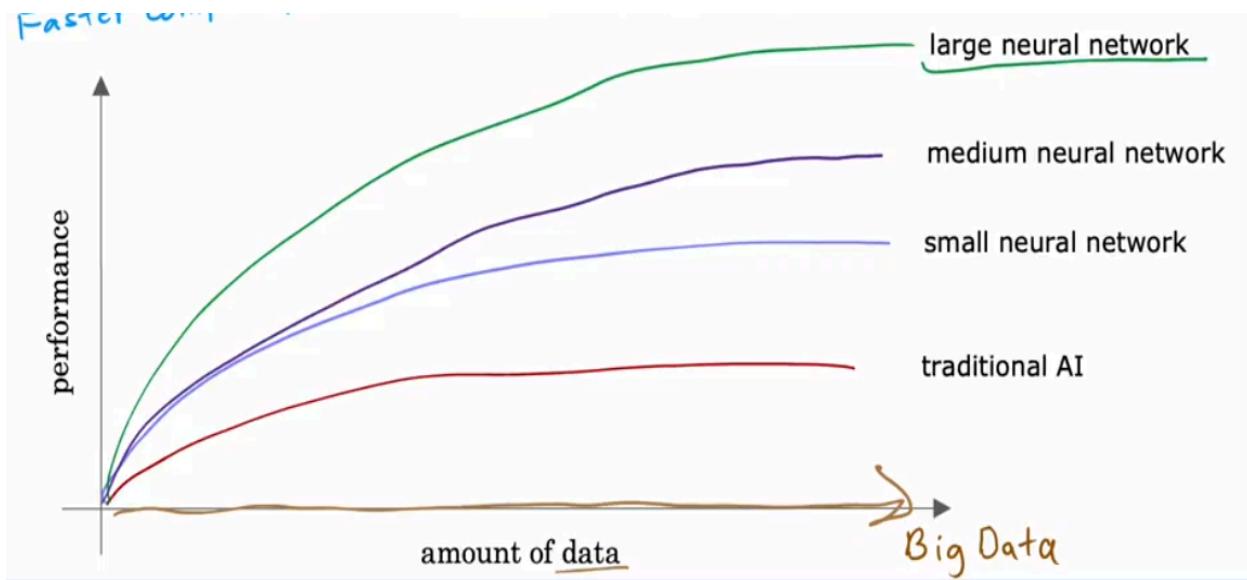


Advanced Learning Algorithm

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.

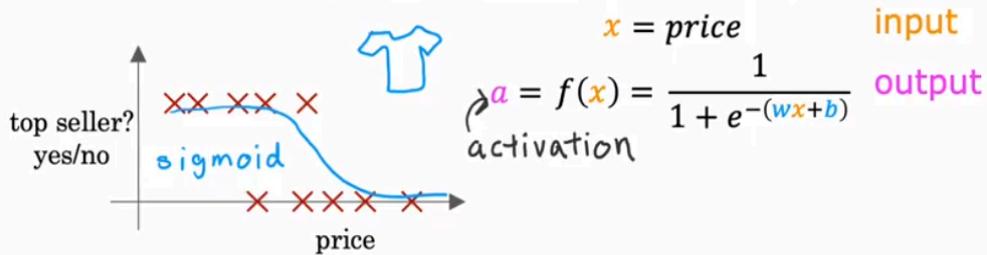


This image represents the relationship between **performance** and the **amount of data** for different types of AI models. The x-axis represents the **amount of data** (increasing towards "Big Data"), while the y-axis represents **performance** (e.g., accuracy, efficiency).

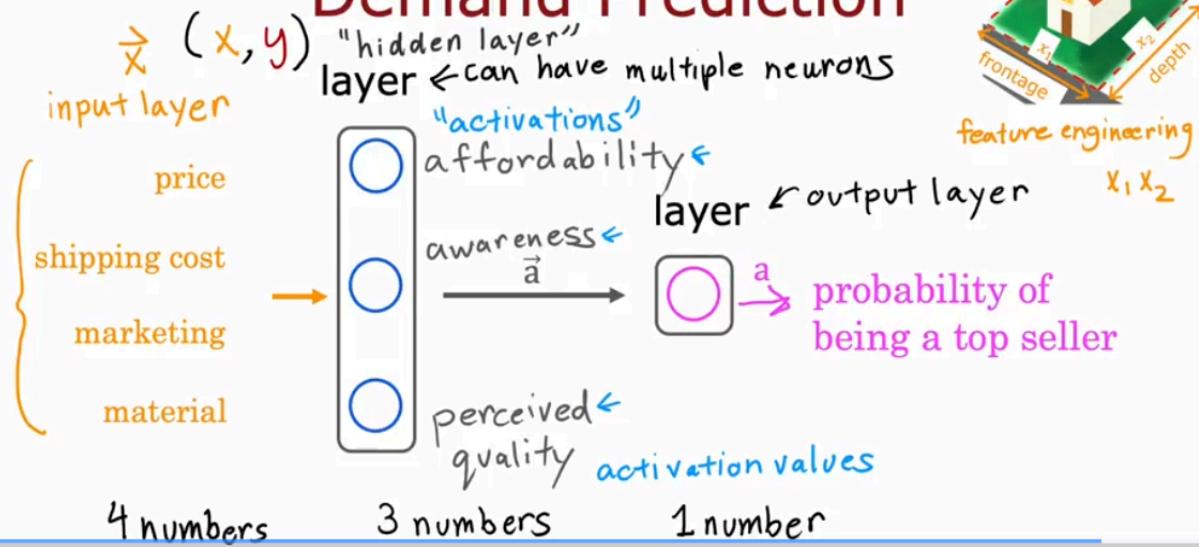
Main Takeaways:

- 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.

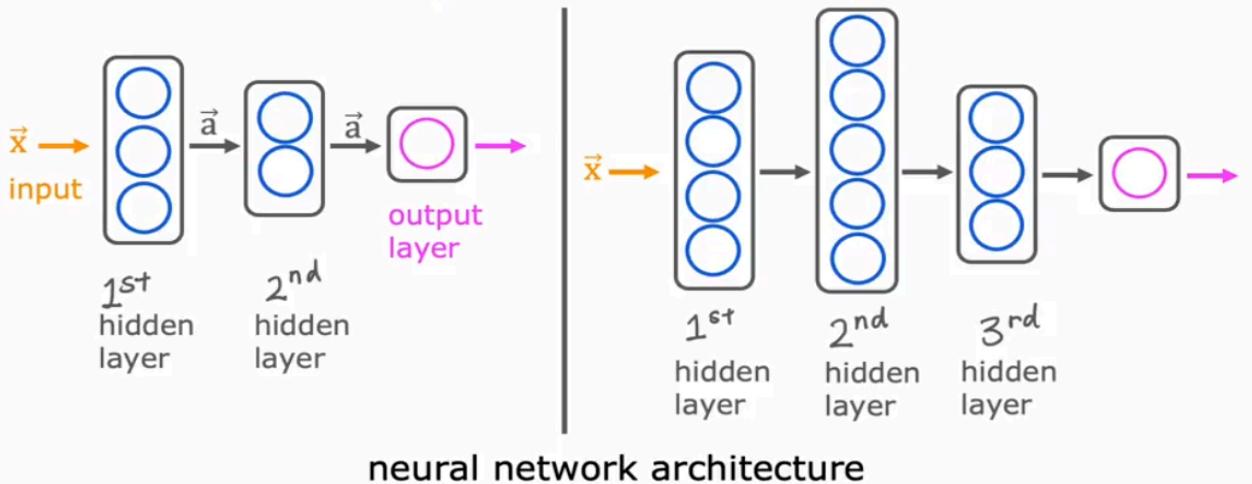
Demand Prediction



Demand Prediction

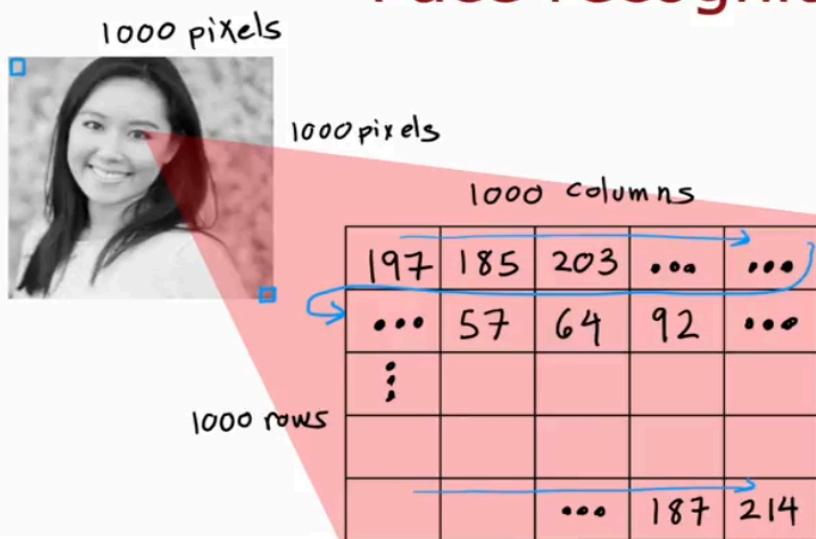


Multiple hidden layers



Recognizing image:

