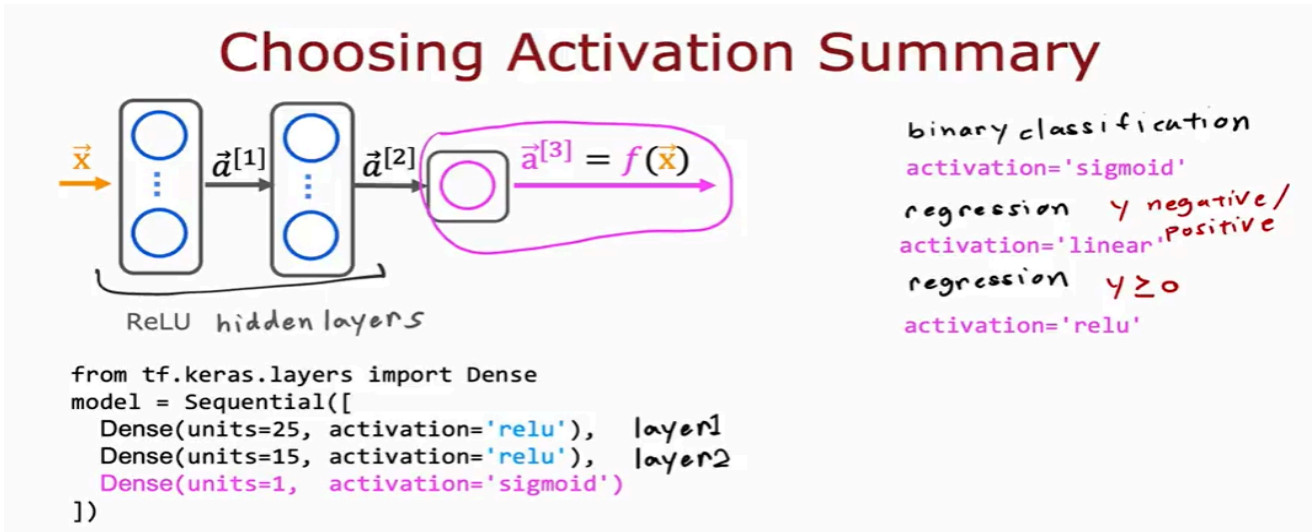


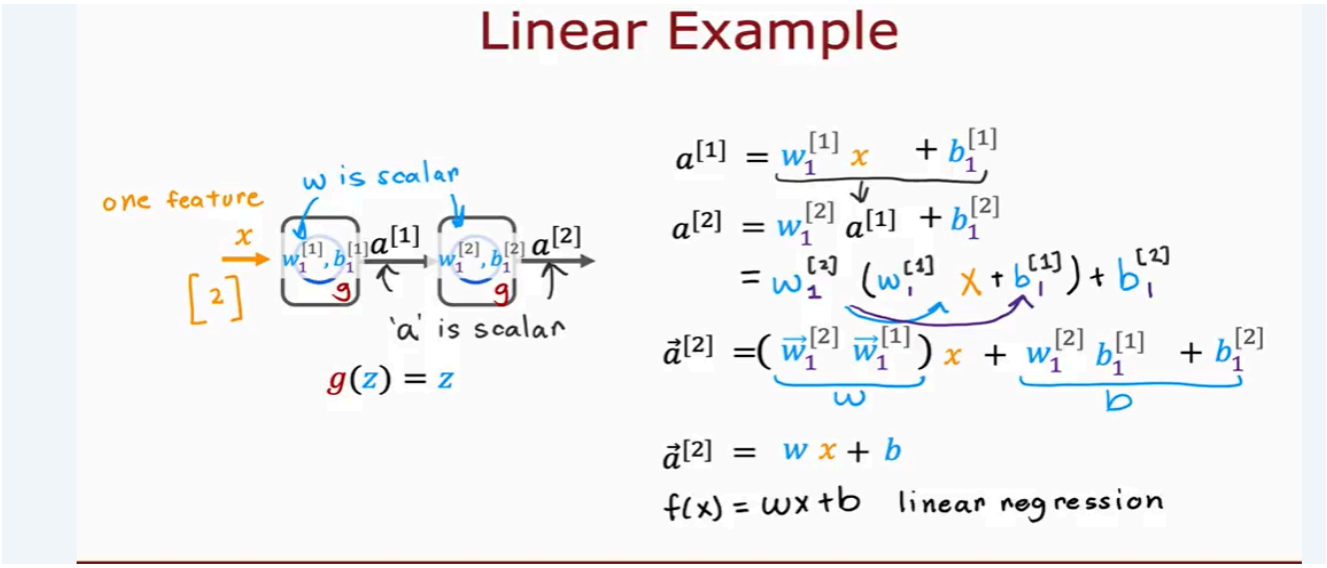
Alternative To The Sigmoid Activations:




relu-> Rectified Linear Activation function

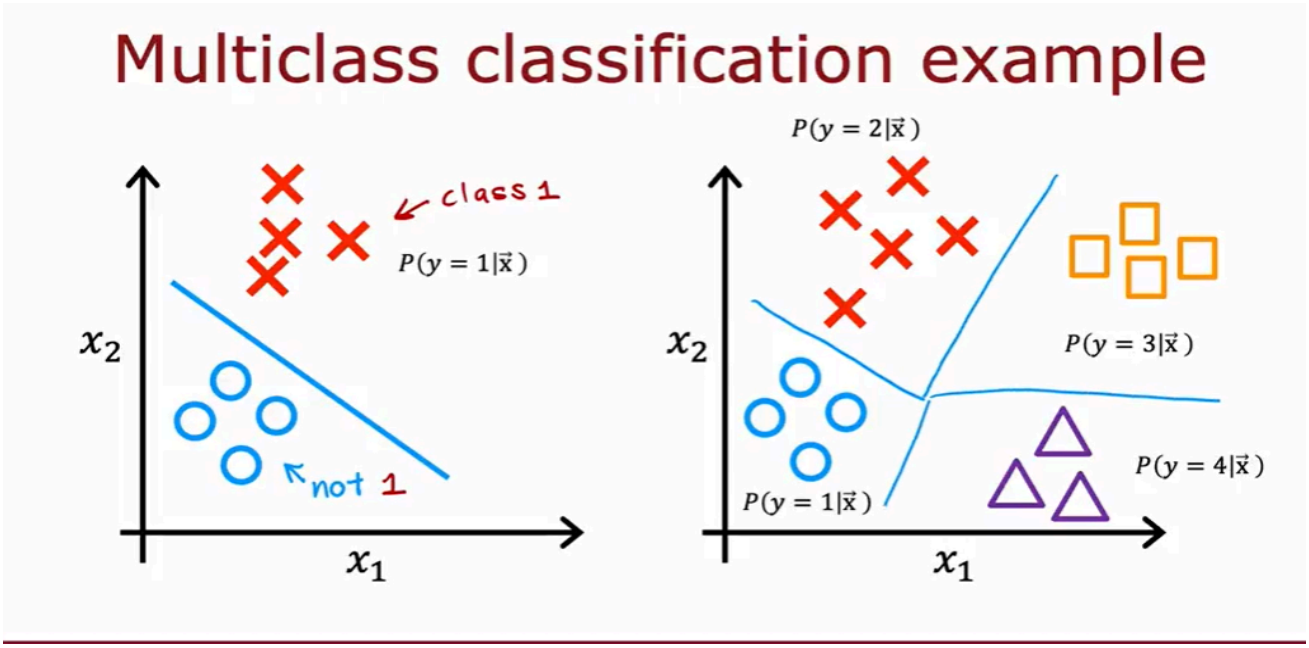
Use of Linear activation function is some times called no use of activation function as it does not change the (wx+b)

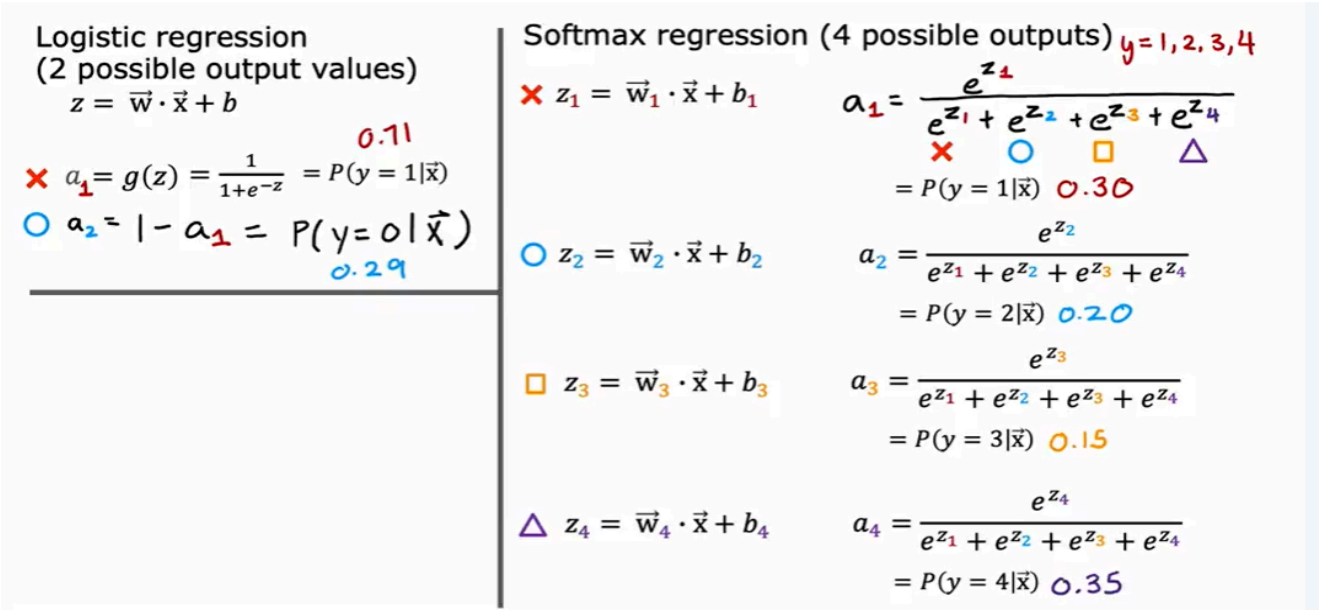




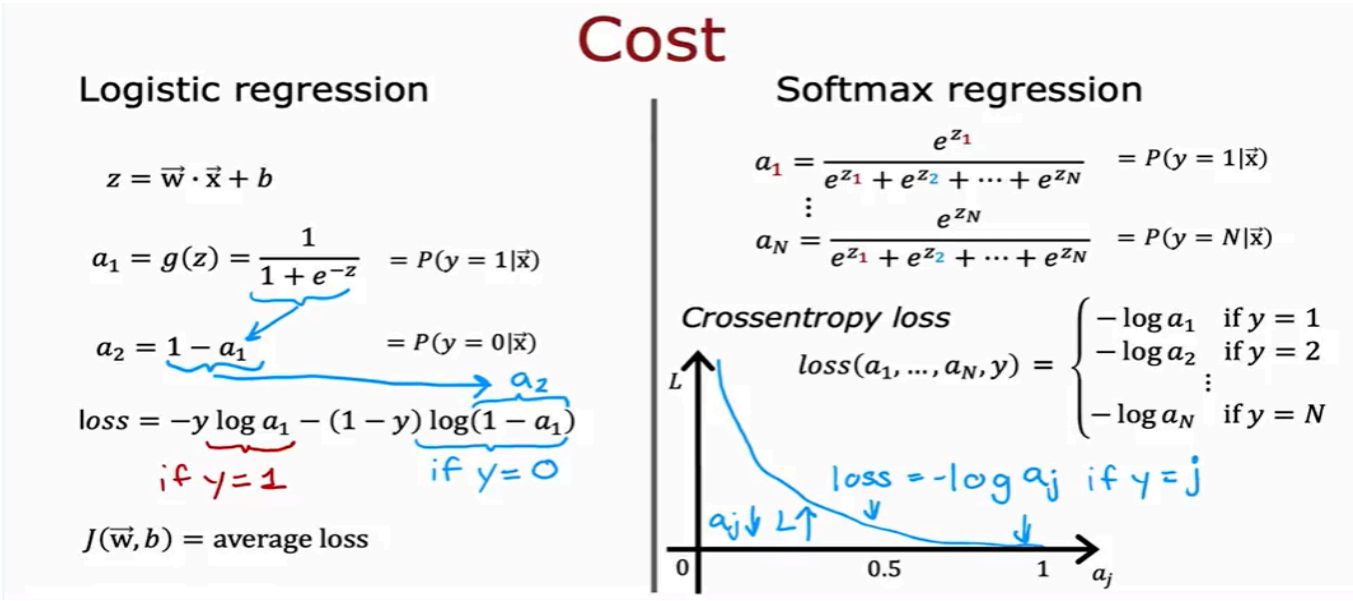

If I have neural network with multiple layers like this and I want to use linear activation function with all hidden layers and also linear activation function is used in output layer then it turns out this model will compute an output that is completely similar to linear regression and like wise if sigmoid activation function is used then this model will work as logistics regression.

