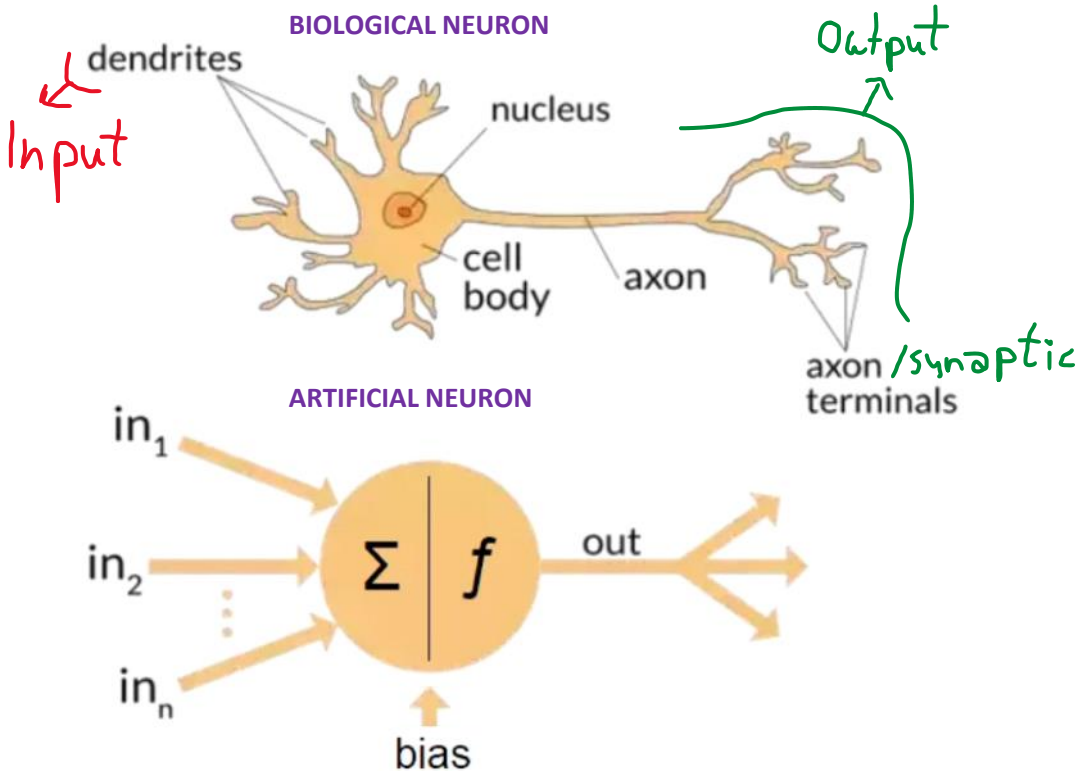
Neurons and the Brain

19 April 2024 15:14

Brief History of Neural Networks

- Neural networks were invented many decades ago with the goal of **mimicking how the human brain learns.**
- Today, neural networks are very different from how the brain works, but some of the biological motivations remain.
- Work in neural networks started in the 1950s, fell out of favor, then gained popularity again in the 1980s and early 1990s for applications like handwritten digit recognition.
- It fell out of favor again in the late 1990s, but re-emerged around 2005 with the term deep learning.
- Deep learning has revolutionized many application areas, including speech recognition, computer vision, and natural language processing.

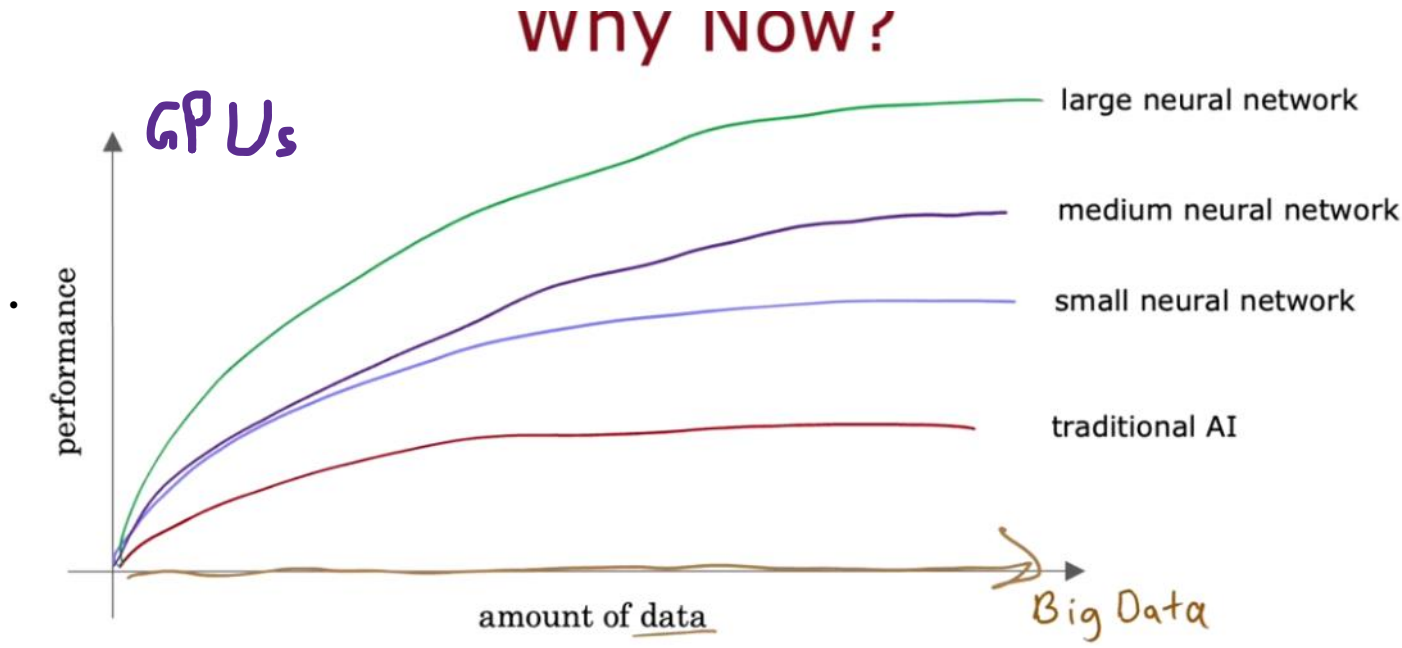
Biological Neurons vs. Artificial Neurons

- A biological neuron has a cell body, dendrites (inputs), and an axon (output).
- 

- Artificial neural networks use a simplified mathematical model of a biological neuron.
- An artificial neuron takes inputs, does some computation, and outputs a number.
- We don't need to memorize the biological terms (dendrites, axon) to build artificial neural networks.

Why Neural Networks Took Off Recently

- The amount of data available has exploded in recent decades.
- Traditional machine learning algorithms couldn't take advantage of this data.



- Neural networks, especially large neural networks, can learn from big data and achieve much better performance on tasks like speech recognition, image recognition, and natural language processing.
- The rise of faster computer processors, especially GPUs, has also enabled the development of deep learning.

Complete History of Neural Networks

History: The 1940's to the 1970's

- In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons might work. In order to describe how neurons in the brain might work, they modeled a simple neural network using electrical circuits.
- In 1949, Donald Hebb wrote *The Organization of Behavior*, a work which pointed out the fact that neural pathways are strengthened each time they are used, a concept fundamentally essential to the ways in which humans learn. If two nerves fire at the same time, he argued, the connection between them is enhanced.
- As computers became more advanced in the 1950's, it was finally possible to simulate a hypothetical neural network. The first step towards this was made by Nathanial Rochester from the IBM research laboratories. Unfortunately for him, the first attempt to do so failed.
- In 1959, Bernard Widrow and Marcian Hoff of Stanford developed models called "ADALINE" and "MADALINE." In a typical display of Stanford's love for acronymns, the names come from their use of Multiple ADAptive LINear Elements. ADALINE was developed to recognize binary patterns so that if it was reading streaming bits from a phone line, it could predict the next bit. MADALINE was the first neural network applied to a real world problem, using an adaptive filter that eliminates echoes on phone lines. While the system is as ancient as air traffic control systems, like air traffic control systems, it is still in commercial use.
- In 1962, Widrow & Hoff developed a learning procedure that examines the value before the weight adjusts it (i.e. 0 or 1) according to the rule: Weight Change = (Pre-Weight line value) * (Error / (Number of Inputs)). It is based on the idea that while one active perceptron may have a big error, one can adjust the weight values to distribute it across the network, or at least to adjacent perceptrons. Applying this rule still results in an error if the line before the weight is 0, although this will eventually correct itself. If the error is conserved so that all of it is distributed to all of the weights than the error is eliminated.
- Despite the later success of the neural network, traditional von Neumann architecture took over the computing scene, and neural research was left behind. Ironically, John von Neumann himself suggested the imitation of neural functions by using telegraph relays or vacuum tubes.
- In the same time period, a paper was written that suggested there could not be an extension from the single layered neural network to a multiple layered neural network. In addition, many people in the field were using a learning function that was fundamentally flawed because it was not differentiable across the entire line. As a result, research and funding went drastically down.
- This was coupled with the fact that the early successes of some neural networks led to an exaggeration of the potential of neural networks, especially considering the practical technology at the time. Promises went unfulfilled, and at times greater philosophical questions led to fear. Writers pondered the effect that the so-called "thinking machines" would have on humans, ideas which are still around today.
- The idea of a computer which programs itself is very appealing. If Microsoft's Windows 2000 could reprogram itself, it might be able to repair the thousands of bugs that the programming staff made. Such ideas were appealing but very difficult to implement. In

addition, von Neumann architecture was gaining in popularity. There were a few advances in the field, but for the most part research was few and far between.

- In 1972, Kohonen and Anderson developed a similar network independently of one another, which we will discuss more about later. They both used matrix mathematics to describe their ideas but did not realize that what they were doing was creating an array of analog ADALINE circuits. The neurons are supposed to activate a set of outputs instead of just one.

- The first multilayered network was developed in 1975, an unsupervised network.

History: The 1980's to the present

- In 1982, interest in the field was renewed. John Hopfield of Caltech presented a paper to the National Academy of Sciences. His approach was to create more useful machines by using bidirectional lines. Previously, the connections between neurons was only one way.

- That same year, Reilly and Cooper used a "Hybrid network" with multiple layers, each layer using a different problem-solving strategy.

- Also in 1982, there was a joint US-Japan conference on Cooperative/Competitive Neural Networks. Japan announced a new Fifth Generation effort on neural networks, and US papers generated worry that the US could be left behind in the field. (Fifth generation computing involves artificial intelligence. First generation used switches and wires, second generation used the transister, third state used solid-state technology like integrated circuits and higher level programming languages, and the fourth generation is code generators.) As a result, there was more funding and thus more research in the field.

- In 1986, with multiple layered neural networks in the news, the problem was how to extend the Widrow-Hoff rule to multiple layers. Three independent groups of researchers, one of which included David Rumelhart, a former member of Stanford's psychology department, came up with similar ideas which are now called back propagation networks because it distributes pattern recognition errors throughout the network. Hybrid networks used just two layers, these back-propagation networks use many. The result is that back-propagation networks are "slow learners," needing possibly thousands of iterations to learn.

- Now, neural networks are used in several applications, some of which we will describe later in our presentation. The fundamental idea behind the nature of neural networks is that if it works in nature, it must be able to work in computers. The future of neural networks, though, lies in the development of hardware. Much like the advanced chess-playing machines like Deep Blue, fast, efficient neural networks depend on hardware being specified for its eventual use.

- Research that concentrates on developing neural networks is relatively slow. Due to the limitations of processors, neural networks take weeks to learn. Some companies are trying to create what is called a "silicon compiler" to generate a specific type of integrated circuit that is optimized for the application of neural networks. Digital, analog, and optical chips are the different types of chips being developed. One might immediately discount analog signals as a thing of the past. However neurons in the brain actually work more like analog signals than digital signals. While digital signals have two distinct states (1 or 0, on or off), analog signals vary between minimum and maximum values. It may be awhile, though, before optical chips can be used in commercial applications.

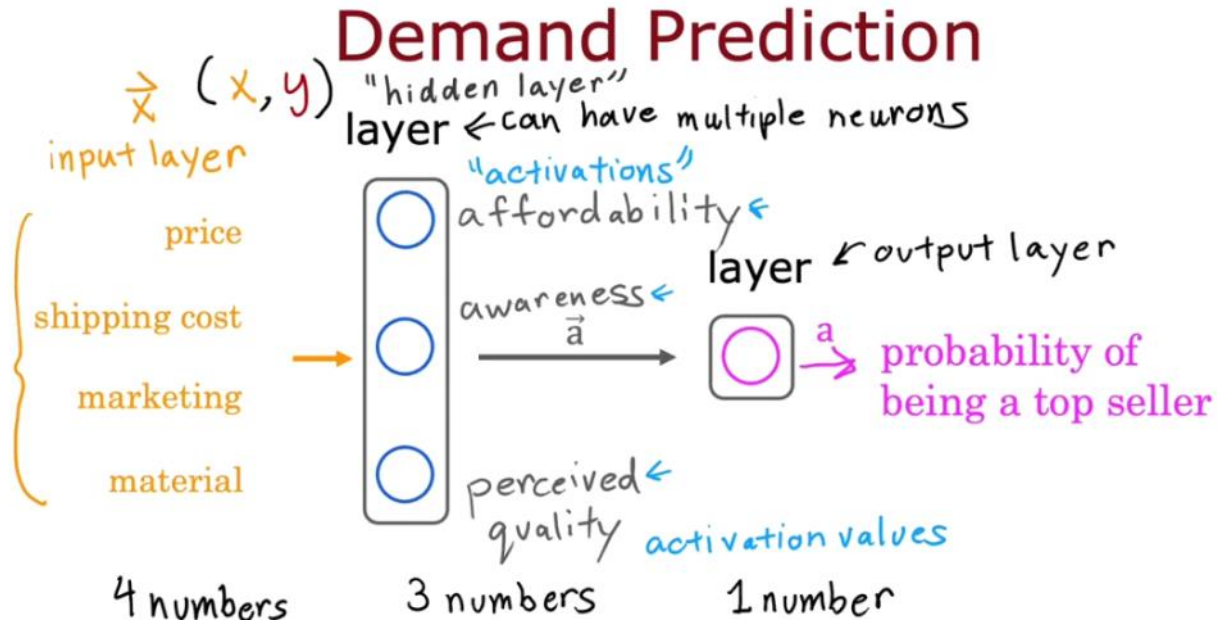
What are neural networks?

14 May 2024 15:51

Introduction

- Neural networks are a type of machine learning program inspired by the structure and function of the human brain.
- They learn by processing data and identifying patterns to make predictions.

Example: Predicting Top-Selling T-Shirts

- Imagine you want to predict if a T-shirt will be a top seller based on features like price, shipping cost, marketing, and material quality.
- 

- A logistic regression algorithm can be used to analyze this data and estimate the probability of a sale.

Building a Neural Network

- A neural network is a network of interconnected processing units called neurons.
- Each neuron receives inputs, performs a calculation, and outputs an **activation value.**
- In our example, we can create multiple neurons to estimate affordability, awareness of the T-shirt (based on marketing), and perceived quality (based on price and material).

Layers in a Neural Network

- Neurons are organized into layers:
  - Input layer: Receives the initial data (e.g., price, shipping cost, marketing, material quality).
  - Hidden layer(s): Processes the data and learns features from it. (This example uses one hidden layer with three neurons).

- Output layer: Produces the final prediction (e.g., probability of the T-shirt being a top seller).

How a Neural Network Works

1. The input layer receives the T-shirt's features.
2. The hidden layer neurons calculate affordability, awareness, and perceived quality based on the input and their connections.
3. The output layer neuron receives these activations and calculates the final prediction (probability of being a top seller).

Benefits of Neural Networks

- Neural networks can learn complex features from data without the need for manual feature engineering.
- Example: Even though this neural network is computing affordability, awareness, and perceived quality, one of the really nice properties of a neural network is when you train it from data, you don't need to go in to explicitly decide what other features, such as affordability and so on, that the neural network should compute instead or figure out all by itself what are the features it wants to use in this hidden layer.
- This makes them powerful tools for various machine learning tasks.

More Complex Neural Networks

- Neural networks can have multiple hidden layers, allowing them to learn even more complex relationships in the data.
- The choice of how many hidden layers and neurons to use is part of the neural network's architecture and can impact its performance.

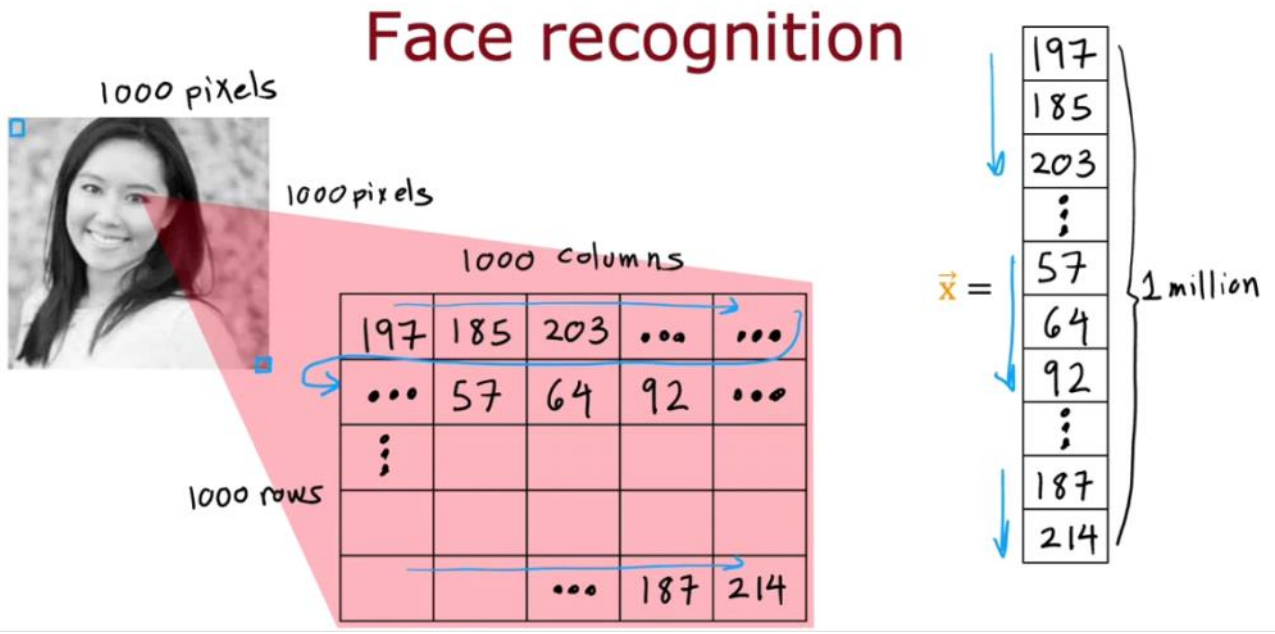
Neural Network Example

14 May 2024 23:13

Face Recognition with Neural Networks

Imagine building a face recognition application. You'd train a neural network to take an image as input and output the person's identity in the picture.

A typical image might be 1,000 by 1,000 pixels. In a computer, this translates to a 1,000 by 1,000 grid, or a matrix, of pixel intensity values. These values typically range from 0 (black) to 255 (white).



To train the network, we can unroll these pixel intensities into a single vector with a million entries (1,000 x 1,000). The challenge is to train a neural network that takes this million-value vector as input and outputs the person's identity.

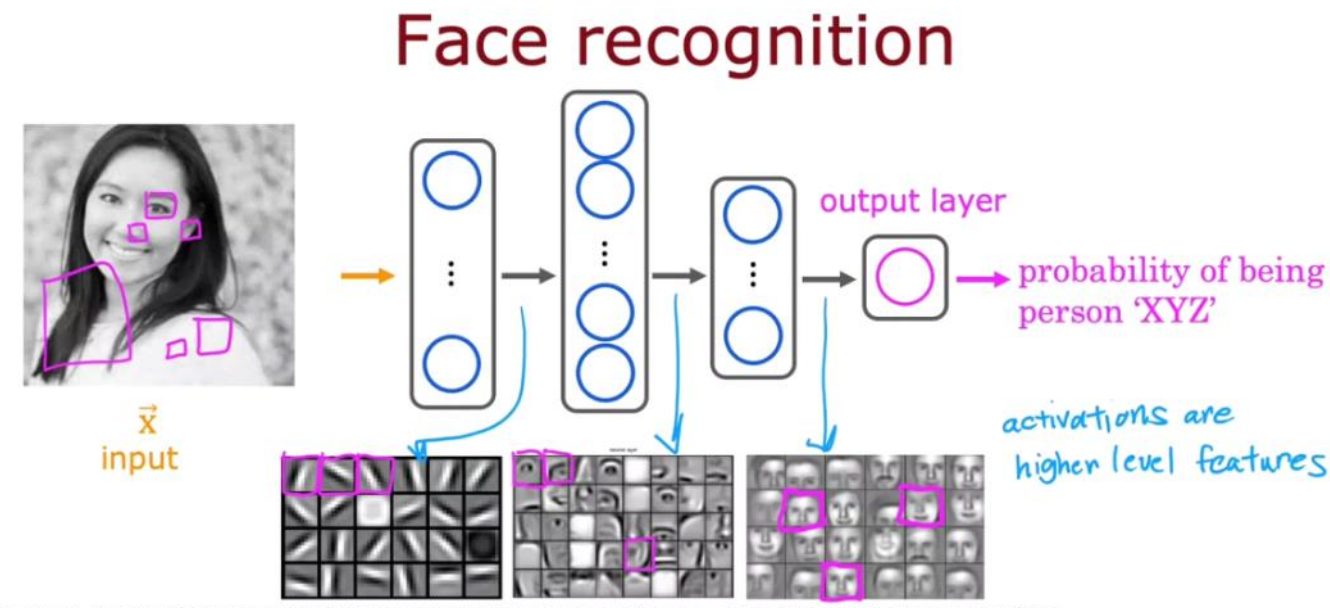
How Neural Networks Extract Features

Here's a simplified view of how a neural network might perform this task:

1. The input image is fed into the first layer of neurons.
2. This first hidden layer extracts some features from the image.
3. The output of this layer is fed to subsequent hidden layers, ultimately reaching the output layer.
4. The output layer estimates, for example, the probability of the image being a specific person.

What's interesting is visualizing what the hidden layers compute. When trained on many images, these layers learn to identify progressively more complex features:

- **Early layers:** Look for basic edges and lines in various orientations.
- **Later layers:** Combine these edges and lines to detect parts of faces (eyes, noses, ears).
- **Final layers:** Aggregate these parts to identify complete face shapes and match them to known identities.



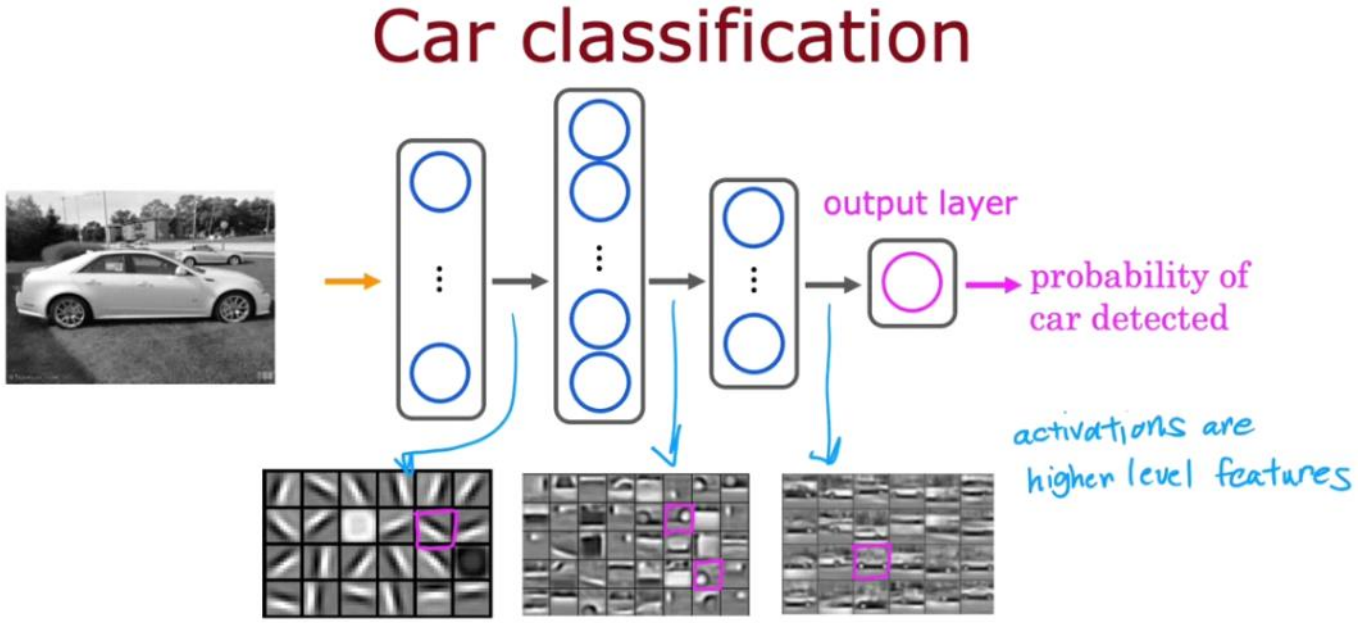
source: Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations by Honglak Lee, Roger Grosse, Ranganath Andrew Y. Ng

Remarkably, the network learns these feature detectors on its own, without explicit programming. It discovers them through the training data.

Adapting to Different Tasks

The beauty of neural networks is their adaptability. Trained on a different dataset (e.g., car pictures), the network would learn to detect:

- Edges in the first layer (similar to faces).
- Car parts (wheels, windows) in the second layer.
- Complete car shapes in the third layer.



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

By feeding it different data, the network automatically learns the relevant features for the specific task (car detection, face

recognition, etc.).

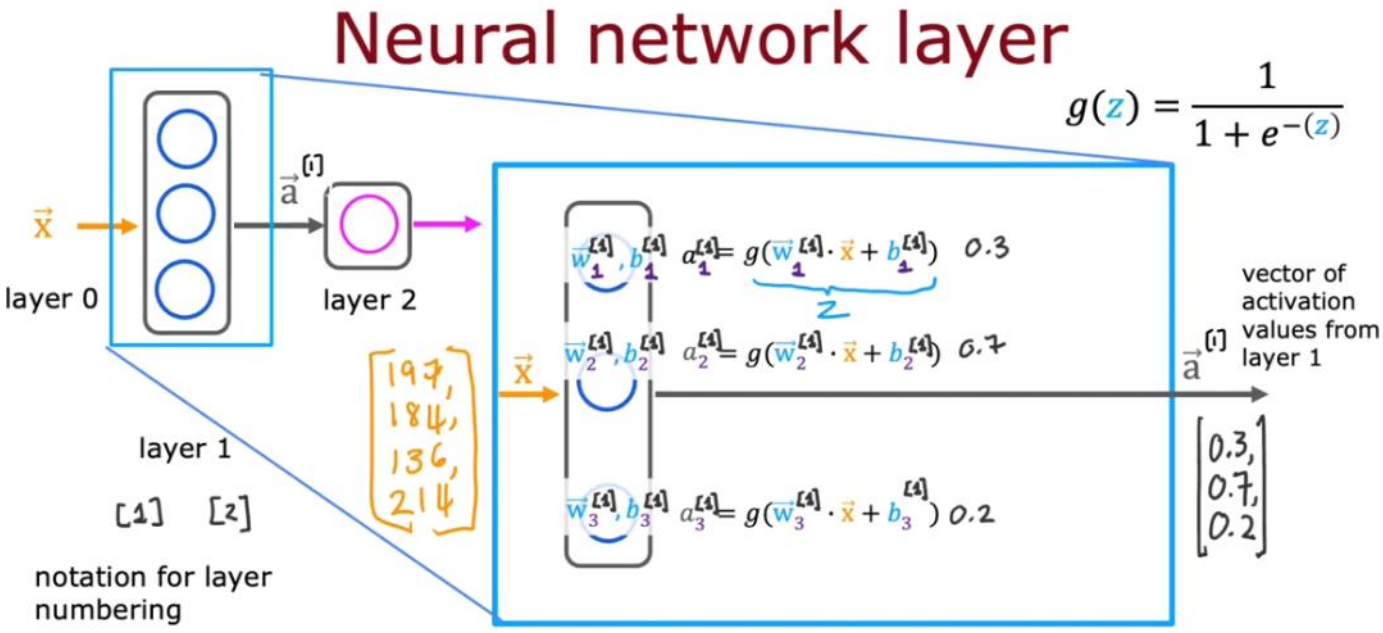
Neural Network Layer

15 May 2024 11:25

Understanding a Layer of Neurons

Let's revisit the demand prediction example from previous lesson. Here, a layer of three hidden neurons receives four input features and sends its output to a single output neuron.

Zooming into the hidden layer, we see each neuron acting like a miniature logistic regression unit. Each neuron takes the weighted sum of its inputs (including a bias term) and applies a sigmoid function to get its activation value.



For example, the first neuron might output an activation value of 0.3, signifying a 30% chance of "highly affordable" based on the input features. Similarly, the second and third neurons might output 0.7 and 0.2, respectively.

These activation values from all three neurons form a vector that becomes the output of this hidden layer (layer 1).

Notation for Multiple Layers

In neural networks with multiple layers, it's crucial to distinguish between them. Here's the convention we'll use:

- Layer 1: This is the hidden layer we just analyzed.
- Layer 2: This is the output layer with a single neuron.
- Layer 0 (optional): This refers to the input layer.
- Modern networks: These can have dozens or even hundreds of layers.

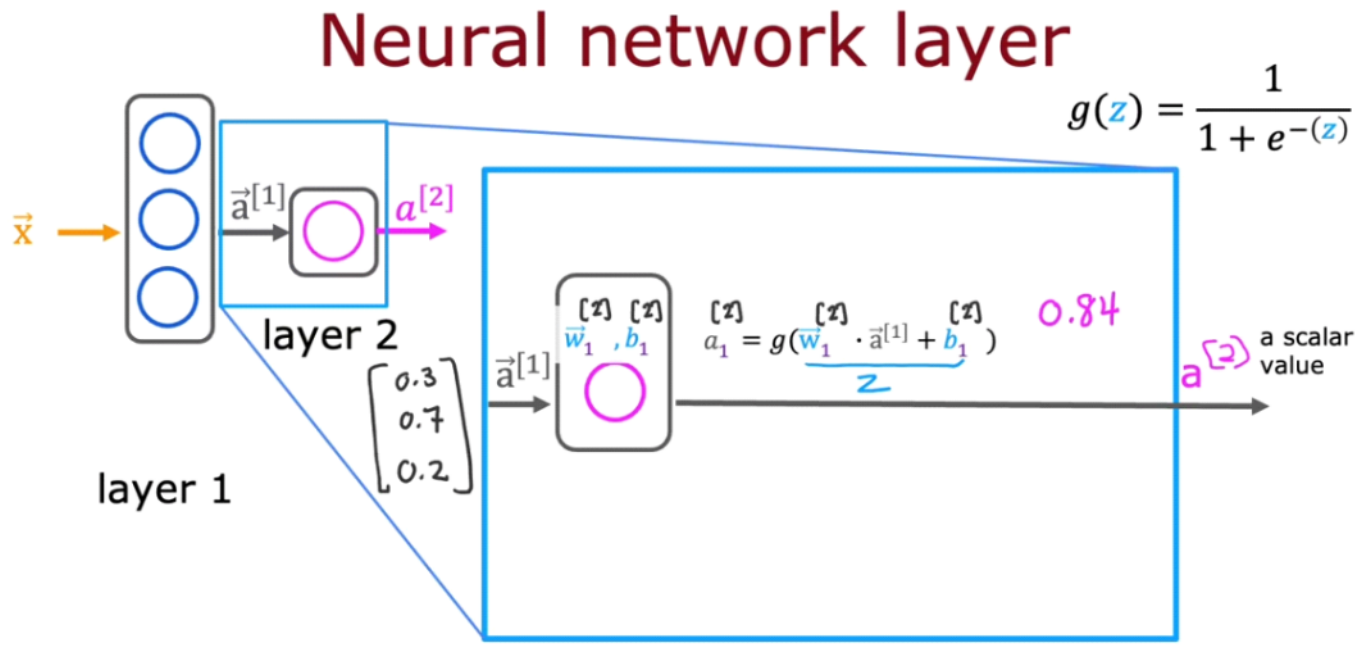
To distinguish elements within a layer, we'll use superscripts in square brackets. For instance:

- $a^{[1]}$: This denotes the activation vector (output) of layer 1.
- $w_1^{[1]}$, $b_1^{[1]}$: These represent the parameters (weights and bias) of the first neuron in layer 1.

This superscript notation applies to all layers (layer 2, layer 3, and so on).

Computation in Multiple Layers

The output (activation vector) of layer 1 ($a^{[1]}$) becomes the input to layer 2 (the output layer). In this layer, the single neuron performs a similar computation as the hidden layer neurons, using its weights and bias to get a final output ($a^{[2]}$).

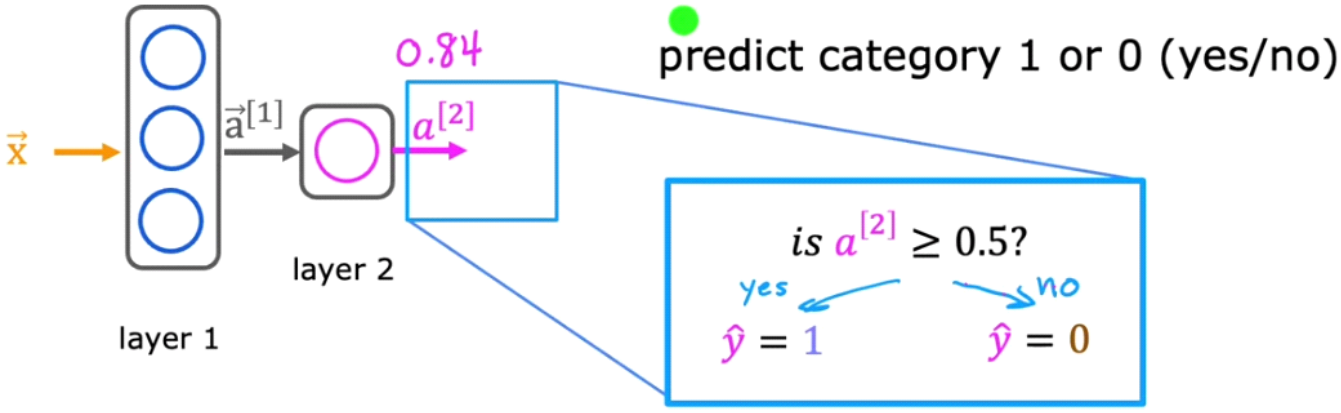


Since the output layer has just one neuron, the final output of the entire network is a single scalar value ($a^{[2]}$).