Face recognition



$\vec{x} =$

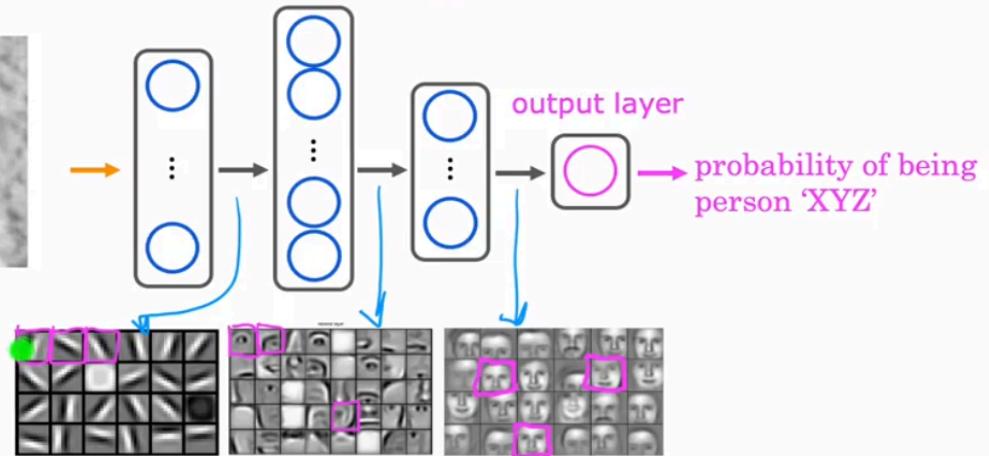
197
185
203
⋮
57
64
92
⋮
187
214

1 million

Face recognition



\vec{x}
input



source: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations
by Honglak Lee, Roger Grosse, Ranganath Andrew Y. Ng

- **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.

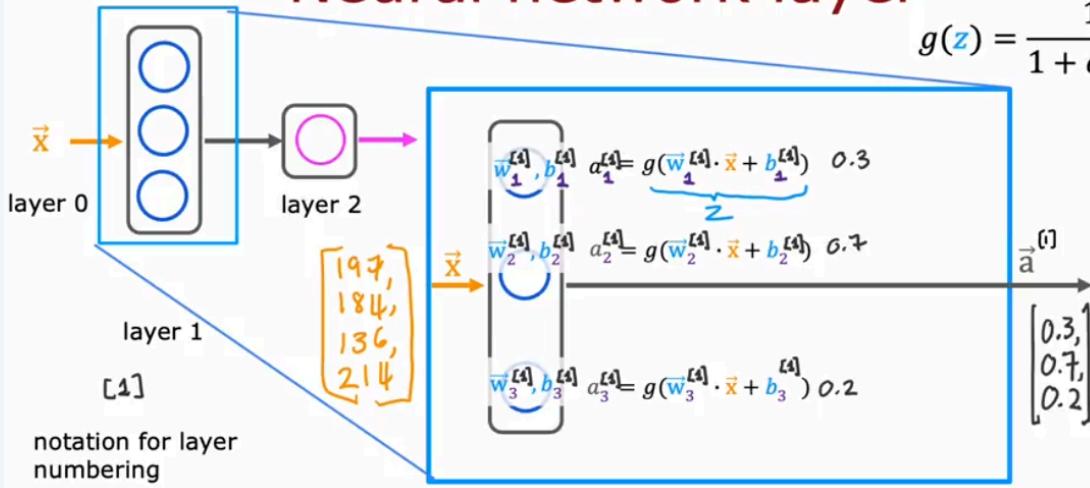
Would you like a more detailed breakdown of how convolutional layers work in this context?



Neural Network Layer:

Neural network layer

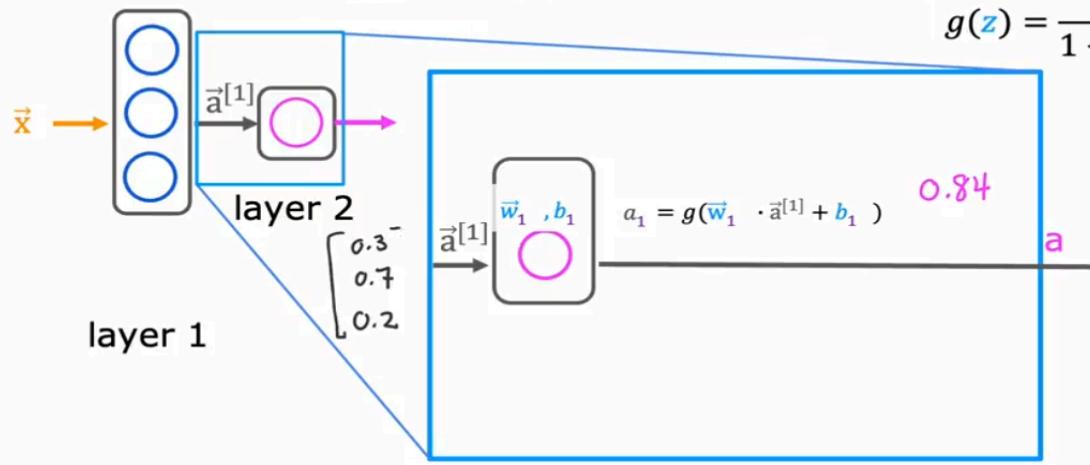
$$g(z) = \frac{1}{1 + e^{-z}}$$



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

Neural network layer

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



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 $a_i^{(1)}$.

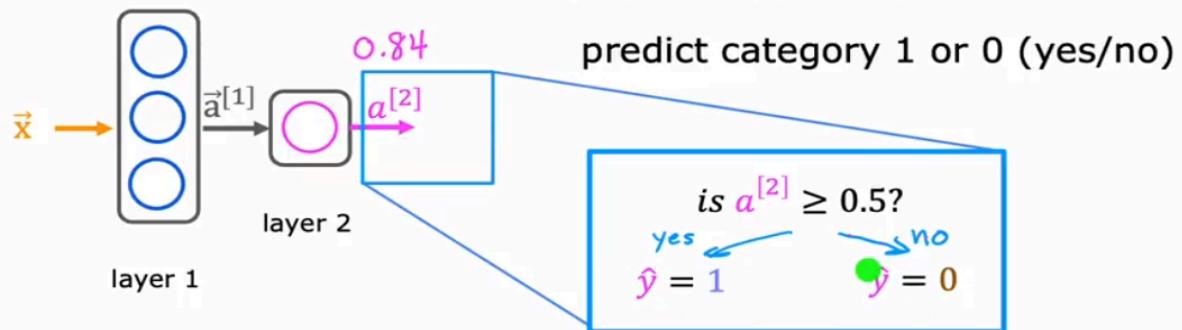
3. Notations in the Zoomed-In Box

- $w_1^{[4]}, w_2^{[4]}, w_3^{[4]}$: 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.

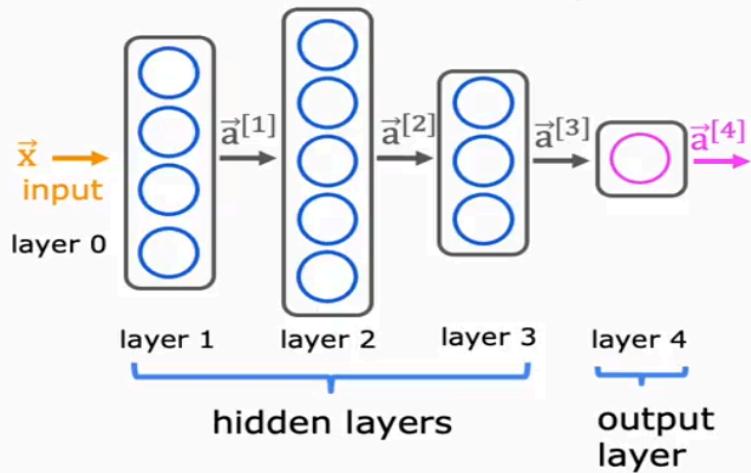
Neural network layer



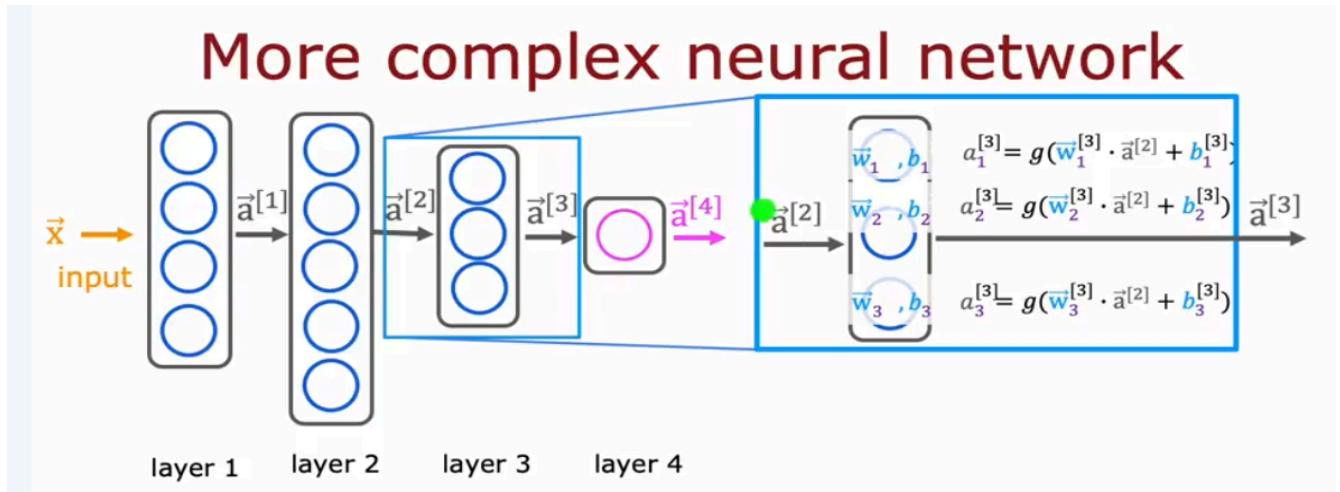
More Complex Neural Network :

Neural Network with 4 layers.