## MultiClass Classification Problem:

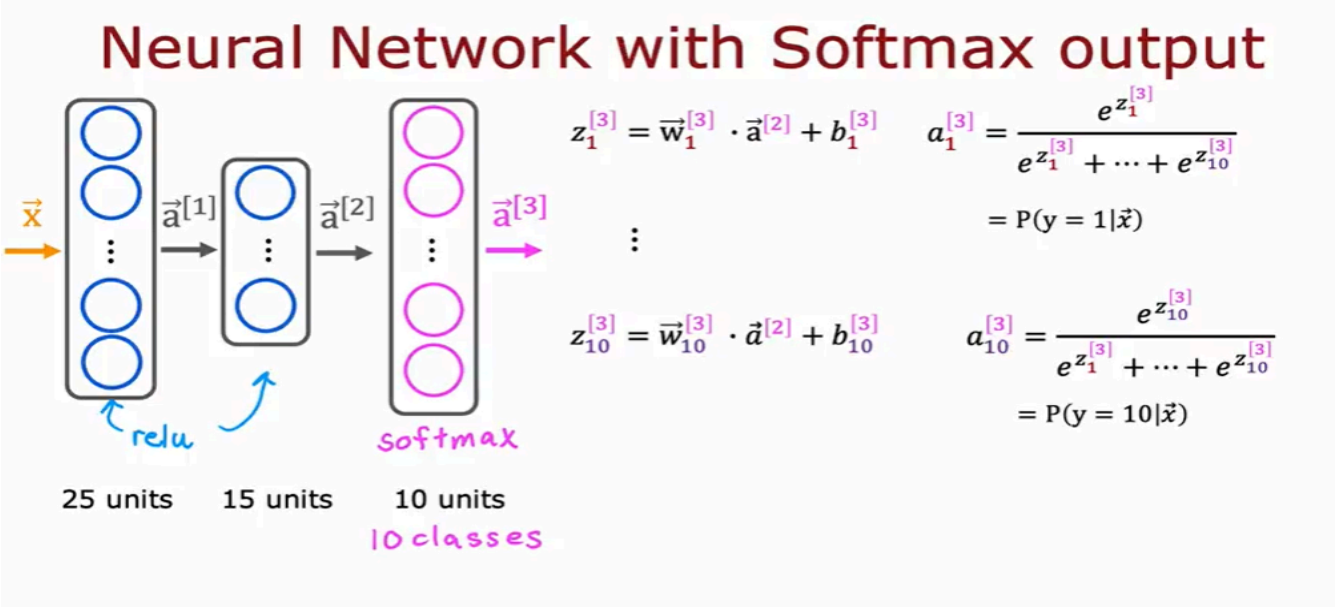


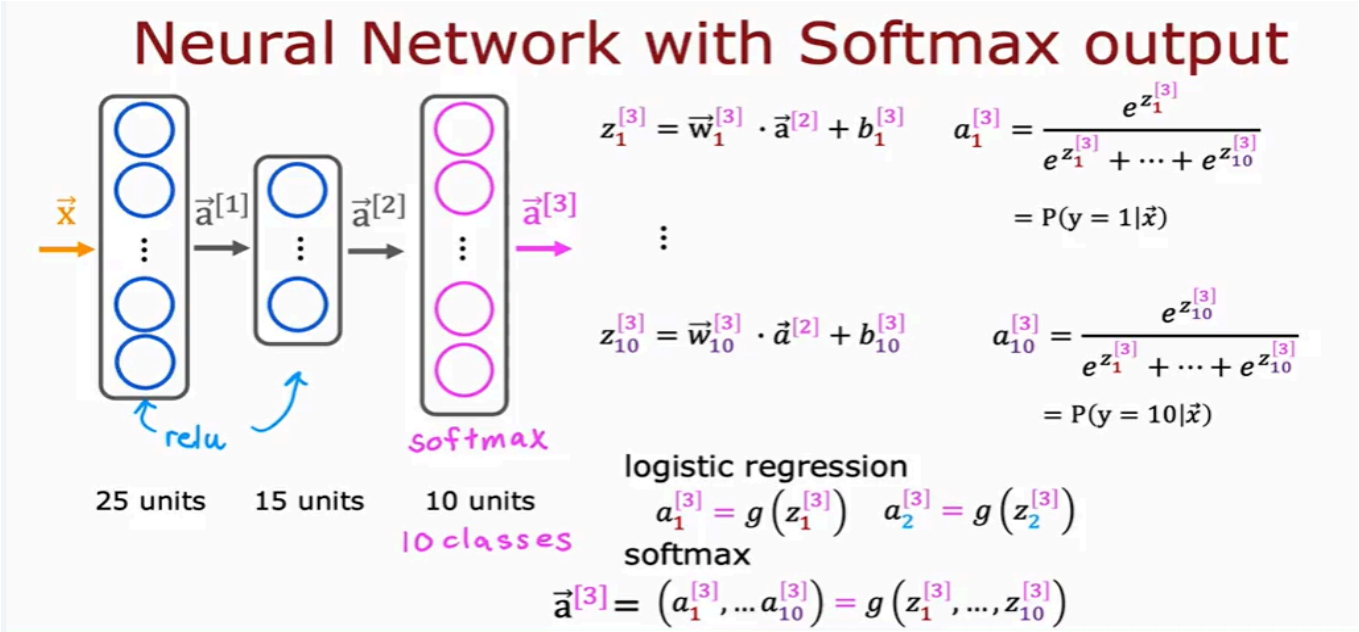


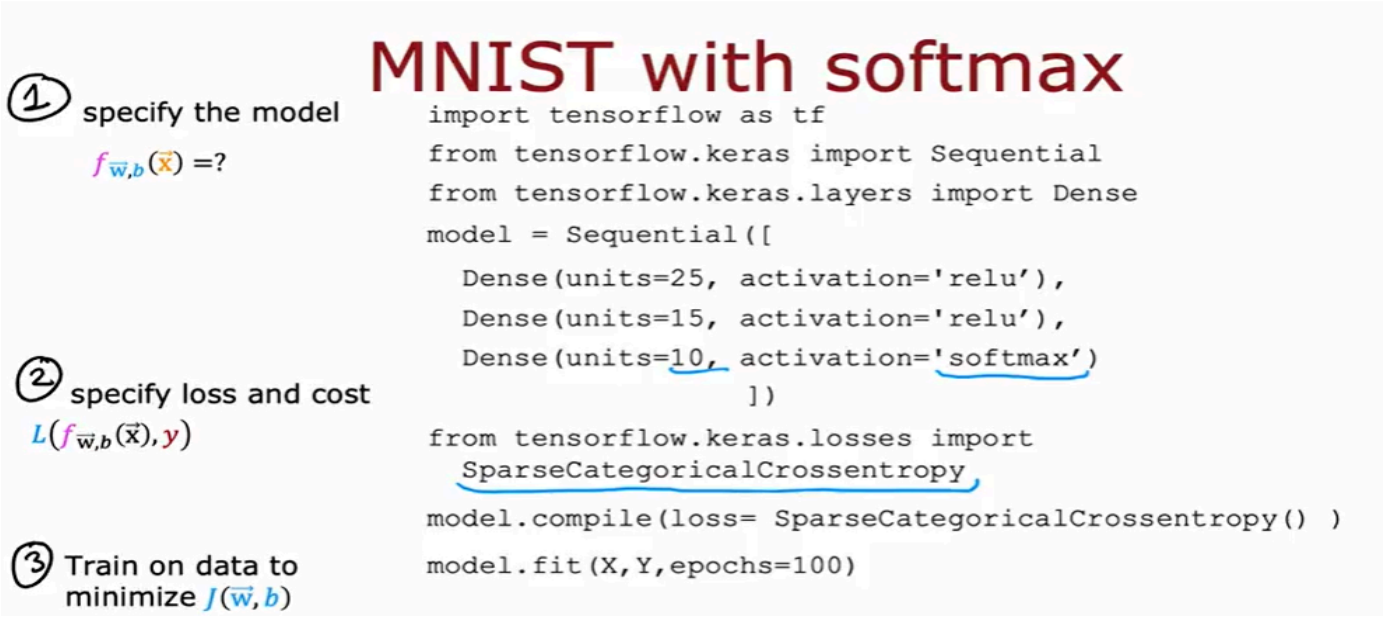

Loss for softmax:



Neural Network for Softmax output:

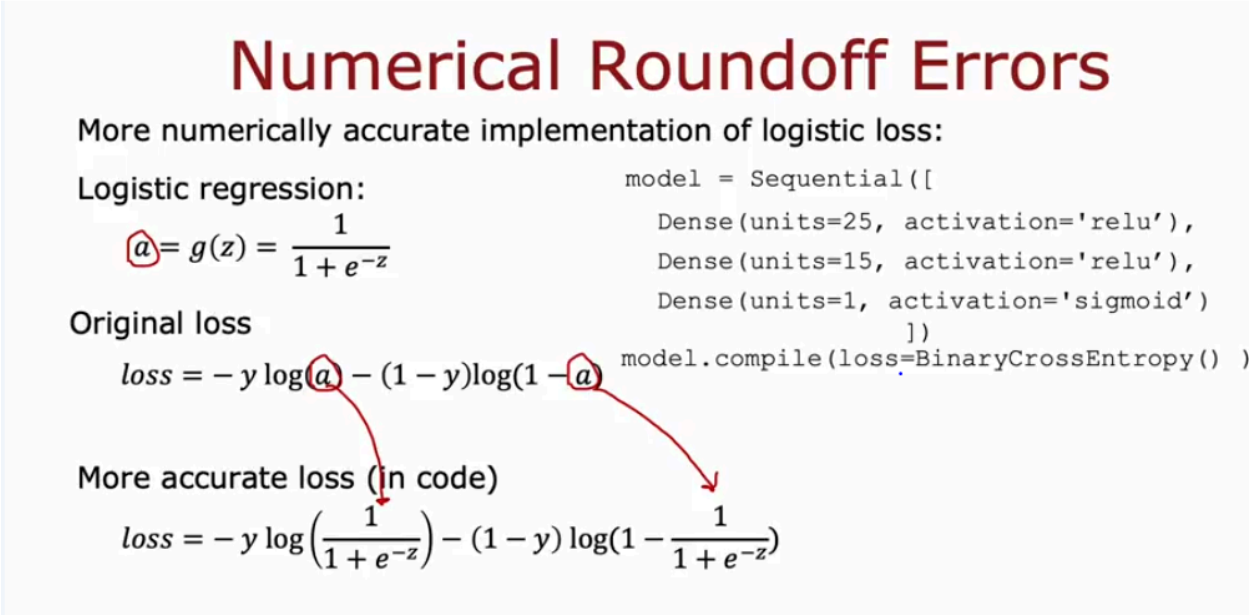


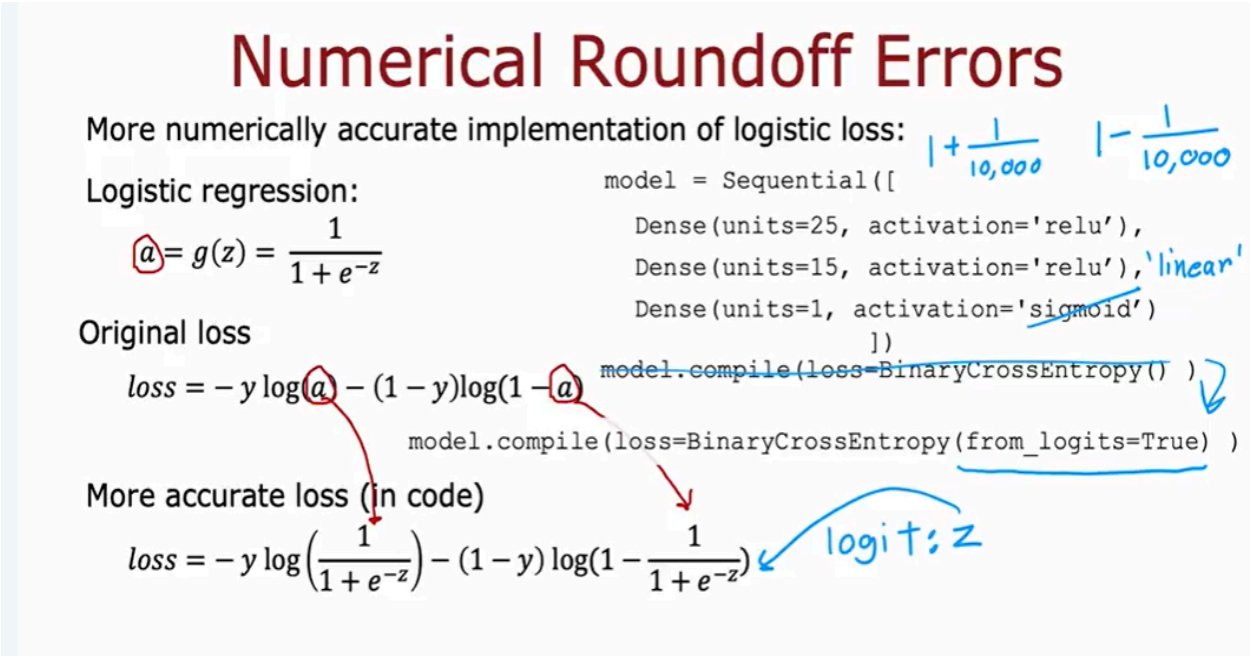


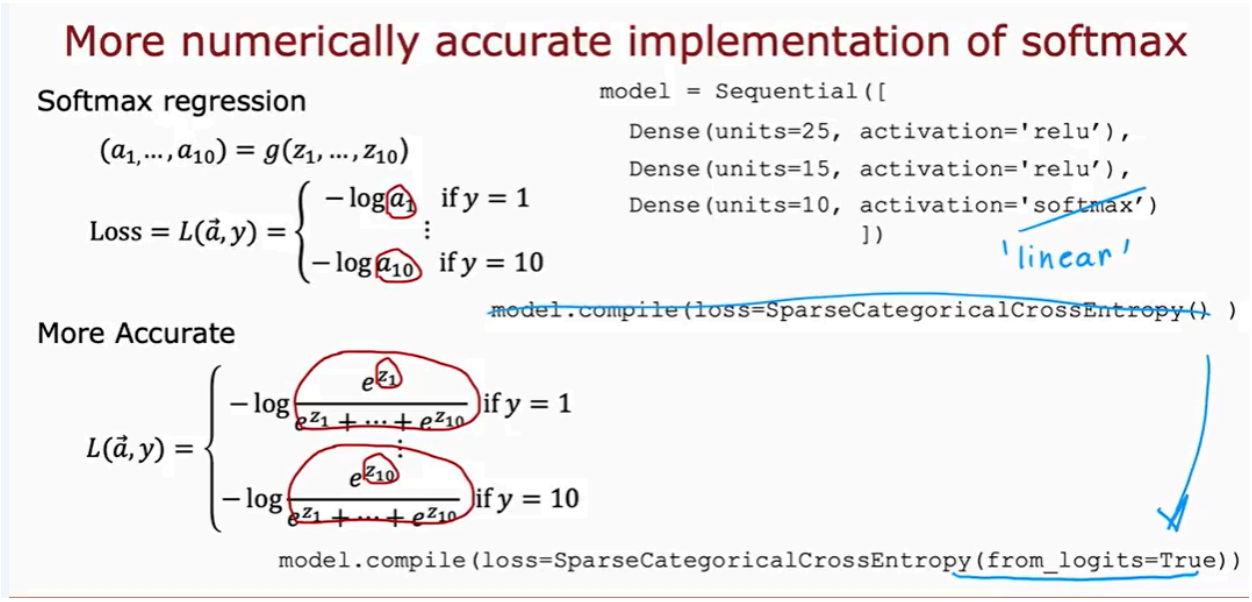

In case of logistic regression binary_crossentropy loss function and in case of multiclass classification like digit recognization sparse categorical cross entropy is used.

## Numerical Round Off Errors:




In case tensorflow if the second wayis used then tensorflow handle the numeric calculation more accurate way.




In deep learning, a **logit** is the output of a model **before** applying an activation function like **softmax** (for multi-class classification) or **sigmoid** (for binary classification). They can be any real numbers — not limited to [0, 1].

# MNIST (more numerically accurate)

**model**

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='linear') ])
```

**loss**

```
from tensorflow.keras.losses import
  SparseCategoricalCrossentropy
model.compile(...,loss=SparseCategoricalCrossentropy(from_logits=True) )
```

**fit**

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

**predict**

```
logits = model(X)
f_x = tf.nn.softmax(logits)
```

not is $a_1 \ldots a_{10}$ $z_1 \ldots z_{10}$

In case logistic regression in more accurate way

# logistic regression (more numerically accurate)

**model**

```
model = Sequential([
  Dense(units=25, activation='sigmoid'),
  Dense(units=15, activation='sigmoid'),
  Dense(units=1, activation='linear')
                  ])
from tensorflow.keras.losses import
  BinaryCrossentropy
```

**loss**

```
model.compile(..., BinaryCrossentropy(from_logits=True))
```

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

**fit**

```
logit = model(X)
```

**predict**

```
f_x = tf.nn.sigmoid(logit)
```

Suppose you're classifying between 3 classes, and your model outputs:

```python
logits = [2.0, 1.0, 0.1]
```

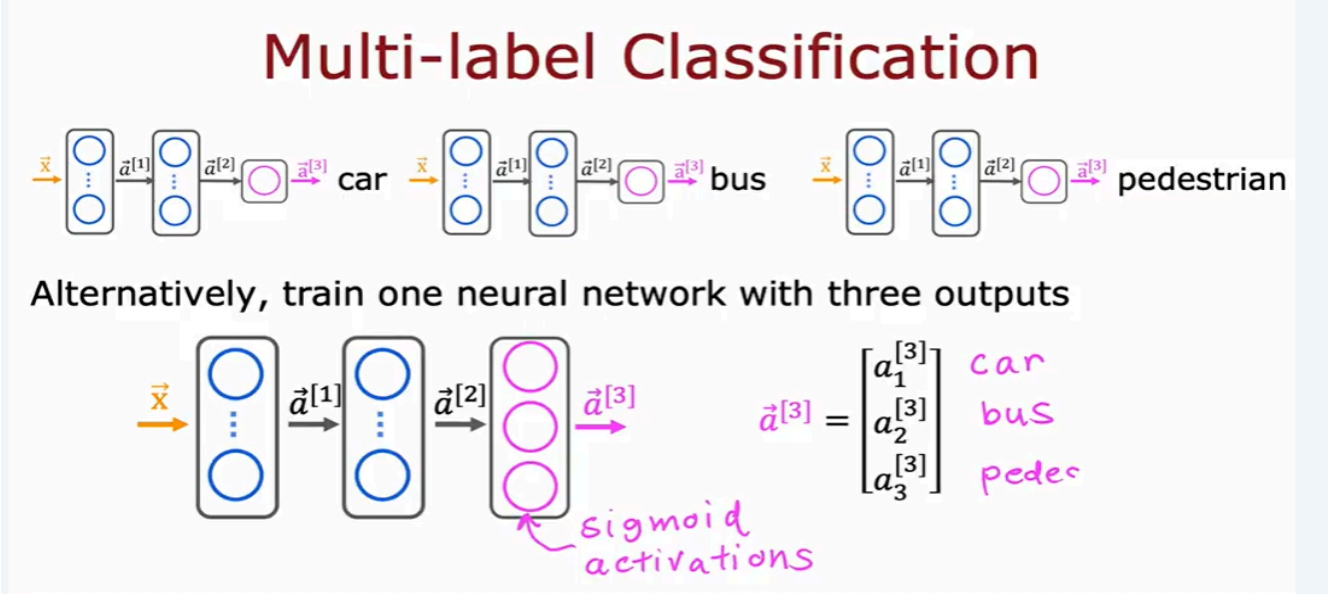
These are the **logits**. If you applied softmax to them, you'd get probabilities like:

```python
softmax(logits) = [0.659, 0.242, 0.099]
```

But when you're using loss functions like `CrossEntropyLoss` in PyTorch or `categorical_crossentropy(from_logits=True)` in TensorFlow/Keras, they often expect the **logits directly**, because they **internally apply softmax** for numerical stability.

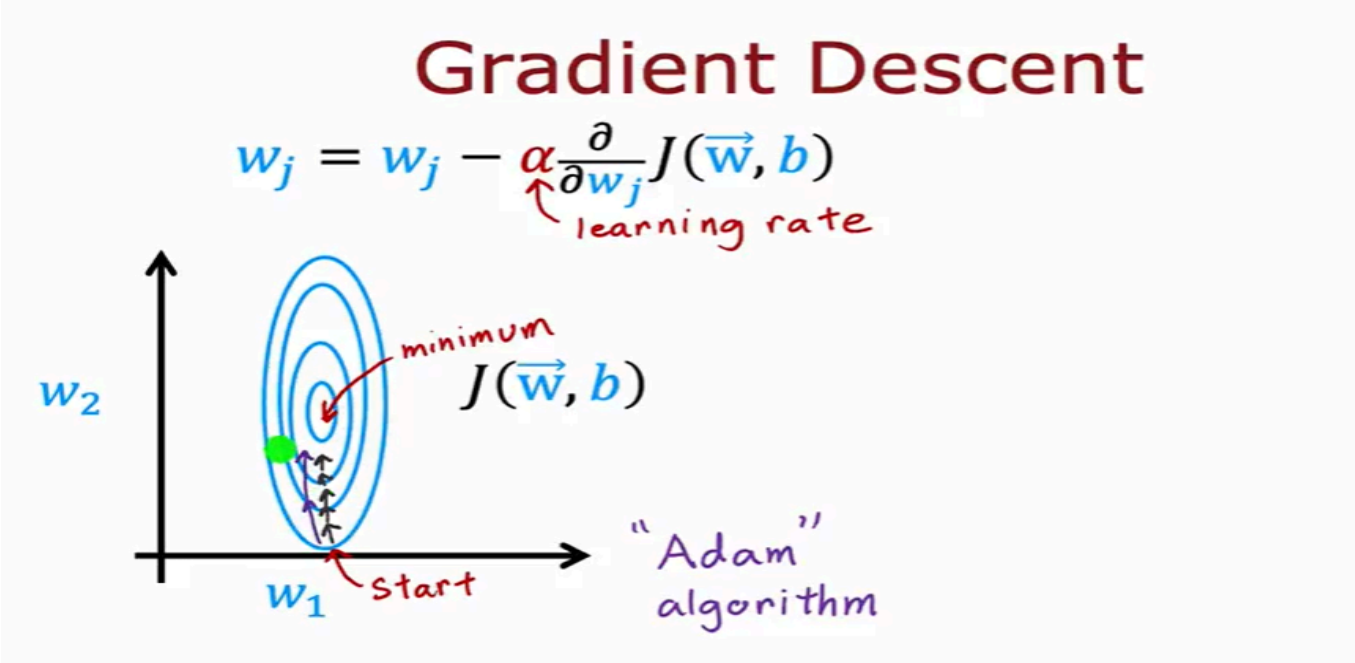
---

## Multi Label Classification:

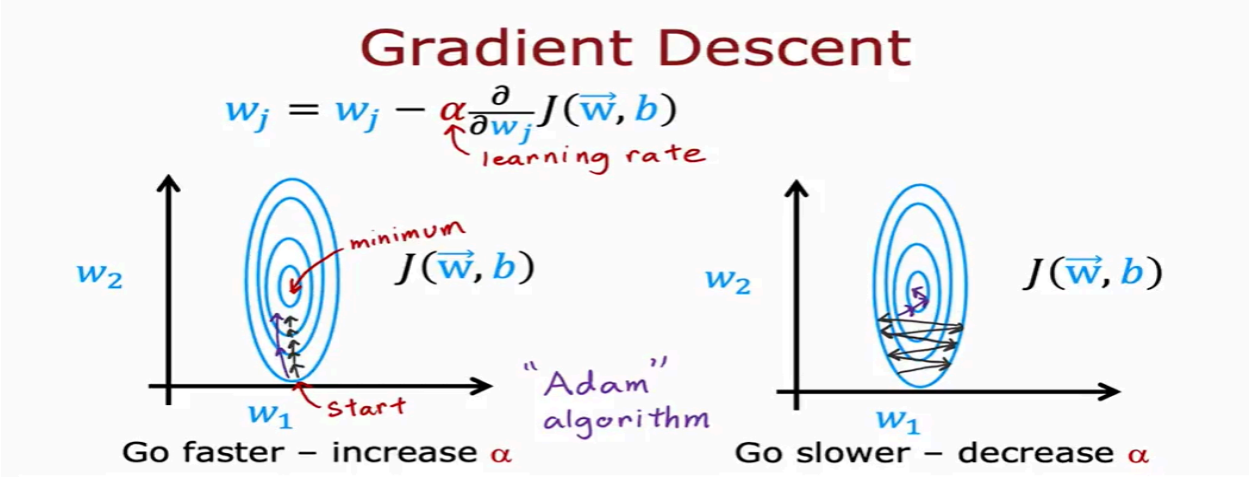


## Advanced Optimization

Gradient descent is an optimization algorithm that is widely used in machine learning, and was the foundation of many algorithms like linear regression and logistic regression and early implementations of neural networks. But it turns out that there are now some other optimization algorithms for minimizing the cost function, that are even better than gradient descent.




Adam algorithm - Adaptive Moment estimation. In case of small learning rate the curve goes to same direction so some times bigger learning rate needs and adam is a type of optimization algorithm which automatically increase or decrease learning rate.



If Adam sees that the learning rate is too small and we are just taking tiny little steps in similar direction over and over again ,it increases the learning rate. Similarly , if adam sees that the learning rate is too big and it is  just oscillating back and forth then , it will decrease the learning rate automatically.

## MNIST Adam

### 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

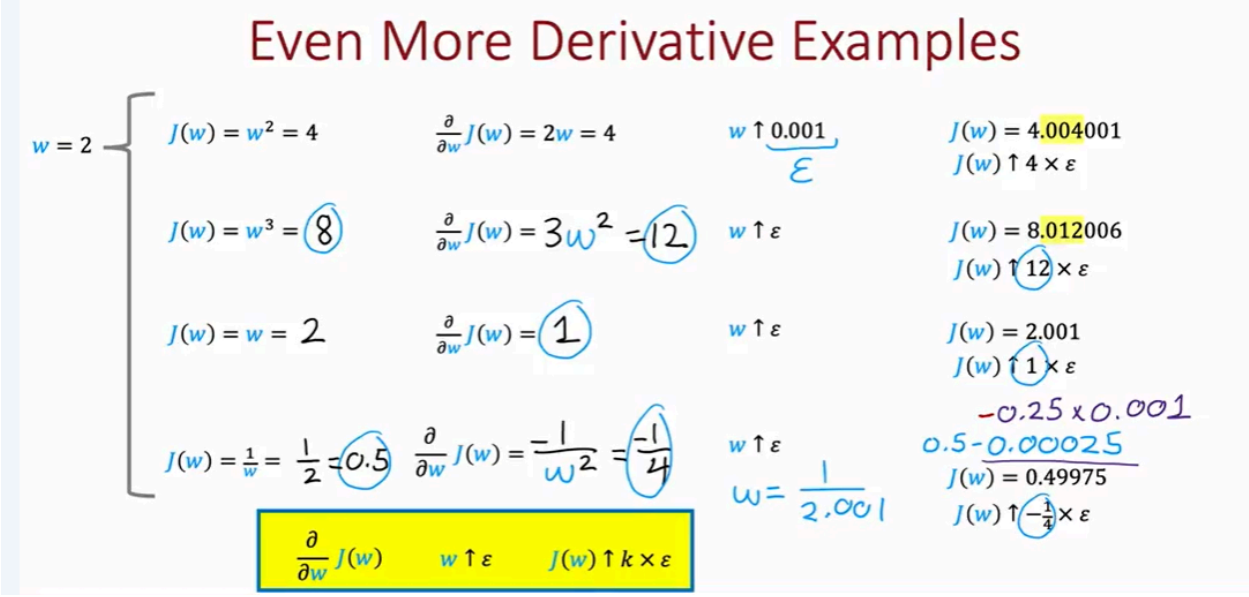
```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

### fit

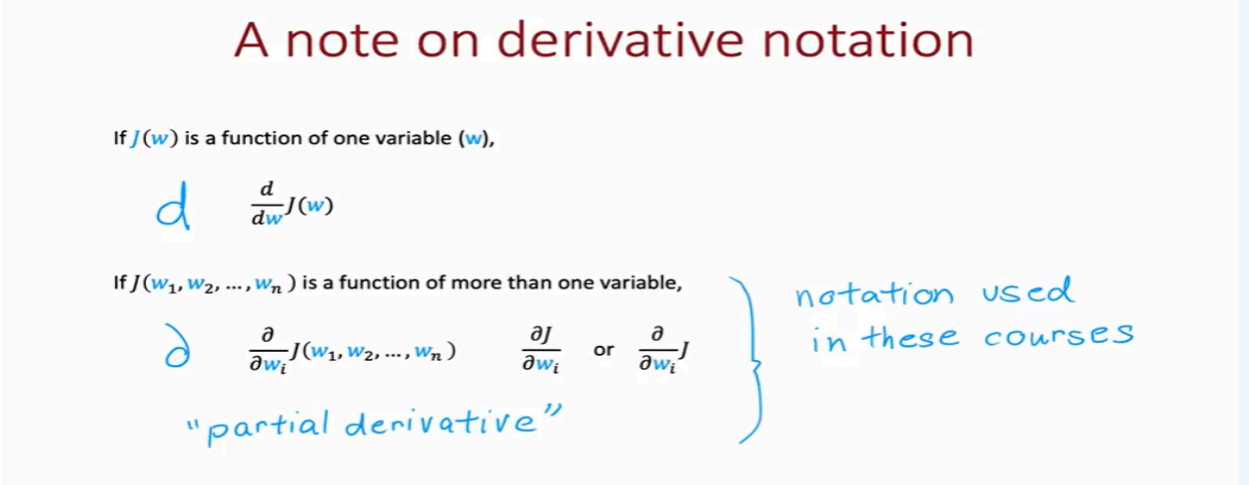
```
model.fit(X,Y,epochs=100)
```

As from_logits=True is here so in the last layer activation =’linear’ has been used so that the softmax calculation is handed over tensorflow and it can do calculation with less error. Here , sparse Categorical Cross entropy is used so softmax function will be used.

Back Propagation:




Here , derivative with respect to w to J(w) simply means that if w goes up epsilon then how much j(w) goes up by some constant and here it is k , so k is derivative.




## 🧠 What is Backpropagation?

**Backpropagation** is the **learning algorithm** used in neural networks to **adjust weights** based on the error (loss). It tells the network:

> **"Hey, here's how much you messed up. Now adjust your weights to improve next time."**

## 🛠️ How It Works (Step-by-Step):

Let's go through this with a **simple example**:
A feedforward neural network with:

- Input layer → Hidden layer → Output layer
- Activation functions (like ReLU or Sigmoid)
- Loss function (like MSE or Cross Entropy)

### Step 1: **Forward Pass**

You give the input $x$ to the network, and it:

1. Calculates outputs at each layer using weights $W$ and biases $b$
2. Applies activation functions
3. Produces a final prediction $\hat{y}$

### Step 2: **Loss Calculation**

The model compares prediction $\hat{y}$ with actual target $y$, using a loss function:

$$\text{Loss} = \mathcal{L}(y, \hat{y})$$

### Step 3: **Backpropagation (Core Part)**

This is where gradients come in. Backpropagation uses **chain rule** from calculus to compute:

> 🔁 **"How much does each weight in the network contribute to the final error?"**

It goes **layer by layer from the end (output) to the start (input)**:

1. **Compute gradient of the loss w.r.t. output**

$$\frac{\partial \mathcal{L}}{\partial \hat{y}}$$

2. **Backpropagate through activation functions and layers** Use chain rule:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial W}$$

3. Do this for **every weight and bias**.

---

### Step 4: **Update Weights (Gradient Descent)**

Now you have gradients $\frac{\partial \mathcal{L}}{\partial W}$, and you update the weights:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}$$

Where $\eta$ is the **learning rate**.

## 🔁 Repeat

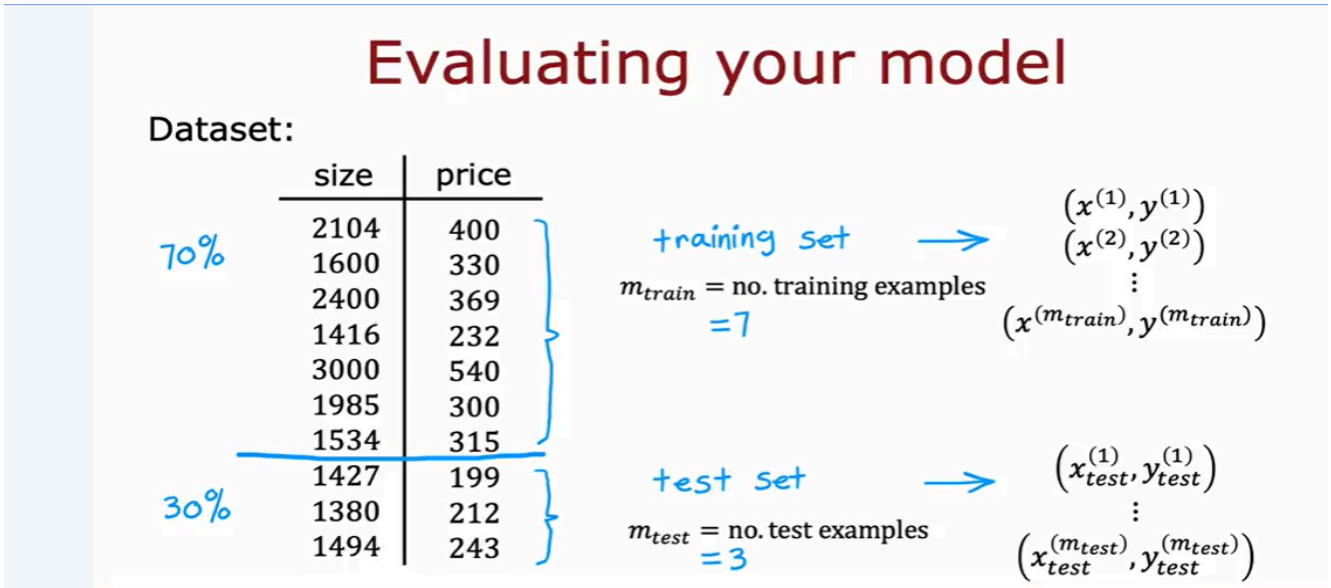
Do this process over and over (called **epochs**) for all your training data — and the network **learns!**

## 💡 Big Picture

In a neural network, **backpropagation** is the algorithm that:

- Calculates **gradients** of the **loss** with respect to all **weights and biases**
- Uses the **chain rule** of calculus to propagate error **from the output to the input**
- Enables **gradient descent** to update weights and reduce loss

Evaluating a model:



# Model Selection in Machine learning:

**Model selection** is the process of **choosing the best algorithm and its best settings (hyperparameters)** for your task.

For example, should you use:

- Logistic Regression?
- Random Forest?
- Support Vector Machine?
- Neural Network?

Each model has **strengths**, **weaknesses**, and **parameters**.

## 🛠️ How do we select?

We compare different models based on performance on **validation data** using techniques like:

- Train/Test Split
- Cross-Validation
- Grid Search / Random Search for hyperparameters
- Evaluation metrics (accuracy, F1-score, RMSE, etc.)

## 📦 Dataset is typically split like this:

| Set | Purpose | Typical % |
|---|---|---|
| **Training set** | Learn model parameters (fit the model) | 60–80% |
| **Validation set** | Tune hyperparameters & select model | 10–20% |
| **Test set** | Final evaluation (never seen before) | 10–20% |

## 4 Model Selection with Cross-Validation: 💡

1. Choose candidate models (e.g., Random Forest, SVM)
2. For each model:
    - Use K-fold CV to estimate performance
3. Select model with **best average CV score**
4. Test on **test set** to check real-world performance

## Bias and Variance in Machine Learning:

In machine learning, **bias** and **variance** are two fundamental sources of error that influence the performance of a model. Understanding how they work helps in building models that generalize well to new, unseen data.

**Bias**: Bias refers to the **error introduced by approximating a real-world problem**, which is often complex, with a simpler model. A high bias typically leads to underfitting.

**Key Points about Bias:**

- **Definition**: Bias is the difference between the average prediction of our model and the true value. A high bias means the model makes strong assumptions and is less flexible.
- **Effect**: High bias can result in underfitting, where the model is too simple to capture the underlying patterns in the data. This leads to poor performance on both training and test data.
- **Example**: A linear regression model used for predicting a highly non-linear relationship (like trying to fit a straight line to data that has a parabolic shape) would have high bias.
- **Model Behavior**: A high-bias model **oversimplifies** the problem, ignoring important relationships in the data.

**Illustration of Bias:**

Imagine fitting a straight line to data that has a curved trend. The line will not capture the curve accurately, resulting in large errors. This is an example of **high bias**.

2. **Variance:** Variance refers to the error introduced by the model's sensitivity to small fluctuations or noise in the training data. A high variance typically leads to overfitting.