Thresholding for Binary Classification

If you desire a binary prediction (1 or 0), you can apply a threshold (typically 0.5) to the output ($a^{[2]}$). Values greater than 0.5 are predicted as 1, and values less than 0.5 are predicted as 0. This is similar to logistic regression.

Neural network layer



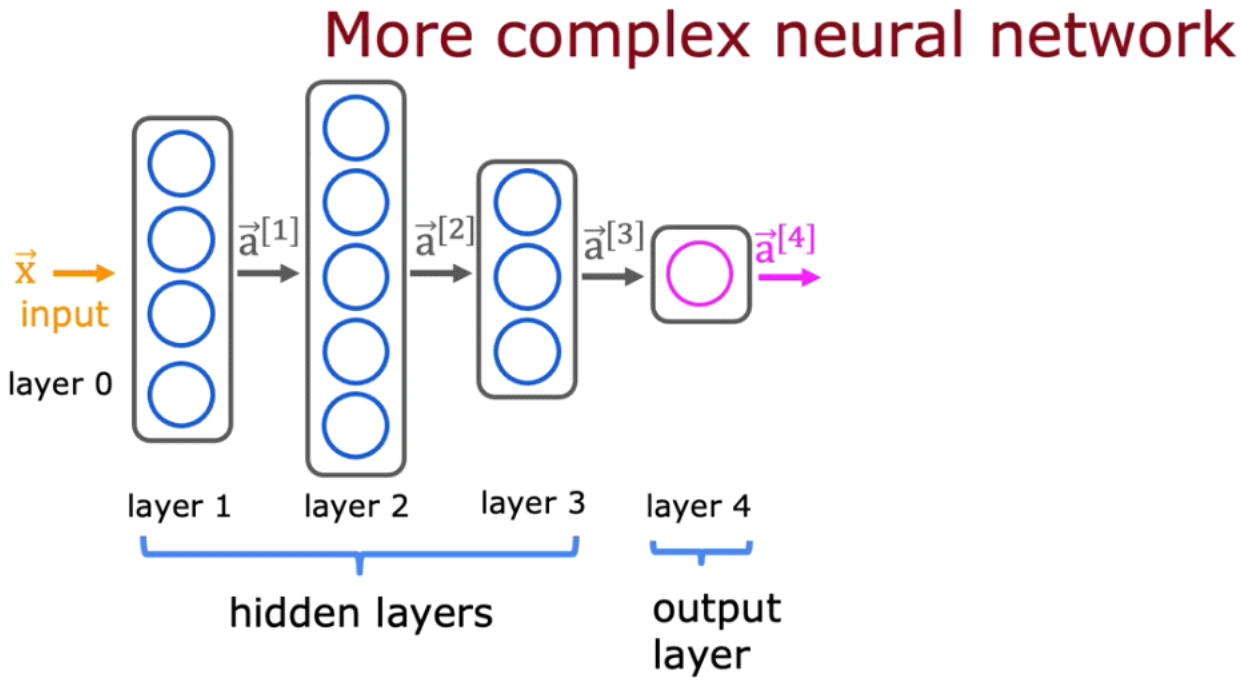
More complex NN

15 May 2024 13:05

A Sample Neural Network with Four Layers

This example network has four layers (excluding the input layer):

- Layer 1, 2, and 3: Hidden layers
- Layer 4: Output layer



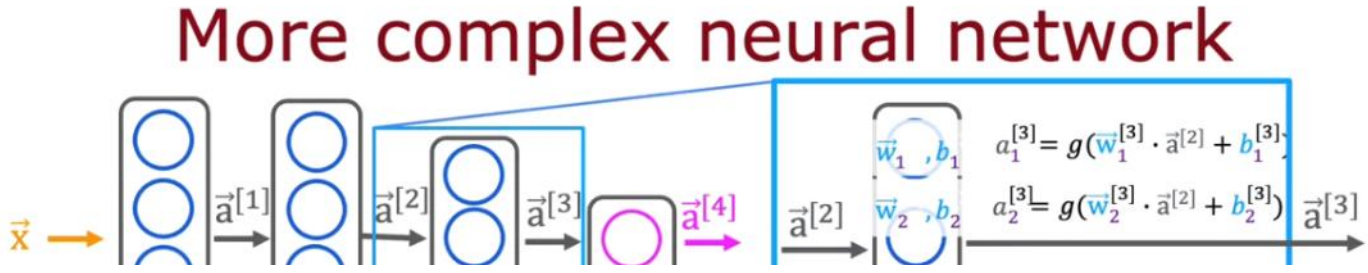
We'll use the conventional way of counting layers, which **excludes the input layer.**

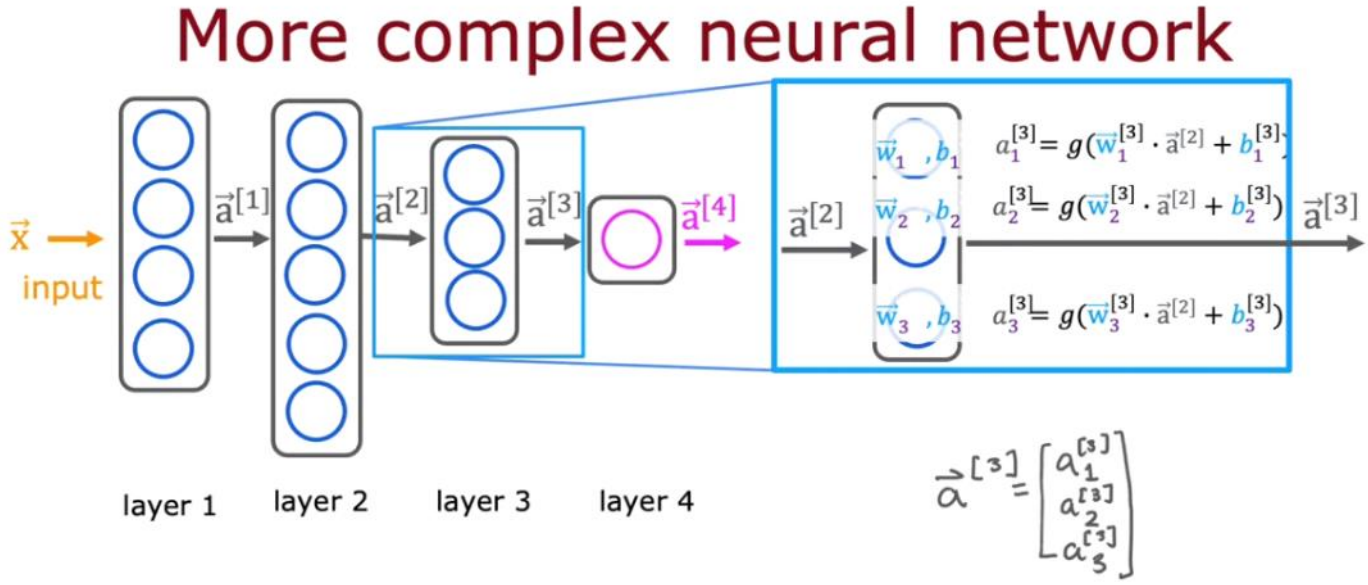
Understanding Computations in Layer 3

Let's focus on Layer 3, the third and final hidden layer. This layer takes an input vector $a^{[2]}$ computed by the previous layer and outputs another vector $a^{[3]}$.

Each neuron in Layer 3 performs the following computation:

1. Applies a sigmoid function (activation function) to a weighted sum of its inputs (including a bias term).
2. The weights (w) and biases (b) are specific to that neuron.





For example, if Layer 3 has three neurons, it might have parameters:

- $w_1^{[3]}$, $b_1^{[3]}$ for the first neuron
- $W_2^{[3]}$, $b_2^{[3]}$ for the second neuron
- $W_3^{[3]}$, $b_3^{[3]}$ for the third neuron

The activation value $a_1^{[3]}$ of the first neuron would be calculated as:

$$a_1^{[3]} = \text{sigmoid}(w_1^{[3]} * \mathbf{a^{[2]}} + b_1^{[3]})$$

where * denotes vector dot product.

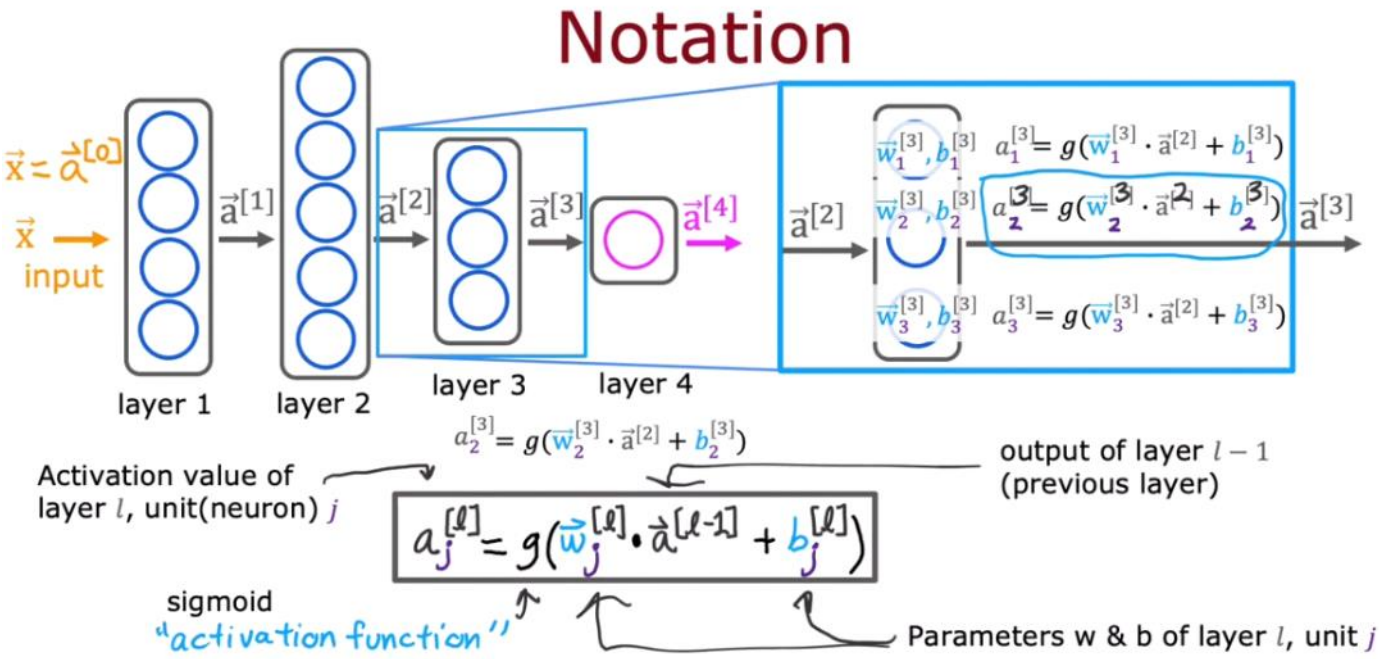
Similarly, we can calculate the activation values (a_2^{[3]} and a_3^{[3]}) for the second and third neurons.

The output of Layer 3 is a vector containing the activation values of all three neurons (a_1^{[3]}, a_2 ^{[3]}, a_3^{[3]}).

Notation Review

We use superscripts in square brackets to denote the layer associated with a parameter or activation value. For instance:

- $a^{[3]}$: Activation vector of Layer 3
- $w_1^{[3]}$, $b_1^{[3]}$: Parameters (weights and bias) of the first neuron in Layer 3



Activation Function and Notation for Input Layer

- The sigmoid function (g) is also called the activation function because it outputs activation values.
- We can use $a^{[0]}$ to denote the input vector (X) for consistency with the notation for other layers.

Forward Propagation

15 May 2024 14:54

Motivating Example: Handwritten Digit Recognition

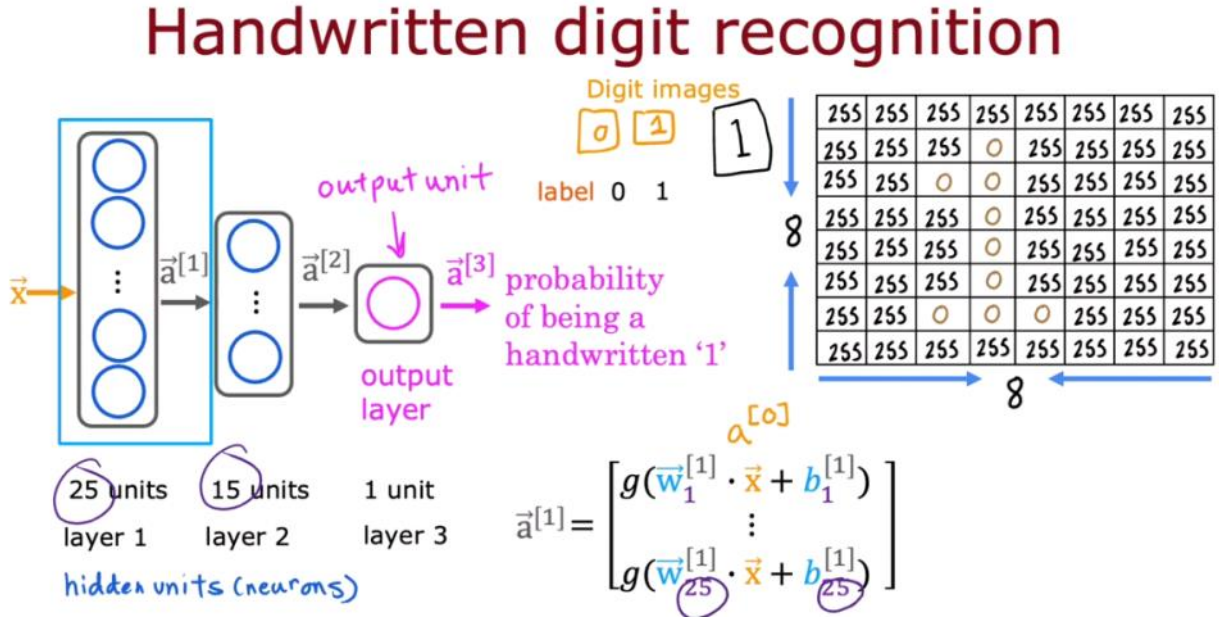
We'll build a neural network to classify handwritten digits as either zero or one. The input will be an 8x8 image (64 pixels), and the output will be the probability of the digit being a one.

Network Architecture

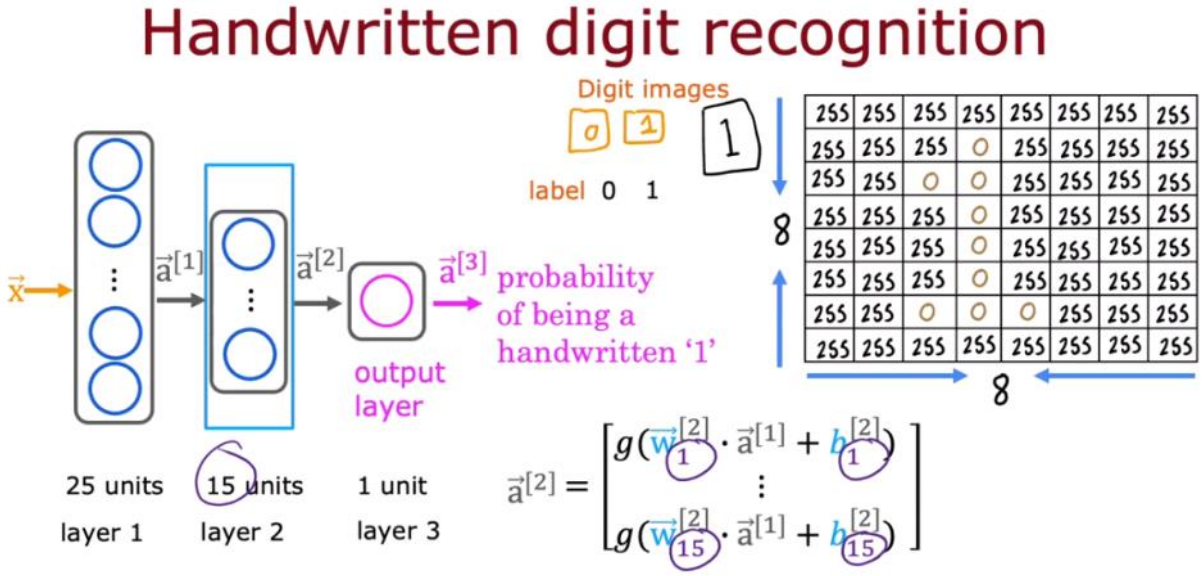
The network will have:

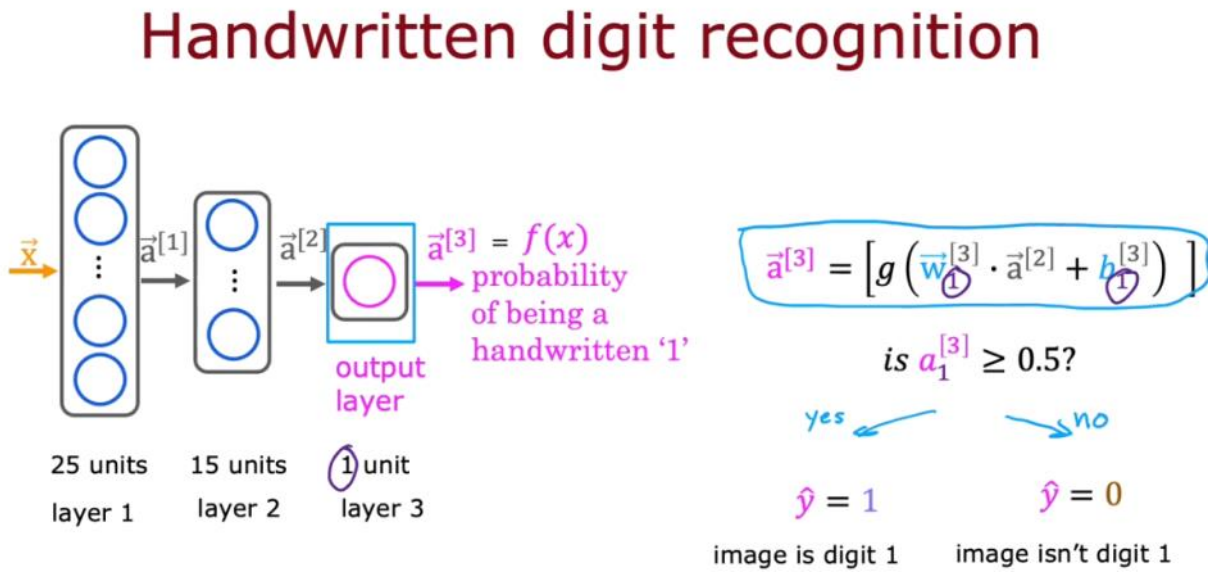
- Input layer: 64 neurons (one for each pixel)
- Hidden layer 1: 25 neurons
- Hidden layer 2: 15 neurons
- Output layer: 1 neuron (probability of digit being one)

Forward Propagation Steps

1. **Input to Hidden Layer 1 (a1):**
    - This step calculates the weighted sum of the inputs (x) and adds a bias term (b).
    - An activation function is applied to introduce non-linearity.
    - 

    -
2. **Hidden Layer 1 to Hidden Layer 2 (a2):**
    - Similar to step 1, we calculate the weighted sum of the previous layer's activations (a1) and add a bias.

- Another activation function is applied.
- 

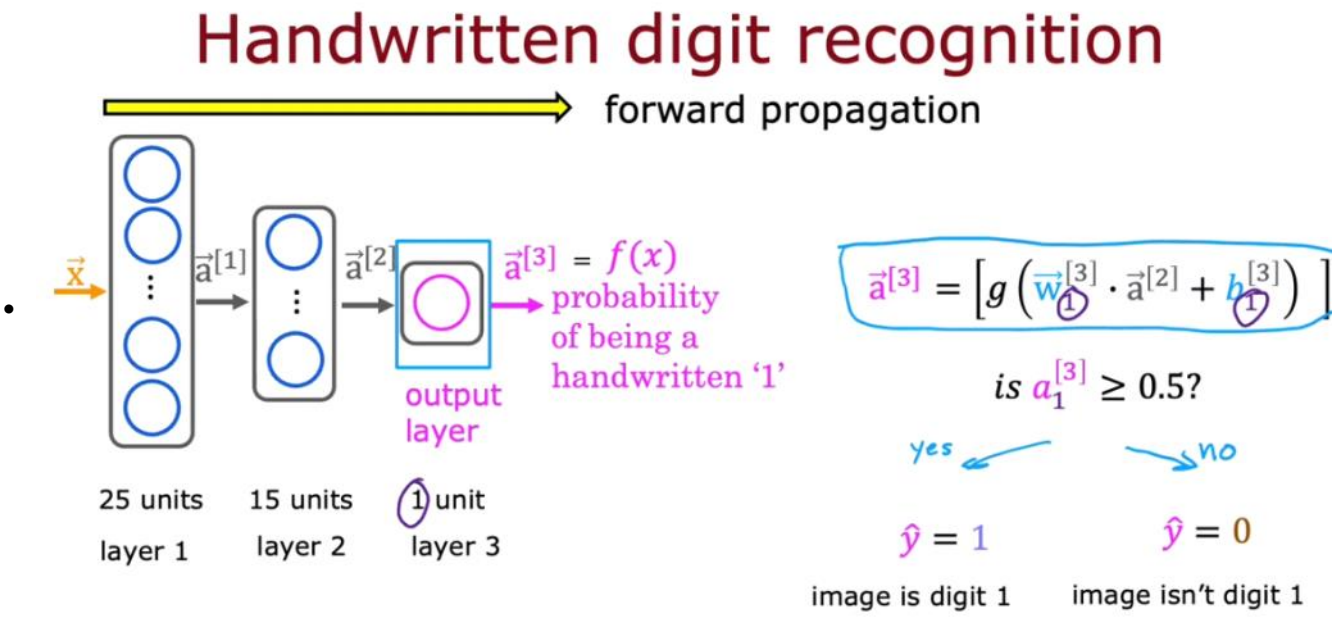
3. **Hidden Layer 2 to Output Layer (a3):**
   - The final step calculates the weighted sum of the previous layer's activations (a2) and adds a bias (for the output layer with one neuron).
   - An activation function (often sigmoid in classification problems) is applied to get the output between 0 and 1, representing the probability of the digit being one.
   - 

f(x): The Neural Network Function

Similar to linear regression, f(x) represents the function computed by the neural network to map the input (x) to the output (a3).

Forward Propagation vs. Backpropagation

- Forward propagation calculates the activations from input to output (inference).

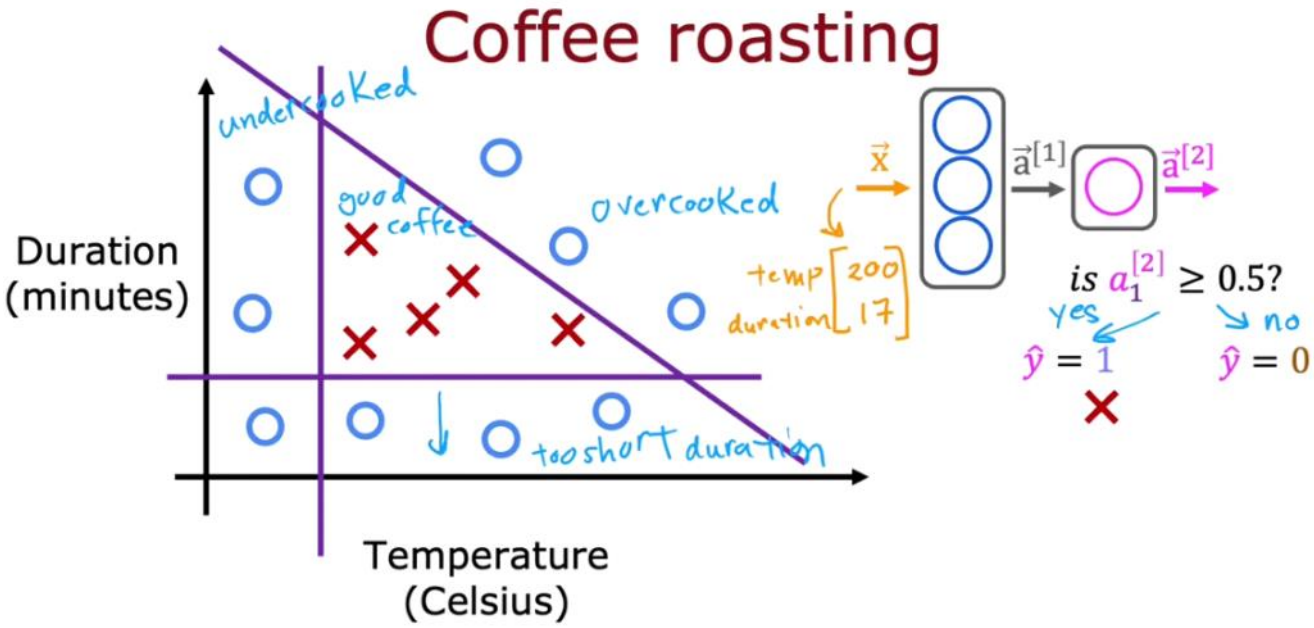


- Backpropagation (covered next week) propagates errors backwards to adjust network weights during training.

15 May 2024 21:15

Coffee Roasting and Neural Networks

Imagine using a machine learning algorithm to optimize coffee roasting! Temperature and duration are two crucial factors affecting the quality of roasted beans. A dataset can be created containing these parameters and labels indicating whether the roasted coffee is good (positive class) or bad (negative class).



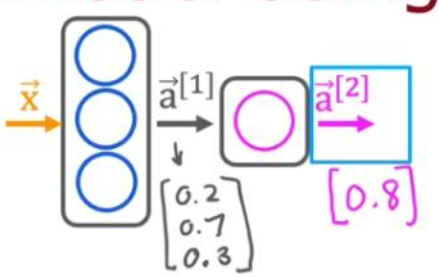
Inference using a Neural Network

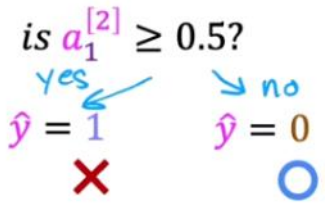
Given a temperature and duration (feature vector x), how can we utilize a neural network to predict if this setting will result in good coffee?

Here's a breakdown of the process:

1. **Define Input (x):**
    - x is a two-element array representing temperature (e.g., 200 degrees Celsius) and duration (e.g., 17 minutes).
2. **Hidden Layer 1 (a1):**
    - This layer (Layer 1) is a dense layer with three units and uses the sigmoid activation function.
    - Applying Layer 1 to x yields a1, a list of three numbers (activations) due to the three units in the layer.
3. **Hidden Layer 2 (a2):**
    - Layer 2 is another dense layer, but with one unit and the sigmoid activation function.
    - a2 is computed by applying Layer 2 to the activations (a1) from Layer 1.
4. **Output and Thresholding (y-hat):**
    - Optionally, a threshold of 0.5 can be applied to a2. If a2 is greater than or equal to 0.5, y-hat (predicted class) is set to 1 (good coffee), otherwise 0 (bad coffee).

Build the model using TensorFlow





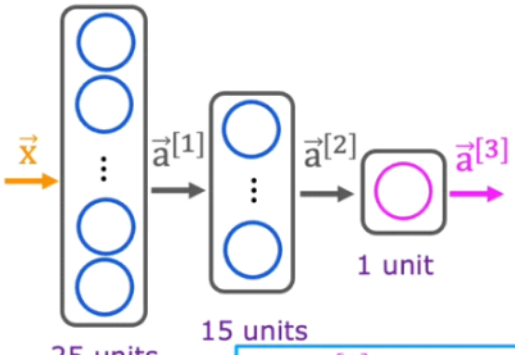

```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)


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

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

Code for previous digit classification problem:

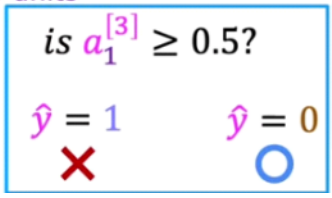
Model for digit classification




```
x = np.array([[0.0,...245,...240...0]])
layer_1 = Dense(units=25, activation='sigmoid')
a1 = layer_1(x)

layer_2 = Dense(units=15, activation='sigmoid')
a2 = layer_2(a1)

layer_3 = Dense(units=1, activation='sigmoid')
a3 = layer_3(a2)
```




```
if a3 >= 0.5:
  yhat = 1
else:
  yhat = 0
```

Data in Tensorflow

16 May 2024 00:33

NumPy vs. TensorFlow Data Representations

- NumPy: A well-established library for linear algebra operations in Python.
- TensorFlow: A framework designed by Google for deep learning, including neural networks.
- One of the unfortunate things about the way things are done in code today is that many, many years ago NumPy was first created and became a standard library for linear algebra and Python. And then much later the Google brain team, created TensorFlow. So there are some inconsistencies between how data is represented in NumPy and in TensorFlow.

TensorFlow Data Representation

TensorFlow utilizes multi-dimensional arrays called tensors to store data. Here's an example:

```
x = tf.constant([[200, 17]])  # Input features (temperature, duration)
```

This code creates a 1x2 matrix representing the input features (200 degrees Celsius and 17 minutes).

NumPy Data Representation

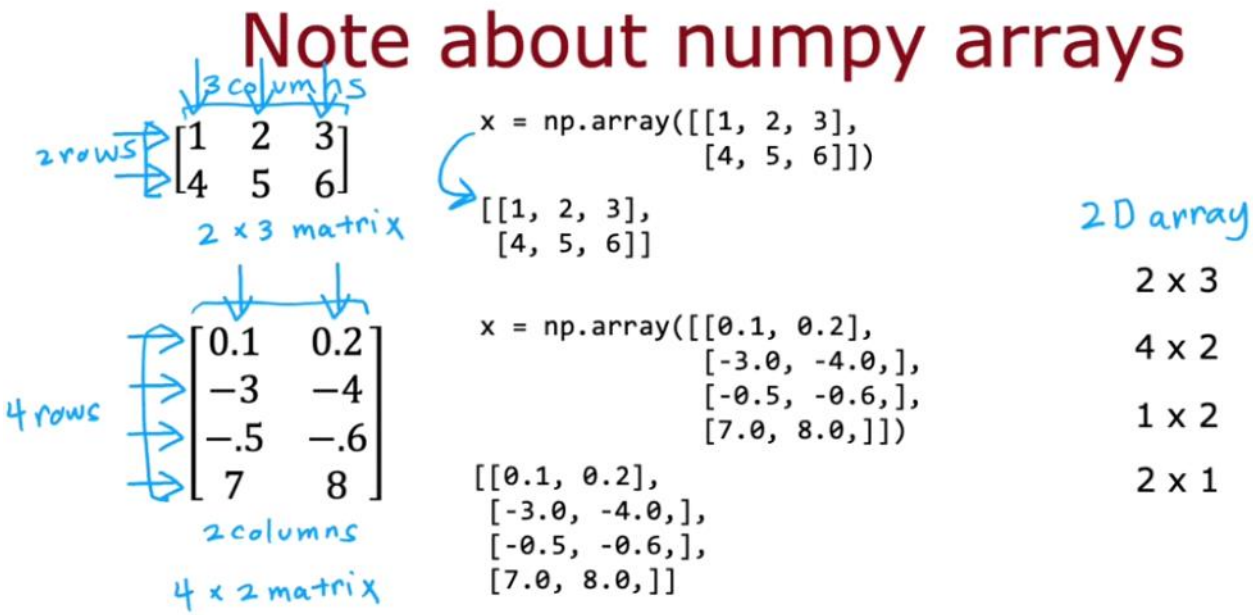
NumPy employs multi-dimensional arrays as well, but with slightly different conventions:

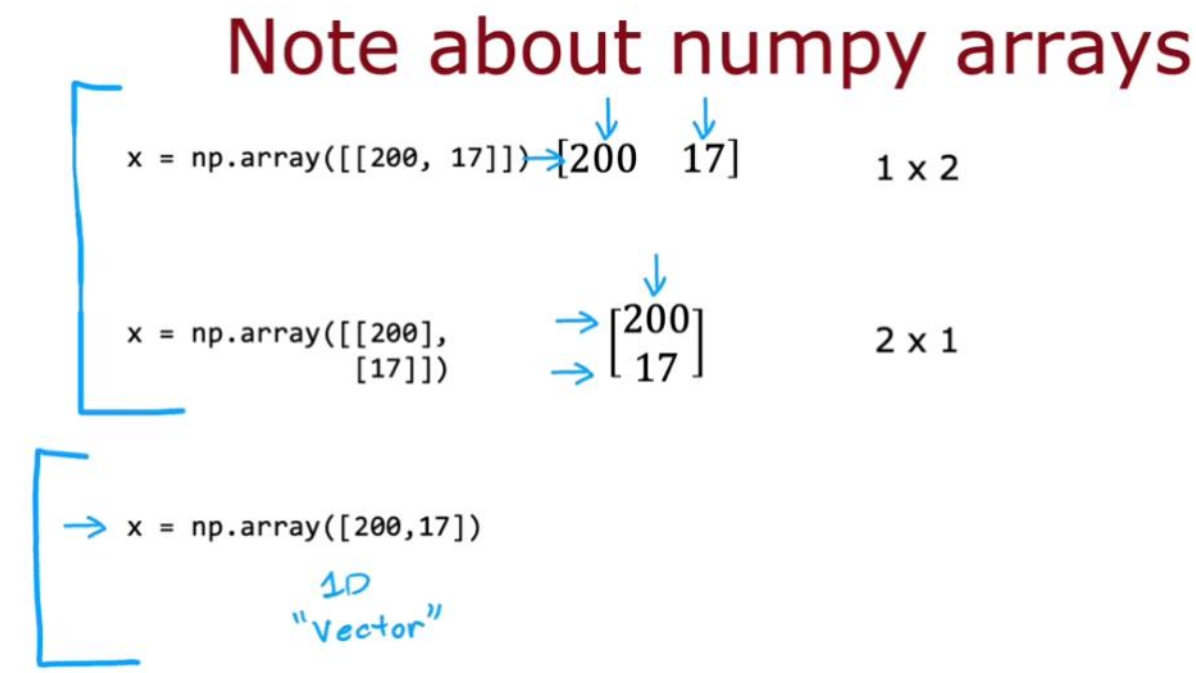
```
import numpy as np
x = np.array([200, 17])  # Input features (temperature, duration)
```

This NumPy code also creates a 1x2 matrix, but the double square brackets indicate a single row.

Key Points

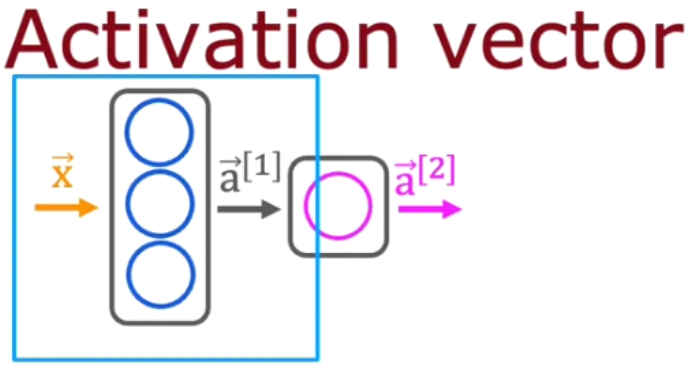
- TensorFlow treats data as 2D matrices by default, even for single rows or columns.
- NumPy allows for more flexibility in representing 1D or 2D data using square brackets.

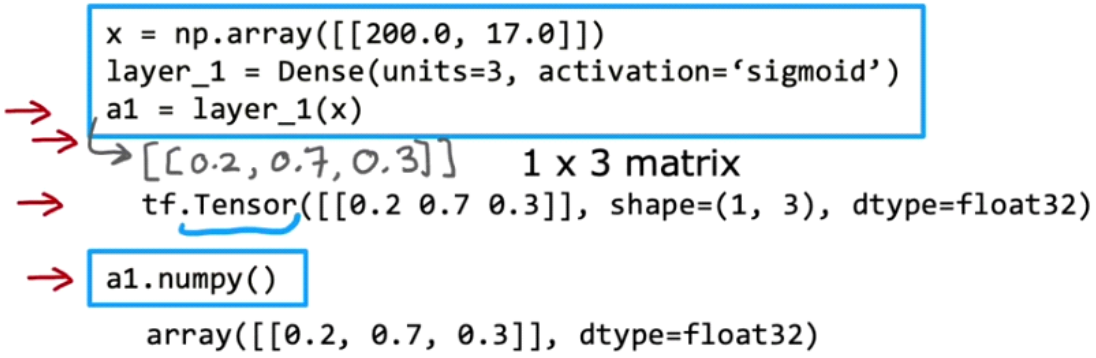




Converting Between NumPy and TensorFlow

- a1.numpy(): Converts a TensorFlow tensor (a1) to a NumPy array.
- np.array(x): Converts a NumPy array (x) to a TensorFlow tensor.



- 

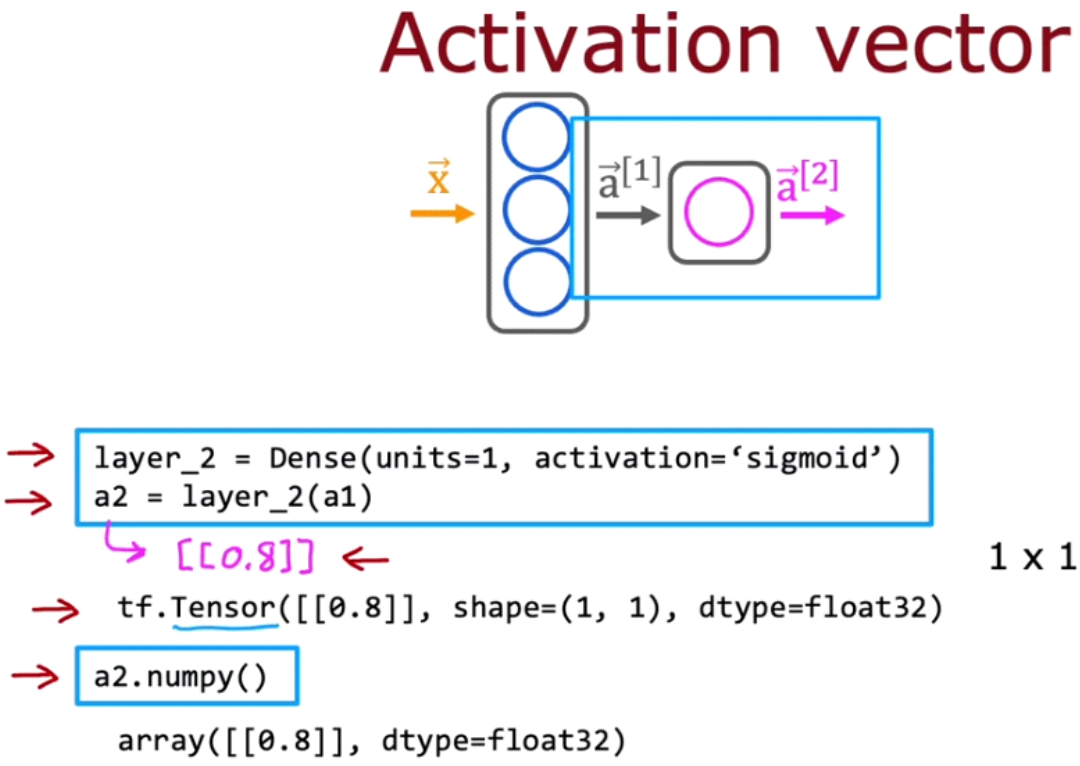
TensorFlow Tensors

- Tensors are a core data structure in TensorFlow, optimized for efficient computations.
- They hold floating-point numbers (float32) by default.
- TensorFlow programs use a tensor data structure to represent all data. You can think of a TensorFlow tensor as **an n-dimensional array or list**. A tensor has a static type and dynamic dimensions. Only tensors may be passed between nodes in the computation graph.

Example: Activation Outputs

a2 = tf.constant([[0.8]]) # Output from layer 2

This TensorFlow code creates a 1x1 matrix representing the output from the second layer (a value of 0.8).



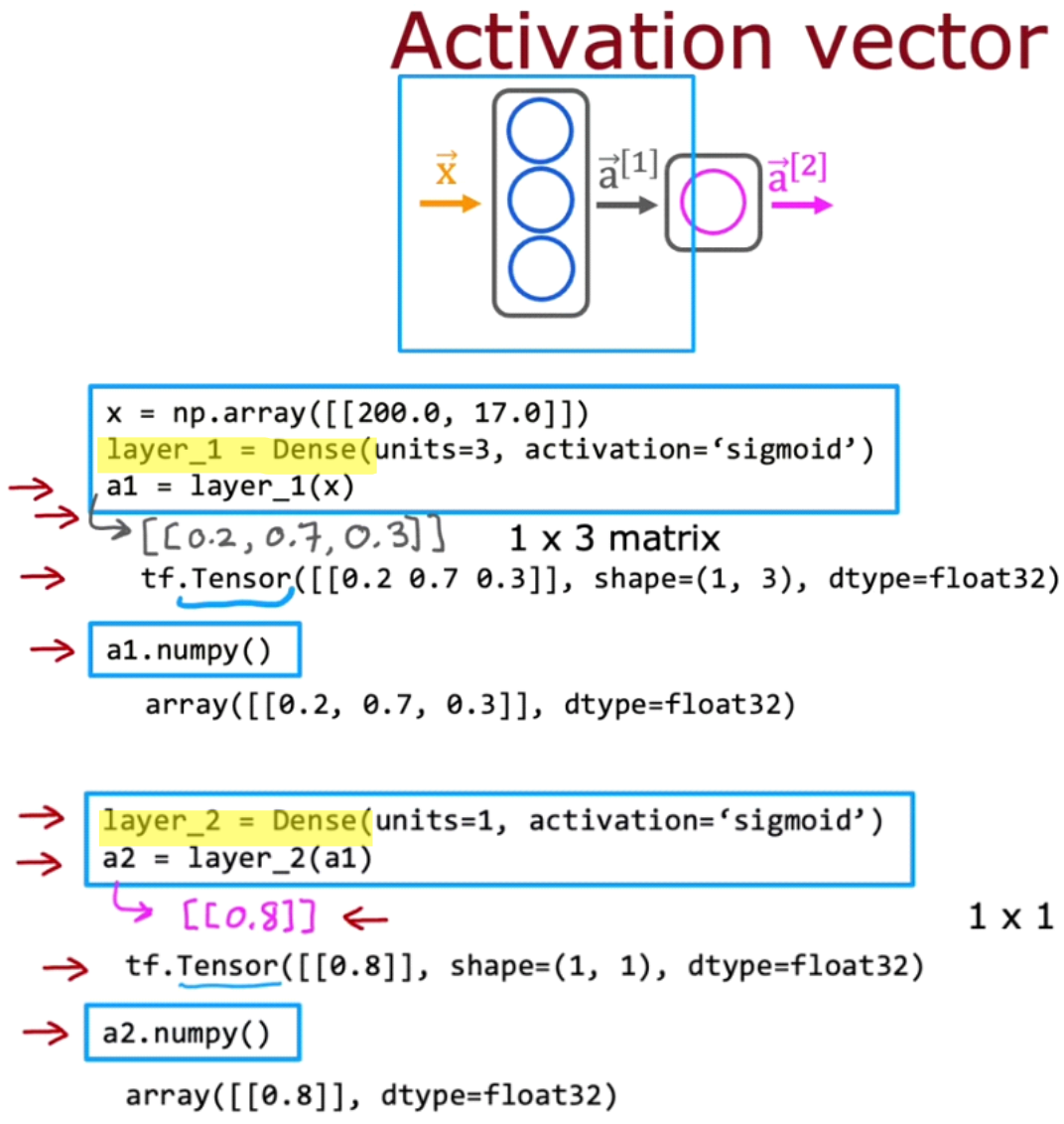
Converting TensorFlow Tensors to NumPy Arrays

a2.numpy() # Converts a2 tensor to a NumPy array

Building a neural network

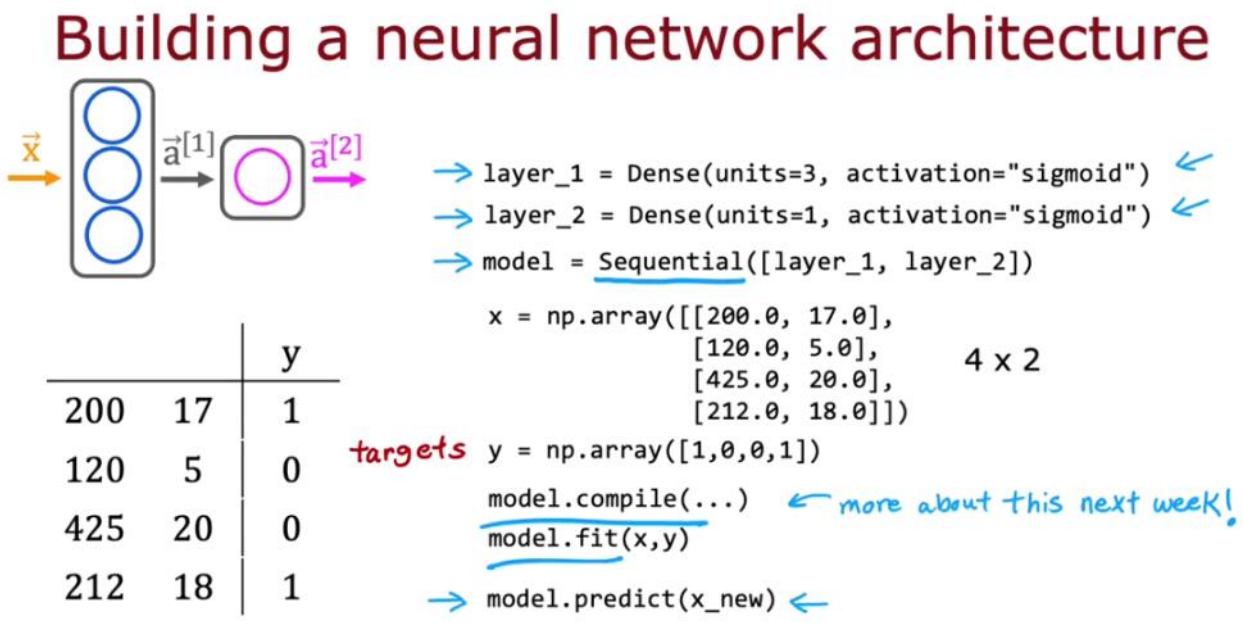
16 May 2024 14:53

Building a Neural Network with Sequential

- **Previous Method:** Explicit layer creation and manual data passing between them. Like this:
- 

If you want to do forward prop, you initialize the data X create layer one then compute a_1, then create layer two and compute a_2. So this was an explicit way of carrying out forward prop one layer of computation at the time.

- **New Method:** Leverage TensorFlow's Sequential function to automatically connect layers.



It turns out that tensor flow has a different way of implementing forward prop as well as learning. But now instead of you manually taking the data and passing it to layer one and then taking the activations from layer one and pass it to layer two. **We can instead tell tensor flow that we would like it to take layer one and layer two and string them together to form a neural network.**

That's what the ***sequential function*** in TensorFlow does which is it says, Dear TensorFlow, please create a neural network for me by sequentially string together these two layers that I just created.

It turns out that with the sequential framework tensor flow can do a lot of work for you. You can then take the training data as inputs X and put them into a numpy array. Similarly you can make a 1D numpy array for values of Y.

If you want to train this neural network, all you need to do is call to functions you need to call **model.compile(…)** with some parameters. And then you need to call model dot fit X Y, which tells tensor flow to take this neural network that are created by sequentially string together layers one and two, and to train it on the data, X and Y.

How do you do forward prop if you have a new example, say X new, which is NP array with these two features than to carry out forward prop instead of having to do it one layer at a time yourself, you just have to call **model.predict()** on X new and this will output the corresponding value of a two for you given this input value of X.

You can also define the layers within the Sequential method like this:

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(3, activation='sigmoid'),  # Layer 1
  tf.keras.layers.Dense(1, activation='sigmoid')   # Layer 2
])
```

This code defines a sequential model with two layers:

- Layer 1: Dense layer with 3 units and sigmoid activation.
- Layer 2: Dense layer with 1 unit and sigmoid activation.

Data and Training

1. **Data Preparation:**

- X: Training data features (NumPy array, 4x2 matrix for the coffee example).
- Y: Target labels (NumPy array, 1D array of length 4).

2. **Model Training:**

```
model.compile(...)  # We'll discuss compilation next week
model.fit(X, Y)
```

- Calling model.compile configures the model for training (details in next video).
- model.fit(X, Y) trains the model on the provided data (X) and labels (Y).

Inference

1. **New Example:** X_new (NumPy array with new features)

2. **Prediction:**

```
prediction = model.predict(X_new)
```

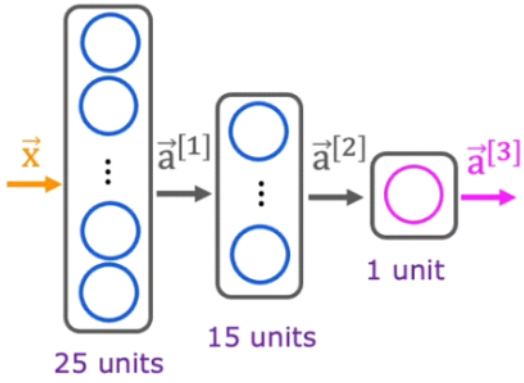
- model.predict(X_new) performs forward propagation on the new data (X_new) and returns the predicted output.

Code Simplification

In TensorFlow, it's common to directly define layers within the Sequential function instead of separate assignments:

```
model = tf.keras.Sequential([
  tf.keras.layers.Dense(3, activation='sigmoid'),
  tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Digit classification model




```
layer_1 = Dense(units=25, activation="sigmoid")
layer_2 = Dense(units=15, activation="sigmoid")
layer_3 = Dense(units=1, activation="sigmoid")

model = Sequential([layer_1, layer_2, layer_3])
model.compile(...)

x = np.array([[0..., 245, ..., 17],
              [0..., 200, ..., 184]])
y = np.array([1,0])

model.fit(x,y)

model.predict(x_new)
```

Forward Propagation from scratch

16 May 2024 22:22

Here we will implement forward propagation (forward prop) in Python from scratch, providing a deeper understanding of neural network libraries like TensorFlow and PyTorch.

Why Implement Forward Prop Yourself?

- Gain intuition about what libraries like TensorFlow and PyTorch do internally.
- Potential future development of even better deep learning frameworks.

Forward Prop in a Single Layer

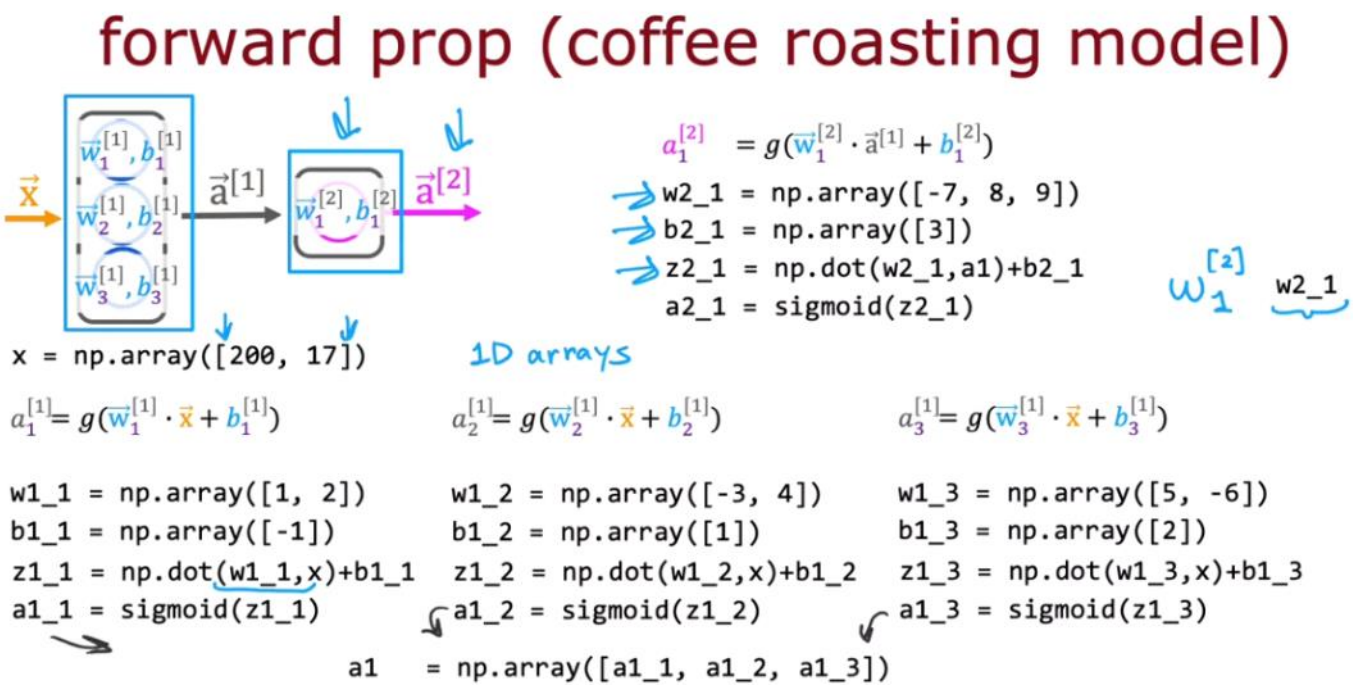
- Example: Coffee roasting model with input x and output a2.
- Using 1D arrays (single square brackets) to represent vectors and parameters.

Computing Activations (a1)

1. **a1_1:**
    - w2_1 and b1_1 are parameters (e.g., 1.2, -1).
    - z1_1 = np.dot(w1_1, x) + b1_1 (dot product of w1_1 and x plus b1_1).
    - a1_1 = g(z1_1) (apply sigmoid function g to z1_1).
2. **a1_2 and a1_3:** Similar calculations using corresponding parameters w1_2, b1_2, w1_3, and b1_3.
3. **a1:** Combine a1_1, a1_2, and a1_3 into a single array using np.array.

Computing Output (a2)

1. a2 is calculated using w2_1, b2_1, and a1 (output from the first layer).
2. z2_1 = np.dot(w2_1, a1) + b2_1.
3. a2 = g(z2_1).



Dense Function for a Single Layer

1. **Function Definition:** dense(a, W, b) takes activation (a) from the previous layer, weight matrix (W), and bias vector (b) as input.

2. **Number of Units:** units = W.shape[1] determines the number of units (neurons) in the current layer based on the number of columns in W.

3. **Initializing Activations:** a = np.zeros(units) creates an array of zeros to store the activations for this layer.

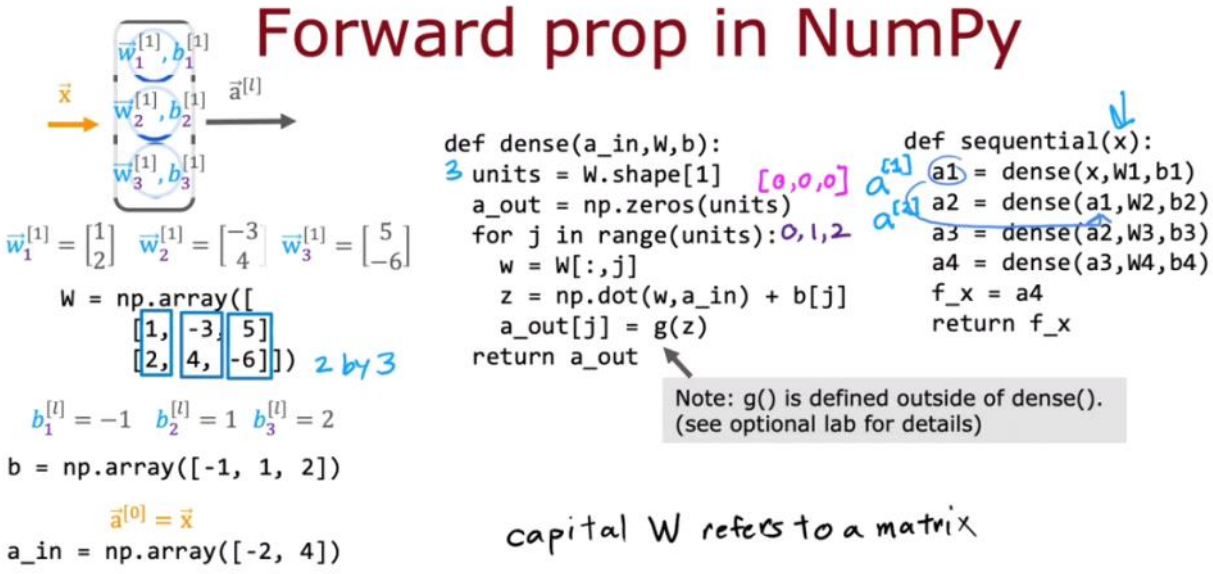
4. **Computing Activations (Loop):**
   - The loop iterates for each unit (j) in the layer.
   - w[:, j] extracts the jth column of W (weights for the current unit).
   - z = np.dot(w[:, j], a) + b[j] calculates the weighted sum (z) for the unit using dot product and bias.
   - a[j] = g(z) applies the activation function (g) to get the activation value for unit j.

5. **Returning Activations:** The function returns the final vector a containing activations for all units in the layer.

Forward Prop with Multiple Layers

1. **Sequential Layer Activation:**
    - a_1 = dense(x, W_1, b_1) computes activations for the first layer using x (input), W_1 (weights), and b_1 (biases).
    - Subsequent layers use the activation from the previous layer as input:
        - a_2 = dense(a_1, W_2, b_2)
        - a_3 = dense(a_2, W_3, b_3) (and so on)

2. **Output:** f(x) = a_L (where L is the last layer) represents the network's output for a given input x.



TensorFlow Notation

Capital W is used for weight matrices (uppercase for matrices, lowercase for vectors/scalars in linear algebra).

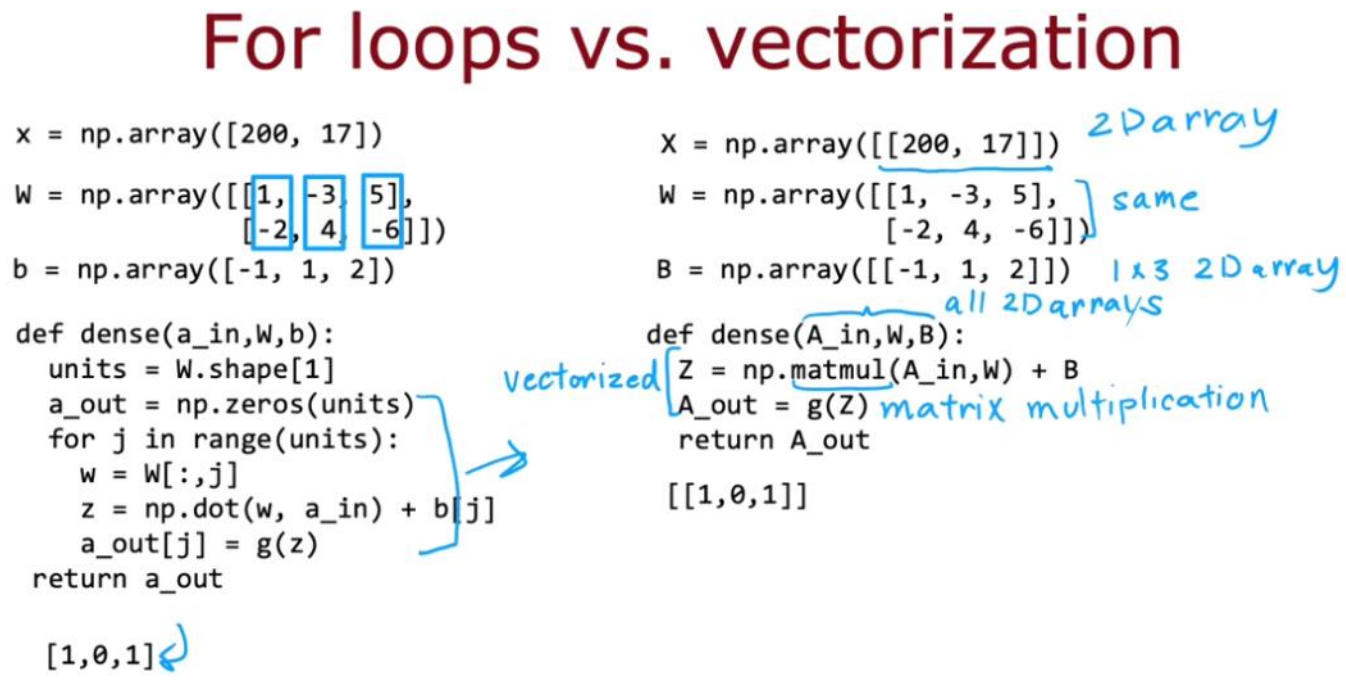
Vectorization (efficient NN implementation)

17 May 2024 00:51

Deep learning researchers have achieved significant progress in scaling up neural networks due to their ability to be vectorized. This efficiency stems from implementing them using matrix multiplications, which are well-suited for parallel computing hardware, particularly GPUs and certain CPU functions. This video explores how these vectorized implementations work.

Non-vectorized Implementation

The code snippet showcases a non-vectorized implementation for forward propagation in a single layer. It includes variables for input (X), weights (W), biases (B), and performs calculations within a loop to generate the output.



Vectorized Implementation

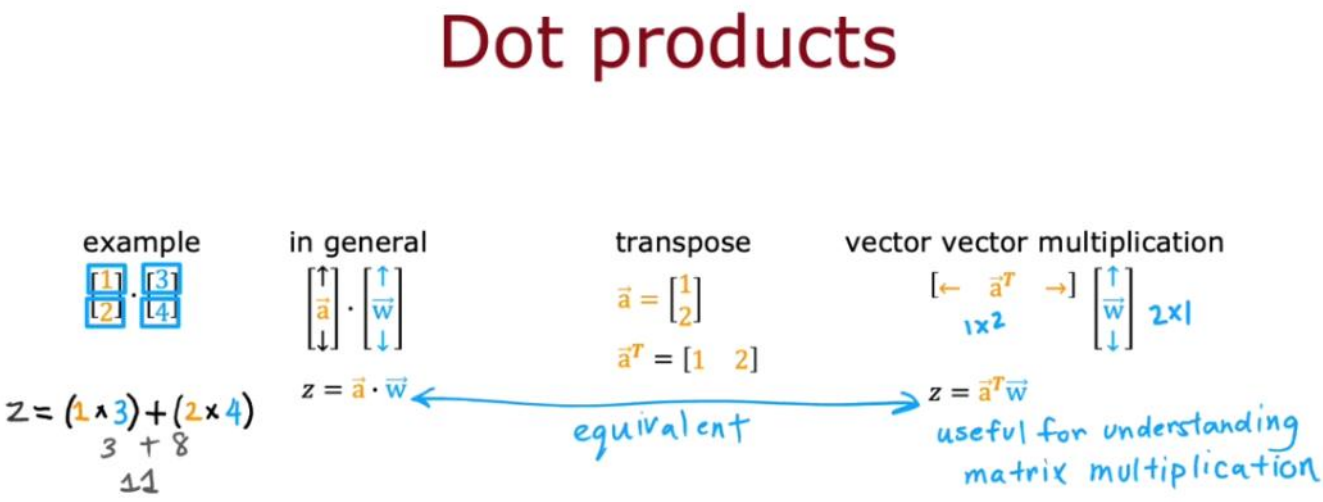
The key concept lies in transforming the non-vectorized code into a vectorized form. This involves:

1. Representing X as a 2D array, similar to TensorFlow.
2. Reshaping B into a one-by-three 2D array.
3. Replacing the loop with a single line using np.matmul (NumPy's matrix multiplication function). This multiplies X and W matrices and adds the B matrix.
4. Applying the activation function (sigmoid) element-wise to the resulting matrix using A_out = g(Z).

This vectorized approach offers a more efficient implementation for a single step of forward propagation in a dense layer.

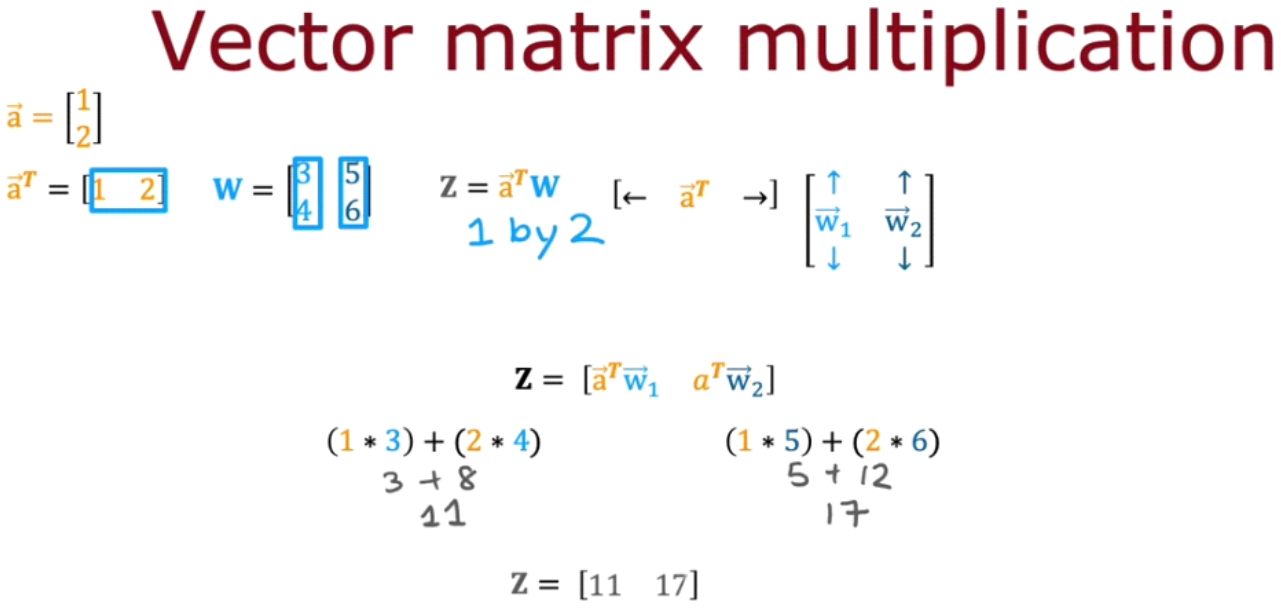
From Vectors to Matrices

1. **Dot Product:** We calculate the dot product between two vectors [1, 2] and [3, 4] by multiplying corresponding elements and summing them (1 * 3 + 2 * 4 = 11).
2. **Vector Transpose:** We can represent a vector as a column or a row. The transpose flips a vector's orientation ([1, 2]^T = [1, 2]).



Vector Matrix Multiplication

We multiply a transposed vector [1, 2]^T with a matrix [[3, 4], [5, 6]] to get a new vector [11, 17]. Each element in the resulting vector is the dot product of the transposed vector and a column of the matrix.



Generalizing to Matrix-Matrix Multiplication

1. **Matrix Transpose:** Similar to vectors, we transpose a matrix by swapping its rows and columns.
2. **Matrix Multiplication as Multiple Dot Products:** We can view matrix multiplication (A * W) as performing multiple dot products between the rows of the transposed first matrix (A^T) and the columns of the second matrix (W). The resulting product matrix (Z) has dimensions based on the number of rows in the first matrix and the number of columns in the second matrix.

matrix matrix multiplication

$$A = \begin{bmatrix} 1 & -1 \\ 2 & -2 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 2 \\ -1 & -2 \end{bmatrix} \quad W = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \quad Z = A^T W = \begin{bmatrix} \leftarrow & \vec{a}_1^T & \rightarrow \\ \leftarrow & \vec{a}_2^T & \rightarrow \end{bmatrix} \begin{bmatrix} \uparrow & \uparrow \\ \vec{w}_1 & \vec{w}_2 \\ \downarrow & \downarrow \end{bmatrix}$$

rows columns

row 1 col 1 / row 1 col 2 / row 2 col 1 / row 2 col 2

$$= \begin{bmatrix} \vec{a}_1^T \vec{w}_1 & \vec{a}_1^T \vec{w}_2 \\ \vec{a}_2^T \vec{w}_1 & \vec{a}_2^T \vec{w}_2 \end{bmatrix}$$

$(-1 \times 3) + (-2 \times 4)$

$-3 + -8$

-11

$(-1 \times 5) + (-2 \times 6)$

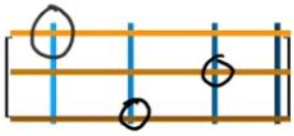
$-5 + -12$

-17

$$= \begin{bmatrix} 11 & 17 \\ -11 & -17 \end{bmatrix}$$

Matrix multiplication rules

$$A = \begin{bmatrix} 1 & -1 & 0.1 \\ 2 & -2 & 0.2 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 2 \\ -1 & -2 \\ 0.1 & 0.2 \end{bmatrix} \quad W = \begin{bmatrix} 3 & 5 & 7 & 9 \\ 4 & 6 & 8 & 0 \end{bmatrix} \quad Z = A^T W =$$



3 by 4 matrix

$\vec{a}_1^T \vec{w}_1 = (1 \times 3) + (2 \times 4) = 11$

row 3 column 2

$\vec{a}_3^T \vec{w}_2 = (0.1 \times 5) + (0.2 \times 6) = 1.7$

$0.5 + 1.2$

row 2 column 3?

$\vec{a}_2^T \vec{w}_3 = (-1 \times 7) + (-2 \times 8)$

$-7 + -16$

Matrix multiplication rules

$$A = \begin{bmatrix} 1 & -1 & 0.1 \\ 2 & -2 & 0.2 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 2 \\ -1 & -2 \\ 0.1 & 0.2 \end{bmatrix} \quad W = \begin{bmatrix} 3 & 5 & 7 & 9 \\ 4 & 6 & 8 & 0 \end{bmatrix} \quad Z = A^T W = \begin{bmatrix} 11 & 17 & 23 & 9 \\ -11 & -17 & -23 & -9 \\ 1.1 & 1.7 & 2.3 & 0.9 \end{bmatrix}$$

$3 \times 2 \quad 2 \times 4$

can only take dot products of vectors that are same length

$[0.1 \ 0.2]$ length 2 $\quad \begin{bmatrix} 5 \\ 6 \end{bmatrix}$ length 2

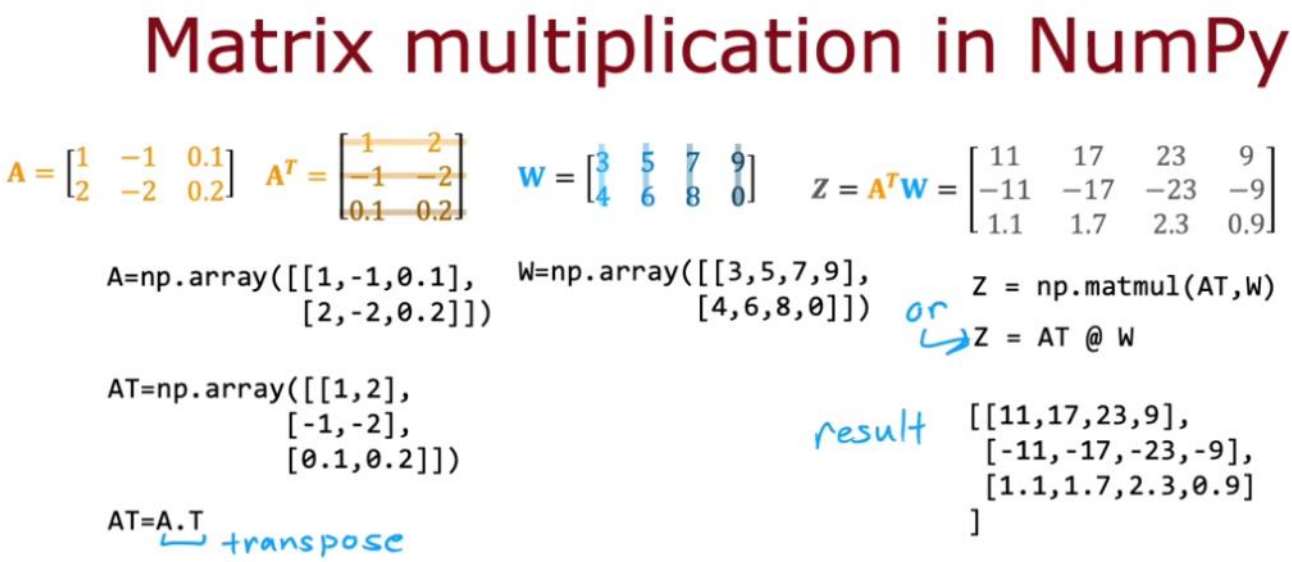
3 by 4 matrix

↳ same # rows as A^T

same # columns as W

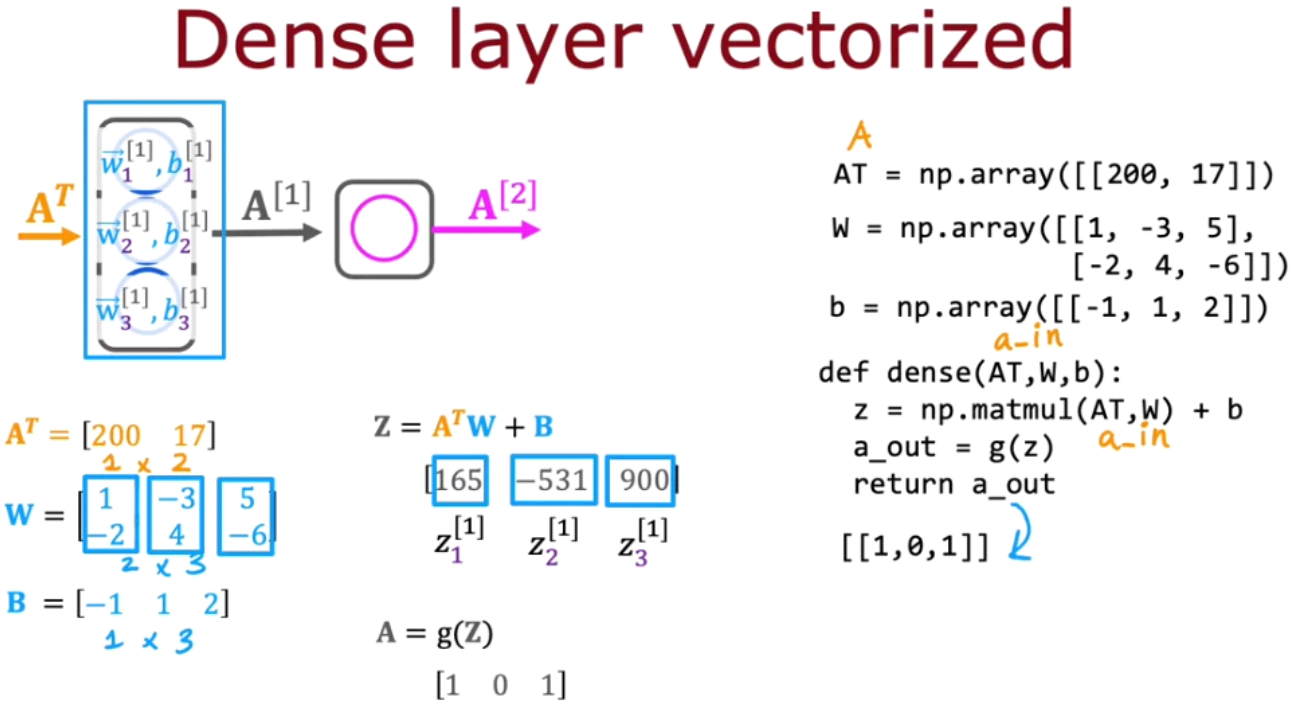
Matrix Dimensions

- **Matrix A:** Represented as a 2x3 matrix, signifying 2 rows and 3 columns.
- **A Transpose (A^T):** Obtained by swapping the positions of rows and columns in matrix A.
- **Matrix W:** A 2x4 matrix, indicating 2 rows and 4 columns.
    - W can be visualized as a combination of vectors w1, w2, w3, and w4 stacked together.



Computing Elements of Z

- **Z:** The resulting product matrix, with dimensions 3x4 (3 rows and 4 columns).
- **Element (1, 1) of Z:** Calculated by finding the dot product of the first row in A^T and the first column in W.



Compatibility for Matrix Multiplication

- **Requirement:** The number of columns in the first matrix (number of rows in A^T) must be equal to the number of rows in the second matrix (W).
- **Reasoning:** Ensures that vectors used in dot products have the same length.
- **Output Dimensions:** The resulting product matrix (Z) has the same number of rows as the first matrix's transposed rows (A^T) and the same number of columns as the second matrix (W).

Training a neural network

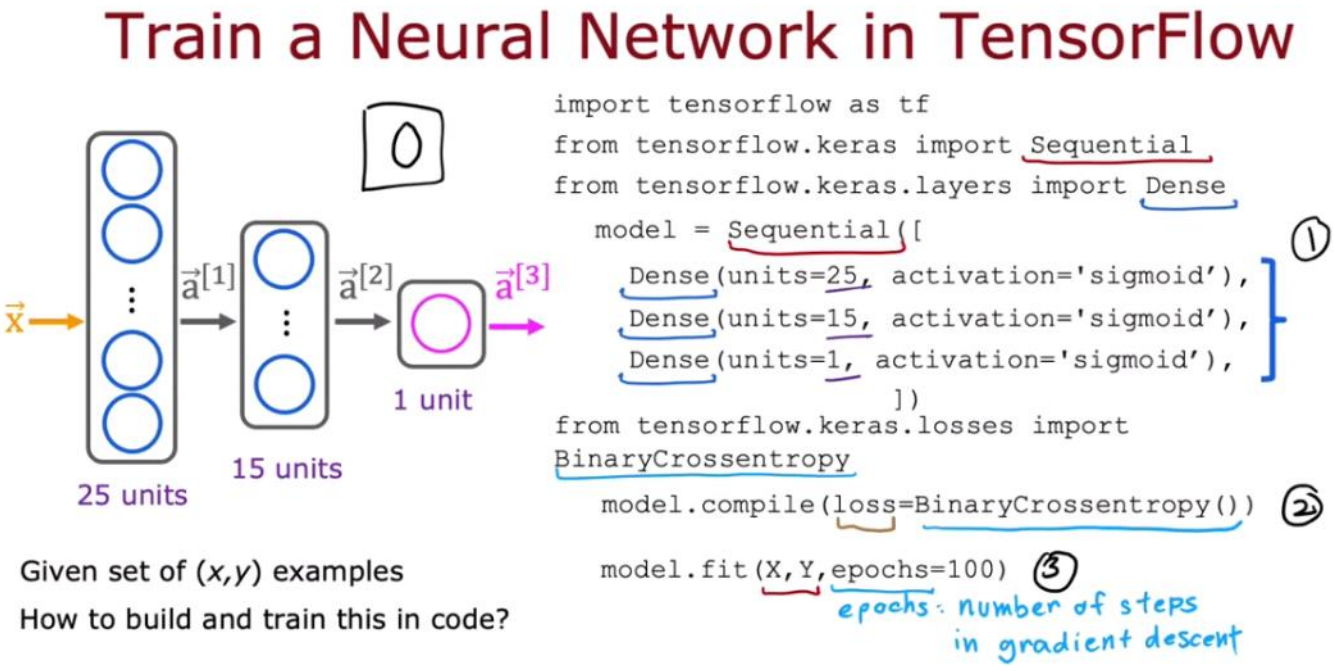
20 May 2024 08:55

Steps to Train a Neural Network:

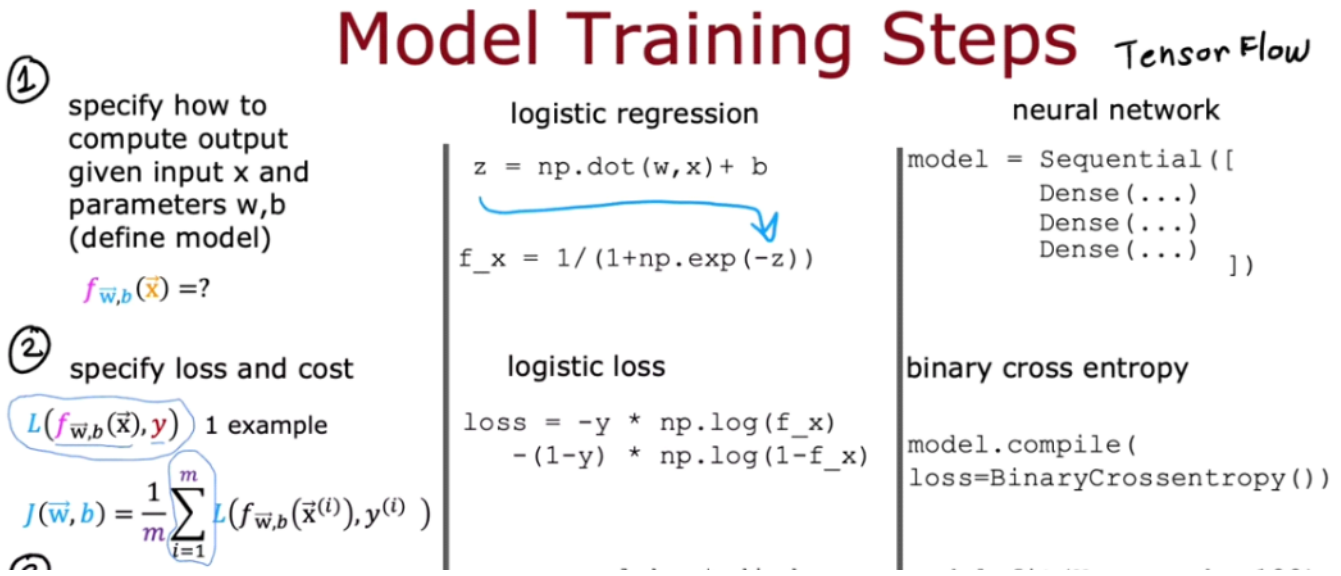
1. **Define the Model Architecture:**
   Use TensorFlow to sequentially create the layers:
   - First hidden layer with 25 units and sigmoid activation
   - Second hidden layer
   - Output layer

2. **Compile the Model:** Specify the loss function (e.g., binary crossentropy)

3. **Train the Model:**
   - Use the fit function to train the model on your dataset (X, Y)
   - **Epochs: The number of times to run the training algorithm**



Example of training a logistic regression model vs a neural network:



$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$

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

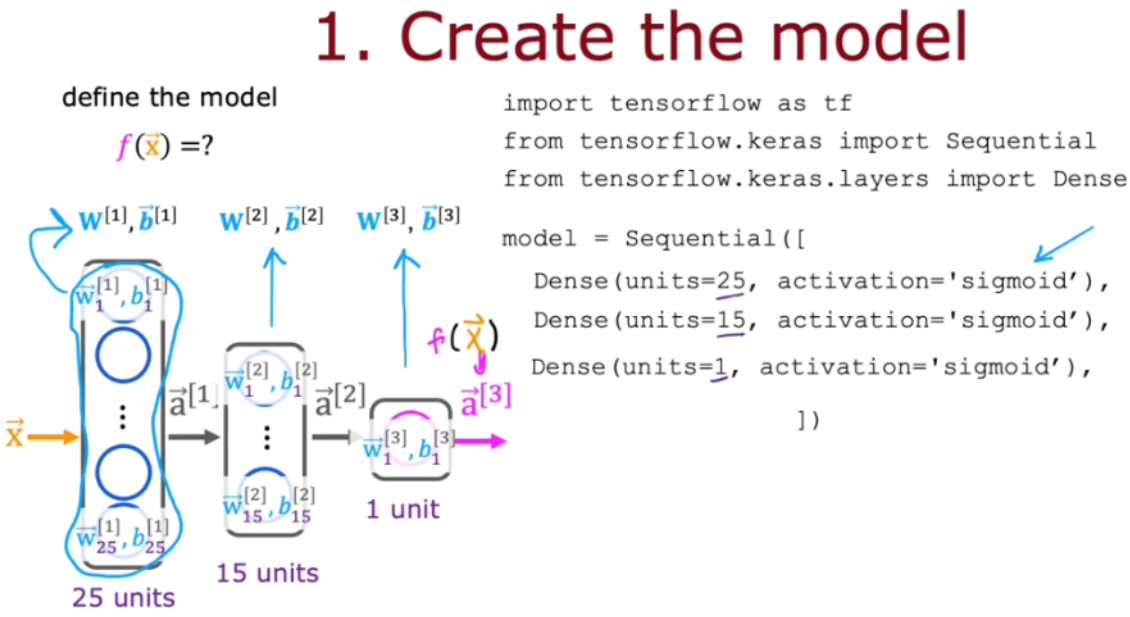
```
w = w - alpha * dj_dw
b = b - alpha * dj_db
```