More complex neural network



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.

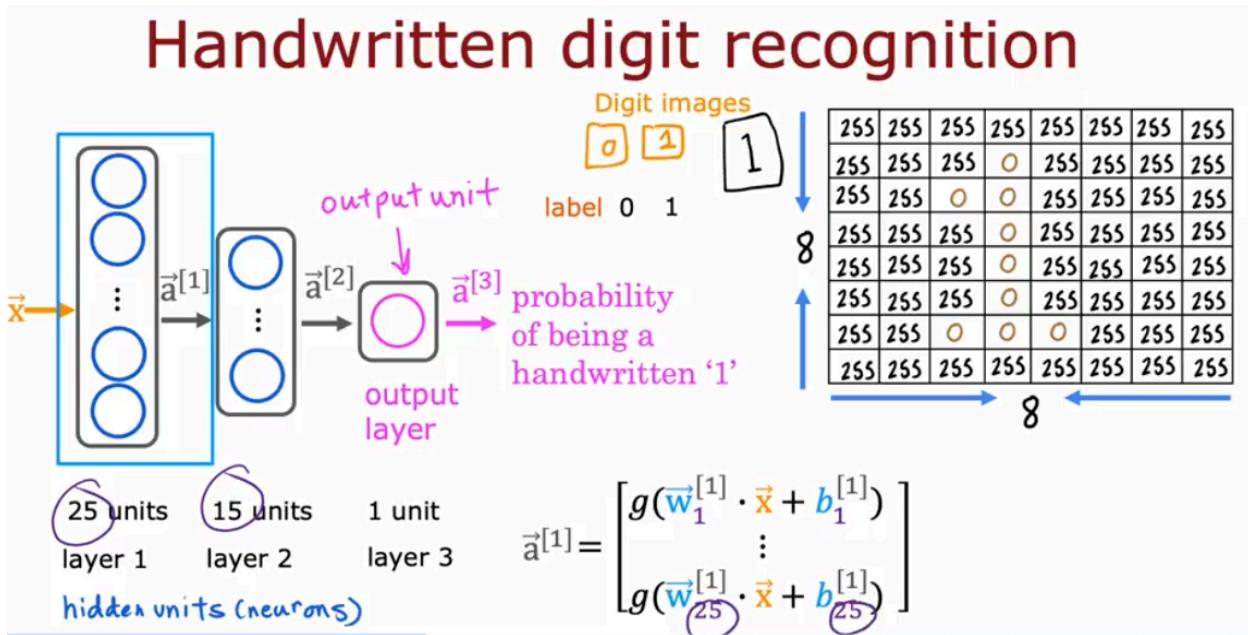
4. Key Observations

- 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) :



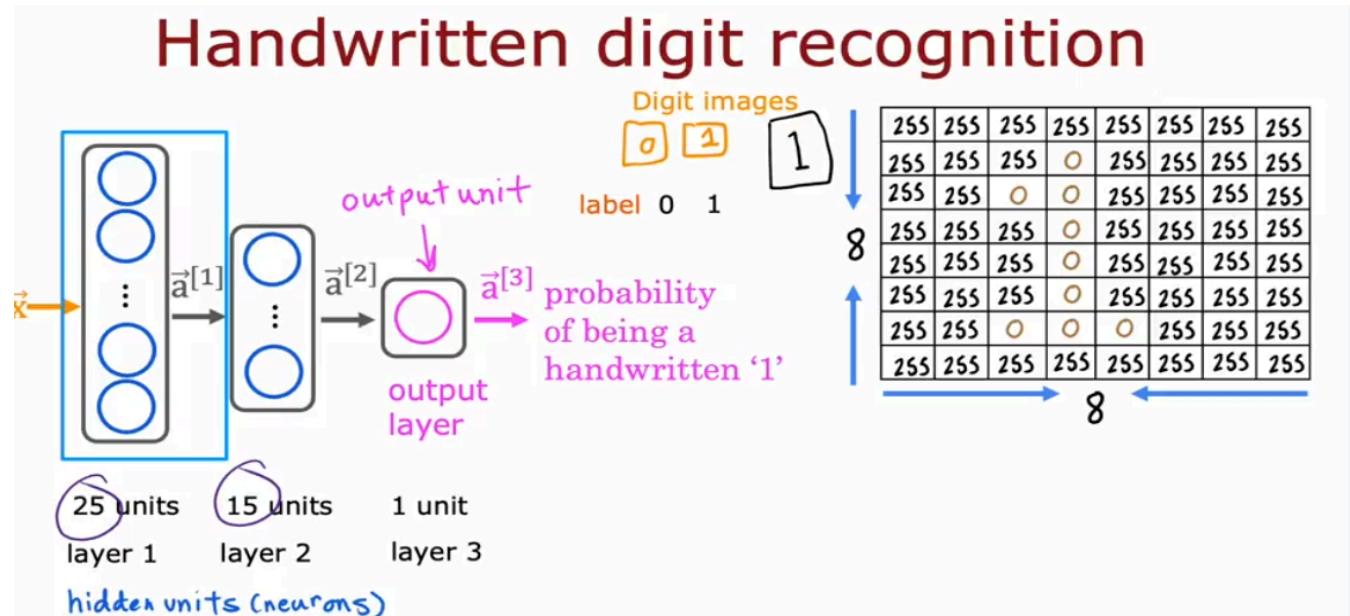
1. Input Representation (Right Side)

- 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 probability 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'.