Key Points about Variance:

- Definition: Variance is the variability of model predictions for different training datasets. A model with high variance will change significantly when trained on different subsets of the data.

- Effect: High variance can lead to overfitting, where the model becomes too complex and captures the noise or random fluctuations in the training data. As a result, it performs well on the training data but poorly on unseen test data.

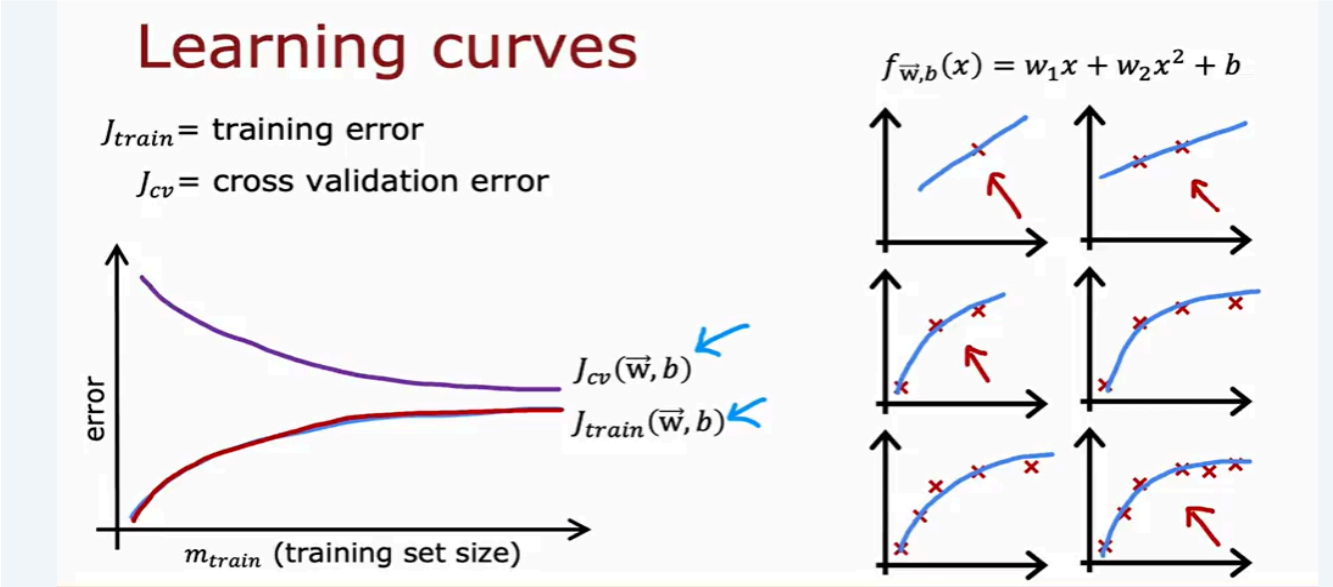
- Example: A decision tree with no pruning that fits perfectly to every point in the training data, even noise or outliers, has high variance.

- Model Behavior: A high-variance model overfits the data, meaning it learns not only the underlying patterns but also the noise in the dataset.

**Illustration of Variance:**

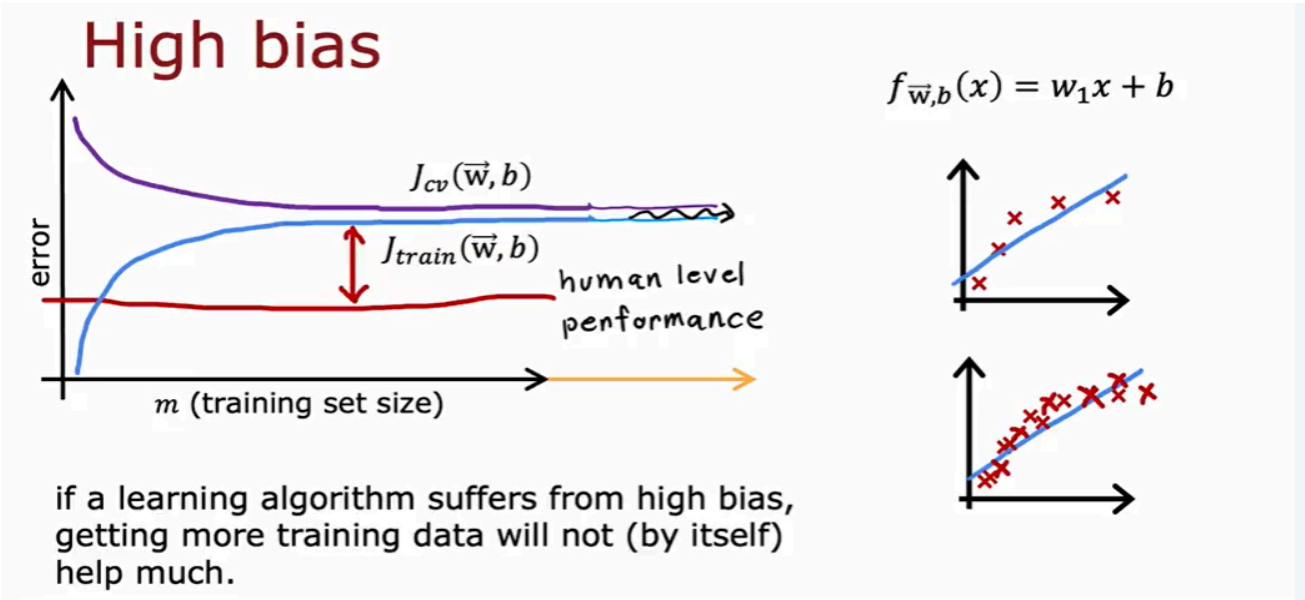
Imagine fitting a model that goes through every single data point in the training set, even the outliers. This model will have low bias (it fits the data perfectly) but high variance (it’s too sensitive to training data, leading to poor generalization to new data).

Learning Curve:




Training Error : **Training error** refers to the error (or difference) between the predicted values and the actual values for the **training dataset**. It is a measure of how well the model performs on the data it was trained on.

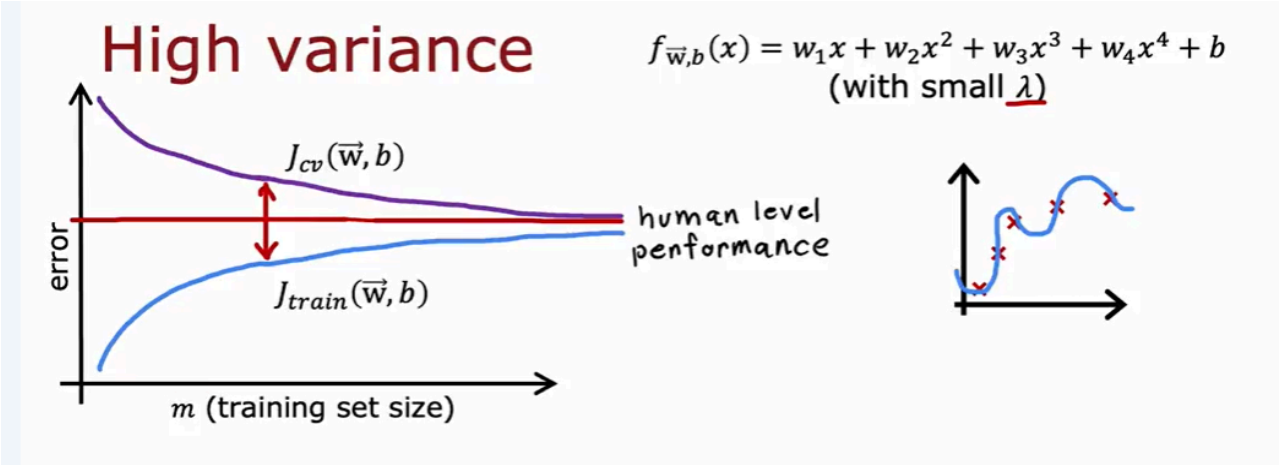
Cross Validation Error: **Cross-validation error** is the error computed using **cross-validation** (e.g., K-fold cross-validation), where the entire dataset is divided into multiple folds, and the model is trained and validated multiple times, each time using a different fold as the validation set.



Bias : In machine learning, **bias** refers to the **error introduced** by making **simplified assumptions** about the data. It is the difference between the predicted values and the true values (ground truth) of the data, reflecting the model's **incorrect assumptions** about the underlying data distribution.

In simpler terms, bias is how much the model **deviates from the actual values** because it assumes the relationship between features and the target is too simplistic.

Variance : In the context of machine learning, **variance** refers to the **sensitivity of a model to small changes in the training data**. It measures how much the model's predictions would change if it were trained on a different subset of the data. High variance means that the model is **overfitting** to the training data, capturing noise and random fluctuations rather than the underlying patterns.




If a learning algorithm suffers from high variance ,getting more training data is likely to help.

Bias and Variance Tradeoff :

# The bias variance tradeoff

$f_{\vec{w},b}(x) = w_1 x + b$

Simple model

High bias

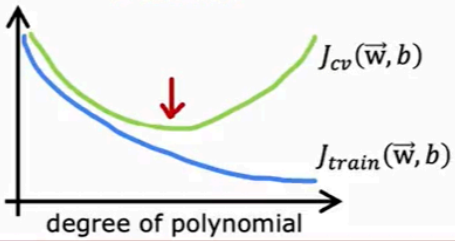
$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$

$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

Complex model

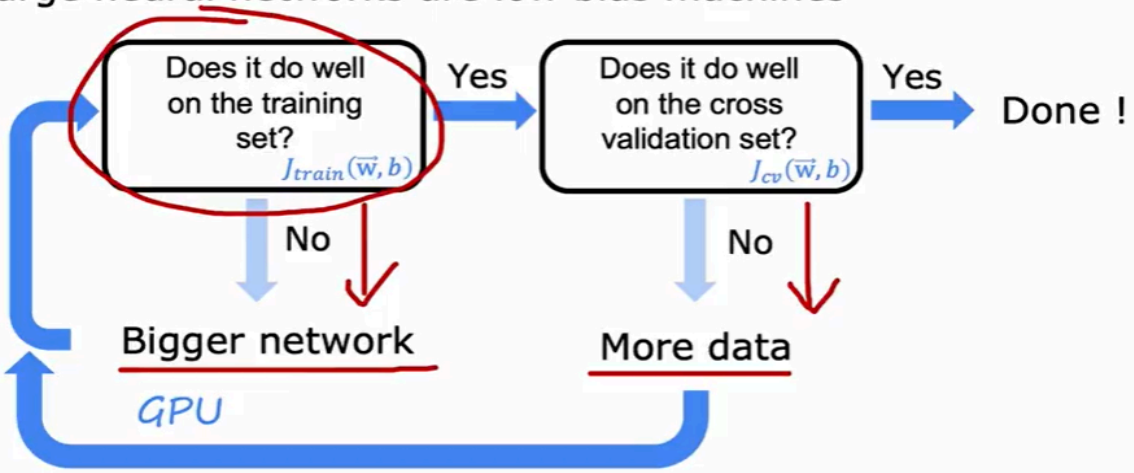
High variance

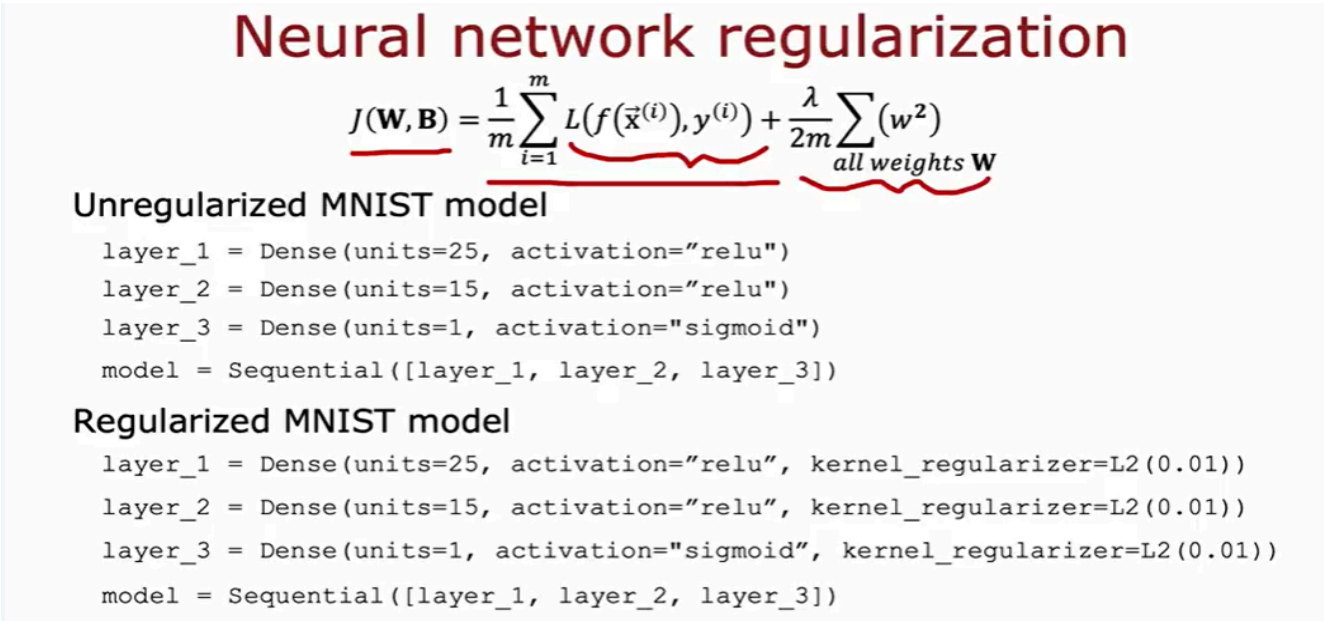
tradeoff




# Neural networks and bias variance

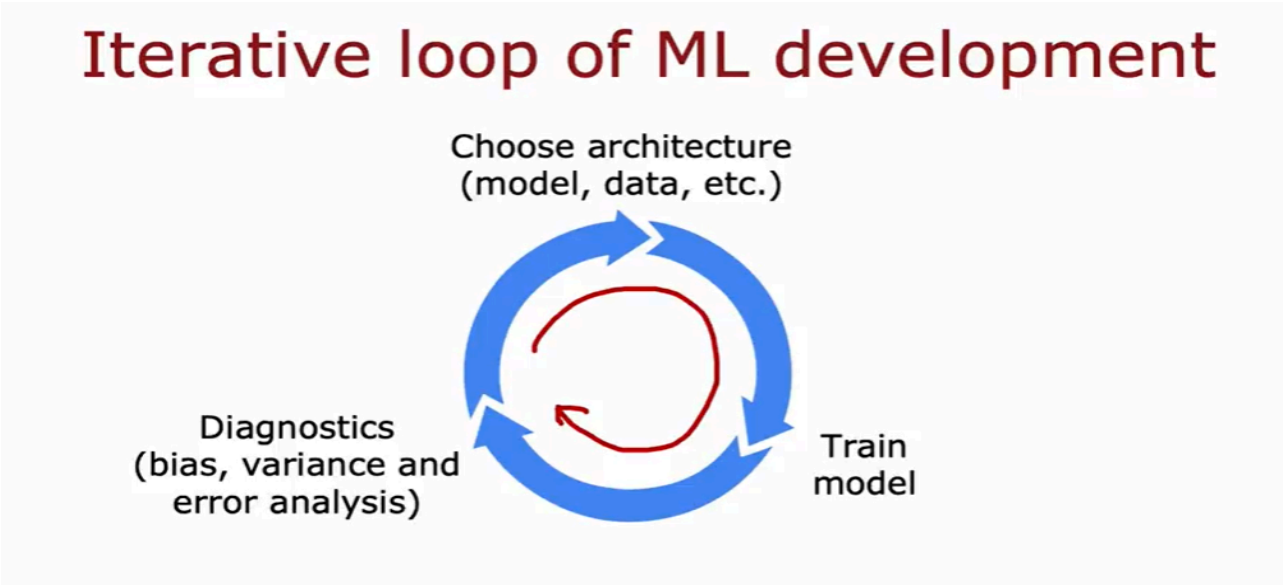
Large neural networks are low bias machines






Neural network regularization techniques are methods used to reduce overfitting and improve the generalization of a model. Overfitting occurs when the model learns to memorize the training data instead of generalizing well to unseen data.

## Iterative Loop of ML Development:




## Error Analysis in Machine Learning :

**Error analysis** is the process of systematically examining where and why your model makes mistakes, with the goal of improving its performance. It's one of the most important tools during the **evaluation and iteration** stages of ML development.