```
loss=BinaryCrossentropy())

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

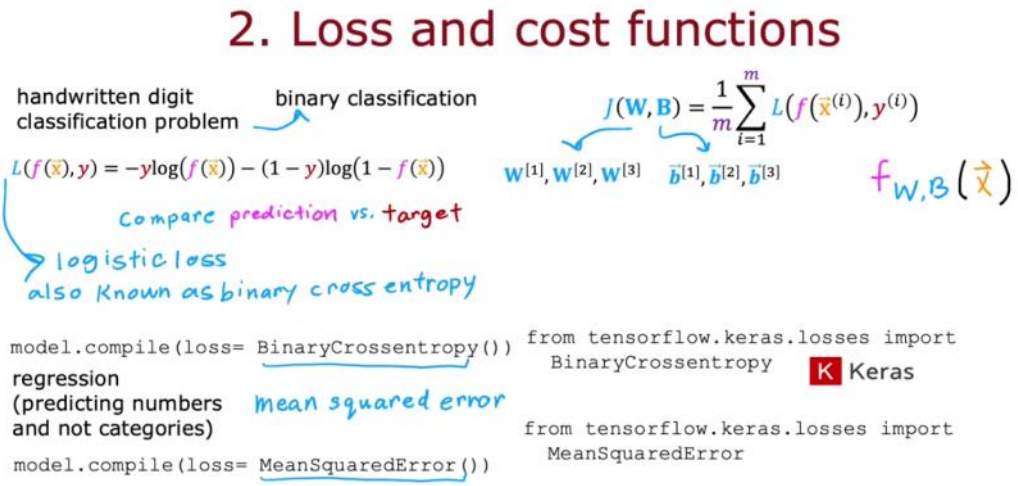
Steps in Detail

1. Specifying Output Computation:

- The code defines the network architecture:
  - 25 hidden units in the first layer with sigmoid activation
  - 15 units in the second layer
  - 1 output unit
- 

- This code snippet tells TensorFlow how to compute the output (f(x)) given the input (x) and parameters (weights and biases).

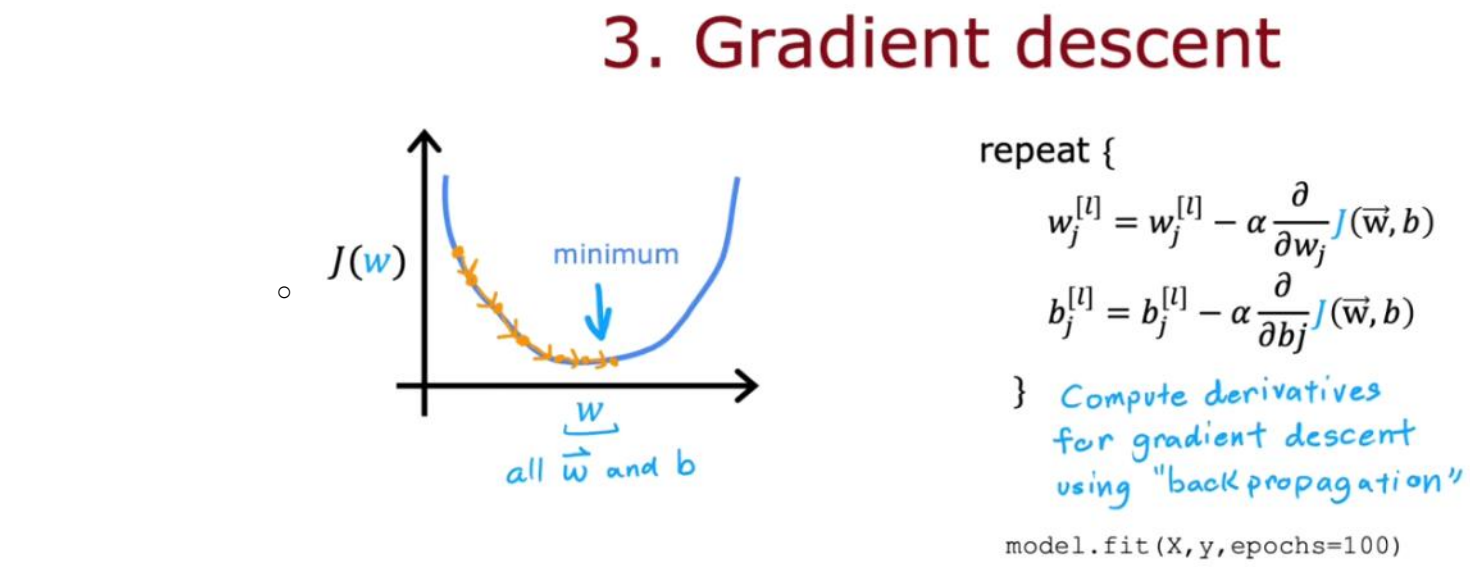
2. Specifying Loss and Cost Function:

- The most common loss function for our binary classification problem (images of zeros or ones) is the binary crossentropy loss function, similar to logistic regression.
  - It measures how well the network performs on a single training example (x, y).
- 

- Compiling the model with this loss function calculates the cost function (average loss over the entire training set).

3. Minimizing the Cost Function:

- This step resembles minimizing the cost function in logistic regression using gradient descent.
- We update the weights (W) and biases (b) to minimize the cost function J(W, b).
-



- TensorFlow's fit function implements backpropagation to compute the partial derivatives needed for gradient descent.

Alternatives to Sigmoid Function

20 May 2024 09:42

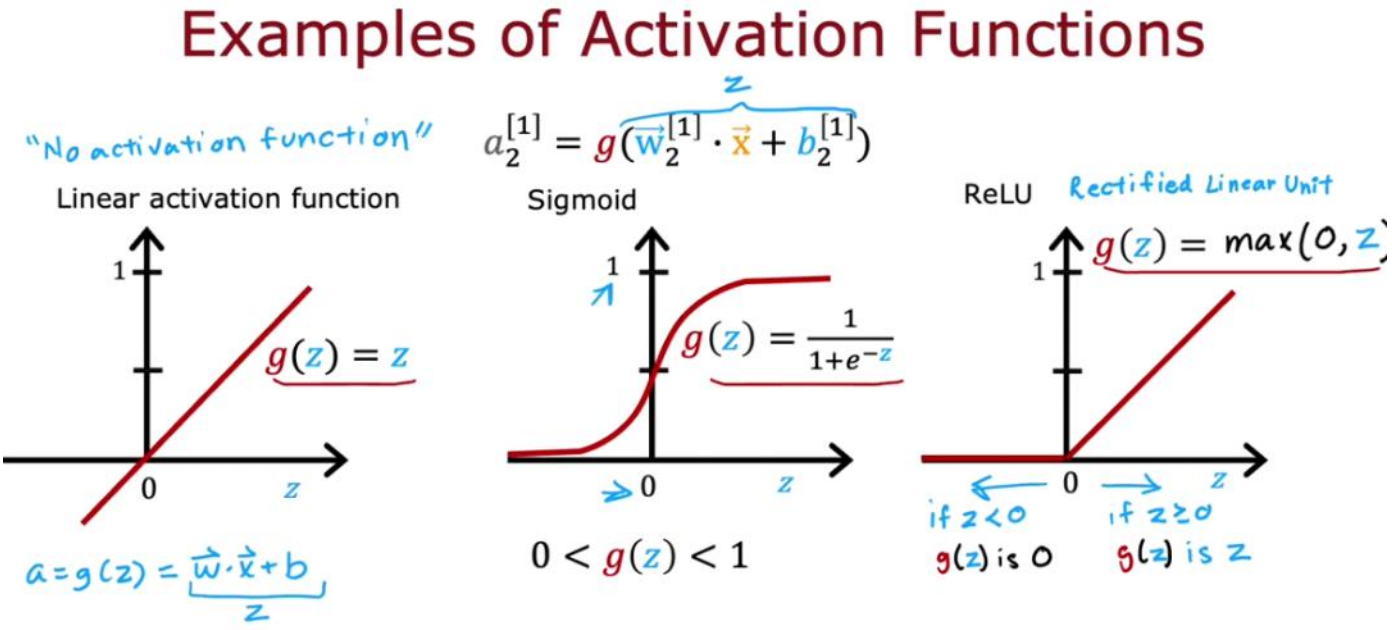
We've been using the sigmoid activation function throughout the hidden and output layers. This choice stemmed from building neural networks from logistic regression units. However, other activation functions can significantly enhance a network's power.

Introducing the ReLU Activation Function

Instead of modeling awareness as a binary value (0 or 1), we can estimate it as a non-negative number (0 to very large values). The sigmoid function limits the output to 0 and 1. To allow for a wider range, we can use the ReLU (Rectified Linear Unit) activation function:
g(z) = { 0, if z < 0
z, if z >= 0 }
This function allows the activation value (a) to take on any non-negative value.

Other Common Activation Functions

- Sigmoid: g(z) = sigmoid function (already seen)
- ReLU: g(z) = max(0, z) (just discussed)
- Linear: g(z) = z (sometimes considered "no activation function")

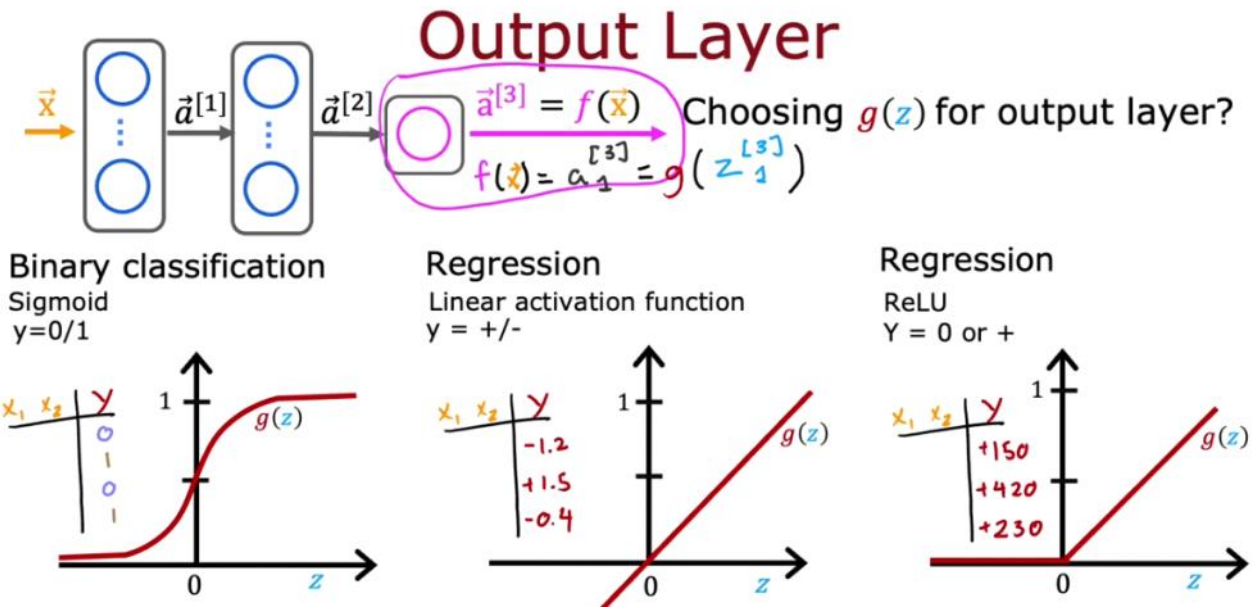


These three are the most frequently used activation functions.
The selection of an activation function for each neuron (sigmoid, ReLU, or linear) is crucial.

Choosing the Output Layer Activation Function

The selection depends on the target label (y):

- **Binary Classification (y = 0 or 1):** Use the sigmoid function. The network learns to predict the probability (like logistic regression).
- **Regression (y: continuous values):** Use the linear activation function. This allows the output (f(x)) to take on any real value. Example: Stock Price which can be negative or positive.
- **ReLU** is used for applications that would require only **non-negative outputs**, like House Prices (since house prices cannot be negative).



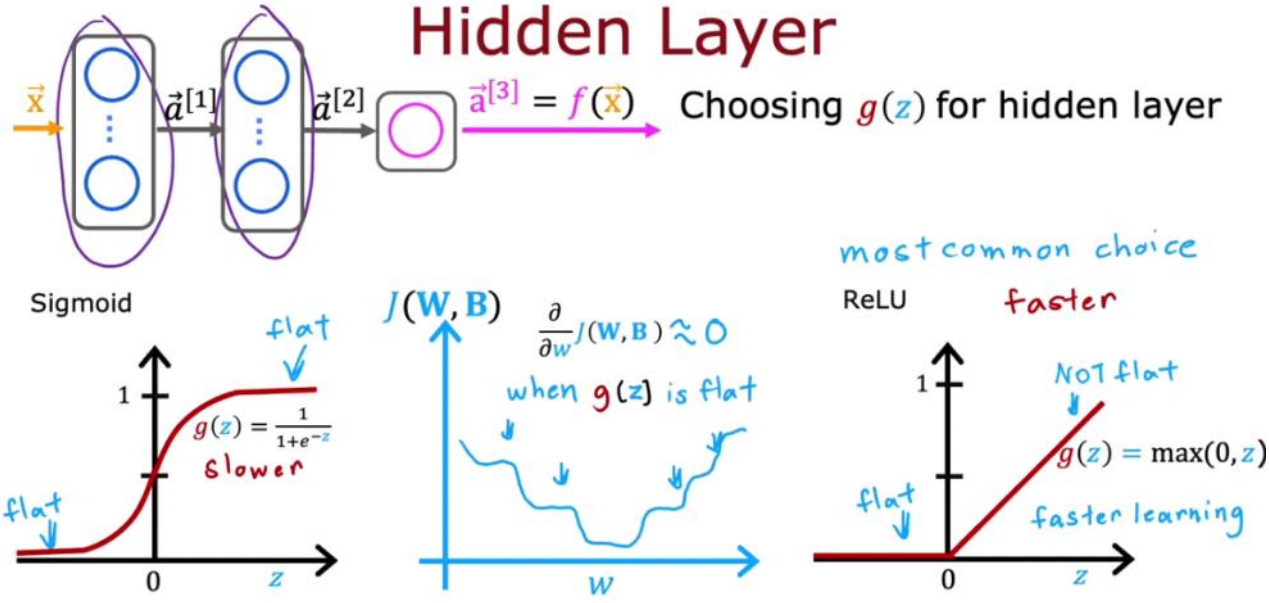
Summary of Output Layer Choices

- Binary Classification: Sigmoid (predict probability)
- Regression: Linear (allow any real number output)
- Non-Negative Outputs: ReLU (only non-negative outputs)

Choosing Activation Functions for Hidden Layers

The ReLU (Rectified Linear Unit) activation function is the most common choice for hidden layers. Here's why ReLU is preferred over sigmoid:

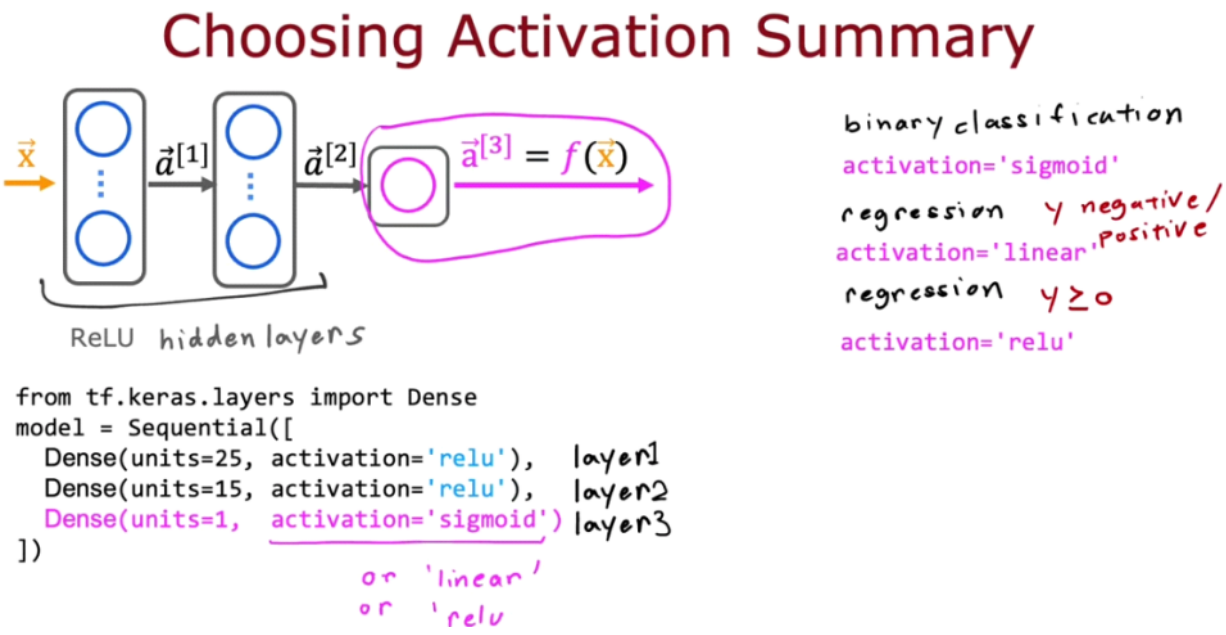
- **Computation Speed:** ReLU is faster to compute (max(0, z)) compared to sigmoid (exponentiation and inversion).
- **Gradient Descent and Learning Speed:** ReLU avoids vanishing gradients, a problem encountered with sigmoid during training with gradient descent. This allows for faster learning.
- ReLU is flat only on one side of the graph whereas sigmoid is flat on two sides of the graph.