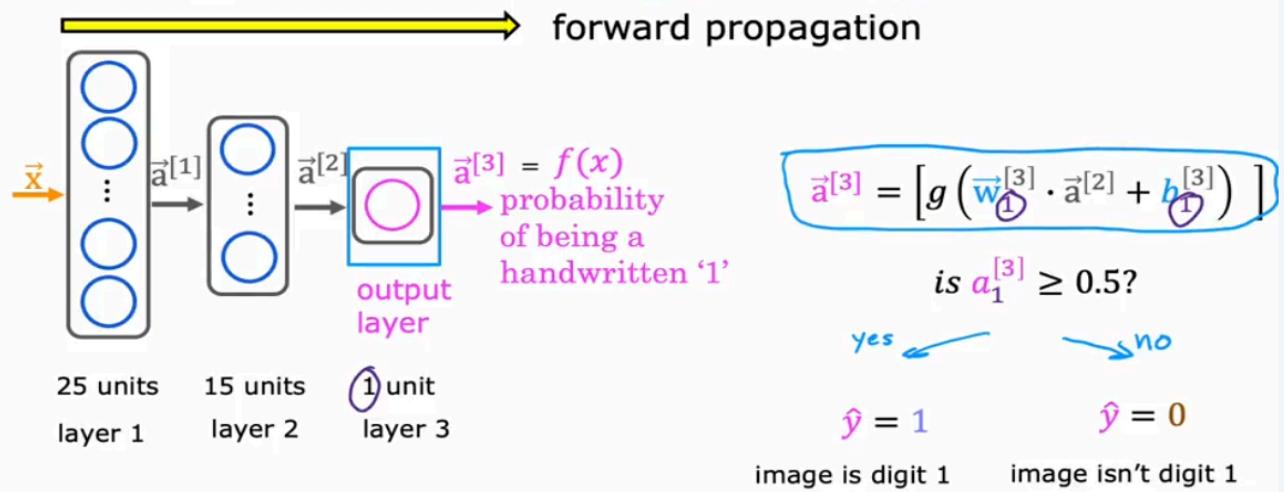
4. Output & Classification

- 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

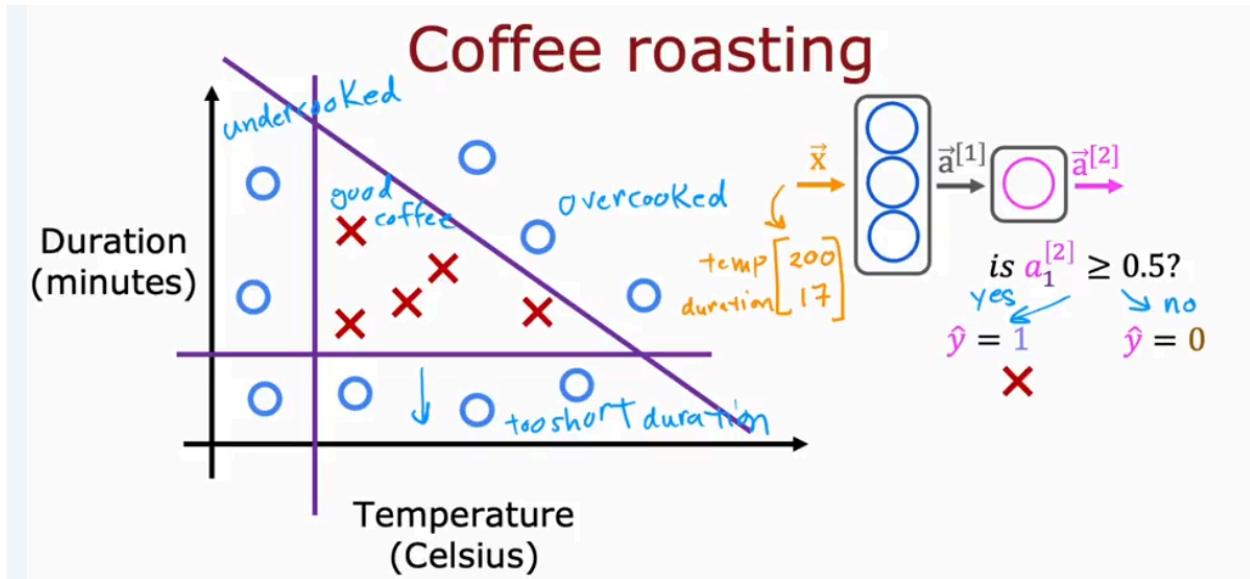
- The model learns features through hidden layers.
- The output layer uses **probabilities**, making it suitable for binary classification (0 vs. 1).
- A more complex neural network (with more layers and neurons) could handle all **digits** (0-9) instead of just 0 and 1.

Handwritten digit recognition

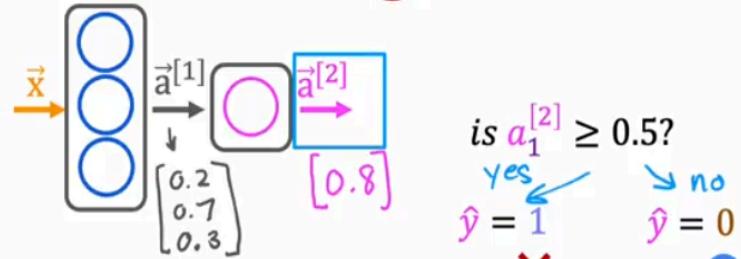


Tensorflow implementation:

Good coffee or not classification using neural network , forward inference



Build the model using TensorFlow



```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)

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

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

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

Copy Edit

```
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**, meaning the outputs are in the range $(0,1)$.

3. Second Layer (Output Layer)

```
python
```

 Copy  Edit

```
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**.

- otherwise, predict 0.

```
python
```

 Copy  Edit

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

- This follows the standard binary classification approach.

Data in Tensorflow:

Note about numpy arrays

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ $x = np.array([[1, 2, 3], [4, 5, 6]])$
2 rows 3 columns
2 × 3 matrix

$\begin{bmatrix} 0.1 & 0.2 \\ -3 & -4 \\ -0.5 & -0.6 \\ 7 & 8 \end{bmatrix}$ $x = np.array([[0.1, 0.2], [-3.0, -4.0], [-0.5, -0.6], [7.0, 8.0]])$

Note about numpy arrays

$x = np.array([[200, 17]])$ $\begin{bmatrix} 200 & 17 \end{bmatrix}$ 1×2

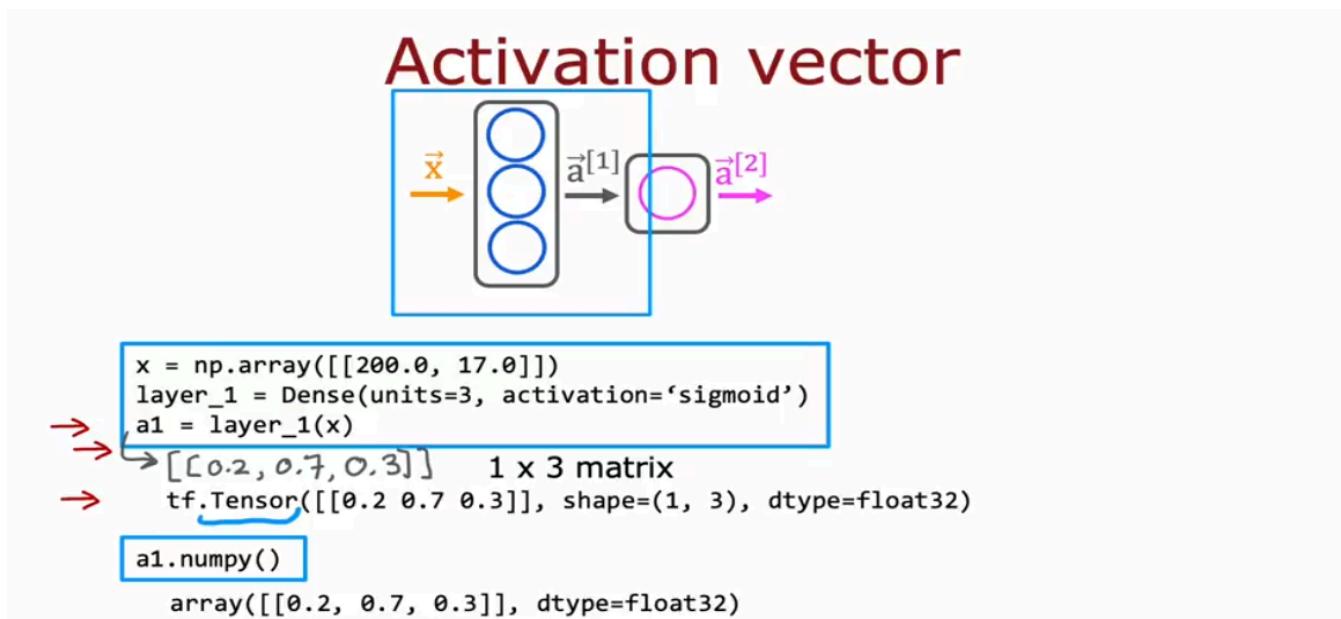
$x = np.array([[200], [17]])$ $\begin{bmatrix} 200 \\ 17 \end{bmatrix}$ 2×1

→ $x = np.array([200, 17])$
1D
"Vector"

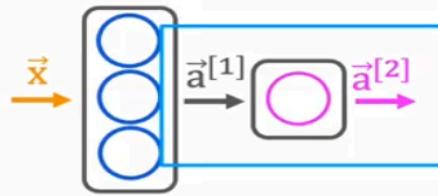
Feature vector :

Feature vectors

temperature (Celsius)	duration (minutes)	Good coffee? (1/0)	x = np.array([[200.0, 17.0]]) ← [[200.0, 17.0]]
200.0	17.0	1	
425.0	18.5	0	
...	1 × 2 → [200.0 17.0]



Activation vector



```
→ layer_2 = Dense(units=1, activation='sigmoid')
→ a2 = layer_2(a1)
  ↲ [[0.8]] ↲
→ tf.Tensor([[0.8]], shape=(1, 1), dtype=float32)
→ a2.numpy()
array([[0.8]], dtype=float32)
```

1×1

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
```

Copy Edit

```
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

 Copy  Edit

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

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

lua

 Copy  Edit

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

- Converting this tensor to a NumPy array:

python

 Copy  Edit

```
a1.numpy()
```

Produces:

lua

 Copy  Edit

```
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
```

Copy Edit

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

- The output is another tensor:

```
lua
```

Copy Edit

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

- Converting this to a NumPy array:

```
python
```

Copy Edit

```
a2.numpy()
```

Results in:

```
lua
```

Copy Edit

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

4. Final Prediction

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