### ✅ Why Do Error Analysis?

- To **understand model limitations**
- To **identify patterns in incorrect predictions**
- To **guide future improvements** (e.g., data quality, features, model choices)

1. **Evaluate Performance Metrics**

Start with standard metrics:

- **Classification:** Accuracy, Precision, Recall, F1 Score, Confusion Matrix
- **Regression:** MAE, MSE, RMSE, $R^2$ Score

> **Look at overall numbers, but don't stop there.**

---

2. **Inspect Misclassified or High-Error Examples**

- **Manual Inspection:** Randomly or systematically go through mispredictions.
- Group errors by:
  - Label type (e.g., confusion between classes)
  - Feature values (e.g., errors mostly occur when `feature_x` is high)
  - Data source or segment (e.g., mobile users, nighttime images)

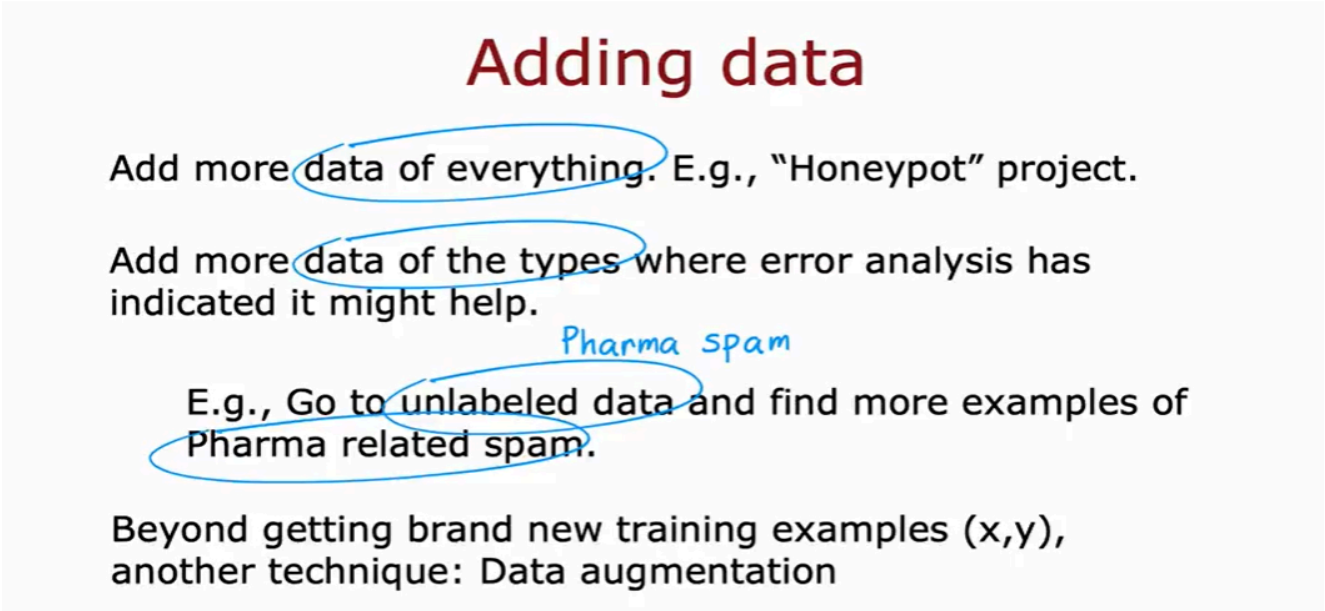
### 3. **Use Confusion Matrix (for classification)**

- Shows how often each class is confused with others
- Helps detect **systematic class confusion**

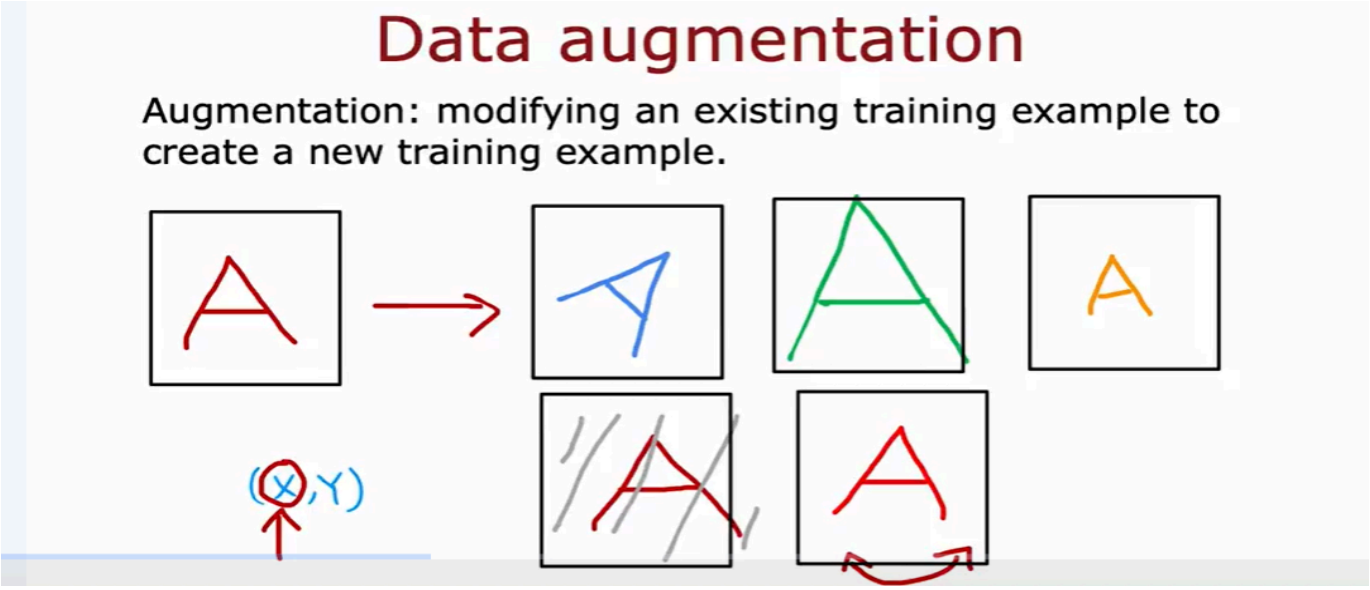
```
plaintext

Predicted →
True ↓      Cat    Dog    Rabbit
Cat          50     5       2
Dog          4      60      10
Rabbit       3      6       40
```

> **High off-diagonal values indicate confusion points.**

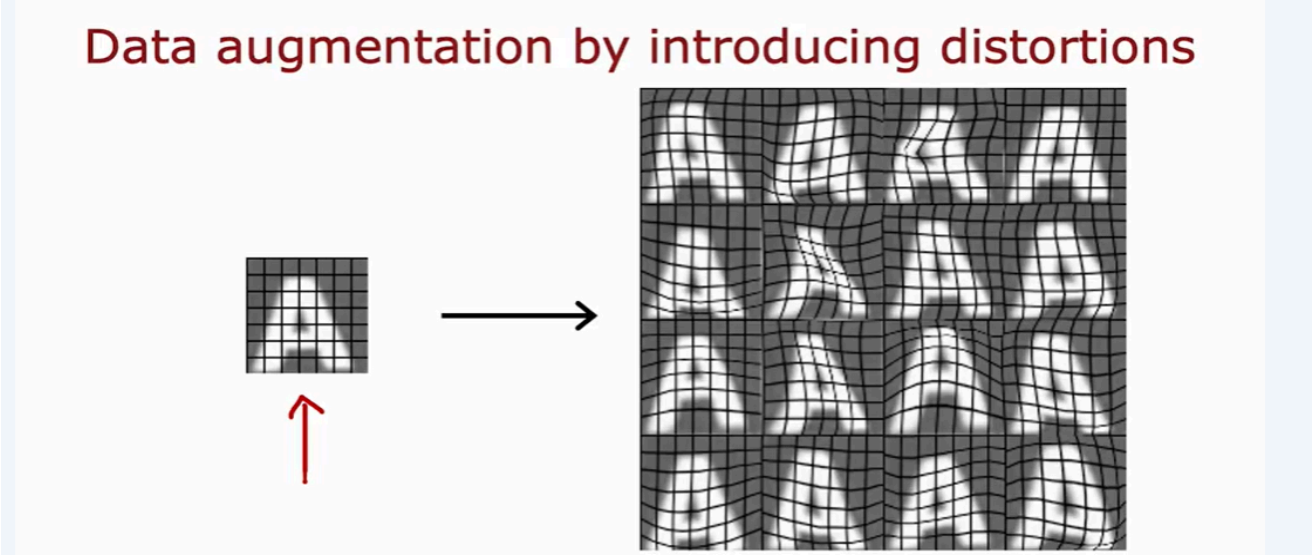
Adding Data:




Data augmentation Specially for audio and image data.




By distortion also data augmentation can be done .




Data augmentation for speech data . Here by adding other background noise

## Data augmentation for speech

Speech recognition example

Original audio (voice search: "What is today's weather?")

+ Noisy background: Crowd

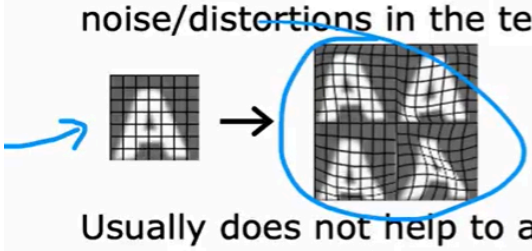
+ Noisy background: Car

+ Audio on bad cellphone connection

Using distortion:

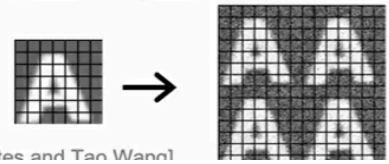
## Data augmentation by introducing distortions

Distortion introduced should be representation of the type of noise/distortions in the test set.



Audio:
Background noise,
bad cellphone connection

Usually does not help to add purely random/meaningless noise to your data.

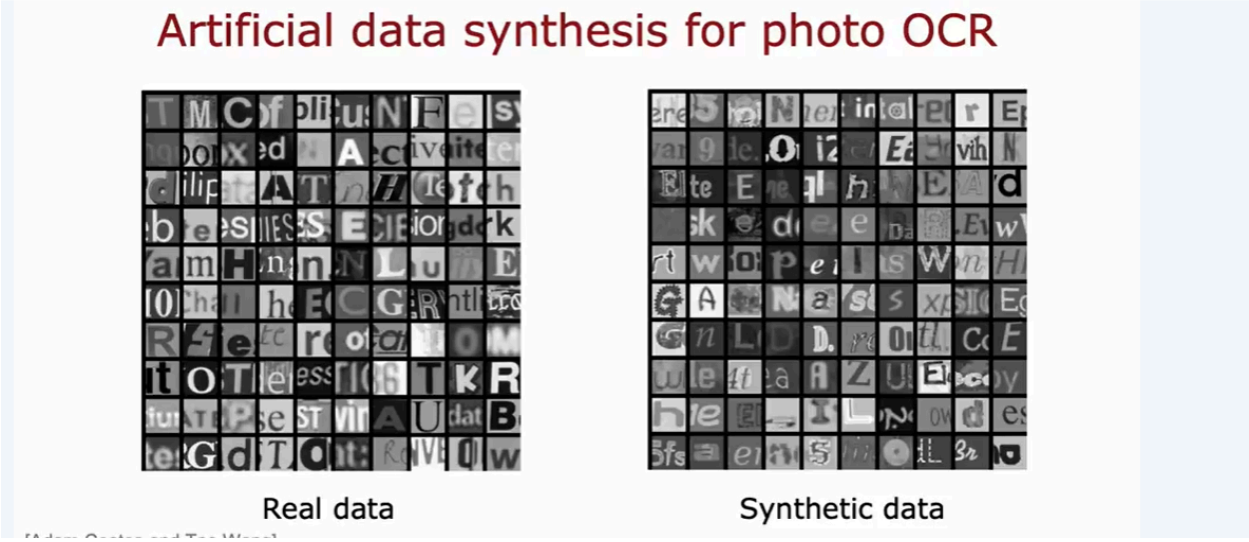


$x_i$=intensity (brightness) of pixel $i$

$x_i \leftarrow x_i$ + random noise

dam Coates and Tao Wang]

Data Synthesis:
This is one kind of approach to augment data.




## Transfer Learning:

**Transfer learning** is a technique where a model developed for one task is **reused (partially or fully)** as the starting point for a model on a second, related task.

It's like using the knowledge gained from learning to ride a cycle to learn how to ride a motorcycle  or bike.

### 🔑 Key Idea

Instead of training a model from scratch, you:

- **Start with a pretrained model** (usually trained on a large dataset like ImageNet, BERT, etc.)
- **Fine-tune it** on your own smaller dataset

## 🧠 When to Use Transfer Learning

- You don't have a lot of labeled data
- Your task is similar to the task the model was pretrained on
- You want faster development and better performance

### 💡 Examples of Pretrained Models

| Domain | Pretrained Models |
| --- | --- |
| Vision | ResNet, VGG, Inception, EfficientNet |
| NLP | BERT, GPT, RoBERTa, T5 |
| Audio | Wav2Vec, YAMNet |
| Time-series | TS-TCC, InceptionTime |

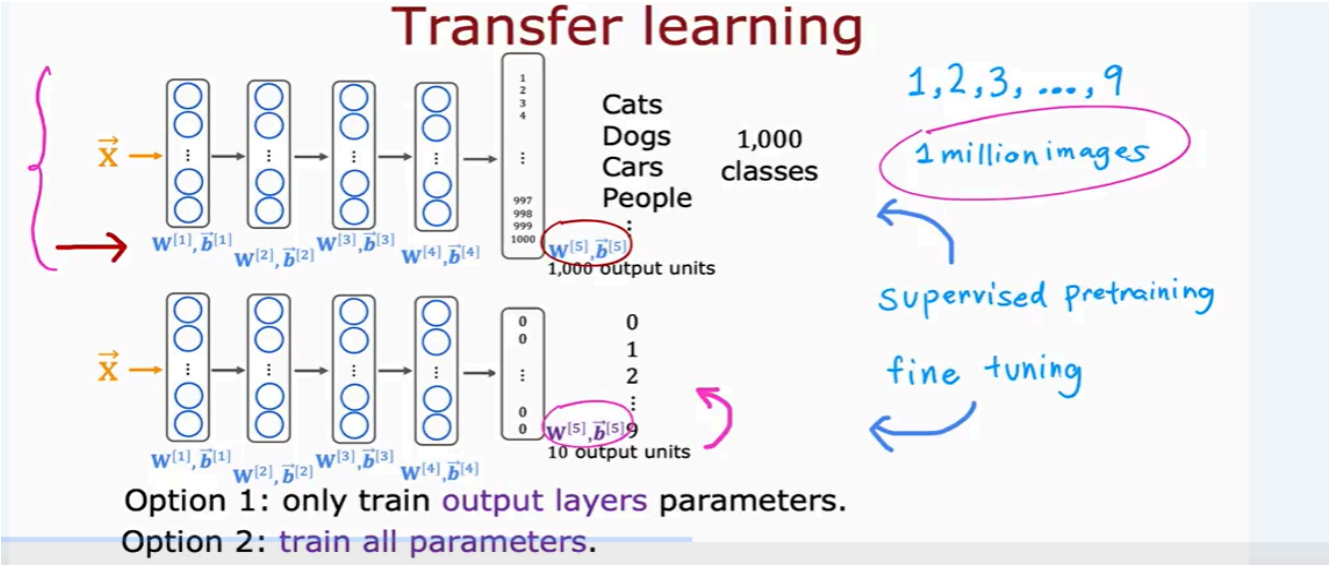
## 🔧 Approaches to Transfer Learning

### 1. **Feature Extraction**

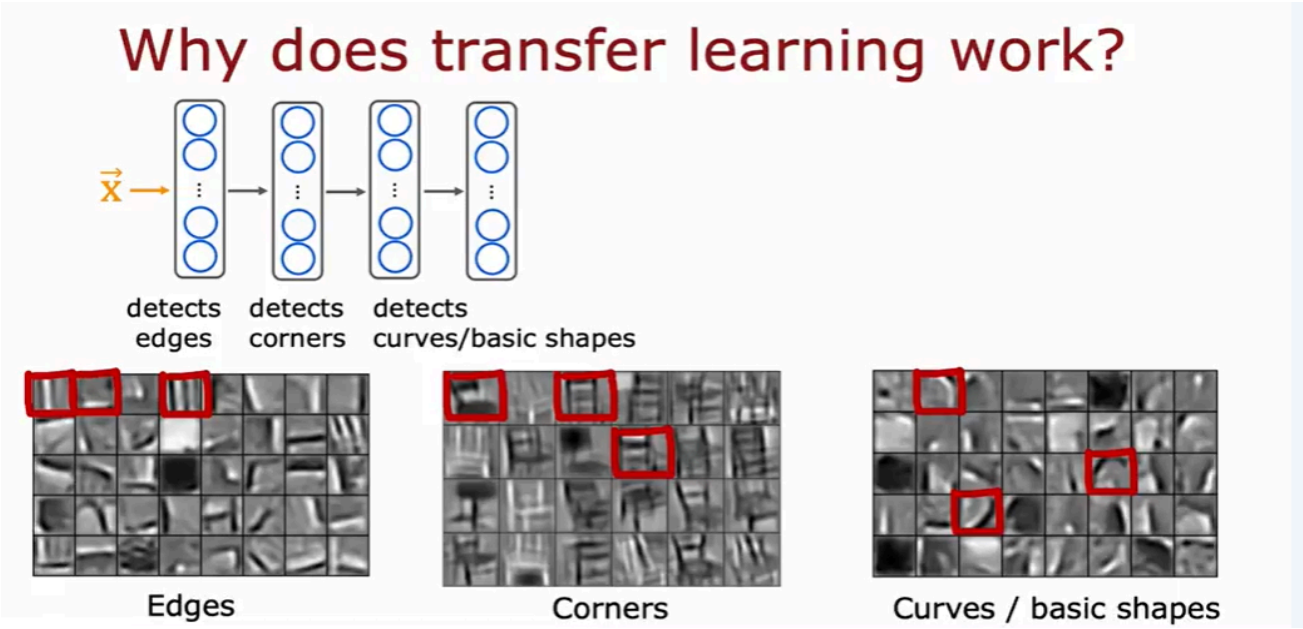
- Freeze all layers except the final one(s)
- Replace the last layer with a new one for your task
- Only train the new layer(s)