Implementing Activation Functions in TensorFlow

TensorFlow provides syntax for specifying activation functions:

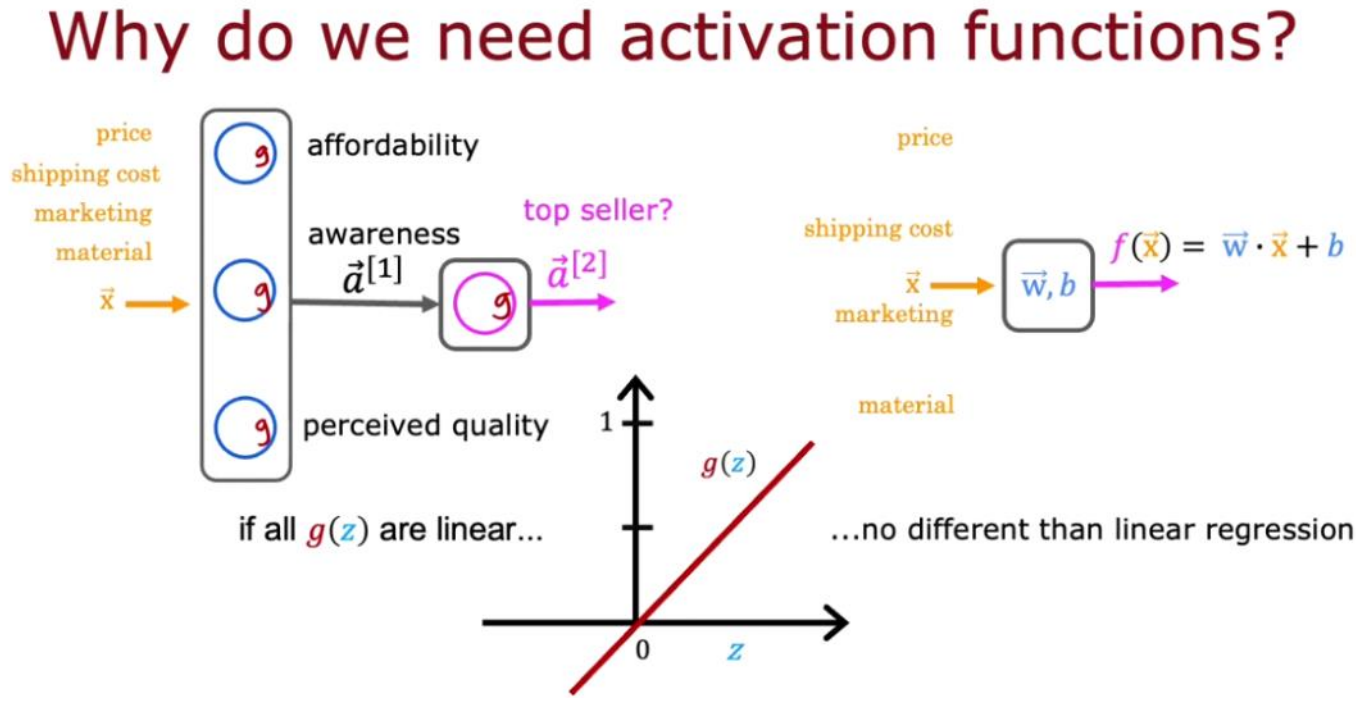
- activation=tf.nn.relu for ReLU
- activation=tf.nn.sigmoid for sigmoid
- activation=tf.nn.linear for linear

- 

Beyond Common Activation Functions

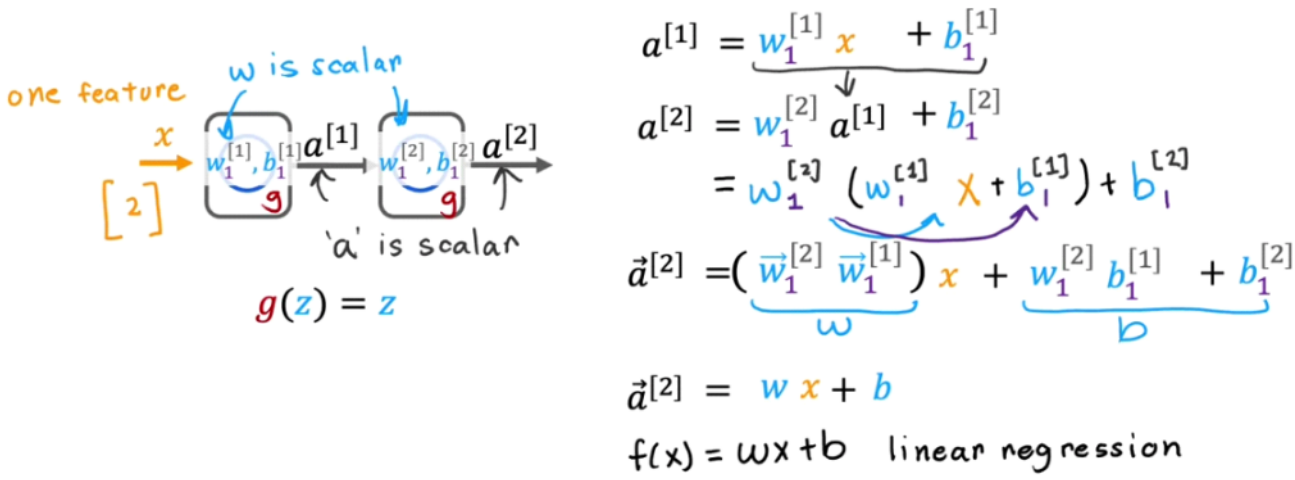
Research explores other activation functions like tanh, LeakyReLU, and swish. While these might show slight improvements, ReLU remains the default choice for most applications.

Linear Activation Function and Reduced Network Capability



Consider the demand prediction example. If we used a linear activation function for all nodes, the complex neural network would become no different from linear regression. It wouldn't be able to learn anything more intricate than the linear regression model.

Linear Example



Example: Simplified Neural Network

Let's analyze a simpler neural network with:

- Input (x): a single number
- One hidden unit with parameters (w1, b1) and output (a1)
- Output layer with one unit, parameters (w2, b2), and output (a2)

Let's see what happens if we use the linear activation function (g(z) = z) everywhere.

Computation with Linear Activation

- a1 = g(w1 * x + b1) = w1 * x + b1 (because g(z) = z)
- a2 = g(w2 * a1 + b2) = w2 * (w1 * x + b1) + b2 (substitute a1)

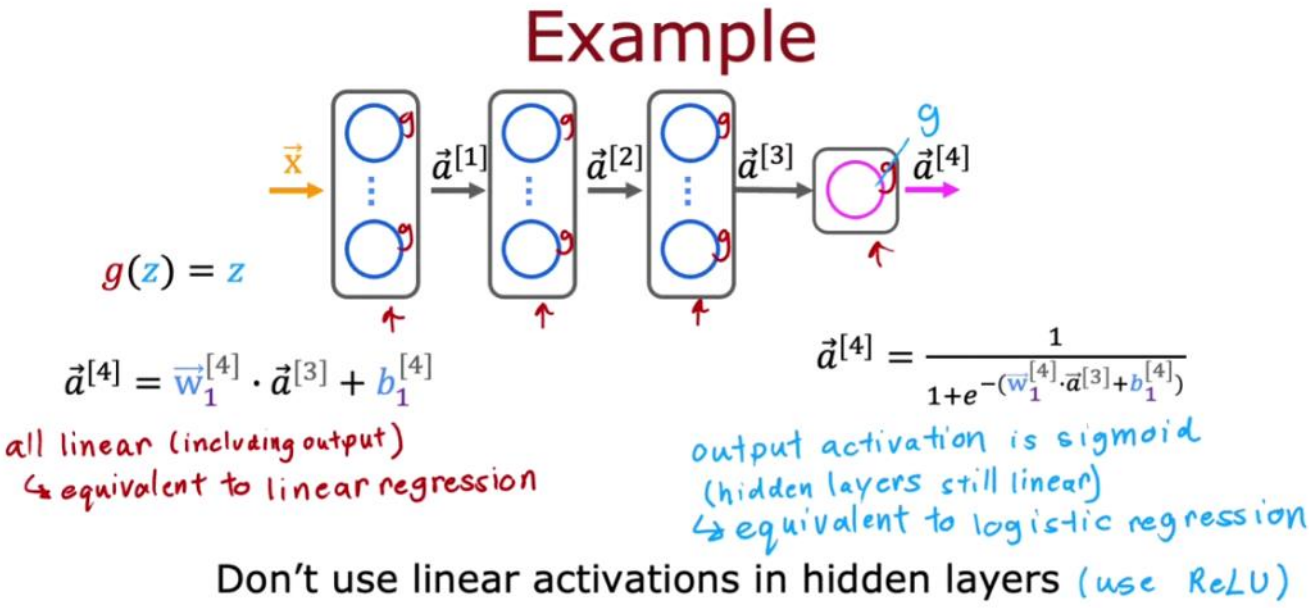
Simplifying the equation for a2:
a2 = w2 * w1 * x + (w2 * b1 + b2)

Equivalence to Linear Regression

If we set w = w2 * w1 and b = w2 * b1 + b2, then a2 becomes:
a2 = w * x + b
This shows that a2 is simply a linear function of the input x. In essence, the neural network with linear activation becomes equivalent to a linear regression model.

Generalization to Multi-Layer Networks

This concept applies to neural networks with multiple layers. If you use linear activation functions throughout, the network's output becomes equivalent to linear regression.



Key Takeaway: Linear Activation Limits Complexity

Neural networks require activation functions other than the linear activation function everywhere. This is because using linear activation throughout the network restricts it from learning complex features or relationships beyond what linear regression can handle.

Common Best Practice: ReLU Activation

As a rule of thumb, avoid using the linear activation function in hidden layers. The ReLU (Rectified Linear Unit) activation function is a generally good choice.

Multiclass Classification

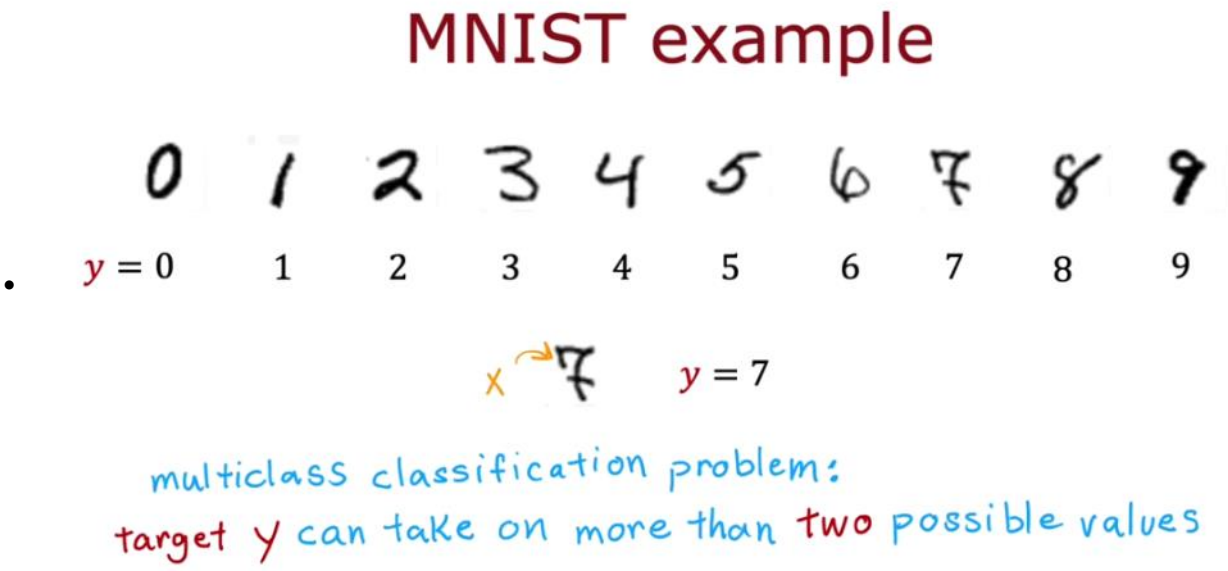
20 May 2024 11:59

Multi-Class Classification: More Than Two Options

- Binary classification: y can only be 0 or 1 (e.g., handwritten digit classification: 0 or 1)
- Multi-class classification: y can take on more than two possible values (e.g., classifying handwritten digits 0-9, disease classification with multiple diseases)

Examples of Multi-Class Classification

- Handwritten Digit Recognition (10 possible digits: 0-9)
- Disease Classification (3 or 5 possible diseases)
- Visual Defect Inspection (scratch, discoloration, chip defect, etc. in pills)



Data Representation in Multi-Class Classification

Unlike binary classification (where y is 0 or 1), y in multi-class classification can have multiple categories (e.g., 4 classes represented by circles, Xs, triangles, and squares).

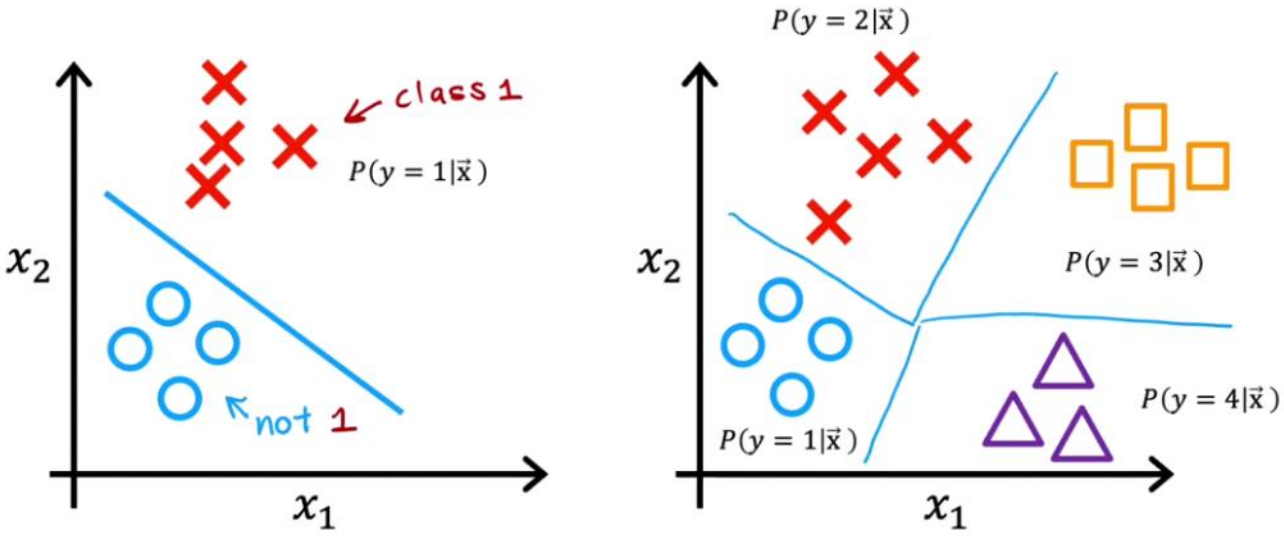
Goal: Estimating Probabilities for Multiple Classes

Instead of estimating the probability of y being 1 (like in logistic regression for binary classification), we now aim to estimate the probability of y belonging to any of the multiple classes (e.g., probability of y being 1, 2, 3, or 4).

Multi-Class Classification with Decision Boundaries

The model learns a decision boundary that divides the feature space into multiple regions, corresponding to the different classes (e.g., a more complex boundary compared to binary classification).

Multiclass classification example



Softmax Function

20 May 2024 12:22

Logistic Regression Recap

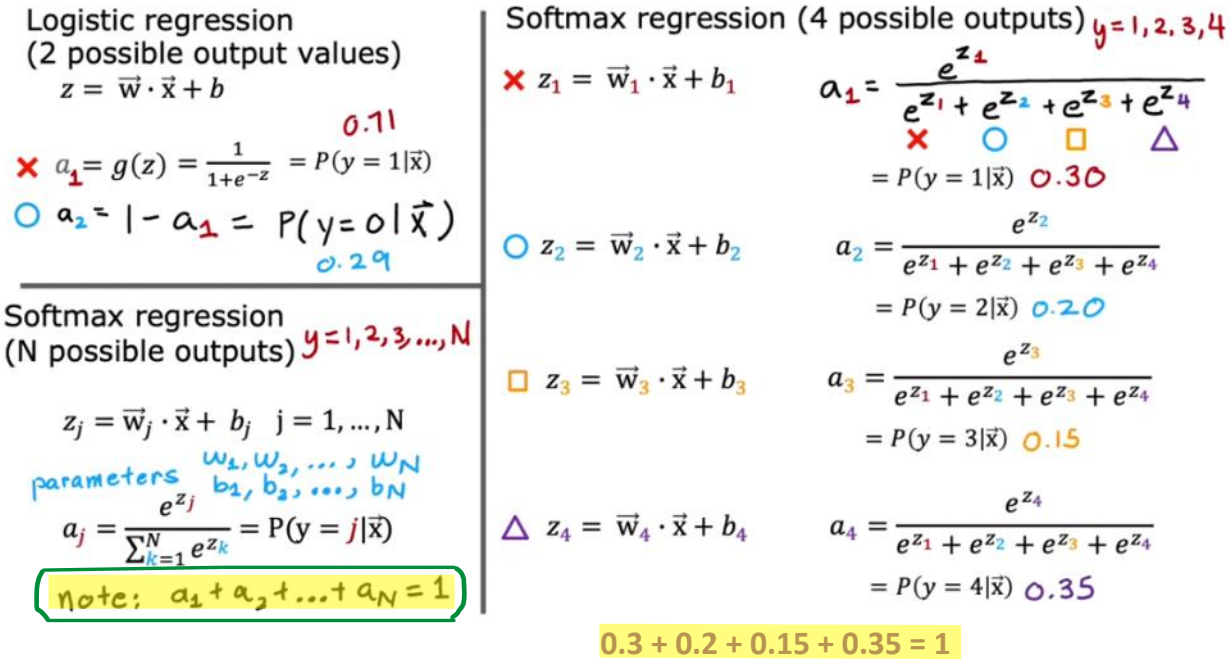
- Applicable when y can be 0 or 1 (e.g., handwritten digit classification)
- Calculates z = w * x + b (linear combination of weights and features)
- Applies the sigmoid function (g(z)) to z to get a value between 0 and 1, interpreted as the probability of y being 1 (a)
- We know that a (probability of y being 1) + (1 - a) (probability of y being 0) = 1

Extending Logistic Regression to Softmax Regression

- **Scenario:** Let's consider a case where y can take on four possible values (1, 2, 3, or 4).
- **Softmax Regression Steps:**
  1. Compute z_i (linear combinations of weights and features) for each class (i = 1 to 4): z_i = w_i * x + b_i
  2. Apply the softmax function to each z_i to get probabilities (a_i) for each class: a_i = e^z_i / (e^z_1 + e^z_2 + e^z_3 + e^z_4)
  3. Interpretation: a_i represents the model's estimated probability of y being class i given the input features (x).

Key Points about Softmax Regression

- The probabilities (a_i) always sum up to 1 (a_1 + a_2 + ... + a_n = 1).
- Softmax regression reduces to logistic regression when there are only two output classes (n = 2).

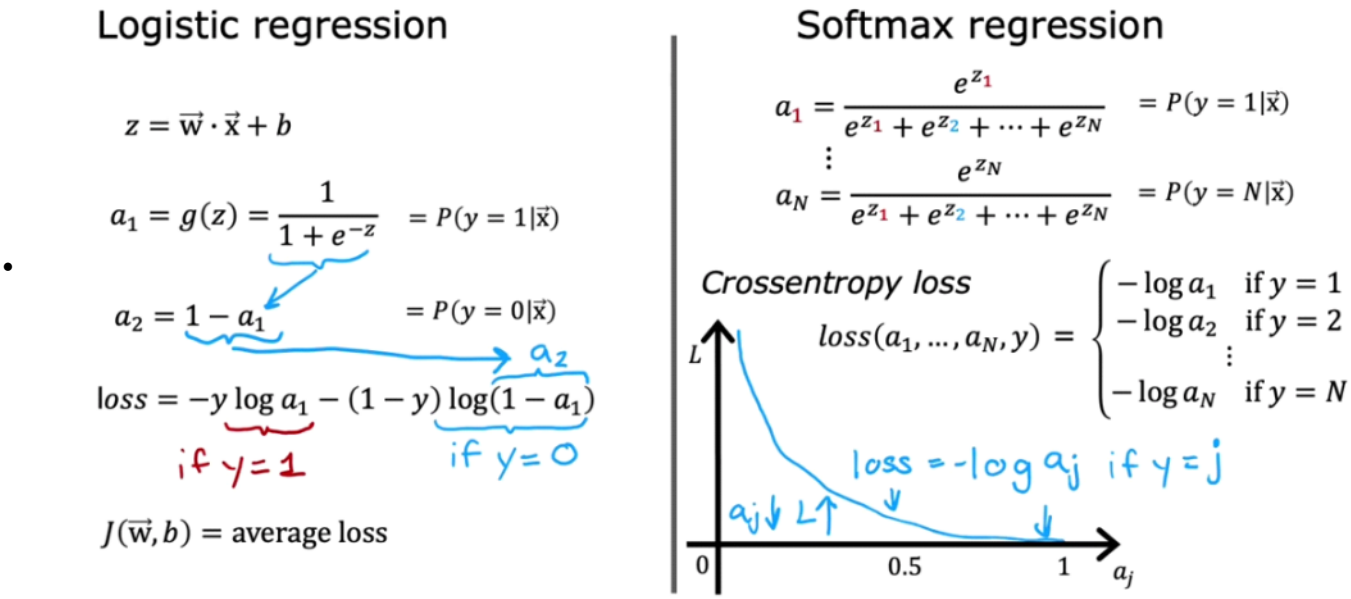


Cost Function for Softmax Regression

- Similar to logistic regression, we use a negative log-likelihood loss function.
- The loss depends on the ground truth label (y) and the predicted probabilities (a_i).
- If the actual class is j (y = j), the loss is negative log(a_j). This encourages the model to assign a higher probability to the correct class.
- The overall cost function is the average loss over all training examples.

Visualization of Loss Function

Cost



- Lower a_j values lead to higher losses.
- This encourages the model to make a_j (probability of class j) as close to 1 as possible (correct classification).

Uniqueness of Loss Function Calculation

- During training, for each training example, we only calculate the negative log term for the actual class value (y).
- For example, if y = 2, we only compute negative log(a_2), not the negative log terms for other classes (a_1, a_3, etc.).

Neural Network with Softmax Output

20 May 2024 23:45

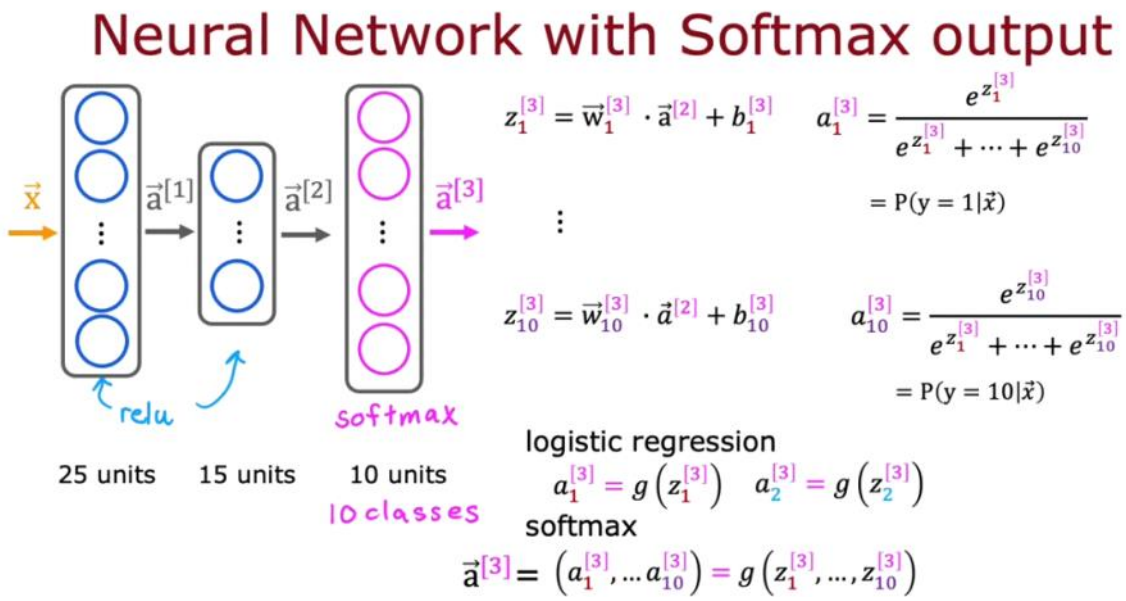
From Single-Class to Multi-Class Neural Networks

- Previously used a neural network with one output unit for binary classification (e.g., handwritten digit recognition: 0 or 1).
- For multi-class classification (e.g., recognizing 10 digits: 0-9), the neural network needs to have multiple output units.

Introducing the Softmax Output Layer

- The new output layer in the multi-class neural network will have a softmax activation function.
- This layer estimates the probability of each possible class for a given input.

Forward Propagation in a Multi-Class Neural Network

1. Activation for the first hidden layer (A1) is computed as usual.
2. Activation for the second hidden layer (A2) is also computed as usual.
3. Activations for the output layer (A3) are computed using softmax:
    - Z_i (linear combinations of weights and features) are calculated for each class (i = 1 to total number of classes).
    - The softmax function is applied to each Z_i to get probabilities (a_i) for each class: $a_i = e^{zi} / (e^{z1} + e^{z2} + ... + e^{zn})$
4. 

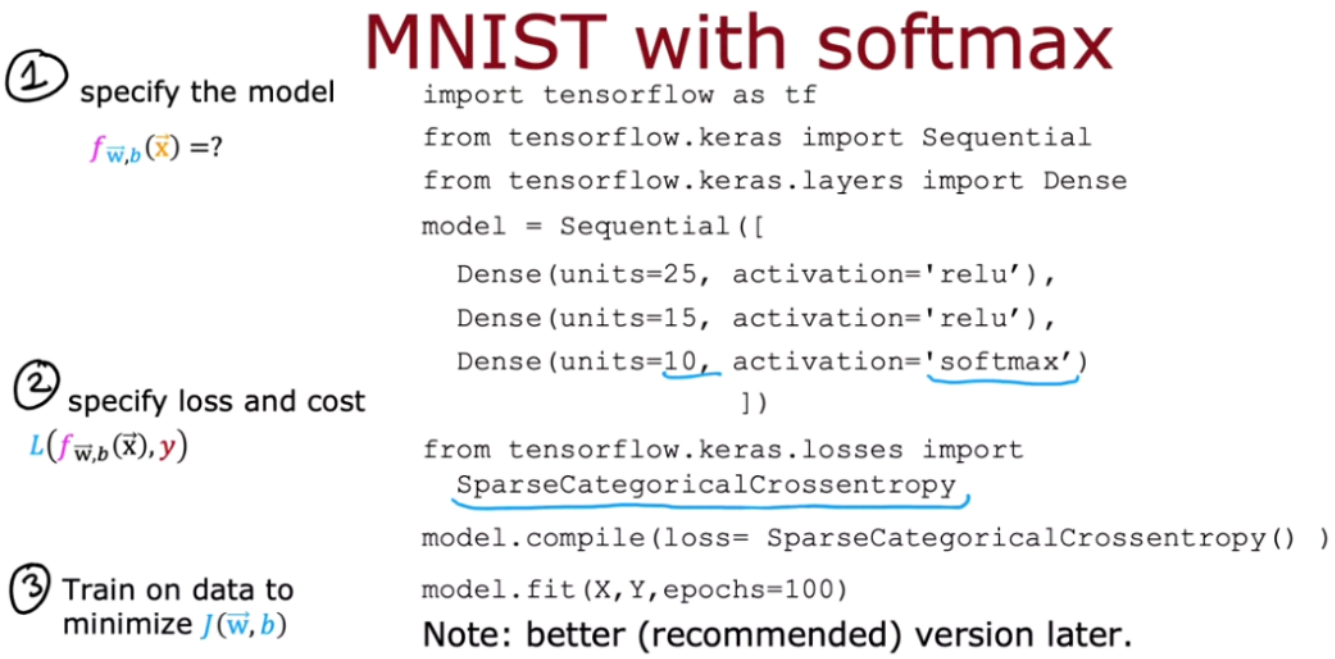
Properties of Softmax Activation

- The output probabilities (a_i) always sum up to 1 ($a_1 + a_2 + ... + a_n = 1$).
- Softmax reduces to logistic regression when there are only two output classes (n = 2).

Uniqueness of Softmax Activation

- Unlike other activation functions (sigmoid, ReLU, etc.), where each activation depends only on its corresponding linear combination (Z_i), the softmax activation in the output layer involves all Z values.
- Each output probability (a_i) depends on all Z values (Z_1 to Z_n).

Implementing a Multi-Class Neural Network with TensorFlow



1. Define the network architecture using TensorFlow:
    - First layer: 25 units with ReLU activation.
    - Second layer: 15 units with ReLU activation.
    - Third (output) layer: 10 units (number of classes) with softmax activation.
2. Cost function: SparseCategoricalCrossentropy (suitable for multi-class classification).
3. Training process: similar to the single-output case.

Improved Implementation of Softmax Function

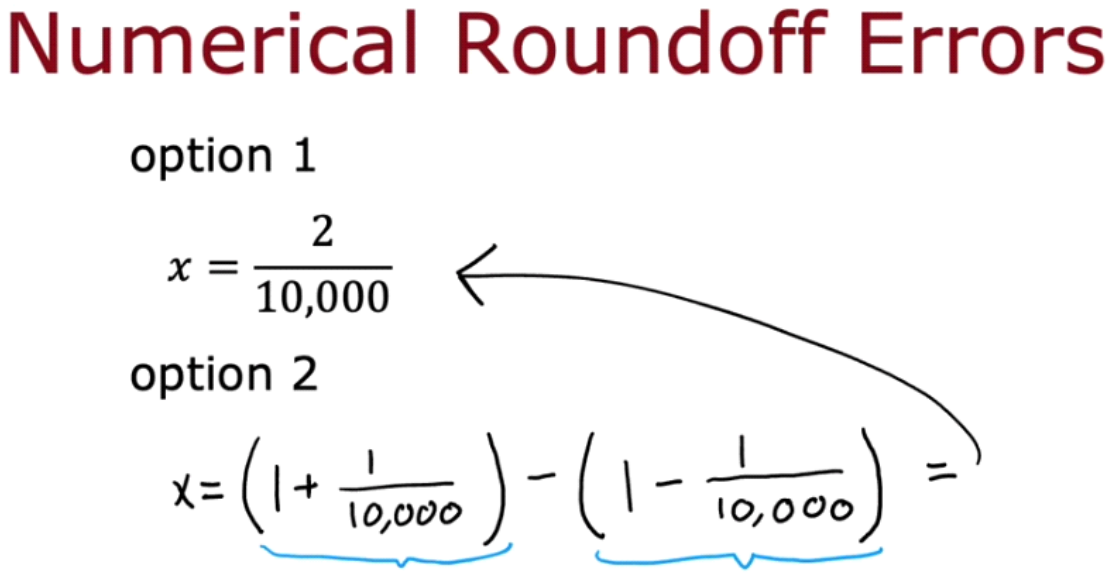
21 May 2024 08:10

Recap: Potential Issue with Previous Implementation

- The previous implementation for computing the cost function in softmax can suffer from numerical round-off errors.
- This can happen when calculations involve very small or very large numbers (e.g., e^z for large/small z).
- These errors can lead to less accurate computations within TensorFlow.

Analogy: Round-Off Errors

- Imagine calculating x = 2/10,000.



- Option 1: Directly set x = 2/10,000 (potentially more accurate).
- Option 2: Set x = 1 + 1/10,000 - 1 - 1/10,000 (may introduce round-off errors during intermediate calculations).

```
In [1]: x1 = 2.0 / 10000
        print(f"{x1:.18f}") # print 18 digits to the right of decimal point

        0.000200000000000000
```

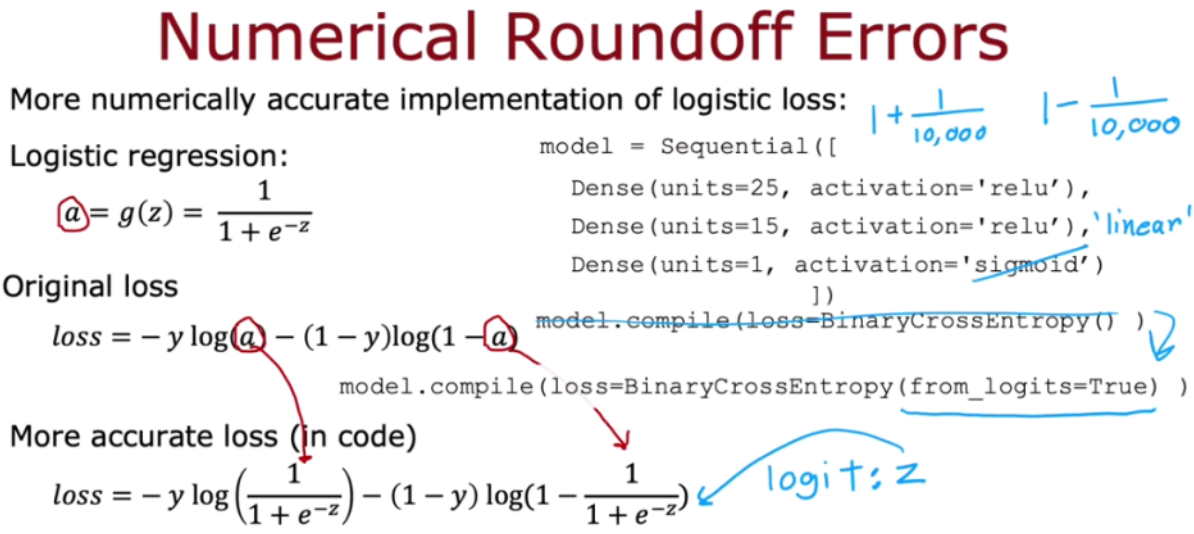
```
In [2]: x2 = 1 + (1/10000) - (1 - 1/10000)
        print(f"{x2: .18f}")

         0.000199999999999978
```

Because the computer has only a finite amount of memory to store each number, called a floating-point number in this case, depending on how you decide to compute the value 2/10,000, the result can have more or less numerical round-off error.
It turns out that while the way we have been computing the cost function for softmax is correct, there's a different way of formulating it that reduces these numerical round-off errors, leading to more accurate computations within TensorFlow.