```
python
```

 Copy  Edit

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

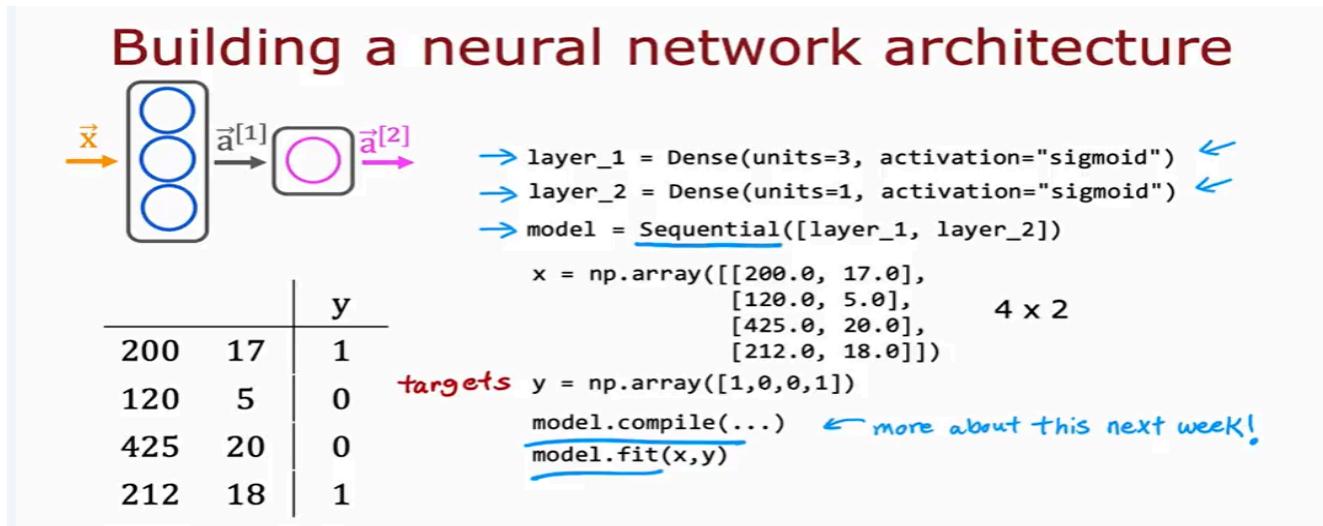
Summary

- 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

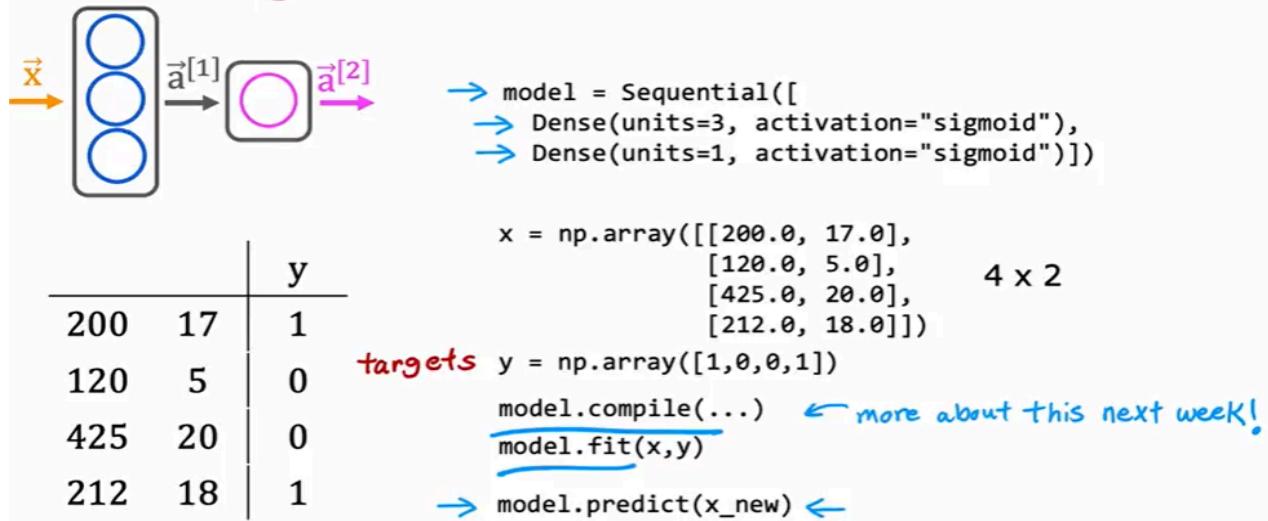
Copy

Edit

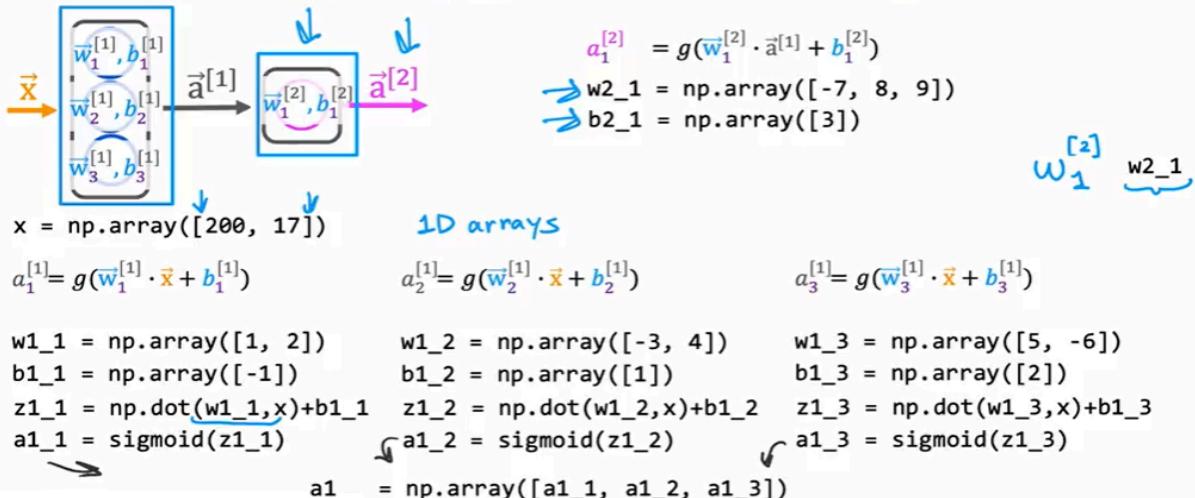
```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

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

Building a neural network architecture



forward prop (coffee roasting model)



Vectorized implementation of Forward Propagation:

For loops vs. vectorization

```

x = np.array([200, 17])
W = np.array([[1, -3, 5],
              [-2, 4, -6]])
b = np.array([-1, 1, 2])

def dense(a_in,W,b):
    units = W.shape[1]
    a_out = np.zeros(units)
    for j in range(units):
        w = W[:,j]
        z = np.dot(w, a_in) + b[j]
        a_out[j] = g(z)
    return a_out

```

vectorized

[[1,0,1]] ↘

```

x = np.array([[200, 17]])           2Darray
W = np.array([[1, -3, 5],           same
              [-2, 4, -6]])          1x3 2Darray
B = np.array([-1, 1, 2])           all 2Darrays

def dense(A_in,W,B):
    Z = np.matmul(A_in,W) + B
    A_out = g(Z) matrix multiplication
    return A_out

```

Dot products

example $\begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ $z = (1 \times 3) + (2 \times 4)$ $= 3 + 8$ $\underline{\underline{= 11}}$	in general $\begin{bmatrix} \uparrow \\ \vec{a} \\ \downarrow \end{bmatrix} \cdot \begin{bmatrix} \uparrow \\ \vec{w} \\ \downarrow \end{bmatrix}$ $z = \vec{a} \cdot \vec{w}$	transpose $\vec{a} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ $\vec{a}^T = [1 \quad 2]$	vector vector multiplication $[\leftarrow \vec{a}^T \rightarrow] \begin{bmatrix} \uparrow \\ \vec{w} \\ \downarrow \end{bmatrix} 2 \times 1$ $z = \vec{a}^T \vec{w}$
--	--	--	--

equivalent ↙ ↘

Neural Network Training:

Train a Neural Network in TensorFlow

Given set of (x, y) examples
How to build and train this in code?

```

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')
])
from tensorflow.keras.losses import BinaryCrossentropy
model.compile(loss=BinaryCrossentropy())
model.fit(X, Y, epochs=100)

```

(1) (2) (3)

How neural network training maps with regression

Model Training Steps

Tensor Flow

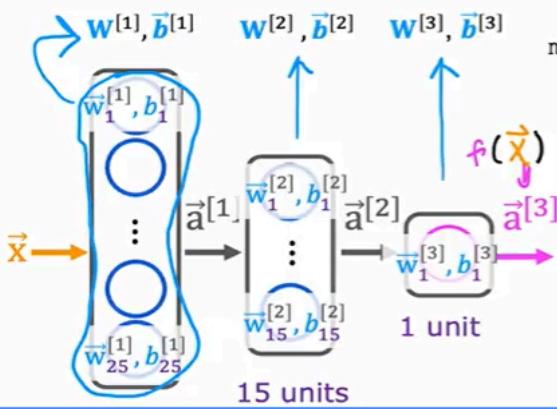
<p>① specify how to compute output given input x and parameters w, b (define model) $f_{\vec{w}, b}(\vec{x}) = ?$</p> <p>② specify loss and cost $L(f_{\vec{w}, b}(\vec{x}), y)$ 1 example $J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^m L(f_{\vec{w}, b}(\vec{x}^{(i)}), y^{(i)})$</p> <p>③ Train on data to minimize $J(\vec{w}, b)$</p>	<p>logistic regression</p> $z = np.dot(w, x) + b$ $f_x = 1 / (1 + np.exp(-z))$ <p>logistic loss</p> $\text{loss} = -y * np.log(f_x) - (1-y) * np.log(1-f_x)$ $w = w - \alpha * dj_dw$ $b = b - \alpha * dj_db$	<p>neural network</p> $\text{model} = \text{Sequential}([\text{Dense}(...), \text{Dense}(...), \text{Dense}(...)])$ <p>binary cross entropy</p> $\text{model.compile(loss=BinaryCrossentropy())}$ $\text{model.fit}(X, y, \text{epochs}=100)$
--	--	---

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 Dense

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

model.compile(loss= BinaryCrossentropy())

regression

(predicting numbers and not categories) mean squared error

model.compile(loss= MeanSquaredError())

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

$\mathbf{w}^{[1]}, \mathbf{w}^{[2]}, \mathbf{w}^{[3]}$ $\vec{b}^{[1]}, \vec{b}^{[2]}, \vec{b}^{[3]}$ $f_{\mathbf{W}, \mathbf{B}}(\vec{x})$

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

Keras

```
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
```

 Copy

 Edit

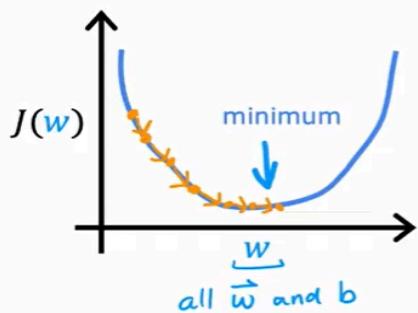
```
from tensorflow.keras.losses import BinaryCrossentropy  
model.compile(loss=BinaryCrossentropy())
```

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.