2. **Fine-Tuning**

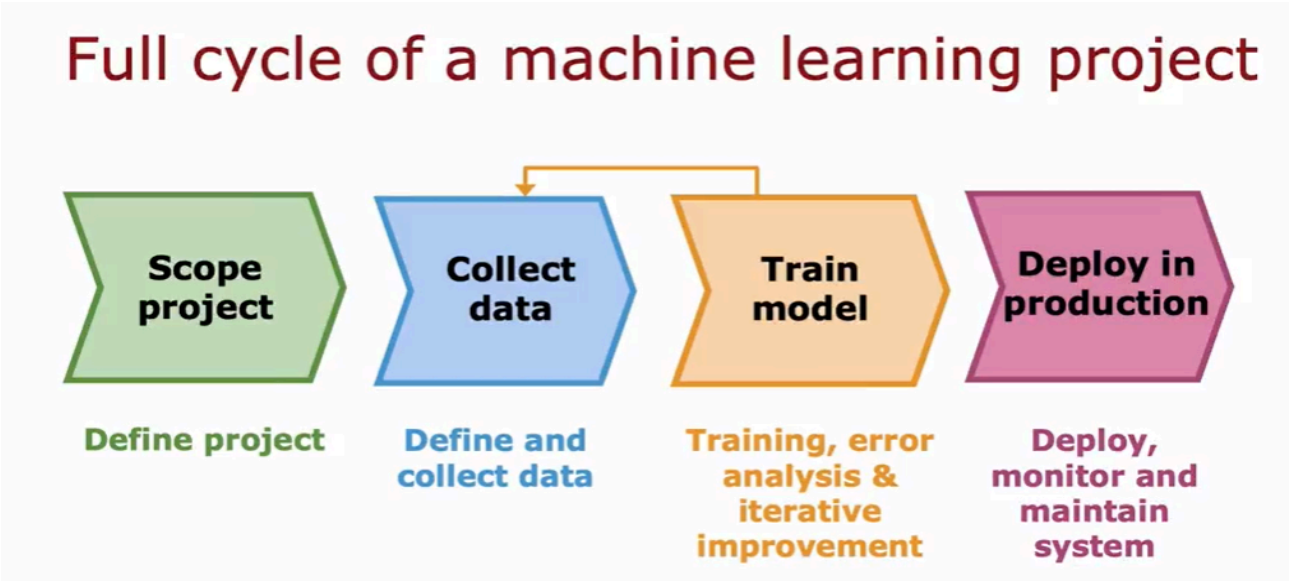
- Load the pretrained model
- **Unfreeze some or all layers**
- Continue training on your new dataset

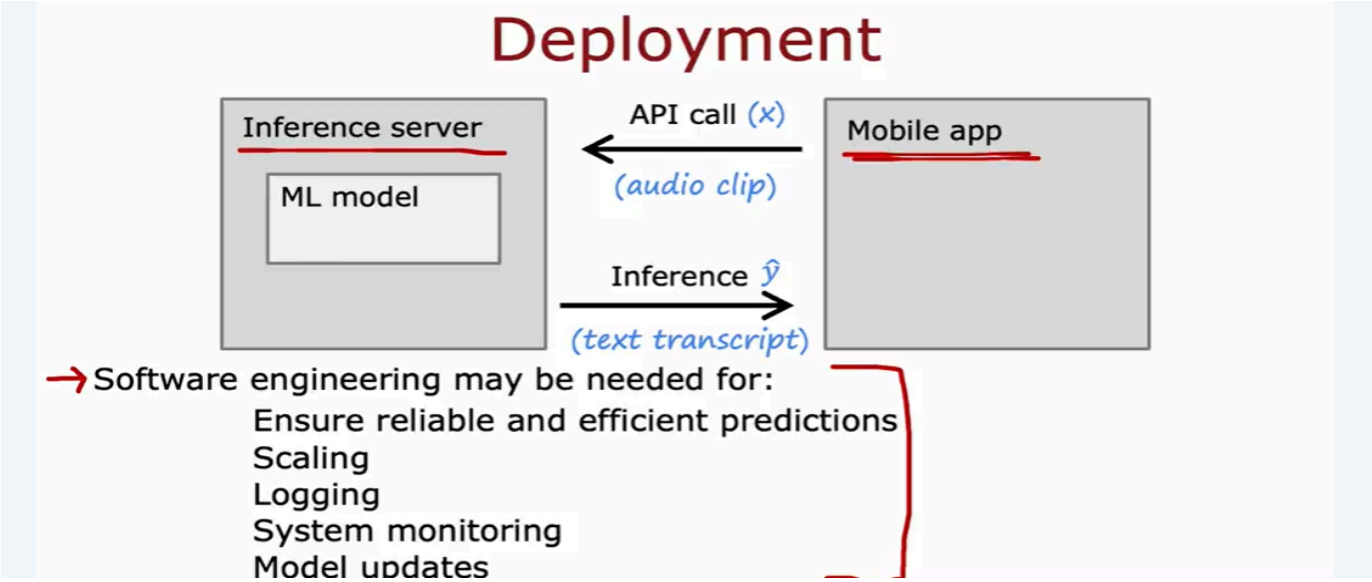



If the training set is smaller then option 1  will be a better approach and otherwise option 2 will be better.




Machine learning development cycle:






## Handling skewed dataset:

When dealing with a **highly skewed dataset** (also called **imbalanced dataset** — where one class is much more frequent than the other), traditional error metrics like **accuracy** can be misleading.
Instead, you should focus on **metrics that better capture the performance of the minority class** (often the more important one).

### ✅ Better Error Metrics for Skewed Data:

**1. Precision**

> **What proportion of positive identifications were actually correct?**

$$\text{Precision} = \frac{TP}{TP + FP}$$

- High precision = fewer **false positives**
- Important when **false alarms** are costly (e.g. cancer diagnosis, fraud detection)

2. **Recall (Sensitivity / True Positive Rate)**

> What proportion of actual positives were correctly identified?

$$\text{Recall} = \frac{TP}{TP + FN}$$

- High recall = fewer **false negatives**
- Important when **missing a positive case** is costly (e.g. disease detection, fire alarms)

3. **F1 Score**

> **Harmonic mean of precision and recall (balances both)**

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- Best when you need a **balance between precision and recall**
- Particularly useful when classes are imbalanced

---

4. **Confusion Matrix**

- Always useful to **manually inspect** how many TP, FP, FN, TN you have.

5. **ROC-AUC (Receiver Operating Characteristic - Area Under Curve)**

> **Measures the ability of the classifier to distinguish between classes at different thresholds.**

- ROC curve plots **True Positive Rate (Recall)** vs. **False Positive Rate**
- AUC closer to 1.0 means better model
- Less useful when positive class is extremely rare

### 6. **PR-AUC (Precision-Recall Curve - Area Under Curve)**

> **Better than ROC-AUC when data is highly imbalanced.**

- Focuses on performance for the **positive class**
- More informative when dealing with rare event detection

### ⚠️ Avoid Using:

❌ **Accuracy**

- Might look very high just because the model always predicts the majority class.
- E.g., in a 99:1 class split, a model that always predicts the majority class will have 99% accuracy but be useless.

---

**Example:**

If you're detecting fraud (very rare), even a model that never detects fraud can have 99% accuracy — but it fails completely.

## What is a Benchmark Dataset?

A **benchmark dataset** is a **standard, publicly available dataset** that is widely used to:

- Evaluate and compare the performance of machine learning (ML) or deep learning (DL) models.
- Serve as a **reference point** or "benchmark" for progress in research or industry.

Think of it like a *universal test* everyone uses to prove how well their model works.

## ✅ Key Features of Benchmark Datasets:

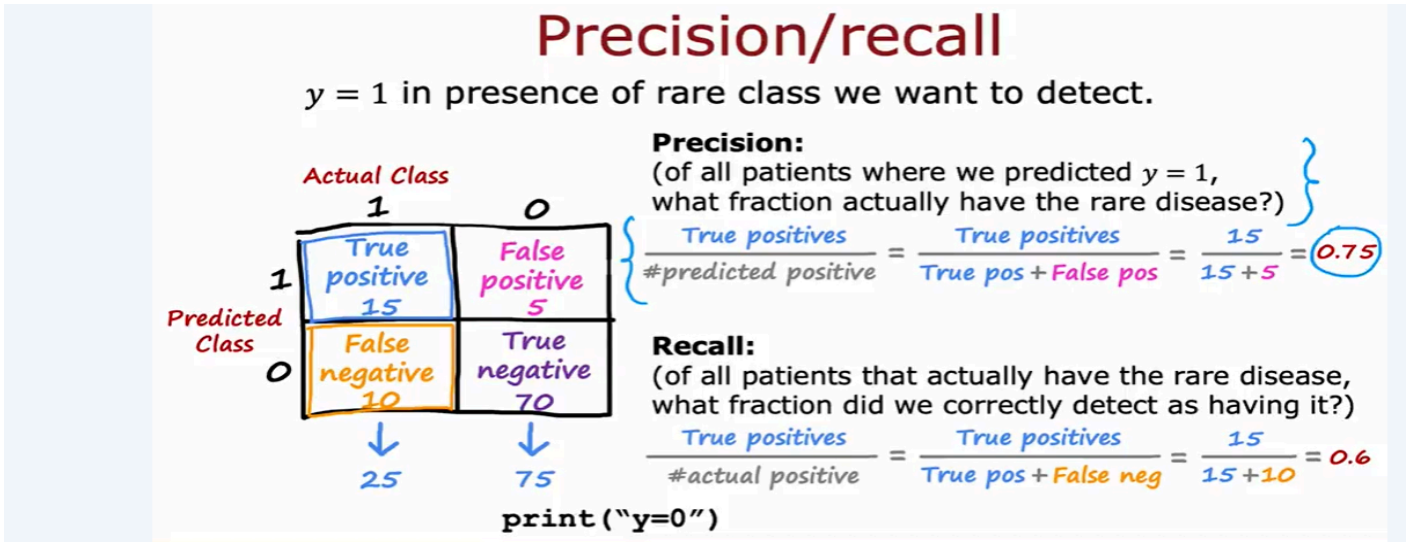
| Feature | Description |
|---|---|
| 📦 **Publicly Available** | Anyone can access and use it for experiments. |
| 🎯 **Task-Specific** | Designed for specific tasks (e.g., classification, detection, translation). |
| 📊 **Standard Metrics** | Comes with defined evaluation metrics (accuracy, F1, BLEU, etc.). |
| 🏆 **Used in Leaderboards** | Often used in competitions (e.g., Kaggle, GLUE, ImageNet Challenge). |
| 🧪 **Fixed Train/Test Splits** | Same splits ensure fair comparison between models. |

## 📚 Examples of Benchmark Datasets by Domain:

| Domain | Dataset | Task |
|---|---|---|
| 🖼️ Computer Vision | ImageNet, CIFAR-10/100, COCO | Image classification, object detection |
| 🧠 NLP | GLUE, SQuAD, IMDB, AG News | Text classification, Q&A, sentiment analysis |
| 📊 Tabular | UCI Adult, Titanic, Credit Card Fraud | Classification, regression |
| 🎤 Audio | LibriSpeech, Speech Commands | Speech recognition, keyword spotting |
| 🌐 Multi-modal | VQA, CLIP, LAION | Visual Q&A, vision-language alignment |

## 🧠 Why Use Benchmark Datasets?

- 🔁 **Reproducibility**: Everyone uses the same data, so results can be fairly compared.
- 🏅 **Progress Tracking**: You can see how your model stacks up against state-of-the-art.
- 🧪 **Standardized Testing**: You don't need to collect your own data to try new ideas.




## Trade off between precision and recall:

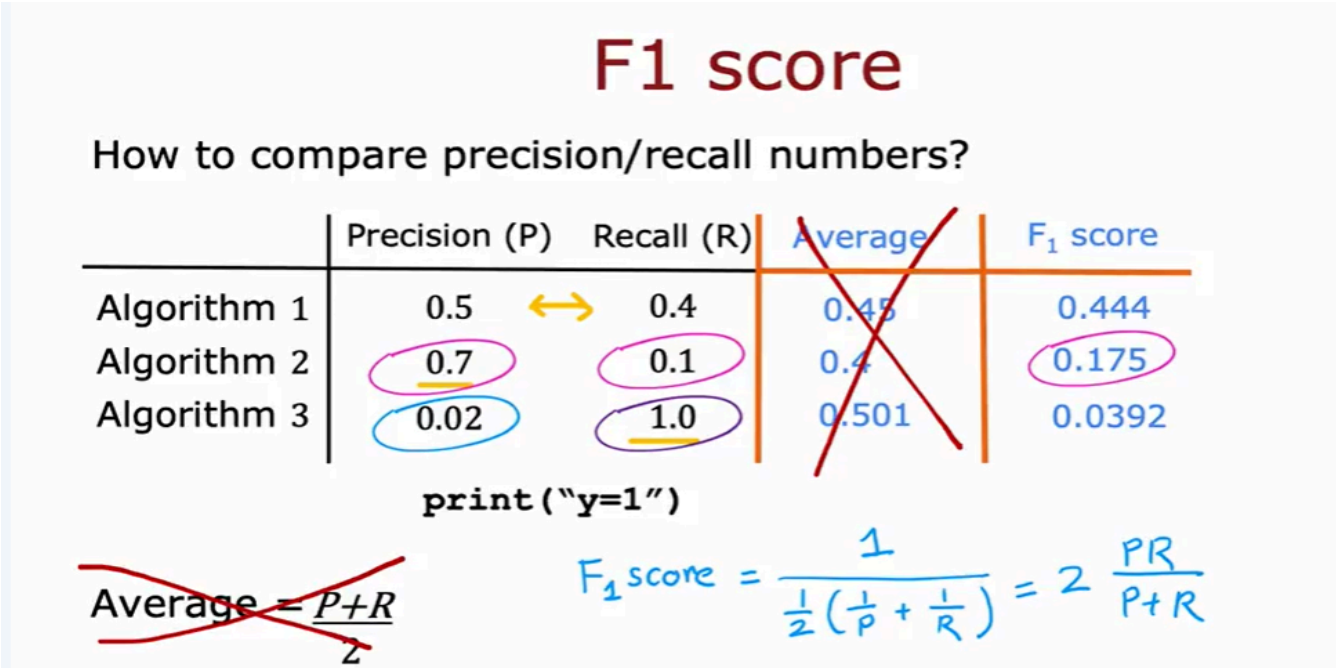
When you try to **increase one**, you often **decrease the other**. This is because they focus on **different types of errors**:

- **Precision** cares about **false positives**
- **Recall** cares about **false negatives**

### 🤝 Want Balance? Use **F1 Score**:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

It's the **harmonic mean** — gives a good balance when classes are skewed and you care about both false positives and false negatives.




F1 score gives a way to trade off between balance precision and recall so Algorithm 1 is better. As F1 score of algo 1 is near of both precision and recall.

## Decision Trees:

A **decision tree** is a **flowchart-like tree structure** where:

- **Each internal node** represents a test on a feature (e.g., "Is age > 30?")
- **Each branch** represents the outcome of that test
- **Each leaf node** represents a class label (for classification) or a value (for regression)

It's like playing **20 Questions** — we keep asking questions to narrow down to a decision.

## How Does a Decision Tree Learn?

The goal is to **split the data** at each step to make the groups **as "pure" as possible** (i.e., all in one class ideally). Here's the learning process step-by-step:

## 🔁 Step-by-Step Learning Process:

1. **Start with the full dataset**

At the root node, all your training data is available.