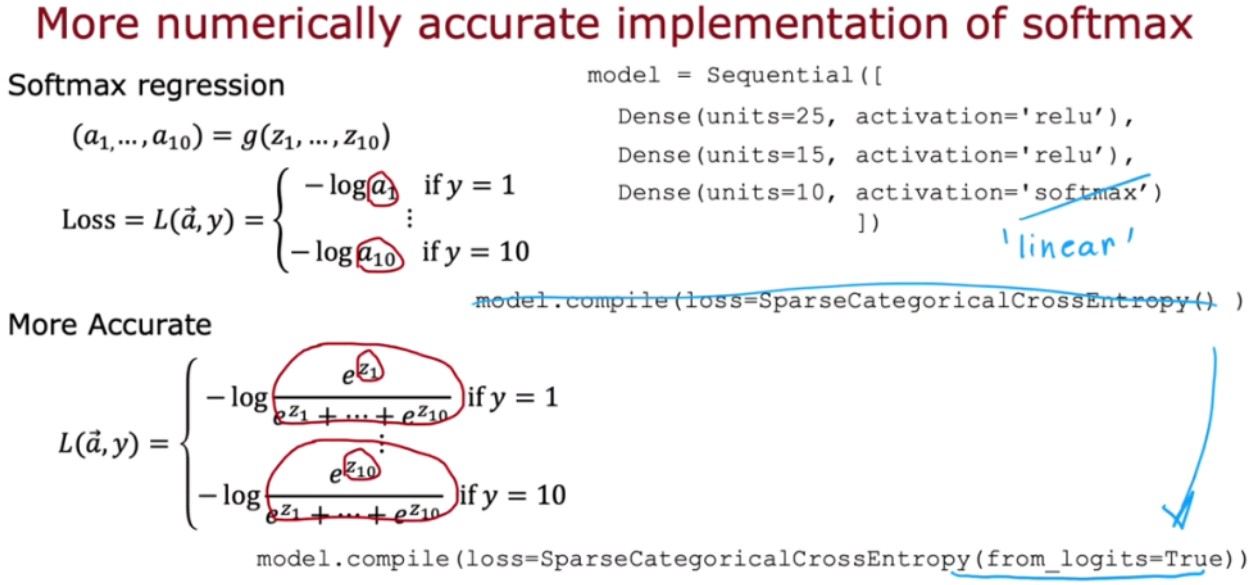
Solution: Leverage TensorFlow's Flexibility

1. **Logistic Regression Example:**
   - Standard cost function computation for logistic regression can also be susceptible to round-off errors.
   - We can provide TensorFlow with more flexibility in calculating the loss function.
   - Instead of explicitly computing the activation (a) as an intermediate step, we can express the loss function directly using the formula involving z (linear combination of weights and features).
   - This allows TensorFlow to rearrange terms and potentially avoid round-off errors.



2. **Softmax Regression Implementation:**
   - The previous implementation calculated activations (a_i) and then the loss function.
   - The improved approach combines the activation and loss function calculations into a single expression within the loss function definition.
   - This gives TensorFlow more control over how to compute the loss function numerically.

Code for Improved Softmax Implementation



- The recommended code uses a linear activation function in the output layer.
- The actual computation of activations and loss is hidden within the loss function definition using from_logits=True.
- This approach is more numerically stable but less readable.

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 $a_1 \ldots a_{10}$ $z_1 \ldots z_{10}$

- The model doesn't predict a1, a2, etc. Instead it predicts z1, z2, etc. now

Key Points

- Both the original and improved implementations achieve similar results conceptually.
- The improved version offers better numerical stability, especially for softmax regression.
- It's recommended to use the improved version (with from_logits=True) for better accuracy.

Classification with multiple outputs

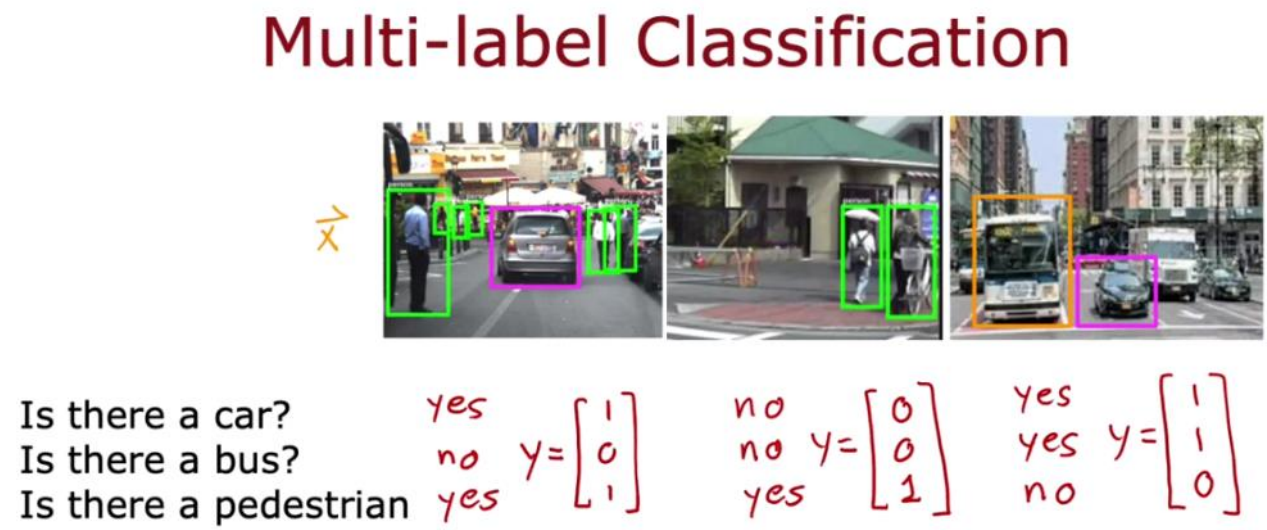
21 May 2024 16:26

Multi-Class Classification

- You learned about multi-class classification in previous lectures.
- In multi-class classification, the output label (Y) can belong to one of two or more possible categories.
    - Example: Handwritten digit recognition (Y can be 0, 1, 2, ..., 9)

Multi-Label Classification

- A different type of classification problem is multi-label classification.
- In multi-label classification, an input (e.g., an image) can be associated with multiple labels simultaneously.



    - Example: Self-driving car perception system classifying objects in an image (car, bus, pedestrian)

Illustrating Multi-Label Classification

- Consider a self-driving car needing to identify objects in front of it:
    - Image 1: Car (Yes), Bus (No), Pedestrian (Yes)
    - Image 2: Car (No), Bus (No), Pedestrian (Yes)
    - Image 3: Car (Yes), Bus (Yes), Pedestrian (No)
- In these examples, each image is assigned multiple labels.

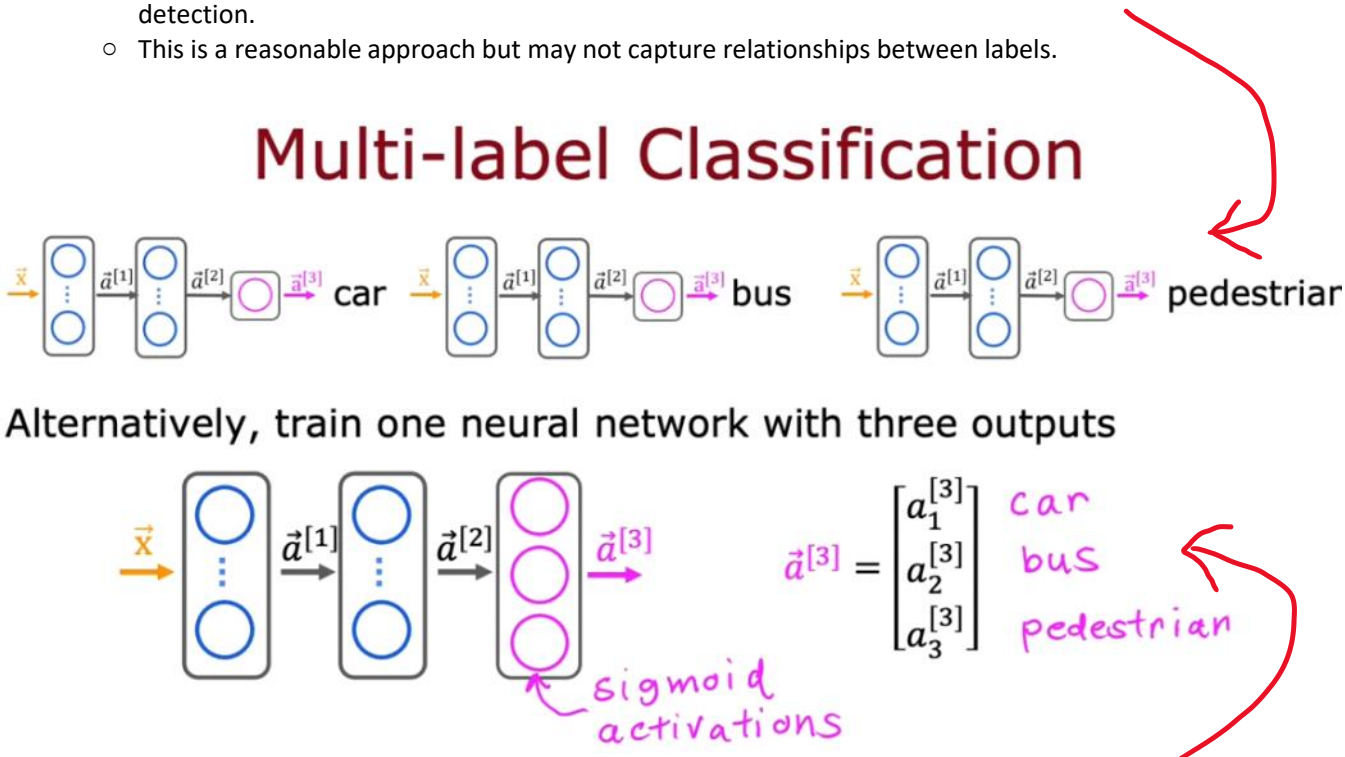
Key Difference from Multi-Class

- The key distinction is in the target variable (Y).
    - Multi-class: Y is a single number (even if it can take on multiple values).
    - Multi-label: Y is a vector of numbers (one for each label), indicating the presence or absence of that label for the input.

Approaches to Building a Multi-Label Neural Network

1. **Independent Neural Networks:**
    - Train three separate neural networks: one for car detection, one for bus detection, and one for pedestrian detection.
    - This is a reasonable approach but may not capture relationships between labels.

detection.
- This is a reasonable approach but may not capture relationships between labels.



2. **Single Neural Network:**
   - Train a single neural network with three output neurons.
   - Each output neuron uses a sigmoid activation function and predicts the probability of its corresponding label (car, bus, pedestrian).

Multi-Class vs. Multi-Label: Avoiding Confusion

- These two classification tasks are sometimes confused.
- Understanding the difference is crucial for choosing the right approach for your application.

Advanced Optimization

23 May 2024 12:27

Adam: An Adaptive Learning Rate Optimization Algorithm for Neural Networks

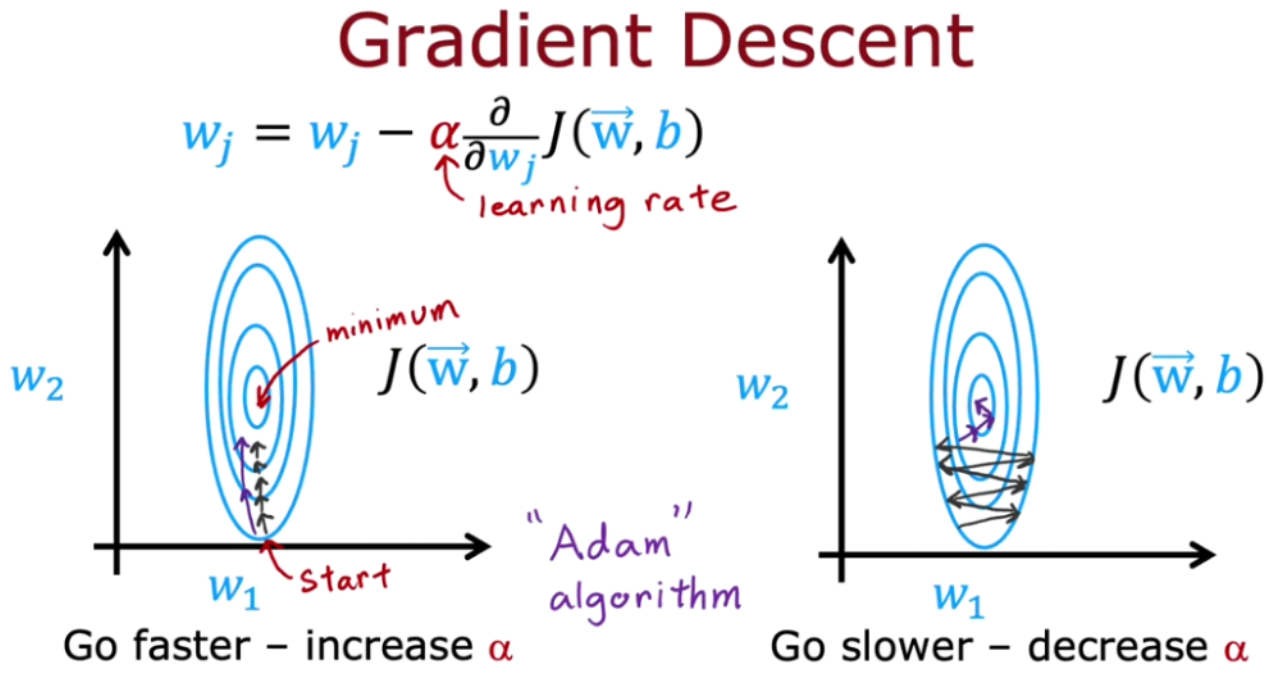
Gradient Descent Review

- Gradient descent is a widely used optimization algorithm for minimizing cost functions in machine learning.
- It updates each parameter based on its learning rate (alpha) and the partial derivative of the cost function with respect to that parameter.

Limitations of Gradient Descent

Choosing a fixed learning rate can be challenging:

- A small learning rate leads to slow convergence (many steps to reach the minimum).
- A large learning rate can cause oscillations or even divergence (overshooting the minimum).



The Adam Algorithm

- Adam (Adaptive Moment Estimation) is an optimization algorithm that addresses these limitations.
- It dynamically adjusts the learning rate for each parameter during training.
- This adaptation is based on historical gradients to improve convergence speed and stability.

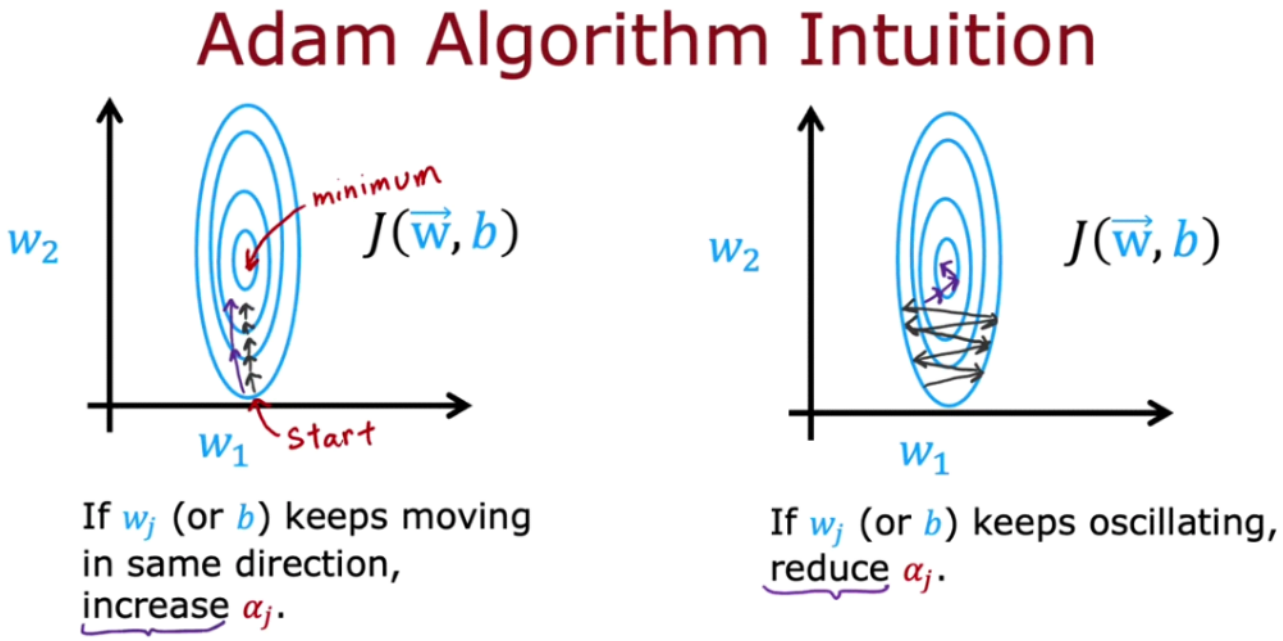
Adam Algorithm Intuition

Adam: Adaptive Moment estimation not just one α

$$w_1 = w_1 - \alpha_1 \frac{\partial}{\partial w_1} J(\vec{w}, b)$$
$$\vdots$$
$$w_{10} = w_{10} - \alpha_{10} \frac{\partial}{\partial w_{10}} J(\vec{w}, b)$$
$$b = b - \alpha_{11} \frac{\partial}{\partial b} J(\vec{w}, b)$$

Visualizing Adam's Behavior

- Imagine a cost function visualized as an elliptical contour plot.
- Gradient descent often takes small steps in a similar direction, potentially inefficient when the minimum lies far away.
- Adam can automatically increase the learning rate in such cases for faster progress.
- Conversely, Adam can decrease the learning rate if the updates oscillate back and forth, preventing divergence.



Key Points about Adam

- **It uses separate learning rates for each parameter (w_1, w_2, ..., b) instead of a single global learning rate.**
- It adjusts learning rates based on estimates of historical gradients (momentum and adaptive learning rate).
- The specific details of these adjustments are beyond the scope of this course but can be explored in advanced deep learning courses.

Implementing Adam in TensorFlow

1. The model architecture remains the same.
2. During compilation, use the tf.keras.optimizers.Adam optimizer instead of the default one.
3. You can optionally specify an initial learning rate (alpha) as an argument to the Adam optimizer.

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

$\alpha = 10^{-3} = 0.001$

```
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)
```

Benefits of Adam

- Compared to gradient descent, Adam often converges faster due to its adaptive learning rates.
- It is less sensitive to the exact choice of the initial learning rate, making it more robust.

Adam as a Default Choice

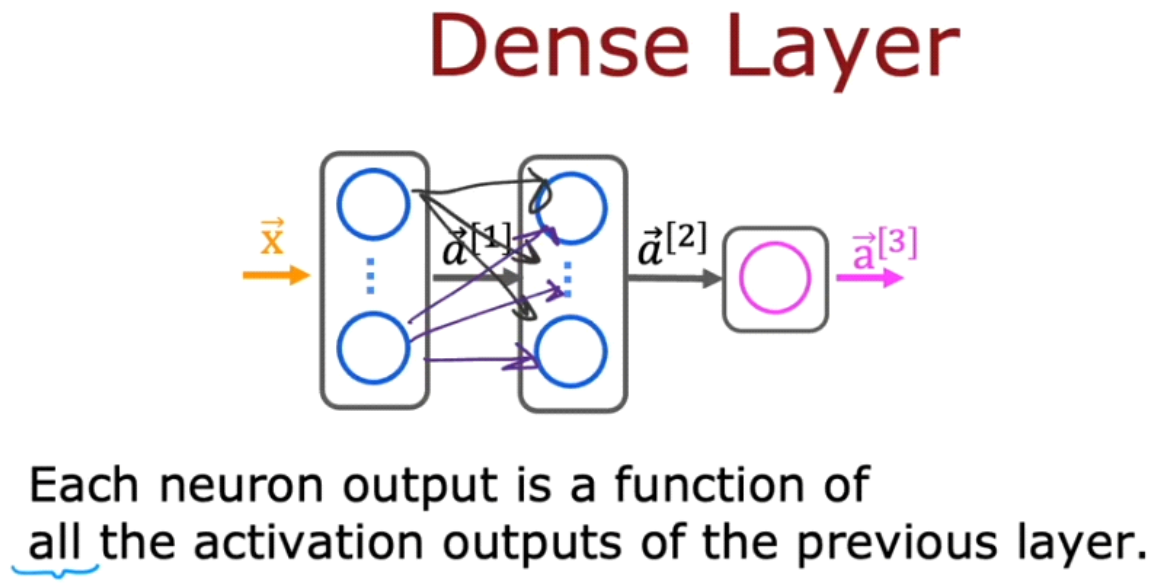
- Adam is a popular and effective optimization algorithm for training neural networks.
- It is often the default choice for practitioners due to its speed and robustness.

Additional Layer Types

23 May 2024 17:48

Dense Layers Revisited

- All neural network layers covered so far have been dense layers.
- In a dense layer, each neuron receives input from all activations in the previous layer.
- Dense layers are powerful for many applications.



$$\vec{a}_1^{[2]} = g\left(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}\right)$$

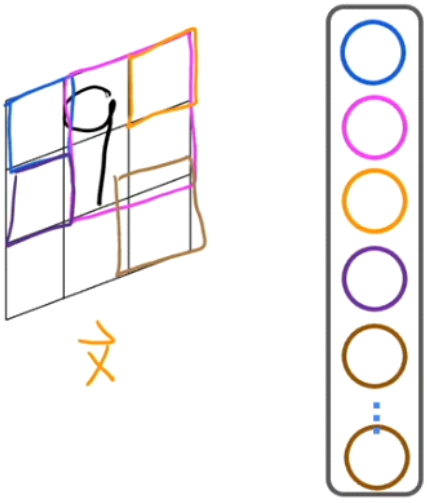
Alternative Layer Types

- Other layer types exist with specific properties suited for different tasks.
- This lecture focuses on convolutional layers, commonly used in convolutional neural networks (CNNs).

Convolutional Layers Explained

- Imagine an input image (e.g., handwritten digit).
- In a dense layer, each neuron in a hidden layer would consider all pixels of the image.
- A convolutional layer works differently:
  - Each neuron in the hidden layer only processes a limited rectangular region of the input image.
  - Different neurons can look at different regions.

Convolutional Layer



Benefits of Convolutional Layers

- Speed up computation by limiting the number of connections each neuron needs.
- Reduce the amount of training data required compared to dense layers.
- Less prone to overfitting (a common problem in machine learning).

Convolutional Neural Networks (CNNs)

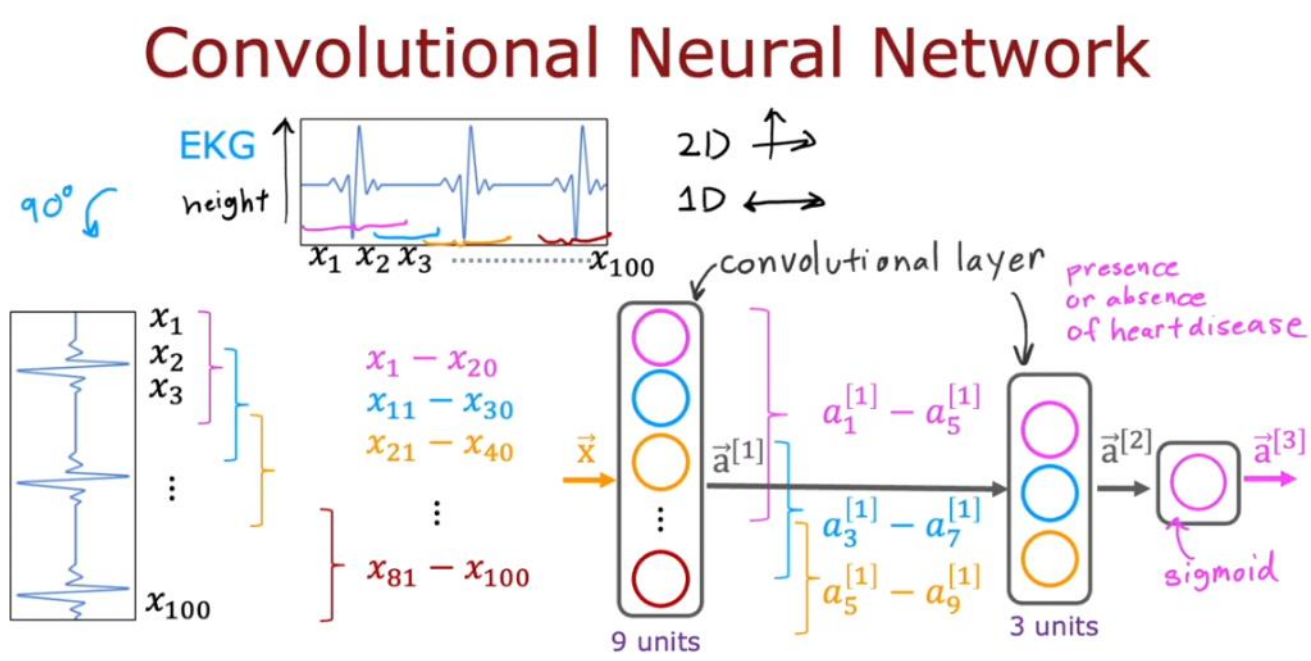
A neural network with multiple convolutional layers is called a CNN.

Example: Classifying EKG Signals

- One-dimensional data (EKG signals) can also be processed with convolutional layers.
- An EKG signal is a time series representing the electrical activity of the heart.
- The goal: classify EKG signals to identify potential heart issues.

Convolutional Layer for EKG Signals

- The EKG signal is represented as a list of values.
- The convolutional layer's neurons only examine "windows" of this list.
- The first neuron looks at the first 20 values, the second looks at the next 20 values, and so on.



Benefits in EKG Classification

This approach allows the network to focus on specific patterns within the EKG signal.

Convolutional Layer Architecture Choices

- The size of the window a neuron examines (filter size) is an architectural choice.
- The number of neurons in the layer is another design decision.
- Effective choices of these parameters can lead to superior performance compared to dense layers for certain tasks.

Neural Network Backpropagation Calculus

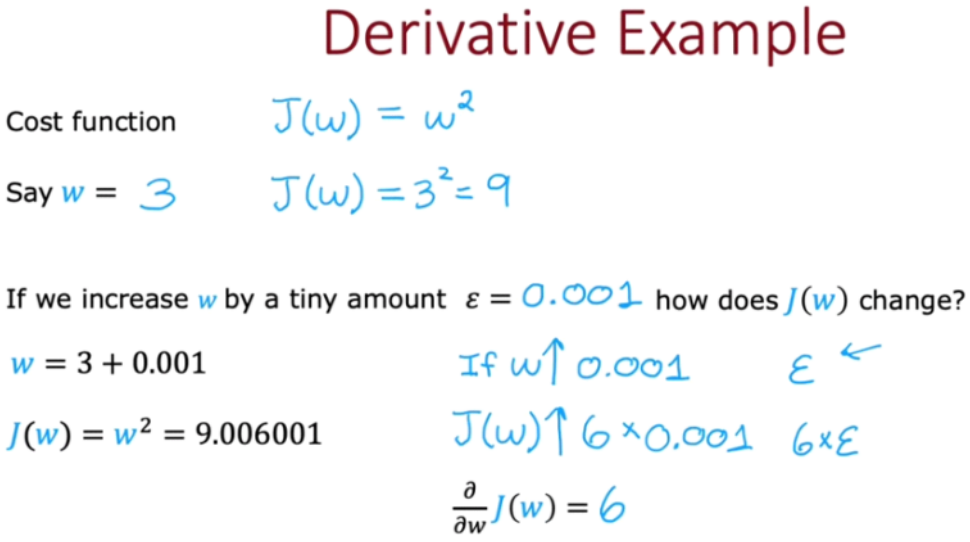
23 May 2024 22:01

Key Points:

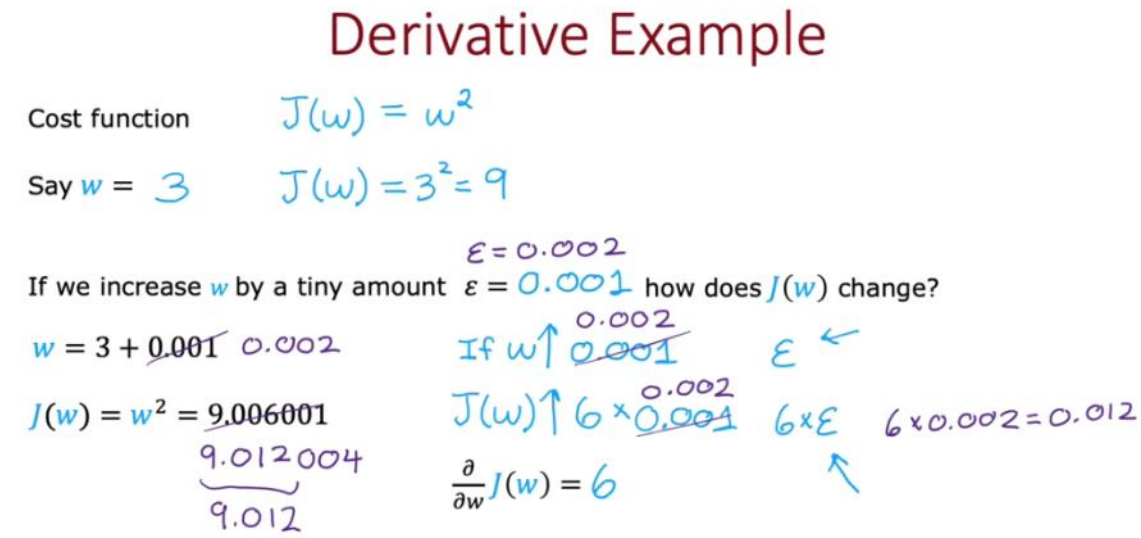
- In TensorFlow, you define a neural network architecture to compute the output y as a function of the input x.
- You also specify a cost function to measure the performance of the network.
- Backpropagation is a crucial algorithm that computes the derivatives of the cost function with respect to the network's parameters (weights and biases).

Understanding Derivatives with a Simplified Example:

- Consider a cost function J(w) = w^2, where w is a parameter.
- We want to know how much J(w) changes if w increases by a tiny amount (Epsilon).
- If w = 3 and Epsilon = 0.001:
    - J(w) increases from 9 to 9.006001 (roughly 6 times Epsilon).



- This suggests a relationship between the change in w and the change in J(w).
- In calculus, the derivative of J(w) with respect to w captures this relationship.
- In this example, the derivative is 6, indicating that a small increase in w leads to a six-fold increase in J(w).



Informal Definition of Derivative

If $w \uparrow \varepsilon$ causes $J(w) \uparrow k \times \varepsilon$ then

$$\frac{\partial}{\partial w} J(w) = k$$

Gradient descent

repeat {

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

}

learning rate

If derivative is small, then this update step will make a small update to w_j

If the derivative is large, then this update step will make a large update to w_j

Derivatives for Different Values of w:

- The derivative of J(w) depends on the current value of w.
  - When w = 2, the derivative is 4.
  - When w = -3, the derivative is -6 (negative because J(w) decreases).

More Derivative Examples

$w = 3$ $J(w) = w^2 = 9$ $w \uparrow 0.001$ $J(w) = J(3.001) = 9.006001$ $\frac{\partial}{\partial w} J(w) = 6$

$J(w) \uparrow 6 \times 0.001$

$w = 2$ $J(w) = w^2 = 4$ $w \uparrow 0.001$ $J(w) = J(2.001) = 4.004001$ $\frac{\partial}{\partial w} J(w) = 4$

$J(w) \uparrow 4 \times 0.001$

$J(w) = w^2$

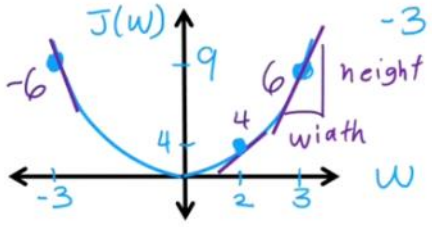
$\downarrow 0.006$

$w = -3$ $J(w) = w^2 = 9$ $w \uparrow 0.001$ $J(w) = J(-2.999) = 8.994001$ $\frac{\partial}{\partial w} J(w) = -6$

$J(w) \downarrow 6 \times 0.001$

$J(w) \uparrow -6 \times 0.001$

-3 + 0.001



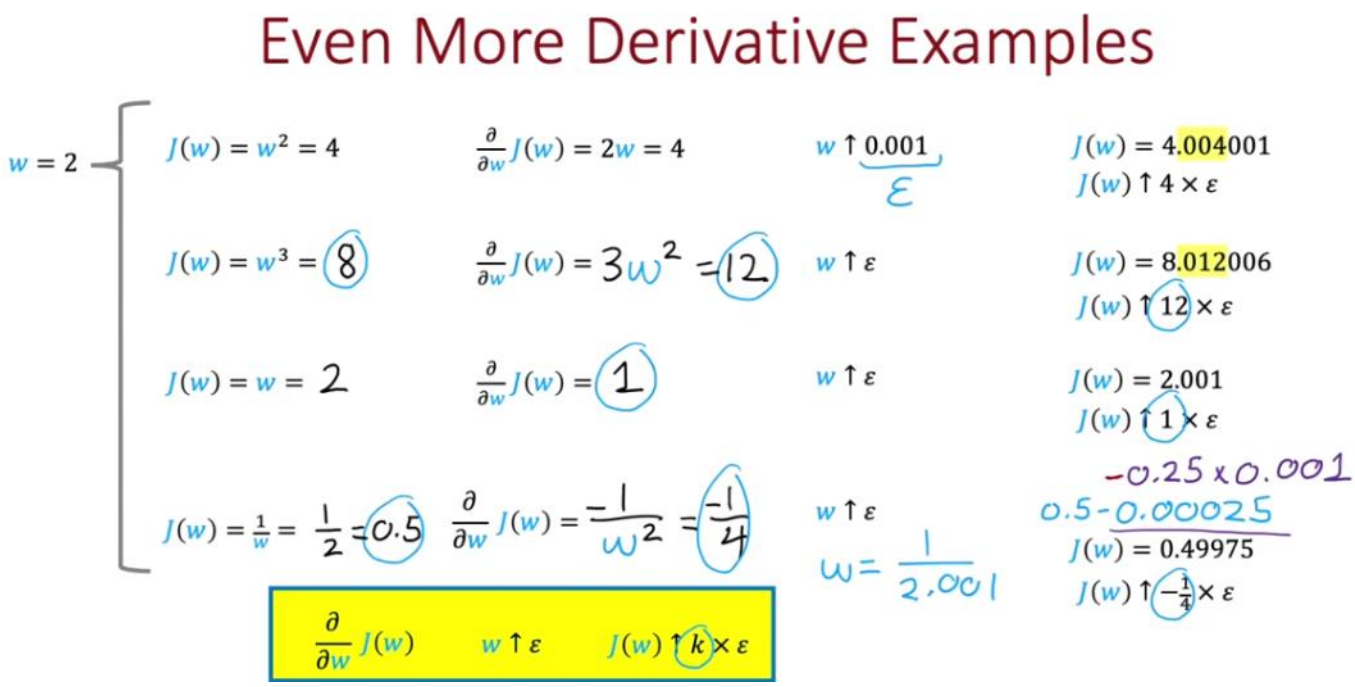
Calculus	w	$\partial J(w)/\partial w$
$\frac{\partial}{\partial w} J(w) = 2w$	3	2 x 3 = 6
	2	2 x 2 = 4
	-3	2 x -3 = -6

Visualizing Derivatives as Slopes:

- Plotting J(w) as a function of w shows how the function changes.
- The derivative at a specific point w corresponds to the slope of the line tangent to the function at that point.
  - A steeper slope (higher derivative) indicates a larger change in J(w) for a small change in w.