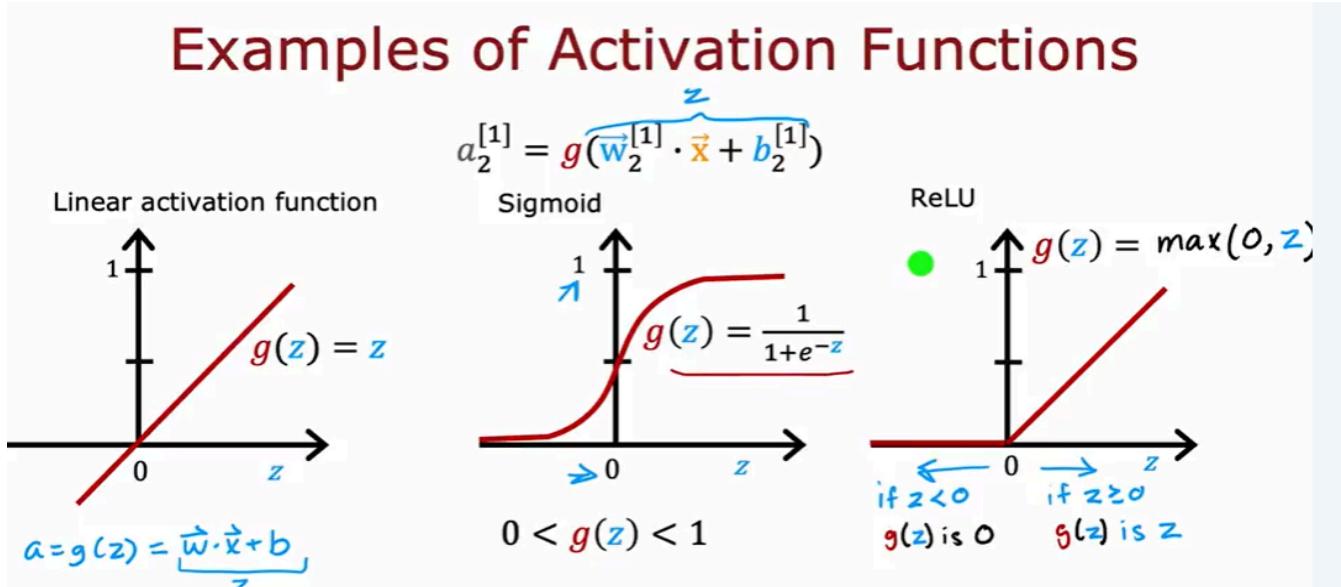
Now compute derivatives for back propagation and updation of weights have been done

3. Gradient descent



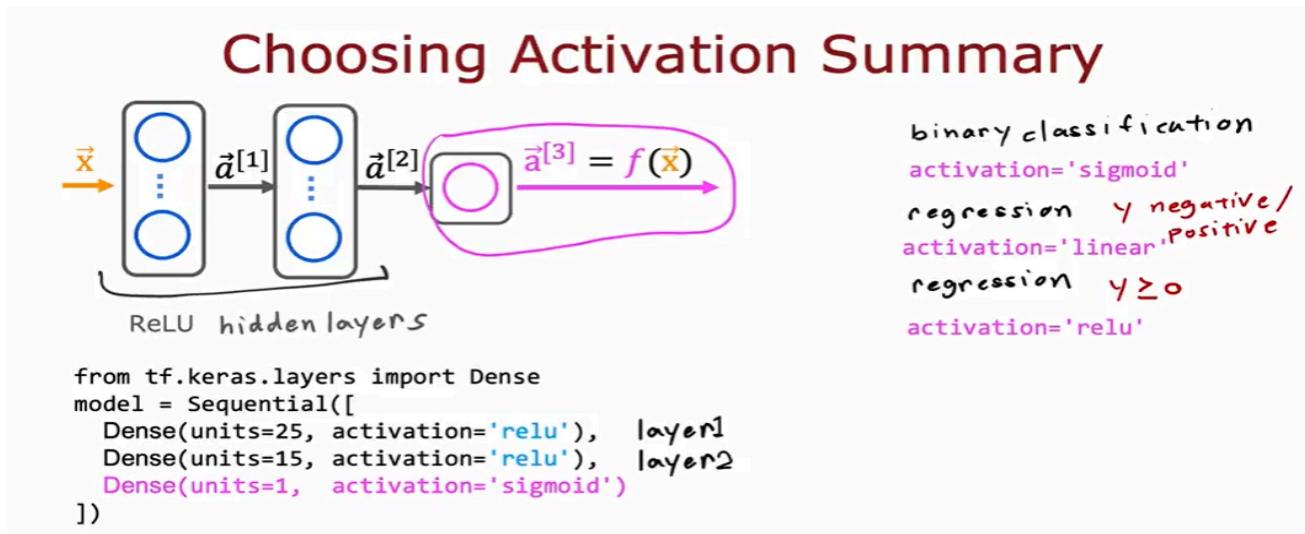
```
repeat {  
     $w_j^{[l]} = w_j^{[l]} - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$   
     $b_j^{[l]} = b_j^{[l]} - \alpha \frac{\partial}{\partial b_j} J(\vec{w}, b)$   
}
```

Alternative To The Sigmoid Activations:

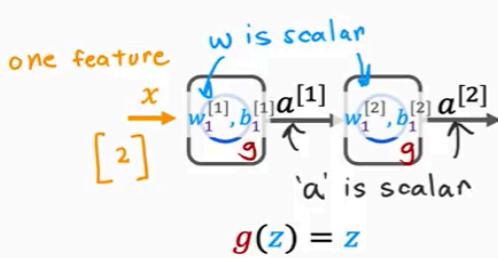


relu-> Rectified Linear Activation function

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



Linear Example



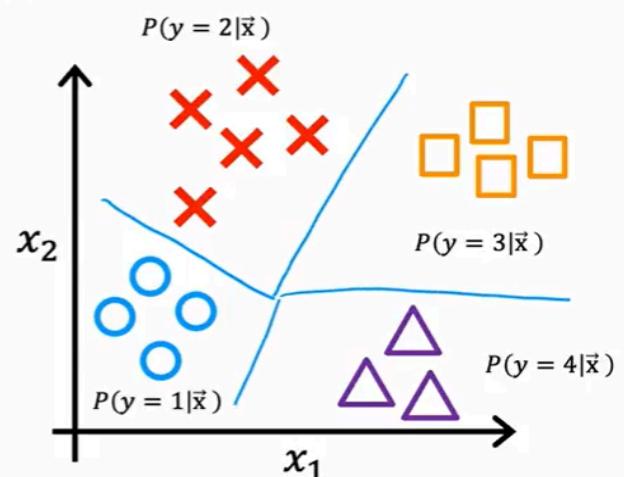
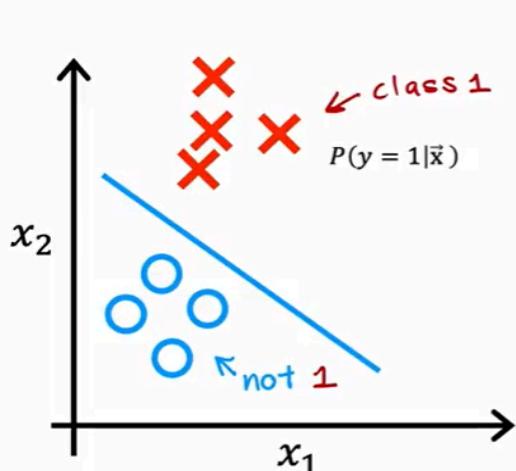
$$\begin{aligned}
 a^{[1]} &= \underbrace{w_1^{[1]} x}_{w} + b_1^{[1]} \\
 a^{[2]} &= \underbrace{w_1^{[2]} a^{[1]} + b_1^{[2]}}_{\vec{a}^{[2]}} \\
 &= w_1^{[2]} (w_1^{[1]} x + b_1^{[1]}) + b_1^{[2]} \\
 \vec{a}^{[2]} &= (\underbrace{\vec{w}_1^{[2]} \vec{w}_1^{[1]}}_{w}) x + \underbrace{w_1^{[2]} b_1^{[1]} + b_1^{[2]}}_{b} \\
 \vec{a}^{[2]} &= w x + b
 \end{aligned}$$

$f(x) = w x + b$ linear regression

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:

Multiclass classification example



Logistic regression
(2 possible output values)

$$z = \vec{w} \cdot \vec{x} + b$$

$$\text{X } a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x}) \quad 0.11$$

$$\text{O } a_2 = 1 - a_1 = P(y=0|\vec{x}) \quad 0.29$$

Softmax regression (4 possible outputs) $y=1, 2, 3, 4$

$$\text{X } z_1 = \vec{w}_1 \cdot \vec{x} + b_1 \quad a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=1|\vec{x}) \quad 0.30$$

$$\text{O } z_2 = \vec{w}_2 \cdot \vec{x} + b_2 \quad a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=2|\vec{x}) \quad 0.20$$

$$\square z_3 = \vec{w}_3 \cdot \vec{x} + b_3 \quad a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=3|\vec{x}) \quad 0.15$$

$$\triangle z_4 = \vec{w}_4 \cdot \vec{x} + b_4 \quad a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y=4|\vec{x}) \quad 0.35$$

Loss for softmax:

Logistic regression

$$z = \vec{w} \cdot \vec{x} + b$$

$$a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$$

$$a_2 = 1 - a_1 = P(y=0|\vec{x})$$

$$\text{loss} = -y \log a_1 - (1-y) \log(1-a_1)$$

if $y=1$ if $y=0$

$$J(\vec{w}, b) = \text{average loss}$$

Cost

Softmax regression

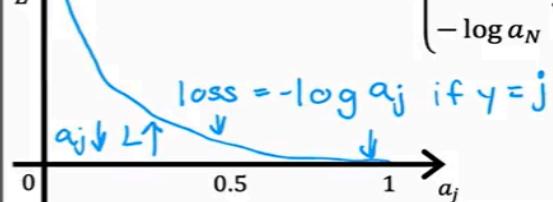
$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \dots + e^{z_N}} = P(y=1|\vec{x})$$

$$\vdots$$

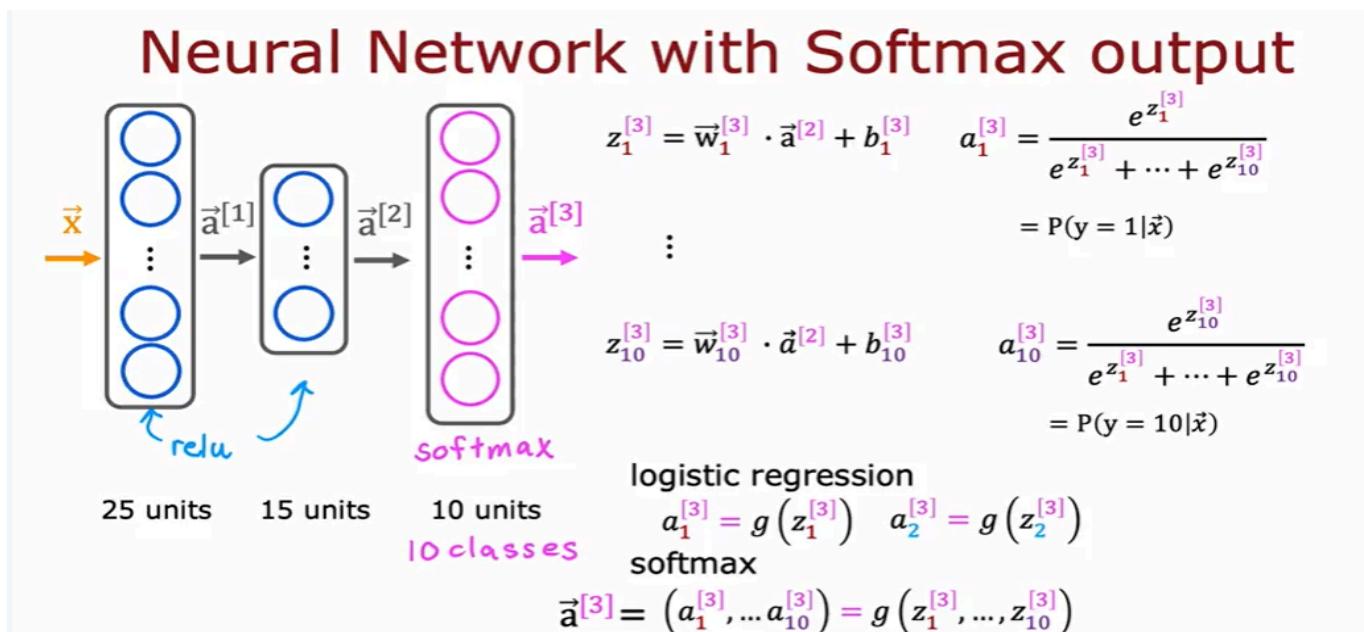
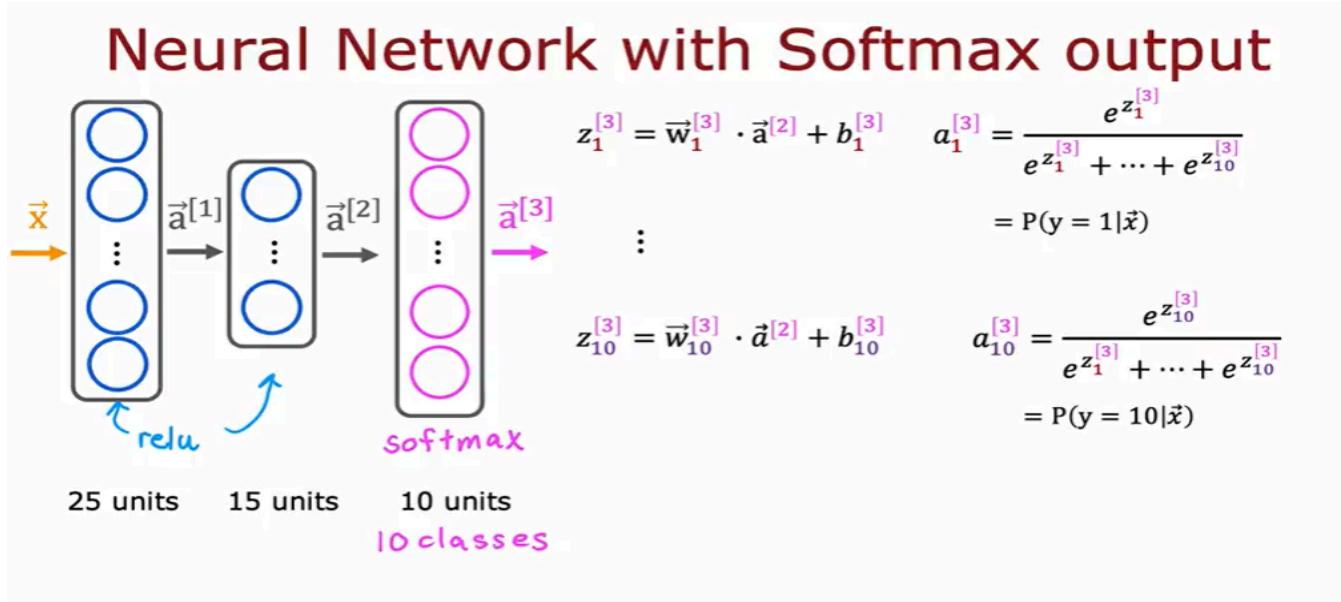
$$a_N = \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \dots + e^{z_N}} = P(y=N|\vec{x})$$

Crossentropy loss

$$\text{loss}(a_1, \dots, a_N, y) = \begin{cases} -\log a_1 & \text{if } y=1 \\ -\log a_2 & \text{if } y=2 \\ \vdots \\ -\log a_N & \text{if } y=N \end{cases}$$



Neural Network for Softmax output:



MNIST with softmax

① specify the model

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

② specify loss and cost

$$L(f_{\vec{w}, b}(\vec{x}), y)$$

③ Train on data to minimize $J(\vec{w}, b)$

```
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='softmax')
])
from tensorflow.keras.losses import
    SparseCategoricalCrossentropy
model.compile(loss= SparseCategoricalCrossentropy() )
model.fit(X, Y, epochs=100)
```

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

Numerical Round Off Errors:

Numerical Roundoff Errors

More numerically accurate implementation of logistic loss:

Logistic regression:

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

Original loss

$$loss = -y \log(\hat{a}) - (1 - y) \log(1 - \hat{a}) \quad \text{model.compile(loss=BinaryCrossEntropy())}$$

More accurate loss (in code)

$$loss = -y \log\left(\frac{1}{1 + e^{-z}}\right) - (1 - y) \log\left(1 - \frac{1}{1 + e^{-z}}\right)$$

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

Numerical Roundoff Errors

More numerically accurate implementation of logistic loss:

Logistic regression:

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

Original loss

$$\text{loss} = -y \log(a) - (1-y) \log(1-a)$$

$| + \frac{1}{10,000}$ $| - \frac{1}{10,000}$

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

~~model.compile(loss=BinaryCrossEntropy())~~

More accurate loss (in code)

$$\text{loss} = -y \log\left(\frac{1}{1 + e^{-z}}\right) - (1-y) \log\left(1 - \frac{1}{1 + e^{-z}}\right)$$

logit: z

More numerically accurate implementation of softmax

Softmax regression

$$(a_1, \dots, a_{10}) = g(z_1, \dots, z_{10})$$

$$\text{Loss} = L(\vec{a}, y) = \begin{cases} -\log a_1 & \text{if } y = 1 \\ \vdots \\ -\log a_{10} & \text{if } y = 10 \end{cases}$$

~~model = Sequential([~~

```
Dense(units=25, activation='relu'),
Dense(units=15, activation='relu'),
Dense(units=10, activation='softmax')
])
```