2. **Choose the best feature to split on**

The model chooses a feature (e.g., "Age", "Income", etc.) that best separates the data.

✅ **How?**

- It uses criteria like:
  - **Gini Impurity** (used in CART)
  - **Entropy** (used in ID3 algorithm — Information Gain)
  - **Variance Reduction** (for regression trees)

3. **Split the dataset**

Partition the dataset based on the selected feature. Each split creates a new branch in the tree.

---

4. **Repeat recursively**

For each subset:

- If it's pure (all examples have same label) → stop and make a leaf.
- If not → repeat the process (select best feature and split again).

5. **Stop when...**

- A maximum depth is reached
- The node has too few samples
- All samples in the node belong to the same class

## 🛠️ How Does It Choose the "Best" Feature?

Let's say we're doing classification:

### 📌 Information Gain (Entropy-based)

$$\text{Information Gain} = \text{Entropy (Parent)} - \sum \text{Weighted Entropy (Children)}$$

- The feature that **maximizes information gain** is selected for the split.

---

### 📌 Gini Impurity (used in CART):

$$\text{Gini}(D) = 1 - \sum_{i=1}^{C} p_i^2$$

Where $p_i$ is the probability of class $i$. Lower Gini = purer split.

### 🧠 Example:

Suppose you're building a tree to decide whether someone buys a phone.

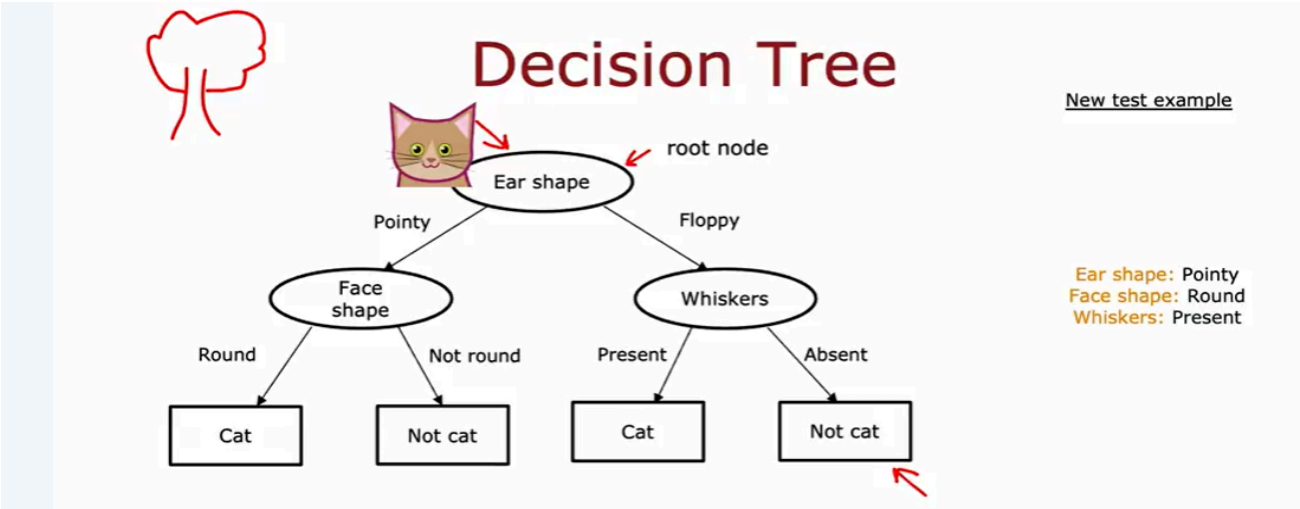
- Root Node: `Age`
  - Is age > 30?
    - Yes → check `Income`
    - No → check `Student`

Each path leads you to a decision like:

- "If age > 30 and income > 50K → Buy"
- "If age ≤ 30 and student = yes → Buy"

## 🔧 Variants and Improvements:

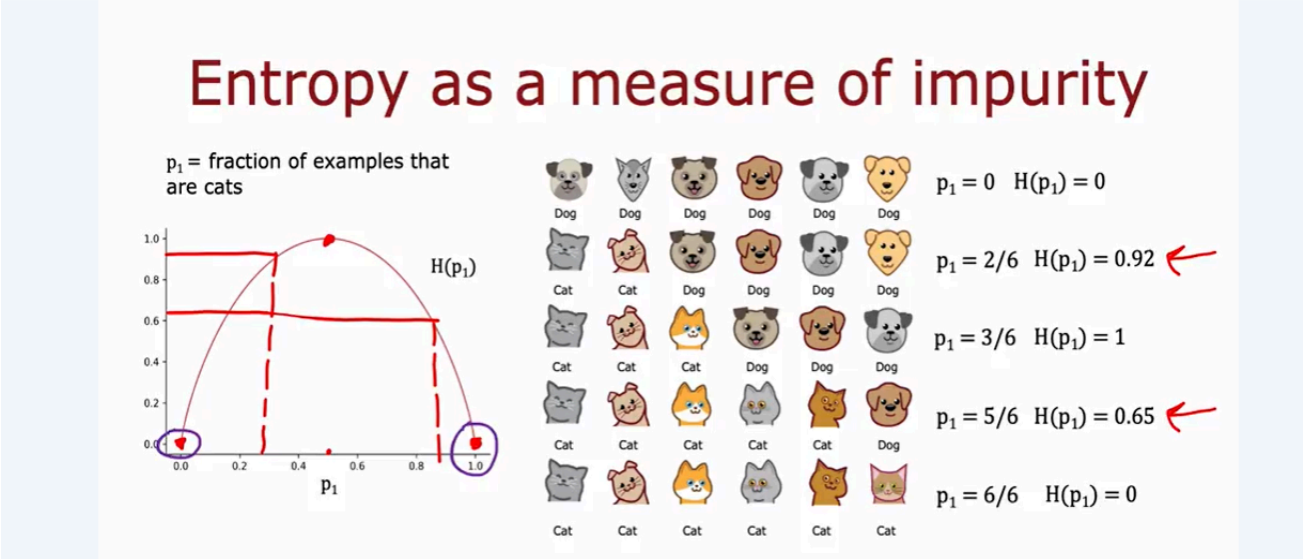
- **Random Forest**: Ensemble of many trees → reduces overfitting
- **Gradient Boosted Trees**: Boosting approach to improve performance
- **Pruning**: Cutting back the tree to avoid overfitting



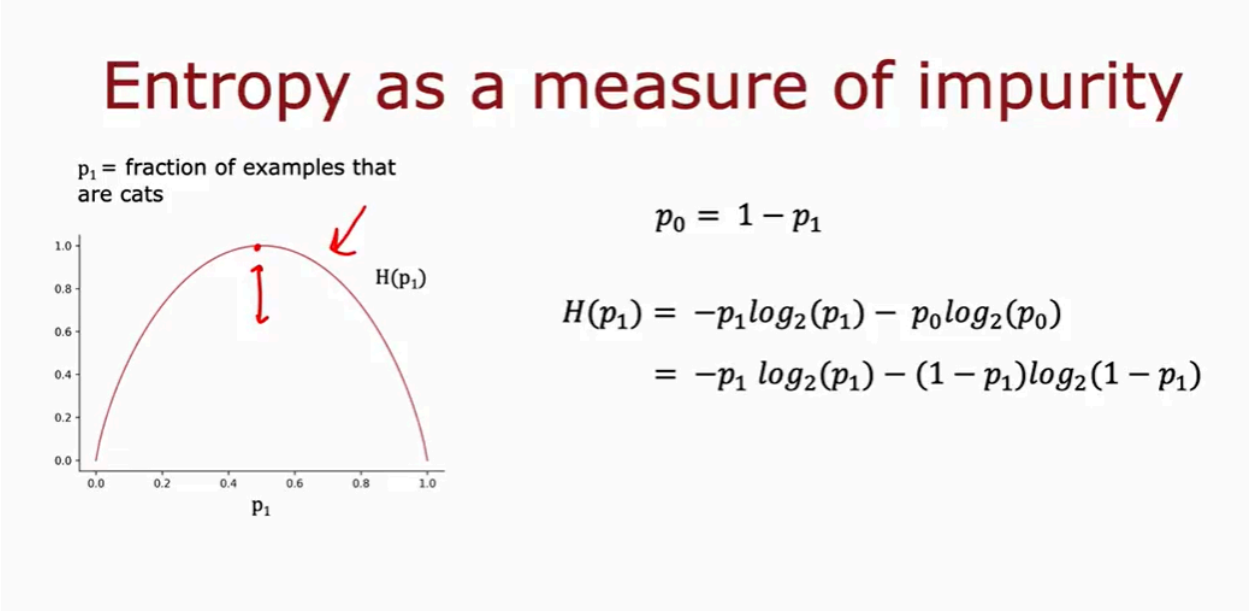
# Decision Tree learning:

Measuring purity:

Here a relation between probability and entropy.



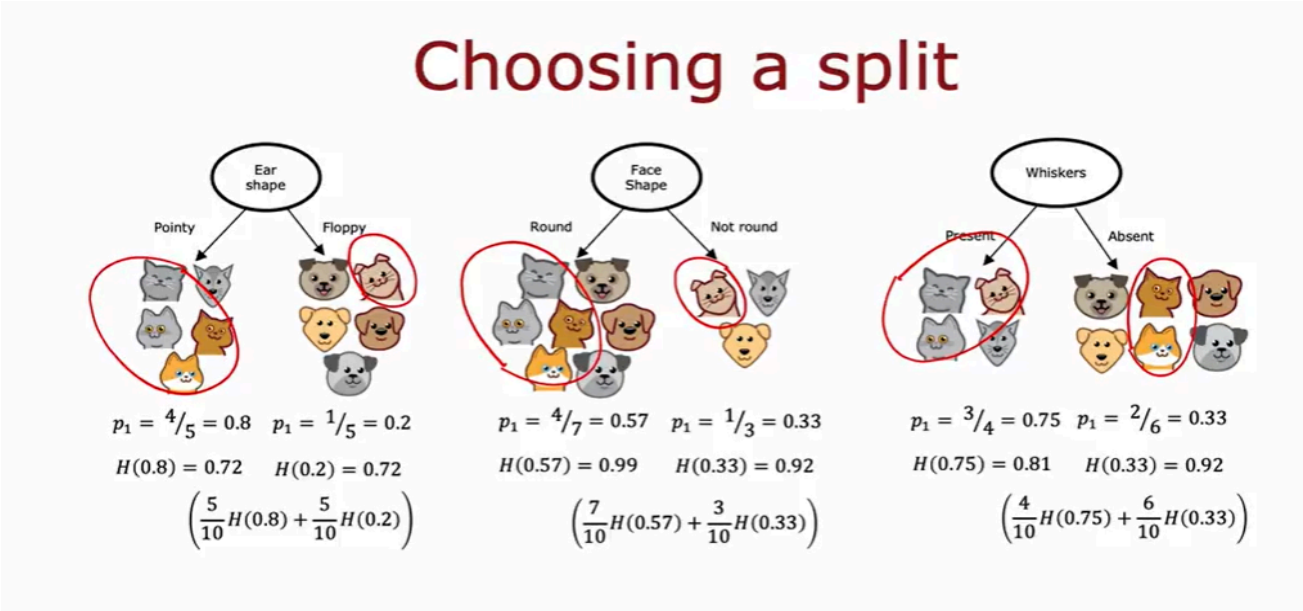

Calculating impurity or entropy



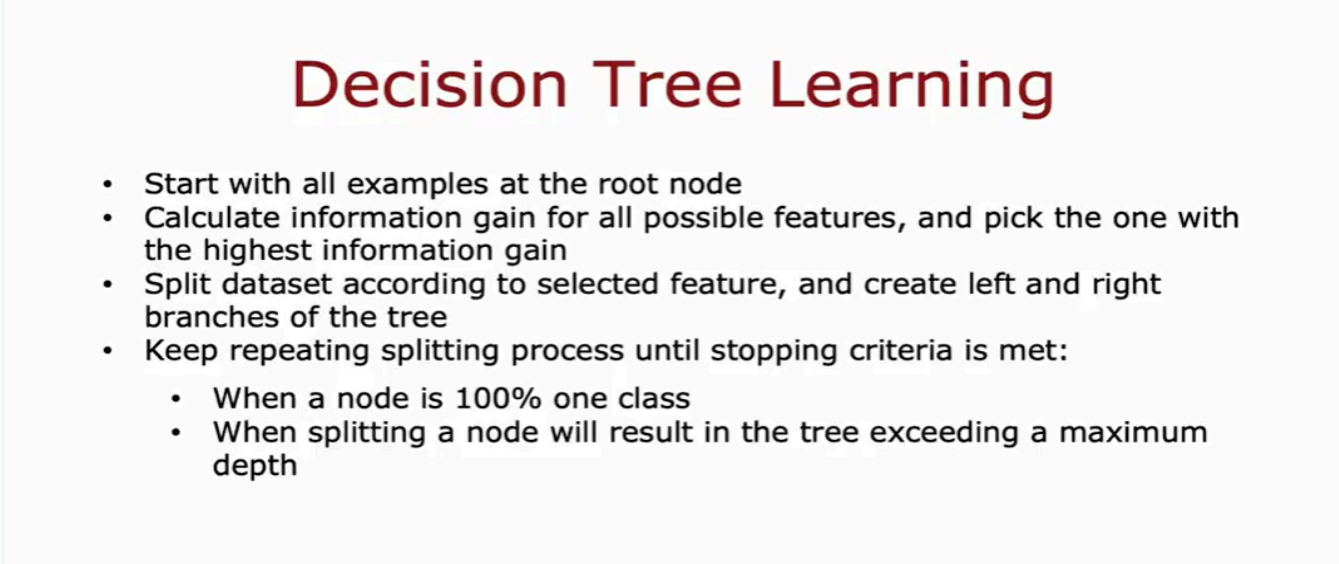

Cats and not cats.

## Choosing Split :

Weighted Entropy Calculation and then subtract them from parent entropy to see which split gains much information, that would be the appropriate splitting node .




H(x)-> entropy under fraction X.



# One hot encoding

If a categorical feature can take on $k$ values, create $k$ binary features (0 or 1 valued).

# One hot encoding and neural networks

| | Pointy ears | Floppy ears | Round ears | Face shape | Whiskers | Cat |
|---|---|---|---|---|---|---|
|  | 1 | 0 | 0 | ~~Round~~ 1 | ~~Present~~ 1 | 1 |
|  | 0 | 0 | 1 | ~~Not round~~ 0 | ~~Present~~ 1 | 1 |
|  | 0 | 0 | 1 | ~~Round~~ 1 | ~~Absent~~ 0 | 0 |
|  | 1 | 0 | 0 | ~~Not round~~ 0 | ~~Present~~ 1 | 0 |
|  | 0 | 0 | 1 | ~~Round~~ 1 | ~~Present~~ 1 | 1 |
|  | 1 | 0 | 0 | ~~Round~~ 1 | ~~Absent~~ 0 | 1 |
|  | 0 | 1 | 0 | ~~Not round~~ 0 | ~~Absent~~ 0 | 1 |
|  | 0 | 0 | 1 | ~~Round~~ 1 | ~~Absent~~ 0 | 1 |
|  | 0 | 1 | 0 | ~~Round~~ 1 | ~~Absent~~ 0 | 1 |
|  | 0 | 1 | 0 | ~~Round~~ 1 | ~~Absent~~ 0 | 1 |

**One-hot encoding** is a method used to **convert categorical (non-numeric) data into numeric format** so that machine learning algorithms can work with it.

### 📦 Example:

Suppose you have a feature:

`Color = [Red, Green, Blue]`

One-hot encoding will turn this into:

| Color | Red | Green | Blue |
|---|---|---|---|
| Red | 1 | 0 | 0 |
| Green | 0 | 1 | 0 |
| Blue | 0 | 0 | 1 |

## 🤔 Why Do We Use It?

Most machine learning models **don't understand text** and might misinterpret **label encoding** (e.g., Red=1, Green=2, Blue=3) as implying order or magnitude.

One-hot encoding avoids this by **removing any notion of order or ranking** in the categories.

## 📉 Can Decision Trees Handle Continuous Features?

✅ **Yes!** Decision trees are **very good at handling continuous (numeric) features**. They **automatically find the best thresholds** to split the data.

## 🧠 How It Works (Step-by-Step)

Suppose you have a continuous feature like `Age`.