Derivatives of Other Functions:

The jupyter notebook for the same explores derivatives of other functions like w^3, w, and 1/w using the SymPy library.



Key Takeaway:

The derivative of a function J(w) with respect to w tells you how much J(w) changes when w increases by a small amount.

Notation:

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

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

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

$\frac{\partial}{\partial w_i}J(w_1, w_2, \ldots, w_n)$

"partial derivative"

Computation Graph and Backpropagation

24 May 2024 16:12

Key Idea:

- Computation graphs are a fundamental concept in deep learning.
- Frameworks like TensorFlow use them to automatically compute derivatives of neural networks.

Understanding with a Small Neural Network:

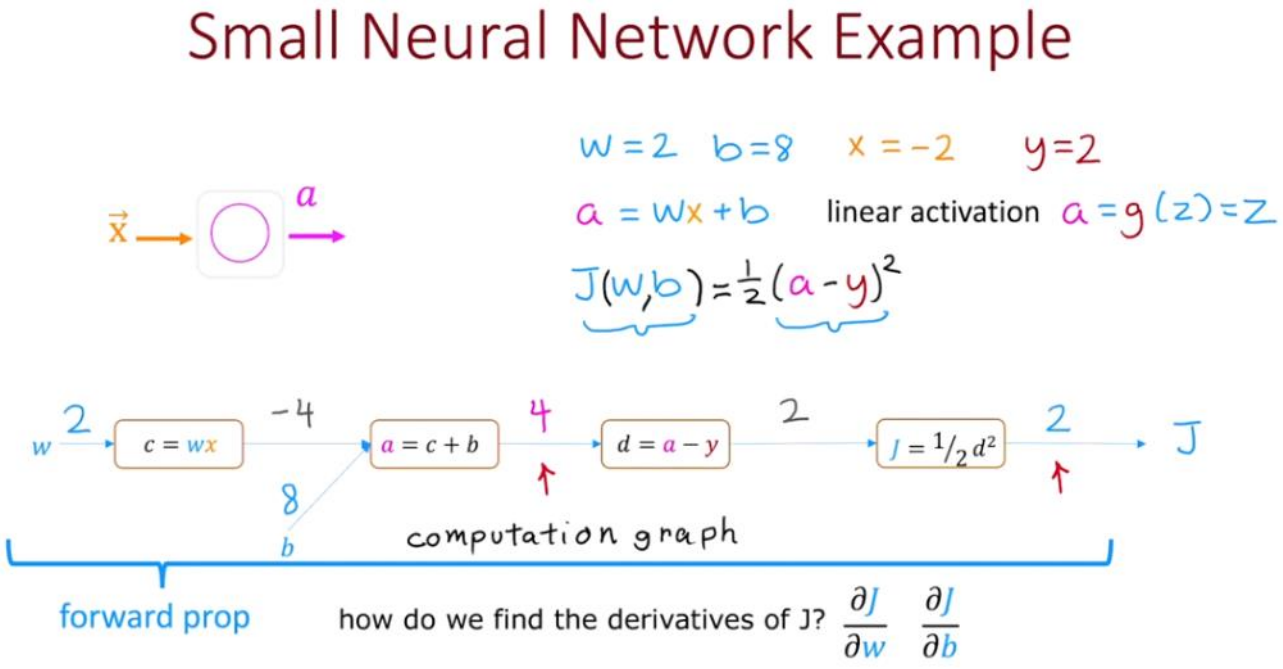
- This example has one layer (output layer) with one unit.
- It takes input x, applies a linear activation function, and outputs a.
    - Mathematically: a = wx + b (linear regression expressed as a neural network).
- The cost function is J = 1/2(a - y)^2, where y is the ground truth value.
- We have a single training example:
    - x = -2
    - y = 2
    - w = 2
    - b = 8

Forward Propagation (Computing the Output and Cost):

1. c = wx: Multiply w and x (here, c = -4).
2. a = c + b: Add c and b to get the output a (here, a = 4).
3. d = a - y: Subtract y from a to get the difference d (here, d = 2).
4. J = 1/2 * d^2: Calculate the cost function J (here, J = 2).

Computation Graph:

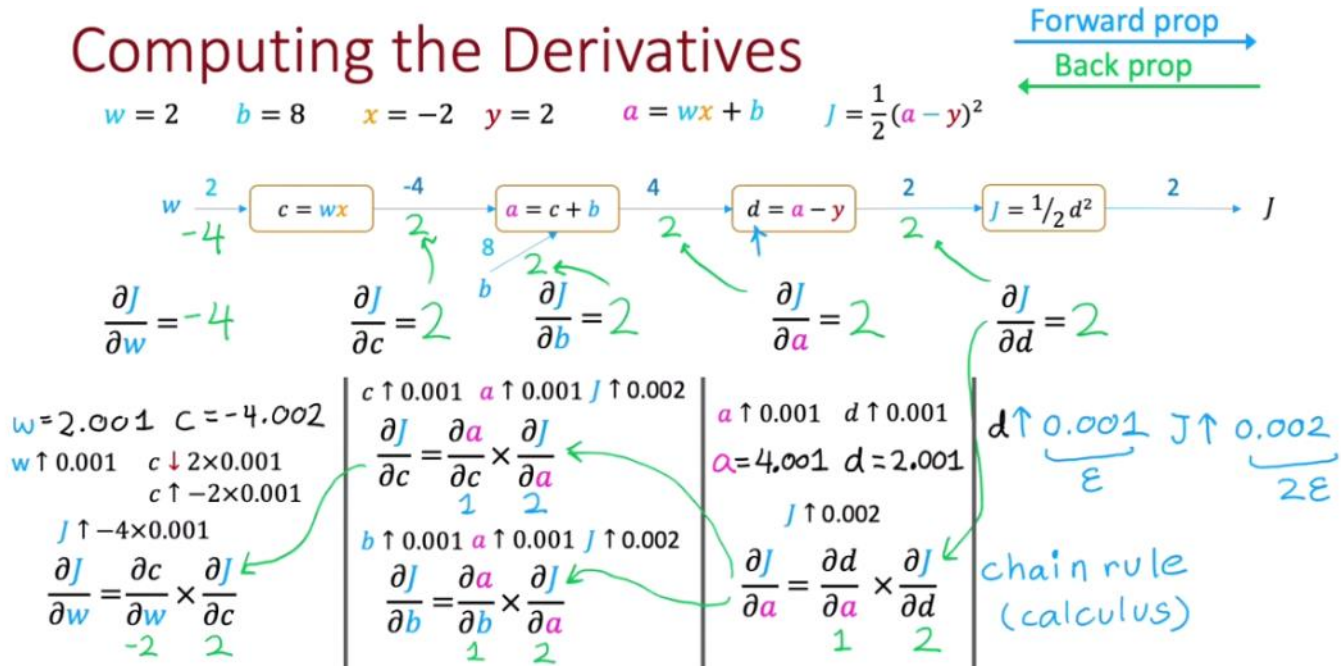
- This graph visually represents the steps involved in forward propagation.
- Arrows show the flow of data between nodes.
- Values on the arrows indicate the results at each step.

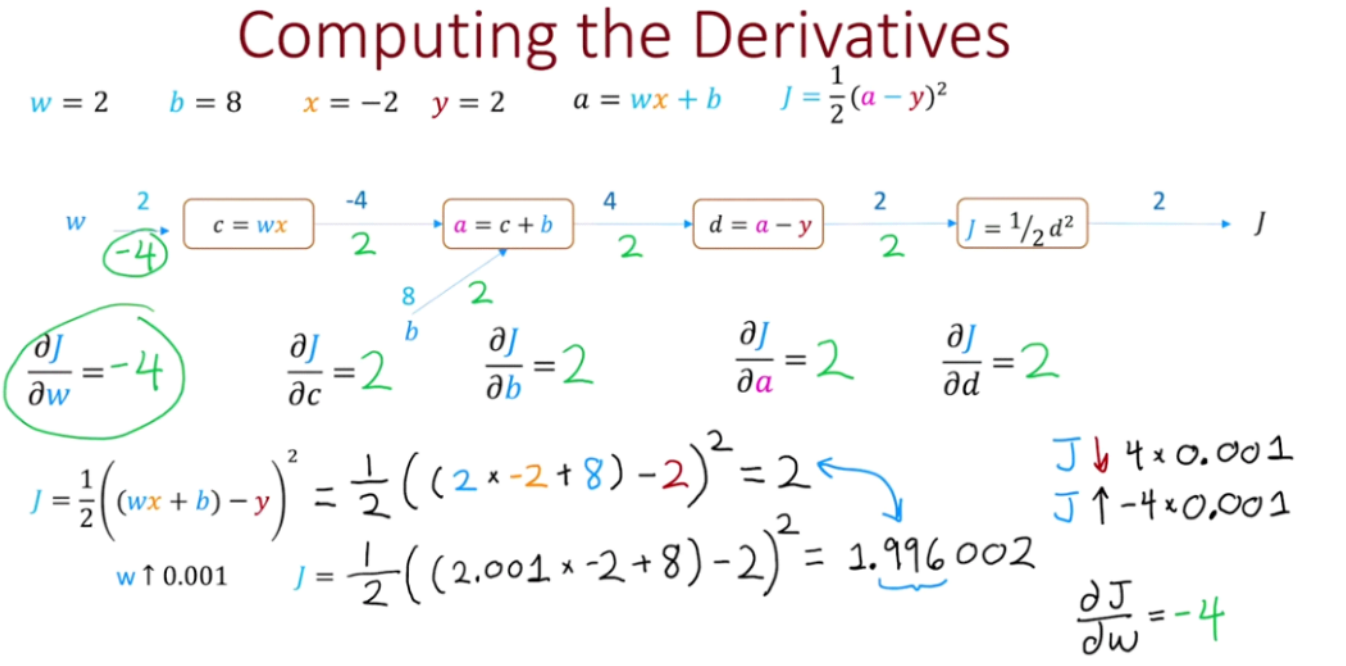


Backpropagation (Computing Derivatives):

- Backpropagation calculates the derivatives of the cost function J with respect to the parameters w and b.
- It's a right-to-left process, unlike forward propagation (left-to-right).

1. **dJ/da (derivative of J w.r.t. a):** How much does J change if a changes slightly? (here, dJ/da = 2).
2. **dJ/dc (derivative of J w.r.t. c):** How much does J change if c changes slightly? (here, dJ/dc = 2 - relies on the chain rule from calculus).
3. **dJ/dw (derivative of J w.r.t. w):** How much does J change if w changes slightly? (here, dJ/dw = -4 - again, chain rule).
4. **dJ/db (derivative of J w.r.t. b):** How much does J change if b changes slightly? (here, dJ/dw = 2 - similar to dJ/dc).



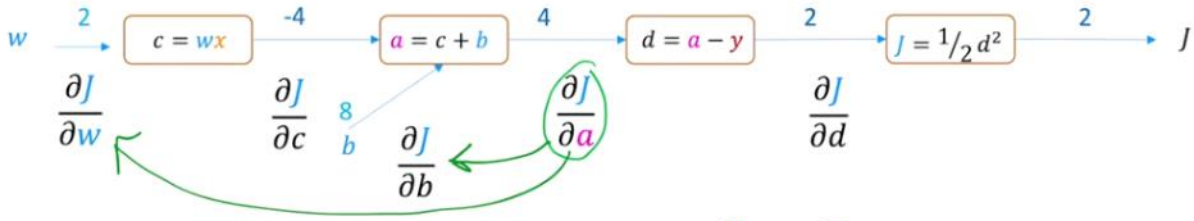


Efficiency of Backpropagation:

- Backprop efficiently computes derivatives by reusing intermediate calculations.

- It takes n + p steps for a graph with n nodes and p parameters, rather than n * p steps (a significant improvement for large networks).

Backprop is an efficient way to compute derivatives



- Compute $\frac{\partial J}{\partial a}$ once and use it to compute both $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$.

  If N nodes and P parameters, compute derivatives in roughly $N + P$ steps rather than $N \times P$ steps.

N	P	N + P	N × P
10,000	100,000	1.1×10^5	10^9

Larger Neural network

24 May 2024 16:46

Key Points:

- We use a ReLU activation function (g(z) = max(0, z)).
- The computation graph visualizes the network's calculations.
- Backpropagation efficiently computes the derivatives needed for learning.
- Modern frameworks (TensorFlow, PyTorch) handle computation graphs and backpropagation automatically.
- Automatic differentiation (autodiff) reduces the need for manual calculus in neural networks.

Breakdown:

1. Network Architecture:

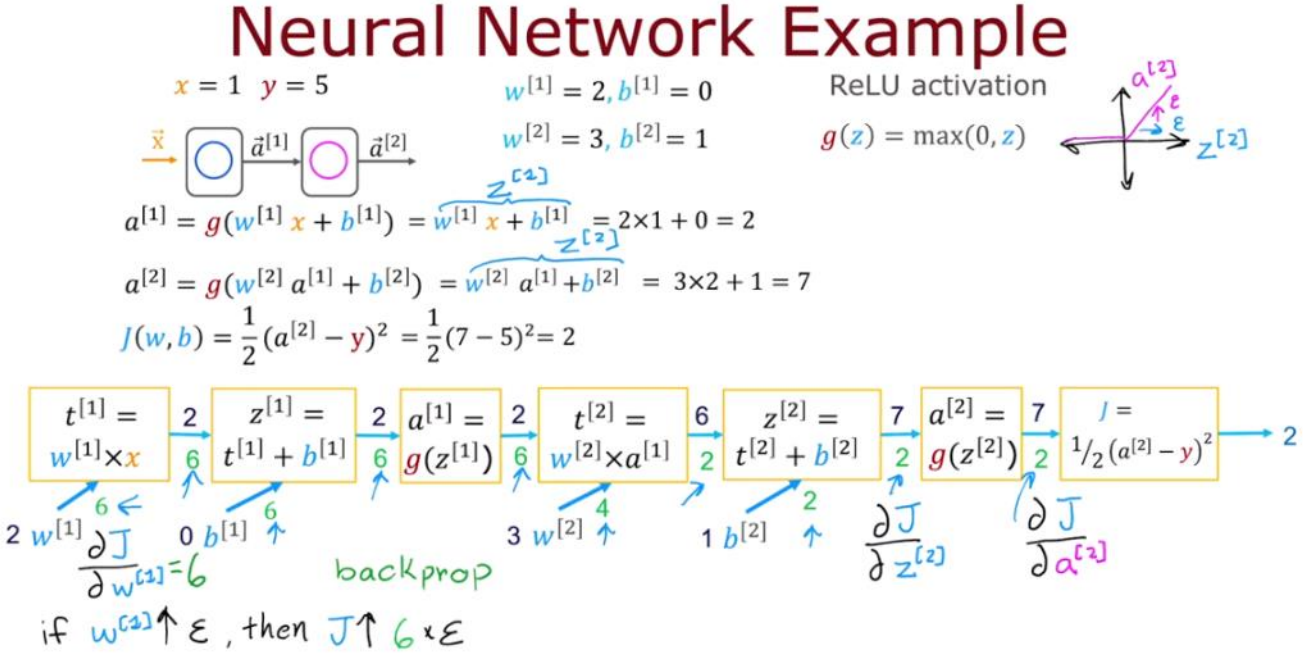
- Single hidden layer with one unit (a1).
- Output layer with one unit (a2).
- ReLU activation function.

2. Forward Propagation:

- Equations for calculating a1, a2, and the cost function (J) are shown step-by-step.
- The computation graph depicts these calculations visually.

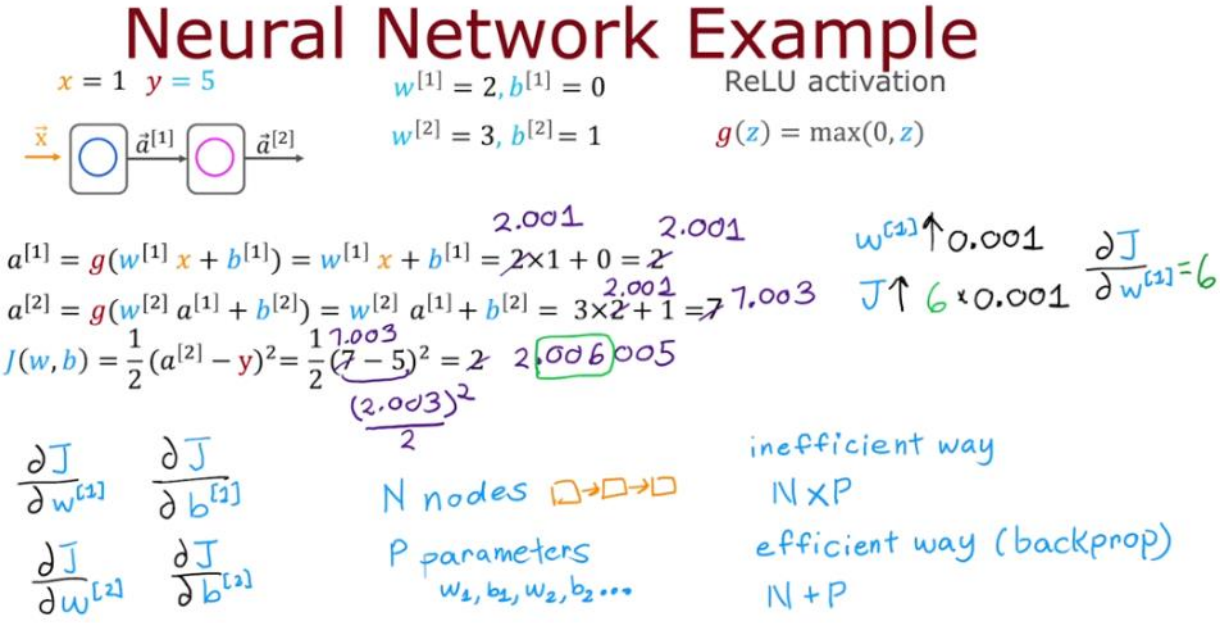
3. Backpropagation (without Explicit Calculations):

- The video mentions backpropagation but doesn't go through detailed calculations.
- It highlights that backprop finds derivatives of the cost function (J) with respect to all parameters (w1, b1, w2, b2).



4. Verification of Backprop (w1 Example):

It demonstrates how a change in w1 affects the cost function (J), confirming backprop's prediction.



5. Efficiency of Backpropagation:

Backprop is efficient because it computes all derivatives in N + P steps (instead of N * P for manual methods), where N is the number of nodes and P is the number of parameters.

6. Benefits of Automatic Differentiation (Autodiff):

- Frameworks handle computation graphs and backpropagation automatically.
- Autodiff reduces the need for manual calculus expertise when using neural networks.
- This has lowered the barrier to entry for learning and implementing neural networks.

Effective Machine Learning Project Management

25 May 2024 08:48

Challenges:

- Building a machine learning system can be time-consuming (e.g., six months vs. a couple of weeks).
- Choosing the most impactful next step is crucial for efficiency.

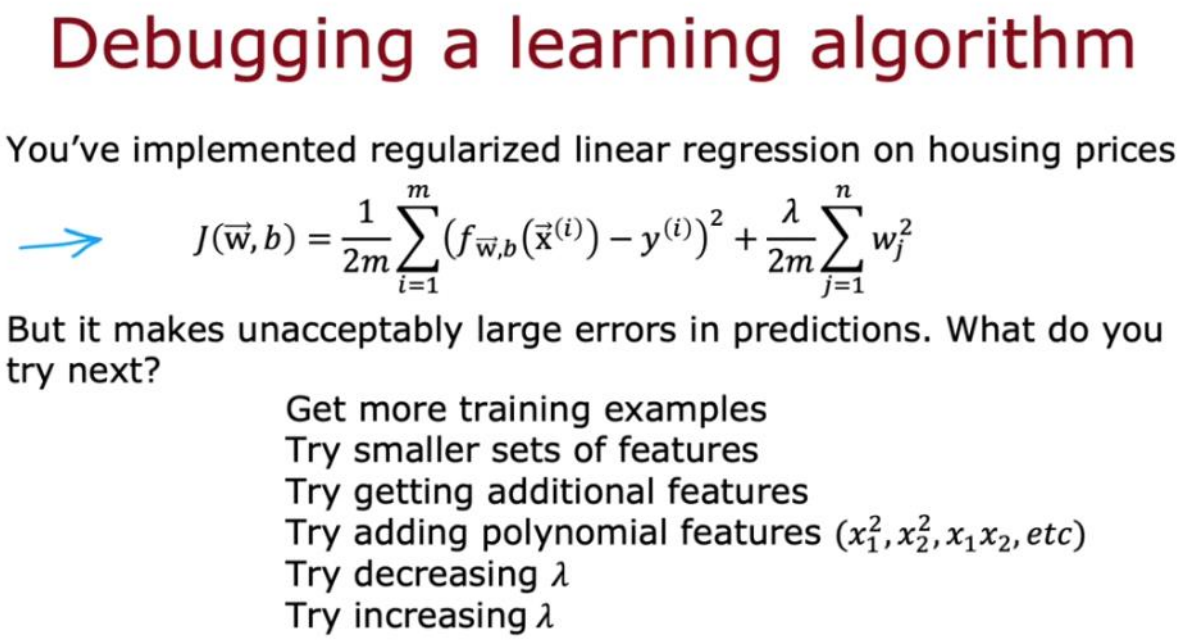
Learning Objectives:

- Develop strategies for making effective decisions in machine learning projects.
- Learn how to conduct diagnostics to improve model performance.

Example: Regularized Linear Regression for Housing Prices

- The model performs poorly with high prediction errors.
- We need to decide what to try next to improve the model.

Potential Next Steps:



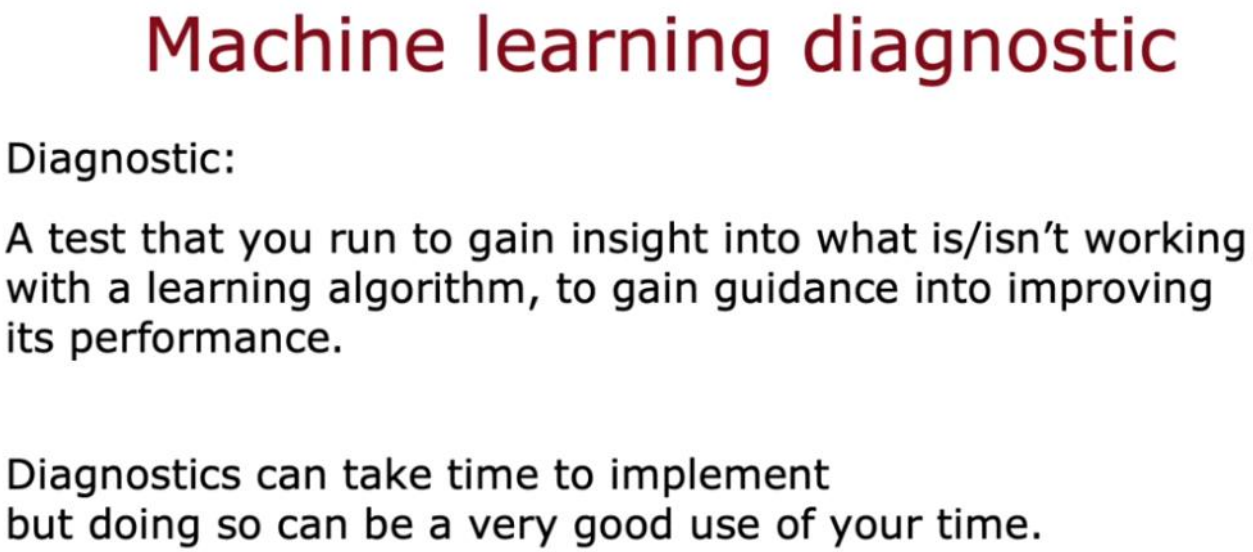
- **Data:**
  - Gather more training data (might or might not be helpful).
  - Reduce the number of features.
  - Include additional features.
  - Engineer new features (e.g., polynomial features).

- **Regularization:**
  - Adjust the lambda value (controls model complexity).

Importance of Effective Decision-Making:

- Prioritize improvements to maximize time investment.
- Avoid spending excessive time on ineffective approaches (e.g., collecting unnecessary data).

Value of Diagnostics:



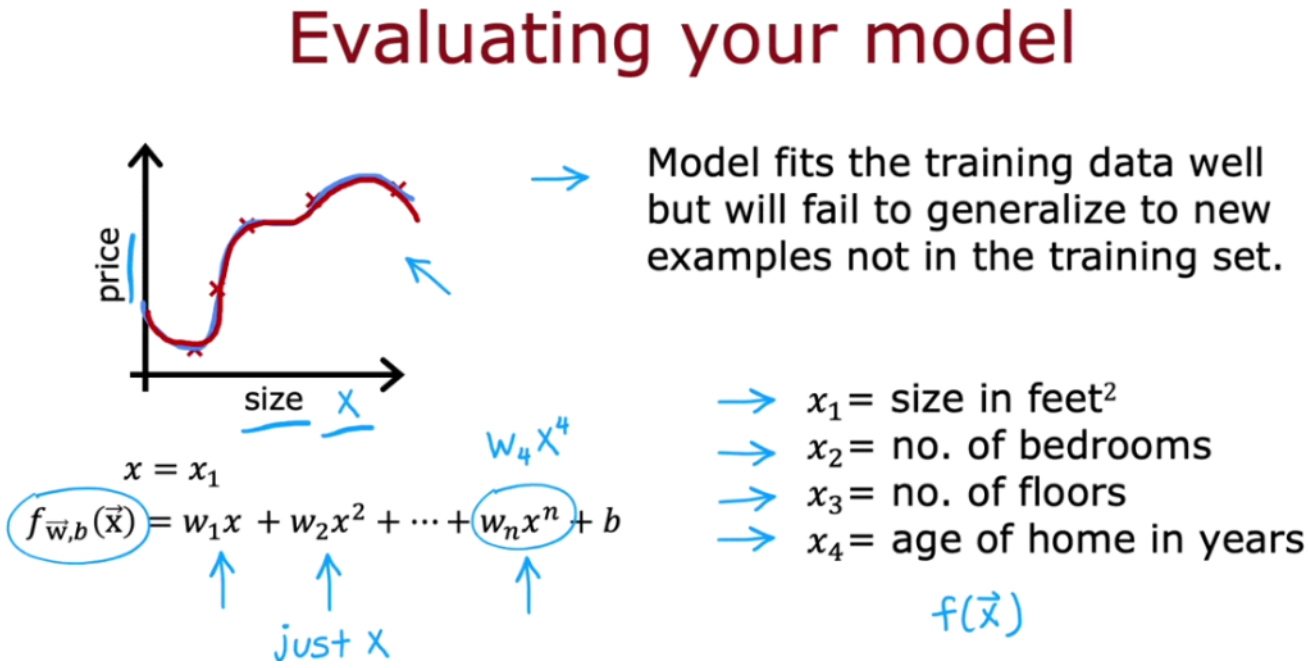
- Diagnostics are tests to assess model behavior and identify improvement opportunities.
- They can save time by guiding resource allocation (e.g., data collection effort).
- Diagnostics may require time to implement, but the benefits outweigh the cost.

Evaluating a Model

25 May 2024 22:53

Challenges of Overfitting:

A complex model (e.g., high-order polynomial) can fit the training data perfectly but fail to generalize to unseen examples.



Also if we have multiple features, say 4 features, how do you go about plotting a 4 dimensional graph of this function?

Evaluation Technique: Train-Test Split

1. Divide the data into two sets:
   - Training set (70%): Used to train the model parameters.
   - Test set (30%): Used to evaluate model performance on unseen data.

2. Notation:
   - x^i, y^i: Training examples (i = 1 to m_{train})
   - x^i_{test}, y^i_{test}: Test examples (i = 1 to m_{test})
   - m_{train}: Number of training examples
   - m_{test}: Number of test examples

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

training set → $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$, ⋮, $(x^{(m_{train})}, y^{(m_{train})})$

m_{train} = no. training examples = 7

test set → $(x_{test}^{(1)}, y_{test}^{(1)})$, ⋮, $(x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

m_{test} = no. test examples = 3

Cost Function and Errors:

- Train the model using the cost function (e.g., squared error for regression) to minimize the error on the training set (J_{train}).
- Evaluate the model's performance on the test set using the test error (J_test).
  - J_{test} = (1 / 2m_test) * Σ(y_i^test - f(x_i^test))^2 (average squared error on test set)
  - **Note:** J_test does not include the regularization term.

Train/test procedure for linear regression (with squared error cost)

Fit parameters by minimizing cost function $J(\vec{w}, b)$

$$J(\vec{w}, b) = \left[\frac{1}{2m_{train}} \sum_{i=1}^{m_{train}} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m_{train}} \sum_{j=1}^{n} w_j^2\right]$$

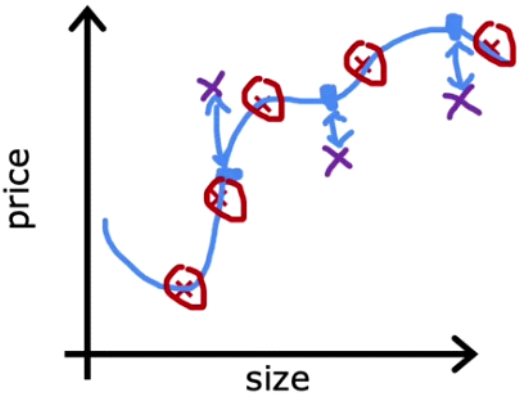
Compute test error:

$$J_{test}(\vec{w}, b) = \frac{1}{2m_{test}} \left[\sum_{i=1}^{m_{test}} \left(f_{\vec{w},b}(\vec{x}_{test}^{(i)}) - y_{test}^{(i)}\right)^2\right]$$

Compute training error:

$$J_{train}(\vec{w}, b) = \frac{1}{2m_{train}} \left[\sum_{i=1}^{m_{train}} \left(f_{\vec{w},b}(\vec{x}_{train}^{(i)}) - y_{train}^{(i)}\right)^2\right]$$

Train/test procedure for linear regression (with squared error cost)



x = train

x = test

$J_{train}(\vec{w}, b)$ will be low

$J_{test}(\vec{w}, b)$ will be high

Train/test procedure for classification problem

0/1

Fit parameters by minimizing $J(\vec{w}, b)$ to find $\vec{w}, b$

E.g.,

$$J(\vec{w}, b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left[y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) \right] + \frac{\lambda}{2m_{train}} \sum_{j=1}^{n} w_j^2$$

Compute test error:

$$J_{test}(\vec{w}, b) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \left[y_{test}^{(i)} \log\left(f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right)\right) + \left(1 - y_{test}^{(i)}\right) \log\left(1 - f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right)\right) \right]$$

Compute train error:

$$J_{train}(\vec{w}, b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left[y_{train}^{(i)} \log\left(f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right)\right) + \left(1 - y_{train}^{(i)}\right) \log\left(1 - f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right)\right) \right]$$

Benefits of Train-Test Split:

- J_test reflects how well the model generalizes to unseen data.
- A high J_test indicates overfitting (model performs well on training data but poorly on unseen data).

Classification Problems:

- Similar approach applies to classification problems (e.g., logistic regression).

Train/test procedure for classification problem

fraction of the test set and the fraction of the train set that the algorithm has misclassified.

- $$\hat{y} = \begin{cases} 1 \text{ if } f_{\vec{w},b}\left(\vec{x}^{(i)}\right) \geq 0.5 \\ 0 \text{ if } f_{\vec{w},b}\left(\vec{x}^{(i)}\right) < 0.5 \end{cases}$$

count $\hat{y} \neq y$

$J_{test}(\vec{w}, b)$ is the fraction of the test set that has been misclassified.

$J_{train}(\vec{w}, b)$ is the fraction of the train set that has been misclassified.