'linear'

~~model.compile(loss=SparseCategoricalCrossEntropy())~~

More Accurate

$$L(\vec{a}, y) = \begin{cases} -\log \frac{e^{z_1}}{e^{z_1} + \dots + e^{z_{10}}} & \text{if } y = 1 \\ \vdots \\ -\log \frac{e^{z_{10}}}{e^{z_1} + \dots + e^{z_{10}}} & \text{if } y = 10 \end{cases}$$

~~model.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))~~

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) ← not  $a_1 \dots a_{10}$ 
         is  $z_1 \dots z_{10}$ 
         f_x = tf.nn.softmax(logits)
```

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
         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
```

Copy

Edit

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

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

```
python
```

Copy

Edit

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

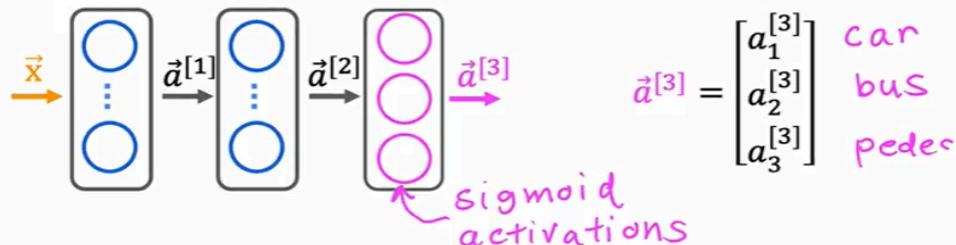
But when you're using loss functions like `crossEntropyLoss` in PyTorch or `categorical_crossentropy(from_logits=True)` in TensorFlow/Keras, they often expect the logits directly, because they internally apply softmax for numerical stability.

Multi Label Classification:

Multi-label Classification

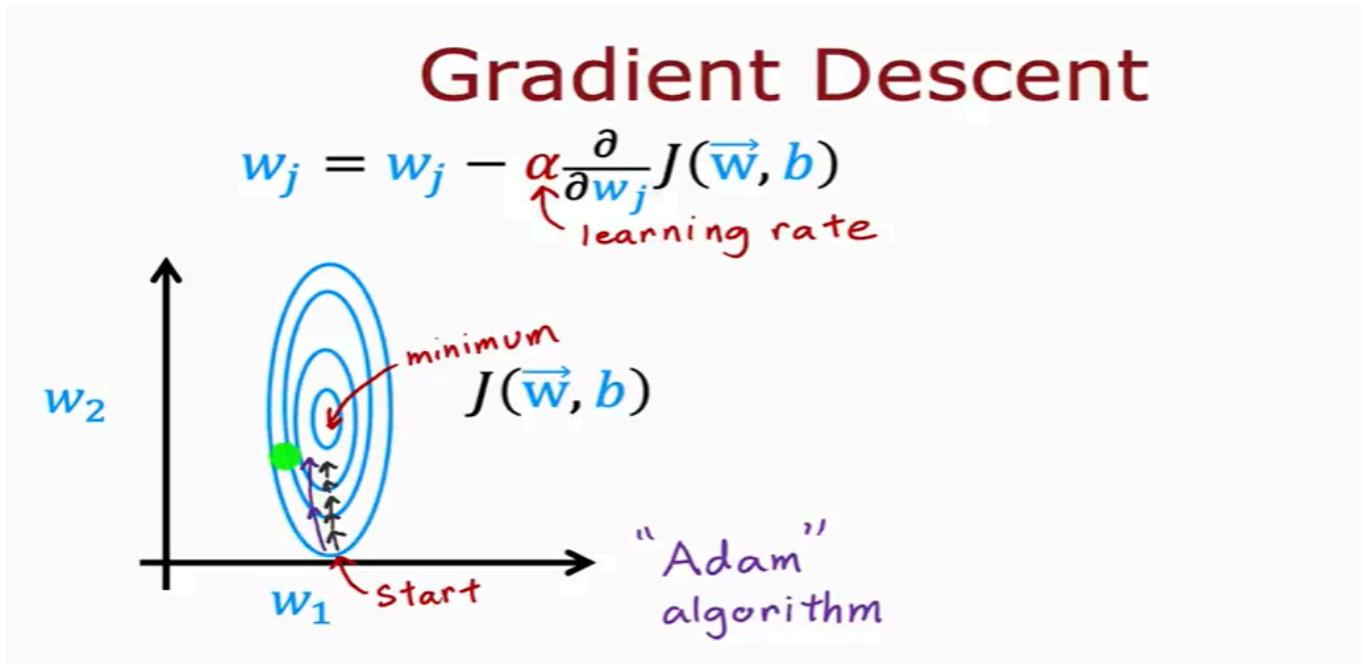


Alternatively, train one neural network with three outputs

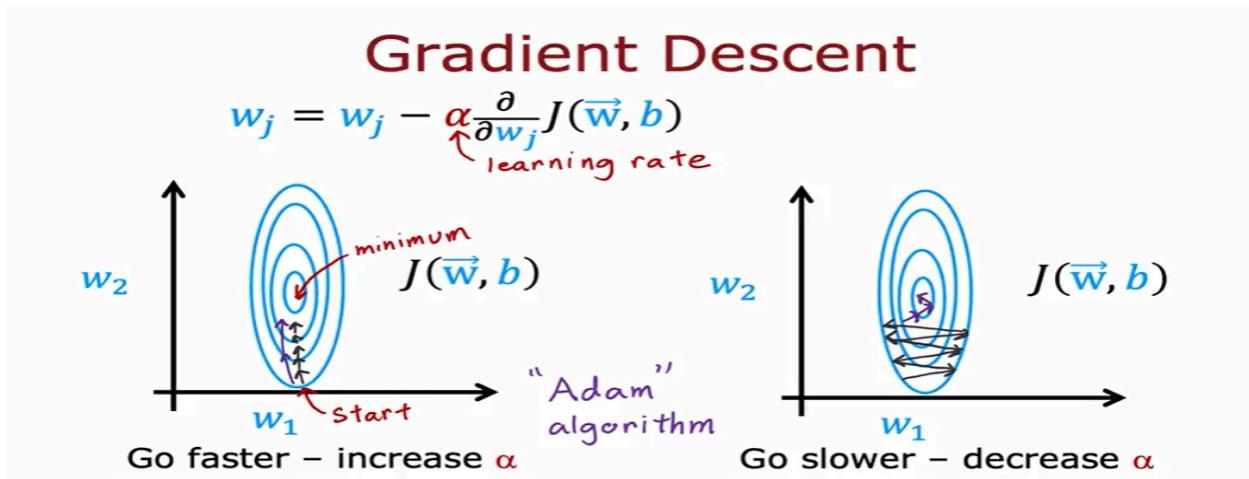


Advanced Optimization

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 sometimes 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:

Even More Derivative Examples

$w = 2$	$J(w) = w^2 = 4$	$\frac{\partial}{\partial w} J(w) = 2w = 4$	$w \uparrow \underbrace{0.001}_{\varepsilon}$	$J(w) = 4.004001$ $J(w) \uparrow 4 \times \varepsilon$
}	$J(w) = w^3 = 8$	$\frac{\partial}{\partial w} J(w) = 3w^2 = 12$	$w \uparrow \varepsilon$	$J(w) = 8.012006$ $J(w) \uparrow 12 \times \varepsilon$
	$J(w) = w = 2$	$\frac{\partial}{\partial w} J(w) = 1$	$w \uparrow \varepsilon$	$J(w) = 2.001$ $J(w) \uparrow 1 \times \varepsilon$
	$J(w) = \frac{1}{w} = \frac{1}{2} = 0.5$	$\frac{\partial}{\partial w} J(w) = -\frac{1}{w^2} = -\frac{1}{4}$	$w \uparrow \varepsilon$ $w = \frac{1}{2.001}$	-0.25×0.001 $0.5 - \underline{0.00025}$ $J(w) = 0.49975$ $J(w) \uparrow -\frac{1}{4} \times \varepsilon$
	$\frac{\partial}{\partial w} J(w) \quad w \uparrow \varepsilon \quad J(w) \uparrow k \times \varepsilon$			

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.

A note on derivative notation

If $J(w)$ is a function of one variable (w),

$$d \quad \frac{d}{dw} J(w)$$

If $J(w_1, w_2, \dots, w_n)$ is a function of more than one variable,

$$\partial \quad \frac{\partial}{\partial w_i} J(w_1, w_2, \dots, w_n) \quad \frac{\partial J}{\partial w_i} \quad \text{or} \quad \frac{\partial}{\partial w_i} J$$

"partial derivative"

notation used
in these courses

🧠 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 η 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:

Evaluating your model

Dataset:

size	price	
70%	2104	400
	1600	330
	2400	369
	1416	232
	3000	540
	1985	300
	1534	315
30%	1427	199
	1380	212
	1494	243

*(x⁽¹⁾, y⁽¹⁾)
(x⁽²⁾, y⁽²⁾)
⋮
(x^(m_{train}), y^(m_{train}))*

*m_{train} = no. training examples
= 7*

*(x_{test}⁽¹⁾, y_{test}⁽¹⁾)
⋮
(x_{test}^(m_{test}), y_{test}^(m_{test}))*

*m_{test} = no. test examples
= 3*

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