1. **Sort all unique values** of the feature
    - Example: Age = [18, 22, 25, 29, 31, 35]
2. For each pair of adjacent values, compute a candidate split point:
    - E.g., (18+22)/2 = 20, (22+25)/2 = 23.5, etc.
3. For each candidate threshold, split the data into:
    - Left: samples where `Age <= threshold`
    - Right: samples where `Age > threshold`
4. For each split, calculate:
    - 🔷 **Information Gain** (if using entropy)
    - 🔷 **Gini Impurity** (for classification)
    - 🔷 **MSE / MAE** (for regression)
5. **Choose the threshold** that gives the **best score (lowest impurity or error)**

📊 **Example (Classification):**

| Age | Buy? |
|---|---|
| 22 | No |
| 25 | Yes |
| 29 | No |
| 31 | Yes |

- Try splitting on Age ≤ 23.5:
  - Left: [22] → No
  - Right: [25, 29, 31] → [Yes, No, Yes]

Then calculate the impurity (e.g., Gini or entropy) and repeat for other thresholds.

## 📈 Example (Regression):

| Age | Salary |
|---|---|
| 22 | 30k |
| 25 | 45k |
| 29 | 50k |
| 31 | 55k |

- Try split on Age ≤ 27:
  - Left mean = avg(30k, 45k) = 37.5k
  - Right mean = avg(50k, 55k) = 52.5k
  - Compute total **MSE**

## 🧰 In `sklearn` (Automatic!)

You don't need to manually do any of that — just feed in continuous features:

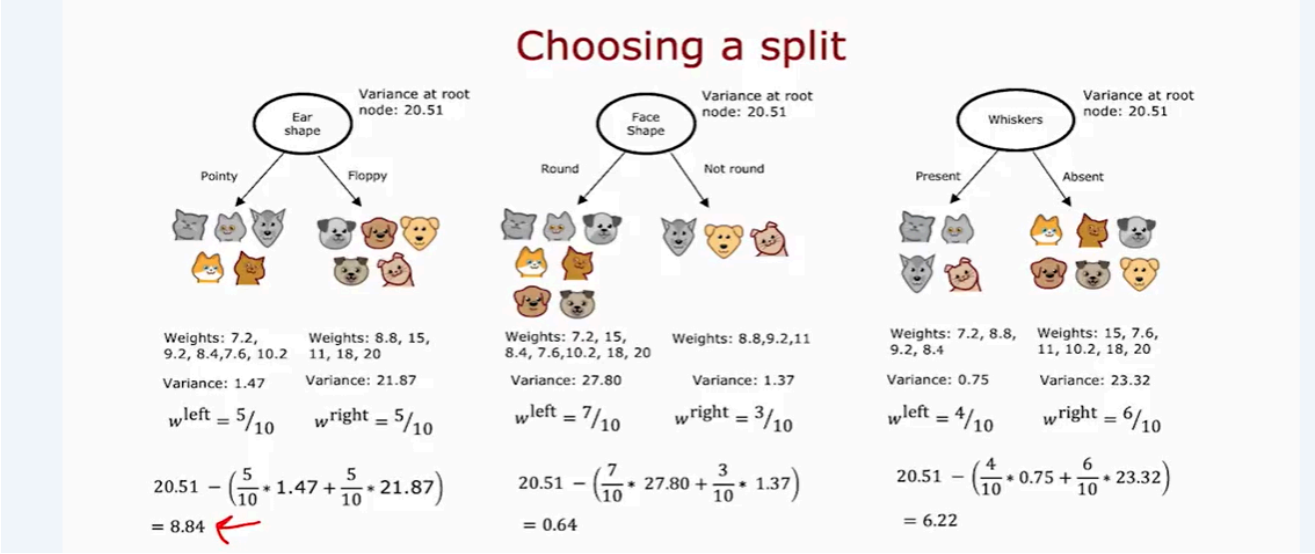
```python
from sklearn.tree import DecisionTreeRegressor

X = [[22], [25], [29], [31]]
y = [30000, 45000, 50000, 55000]

model = DecisionTreeRegressor()
model.fit(X, y)
```

The tree will **automatically pick the best split values** based on MSE.

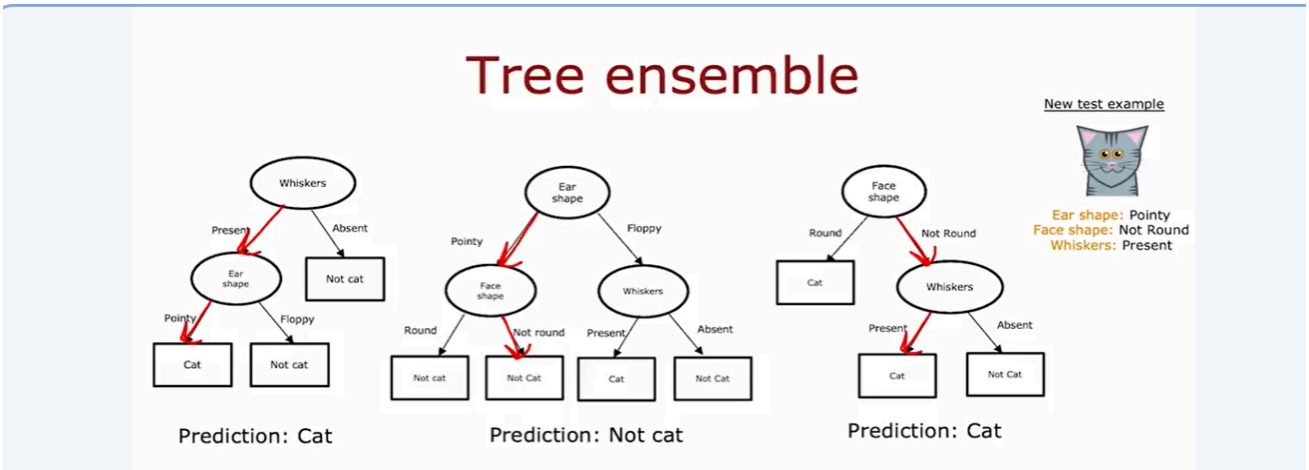
---

## ✅ Advantages

- No need to normalize or scale
- Non-linear splits are easily captured
- Can mix categorical + continuous features



# Tree Ensemble:




A **Tree Ensemble** is a **collection of multiple decision trees** whose predictions are **combined** to make a stronger, more accurate model.

Think of it as:

> “A forest of trees is smarter than a single tree.”

## 📦 Why Use Tree Ensembles?

Individual decision trees are:

- ✅ Fast and interpretable
- ❌ But often **prone to overfitting**

So we **combine many trees** to:

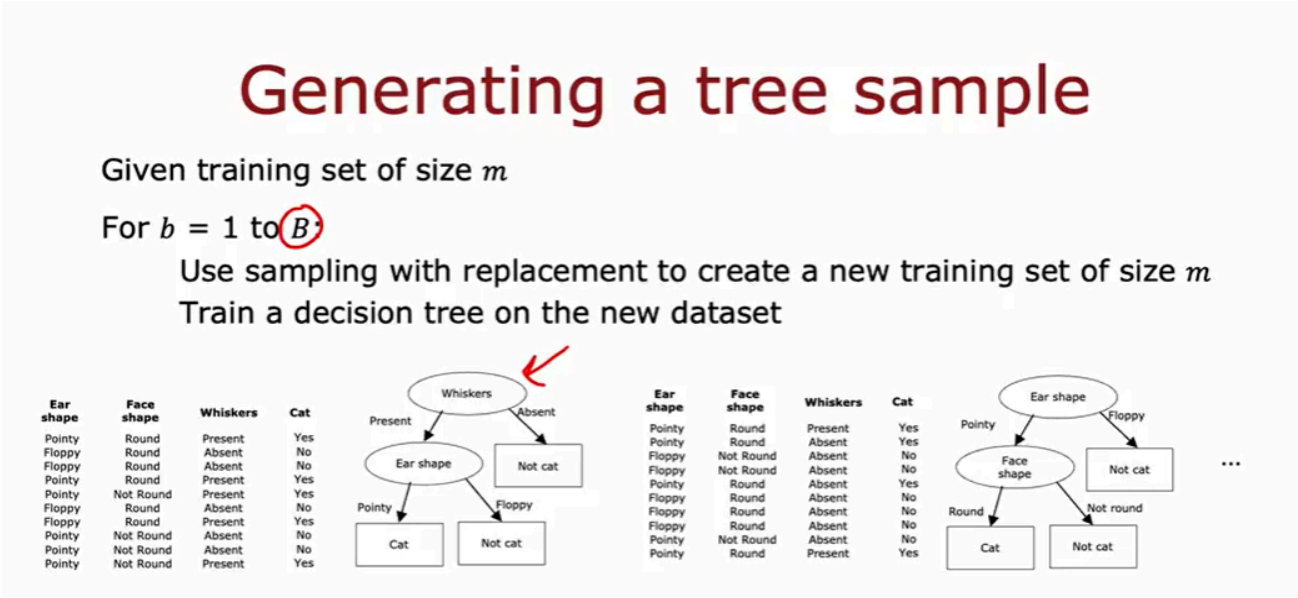
- Reduce variance
- Improve accuracy
- Generalize better

## 🧠 Two Main Types of Tree Ensembles

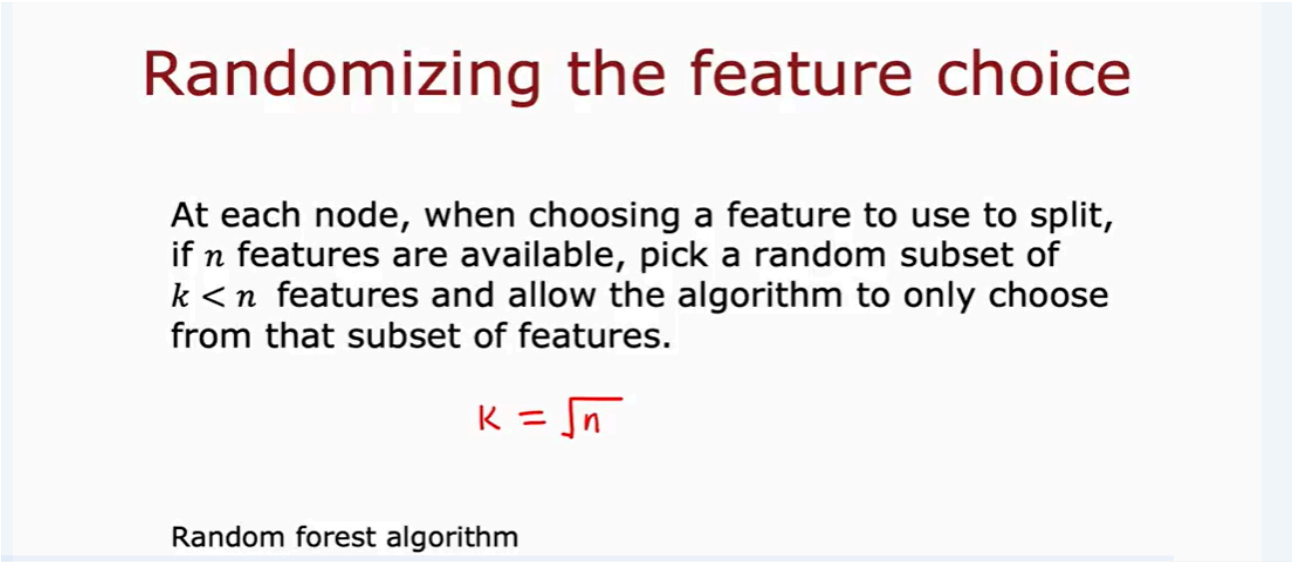
| Method | Idea | Boosts What? | Example |
|---|---|---|---|
| 🟩 Bagging | Train multiple trees independently on random samples | **Stability** (reduces variance) | **Random Forest** |
| 🟨 Boosting | Train trees sequentially, each one correcting the previous | **Accuracy** (reduces bias) | **Gradient Boosting, XGBoost, LightGBM** |

## 🌲 Random Forest (Bagging)

- Trains **many decision trees** on different **bootstrapped subsets** of the data
- At each split, chooses a random subset of features
- Combines results using:
  - **Majority vote** (classification)
  - **Average** (regression)




To make the trees in random forest different from each other a modification can be done




✅ **Strengths:**

- Great performance
- Less overfitting than a single tree
- Easy to use, few parameters to tune

python Copy Edit

```python
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)
```

## 🔥 Gradient Boosting (Boosting)

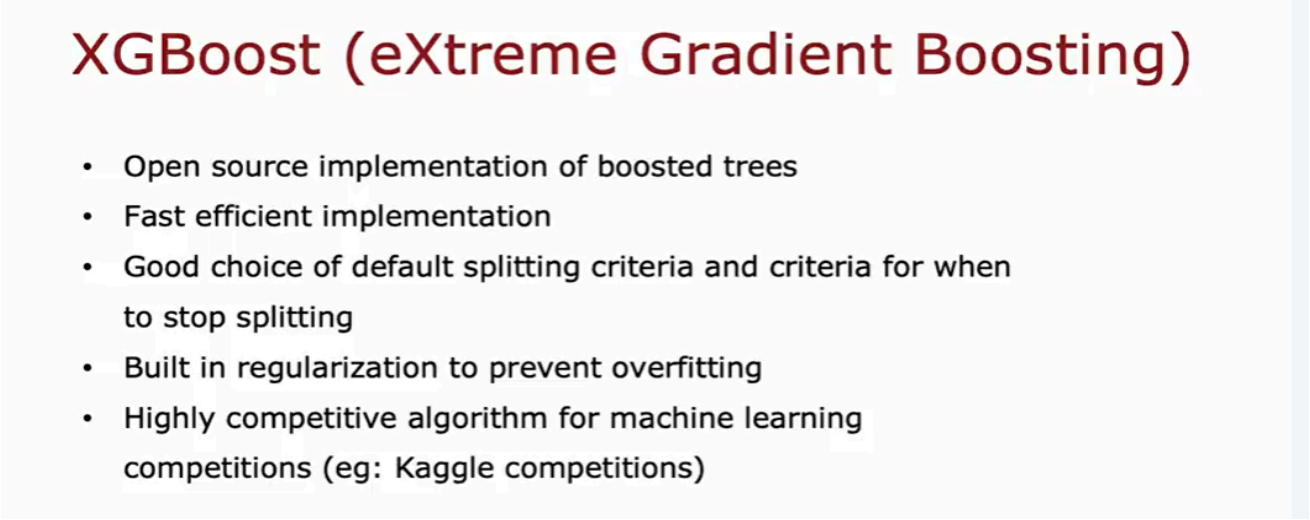
- Trains trees **sequentially**
- Each tree tries to fix the **errors** made by the previous ones
- Learns from **residuals (mistakes)**

Popular implementations:

- `GradientBoostingClassifier` (sklearn)
- `XGBoost` 🚀
- `LightGBM` ⚡
- `CatBoost` 😺

python Copy Edit

```python
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1)
model.fit(X_train, y_train)
```



# Using XGBoost

## Classification

```python
from xgboost import XGBClassifier

model = XGBClassifier()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

## Regression

```python
from xgboost import XGBRegressor

model = XGBRegressor()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

## ⚖️ Comparison: Random Forest vs Boosting

| Feature | Random Forest | Gradient Boosting |
|---|---|---|
| Trees | Independent | Sequential |
| Speed | Faster | Slower |
| Tuning | Less needed | More tuning |
| Bias | Higher | Lower |
| Variance | Lower | May be higher |
| Performance | Very good | Often better |

## 💡 Summary

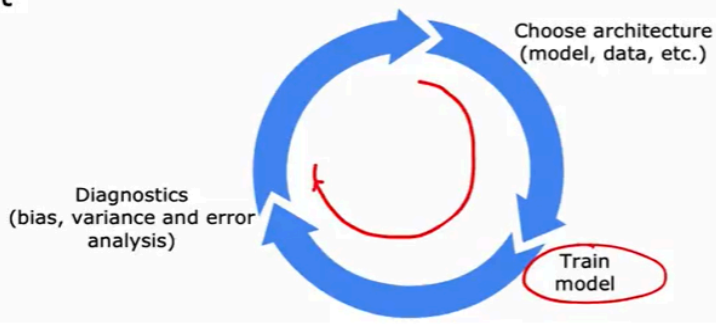
- 🌳 **Tree Ensemble = group of decision trees working together**
- 🟩 **Bagging (Random Forest)** = reduce overfitting
- 🟨 **Boosting (XGBoost, LightGBM)** = correct mistakes, more accurate

When to use decision trees or Decision tree vs Neural network:

## Decision Trees vs Neural Networks

**Decision Trees and Tree ensembles**

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, audio, text)
- Fast




## Decision Trees vs Neural Networks

**Decision Trees and Tree ensembles**

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, audio, text)
- Fast
- Small decision trees may be human interpretable

# Decision Trees vs Neural Networks

**Decision Trees and Tree ensembles**

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, audio, text)
- Fast
- Small decision trees may be human interpretable

**Neural Networks**

- Works well on all types of data, including tabular (structured) and unstructured data
- May be slower than a decision tree
- Works with transfer learning
- When building a system of multiple models working together, it might be easier to string together multiple neural networks