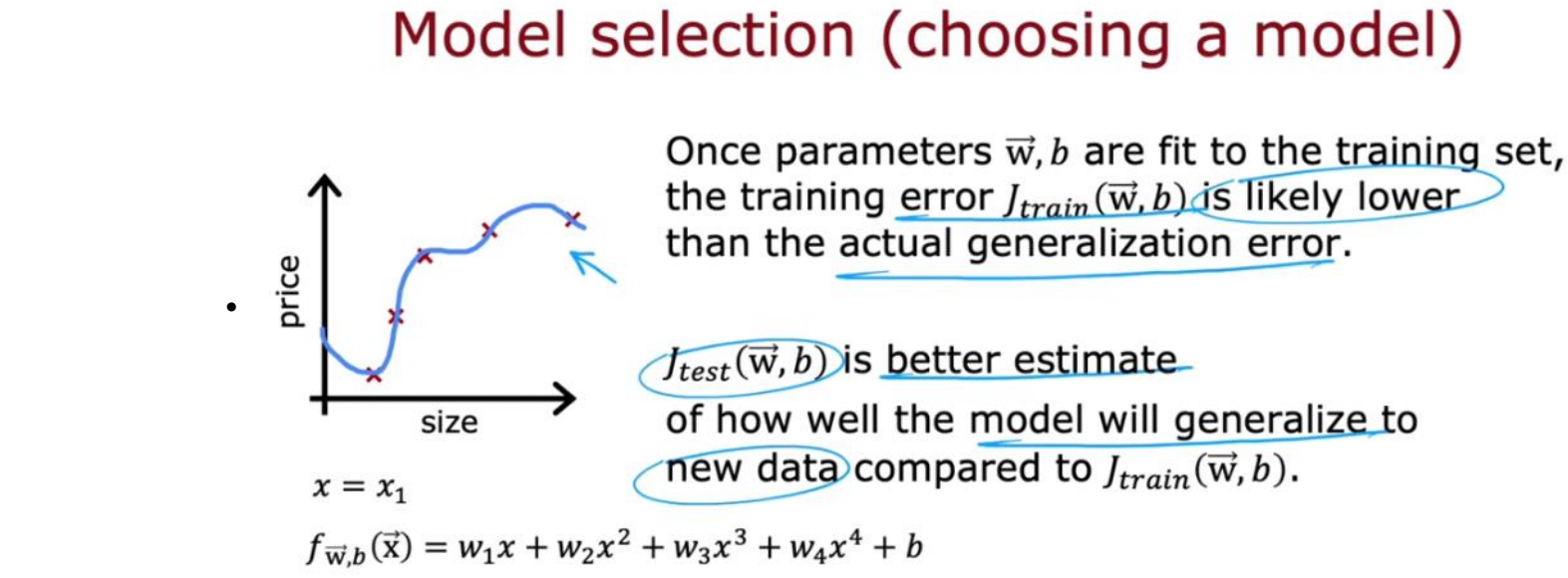
- J_test and J_train can be computed as:
  - Average logistic loss on the test/training set.
  - Fraction of misclassified examples in the test/training set (more commonly used).

Model selection and training/cross validation/test sets

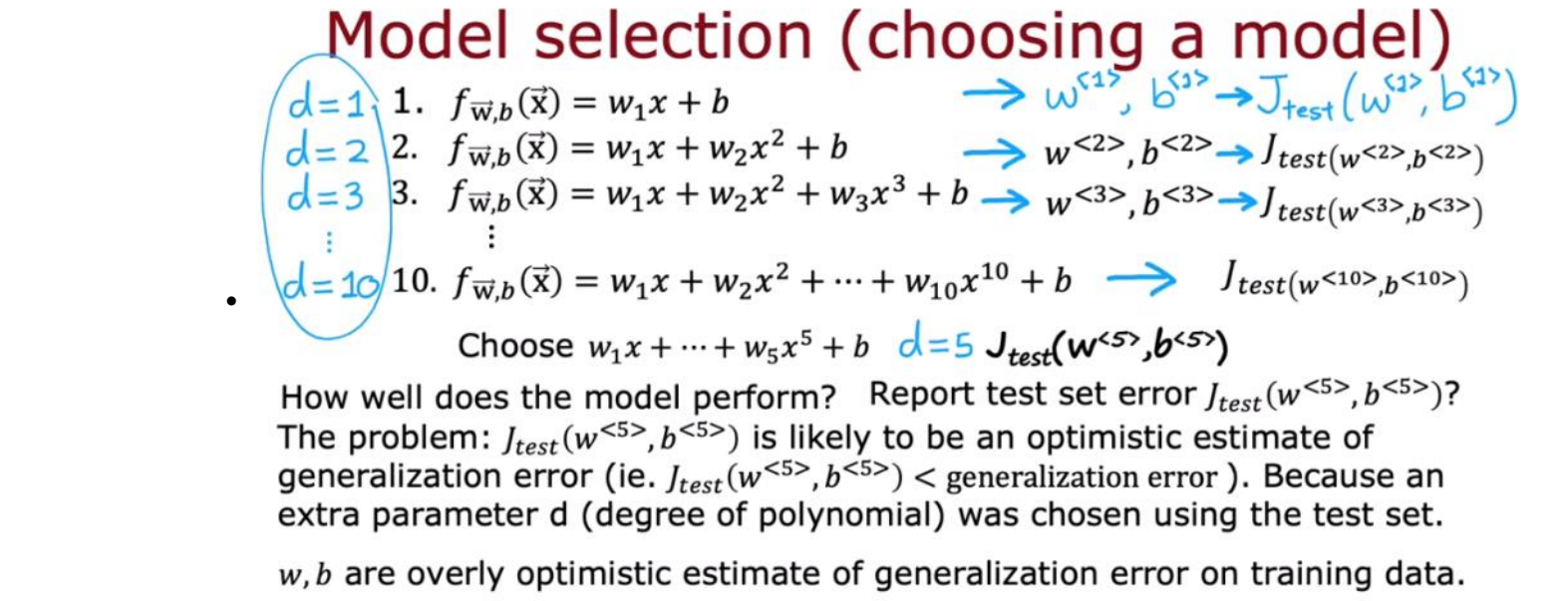
26 May 2024 14:54

Challenges of Choosing a Model:

- Picking the best model complexity (e.g., polynomial degree) is crucial for avoiding overfitting.

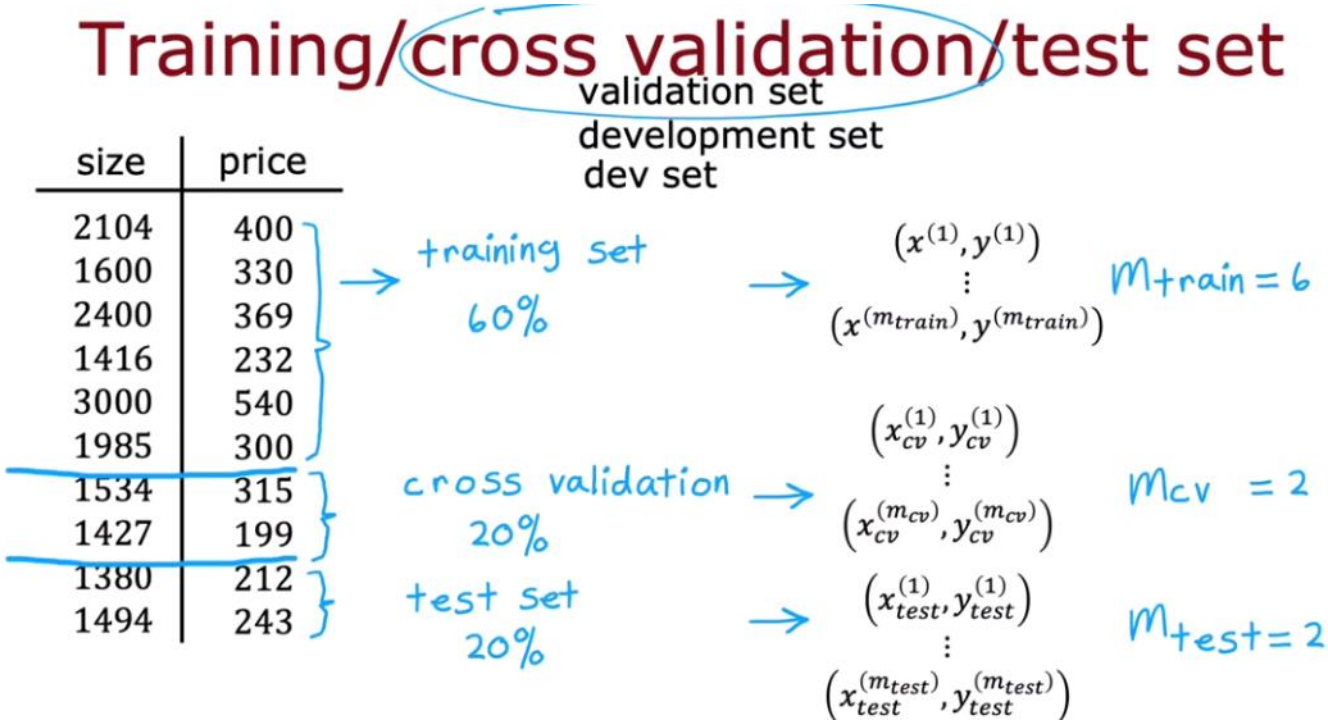


- Using the test set to evaluate multiple models can lead to an overly optimistic estimate of generalization error.



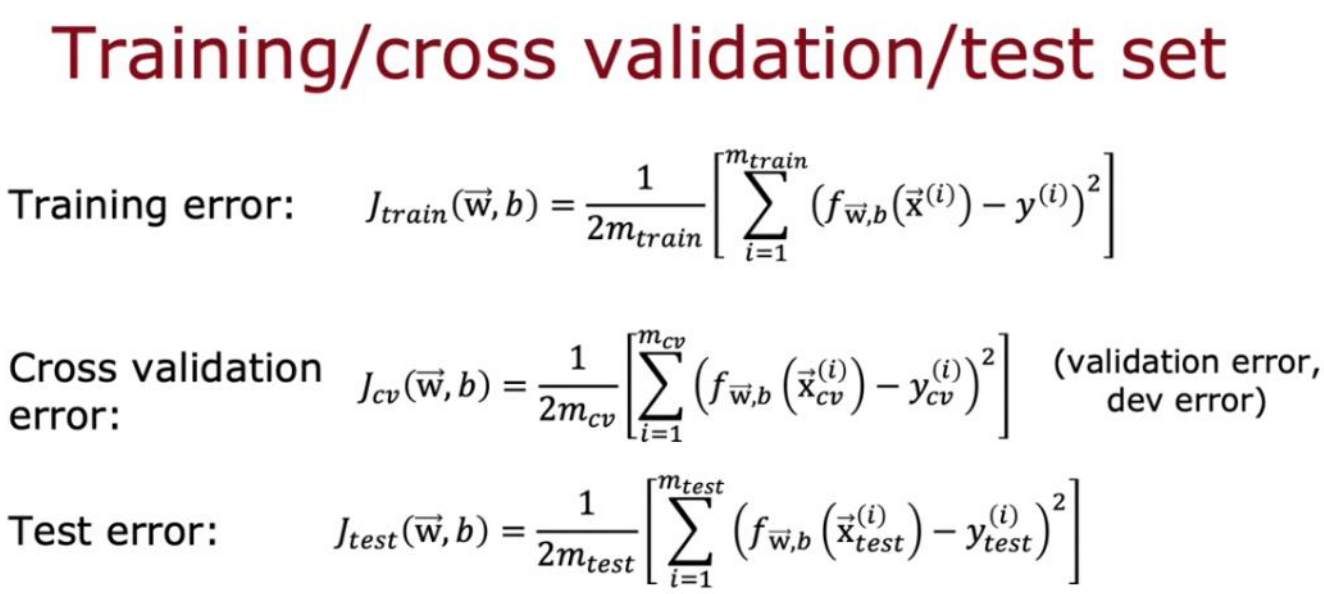
Three-Set Approach:

1. **Training Set (60%):** Used to train model parameters.
2. **Cross-Validation Set (20%):** Used to evaluate different models and choose the best one based on cross-validation error (J_{cv}).
3. **Test Set (20%):** Used to provide an unbiased estimate of the chosen model's generalization error (J_{test}).



Benefits:

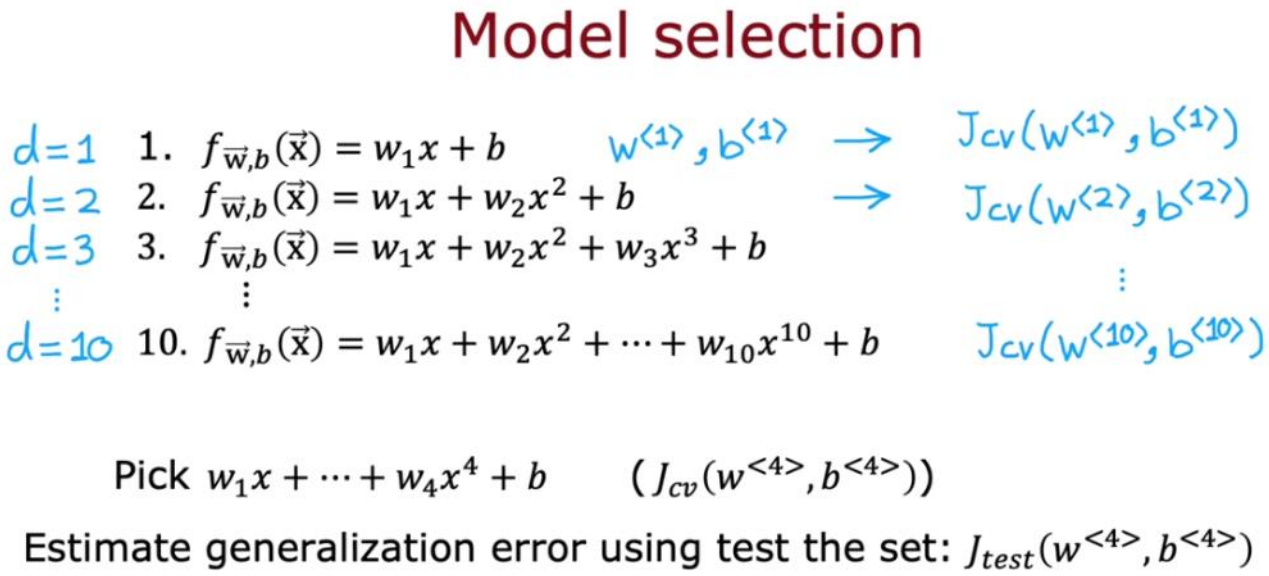
- J_cv helps identify the model with the lowest generalization error without using the test set.
- J_test provides a fair estimate of the chosen model's performance on unseen data.



Model Selection Procedure (Example: Polynomial Regression):

1. Consider multiple models with different complexities (e.g., linear, quadratic, cubic polynomials).
2. Train each model on the training set and obtain parameters (w, b) for each model.
3. Evaluate each model on the cross-validation set to compute J_cv.
4. Choose the model with the lowest J_cv (e.g., fourth-order polynomial).
5. Report the test error (J_test) of the chosen model on the test set for an unbiased estimate of generalization error.

6.



Generalization Error and Model Selection:

This procedure avoids using the test set for model selection, ensuring an unbiased J_test.

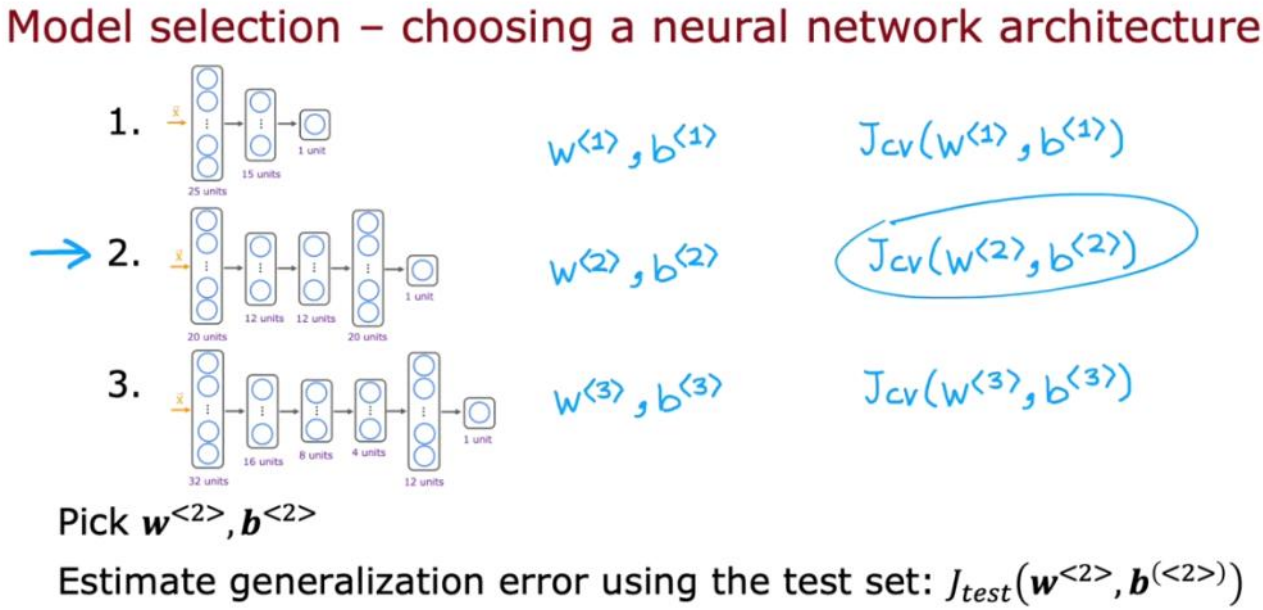
Model Selection Beyond Linear Regression:

This approach applies to various machine learning models, including neural networks.

Example: Neural Network Architecture Selection:

1. Consider multiple neural network architectures with different numbers of layers and hidden units.
2. Train each network on the training set and obtain parameters (w, b) for each network.
3. Evaluate each network on the cross-validation set using J_cv (e.g., misclassification rate).
4. Choose the network with the lowest J_cv.
5. Report the test error (J_test) of the chosen network on the test set.

6.



Best Practices in Machine Learning:

- Use the training set and cross-validation set for all model decisions (parameter fitting, architecture selection).
- Avoid using the test set until the final model selection.
- This ensures an unbiased estimate of model performance on unseen data.

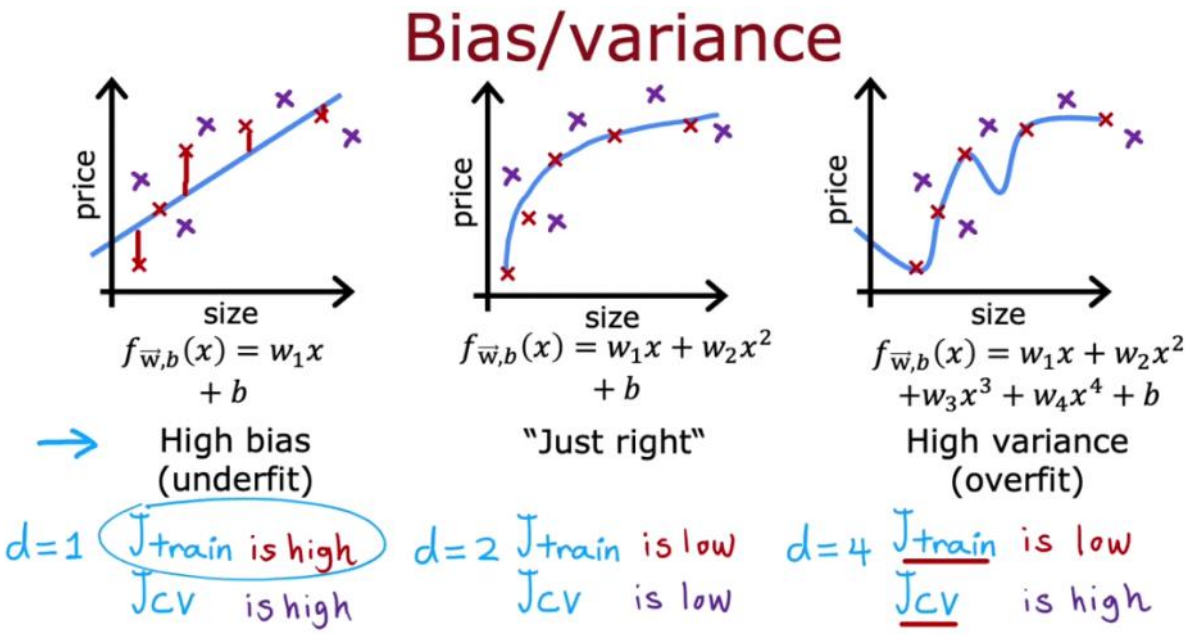
Diagnosis bias and variance

26 May 2024 15:36

Challenges of Model Performance:

- Underfitting (high bias) occurs when a model is too simple to capture the data's complexity, resulting in poor performance on both training and unseen data.
- Overfitting (high variance) occurs when a model is too complex and memorizes the training data without generalizing well to unseen data.

Diagnosing Bias and Variance:

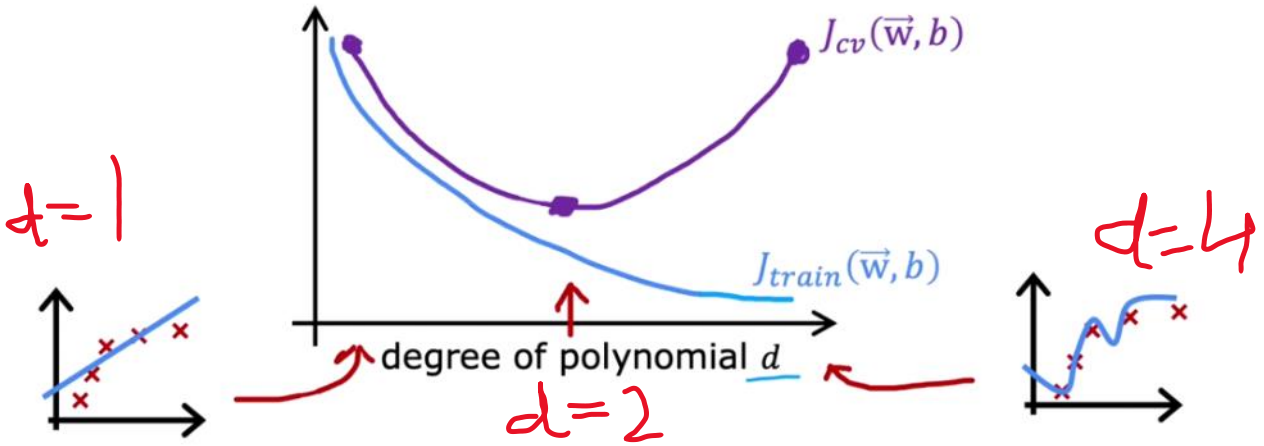
- Analyze the performance on the training set (J_{train}) and the cross-validation set (J_{cv}).
- 

- **High Bias:** J_{train} and J_{cv} are both high. The model underfits the data.
- **High Variance:** J_{train} is low (fits the training data well), but J_{cv} is much higher (doesn't generalize well). The model overfits the training data.

Visualizing Bias and Variance:

- Plot J_{train} and J_{cv} as functions of model complexity (e.g., polynomial degree).

Understanding bias and variance

- 

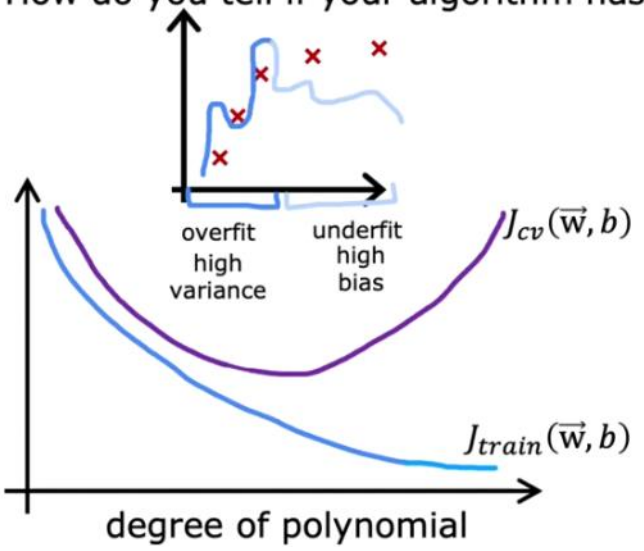
- **Bias-Variance Tradeoff:** For low complexity, J_train and J_{cv} are high (underfitting). For high complexity, J_{train} is low but J_{cv} is high (overfitting). The optimal model complexity lies between these extremes, where J_{train} and J_{cv} are both relatively low.

Additional Considerations:

- **Both High Bias and High Variance (Rare):** J_{train} is high, and J_{cv} is even higher. The model might underfit some parts of the input and overfit others.
- **Bias and Variance in Linear Regression:** Primarily applies to high bias or high variance, not both simultaneously.

Diagnosing bias and variance

How do you tell if your algorithm has a bias or variance problem?

- 

High bias (underfit)
→ J_{train} will be high
($J_{train} \approx J_{cv}$)

High variance (overfit)
→ $J_{cv} \gg J_{train}$
(J_{train} may be low)

High bias and high variance
→ J_{train} will be high
and $J_{cv} \gg J_{train}$

Key Takeaways:

- High bias: J_{train} is high (not performing well on the training set).
- High variance: J_{cv} is much higher than J_{train} (poor generalization to unseen data).

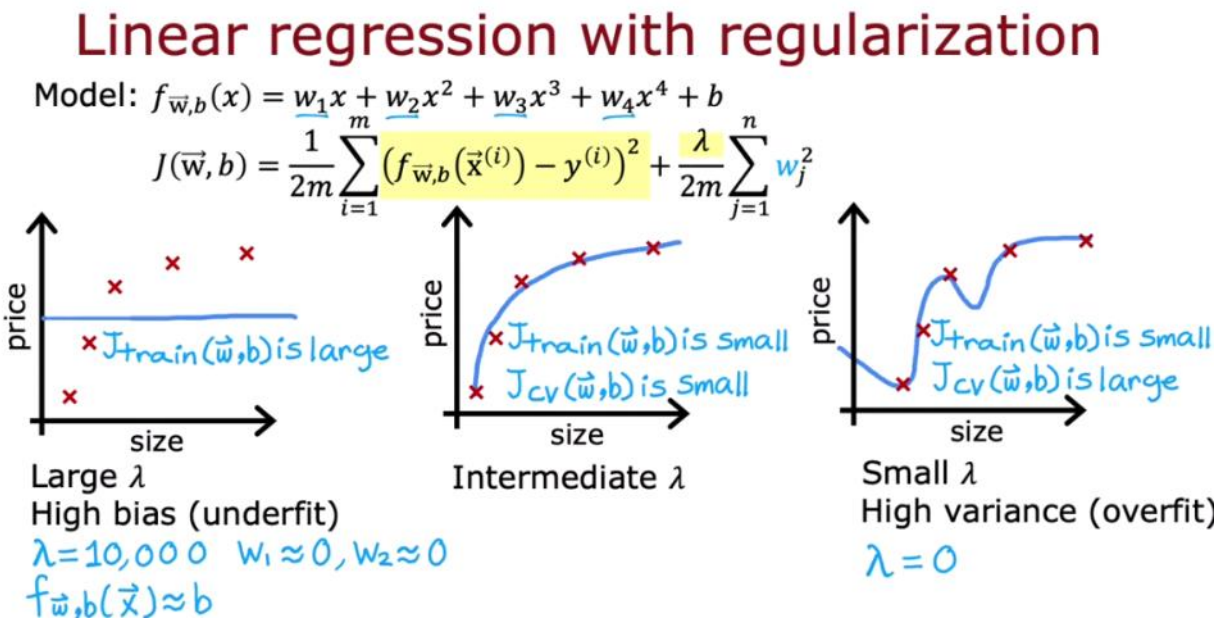
Regularization and Bias/variance

27 May 2024 17:42

Regularization and Model Complexity:

We'll use a fourth-order polynomial model with regularization (λ controls the trade-off between keeping parameters small and fitting the training data well).

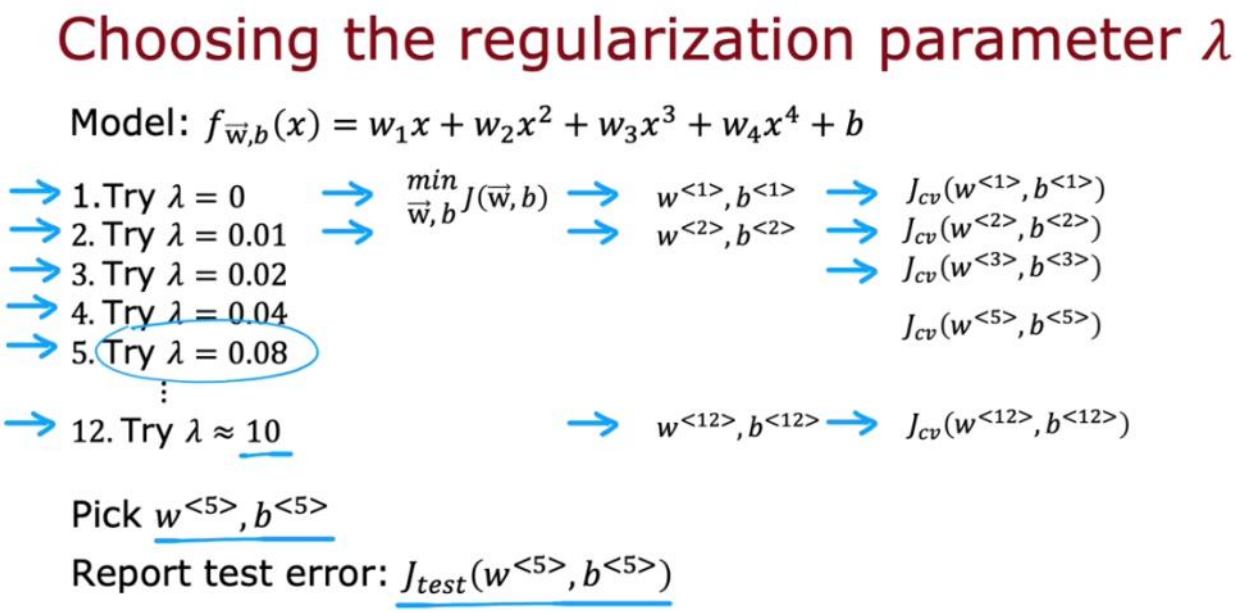
Impact of Lambda on Model Complexity:

- 

- **High Lambda (λ=10,000):** The model becomes a simple horizontal line (high bias, underfits the data).
- **Low Lambda (λ=0):** The model becomes a very wiggly curve (high variance, overfits the data).
- **Intermediate Lambda:** The model fits the data well with a balance between bias and variance.

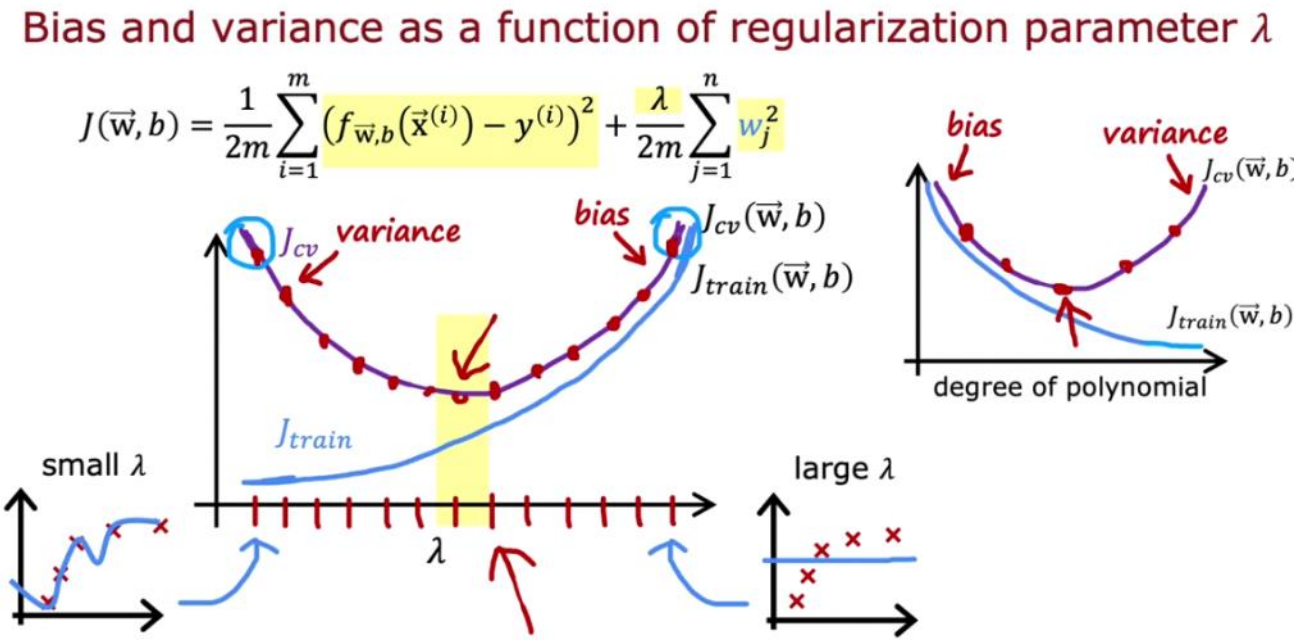
Choosing Lambda Using Cross-Validation:

1. Fit the model with various λ values (e.g., λ=0, 0.01, 0.02, ..., 10).
2. For each λ, compute the cross-validation error (J_cv).
3. Choose the λ that results in the lowest J_cv.



Visualizing Bias-Variance Tradeoff:

- Plot J_train and J_cv as functions of λ.
- **Similar to Model Complexity:** Low λ leads to high variance (low J_train, high J_cv). High λ leads to high bias (high J_train, high J_cv).
- **Minimum J_cv:** The optimal λ minimizes J_cv (good balance between bias and variance).



Comparison with Model Complexity:

- The diagrams for choosing λ and model complexity (polynomial degree) are somewhat like mirror images.
- In both cases, cross-validation helps you choose a good parameter value (λ or polynomial degree).

Key Takeaways:

- High J_train indicates high bias (underfitting).
- High J_cv (much higher than J_train) indicates high variance (overfitting).

- Choosing the right λ balances bias and variance for optimal performance.
- Cross-validation helps determine a good λ value.

Interpreting J-Train and J-CV for Bias and Variance

29 May 2024 11:19

Example: Speech Recognition

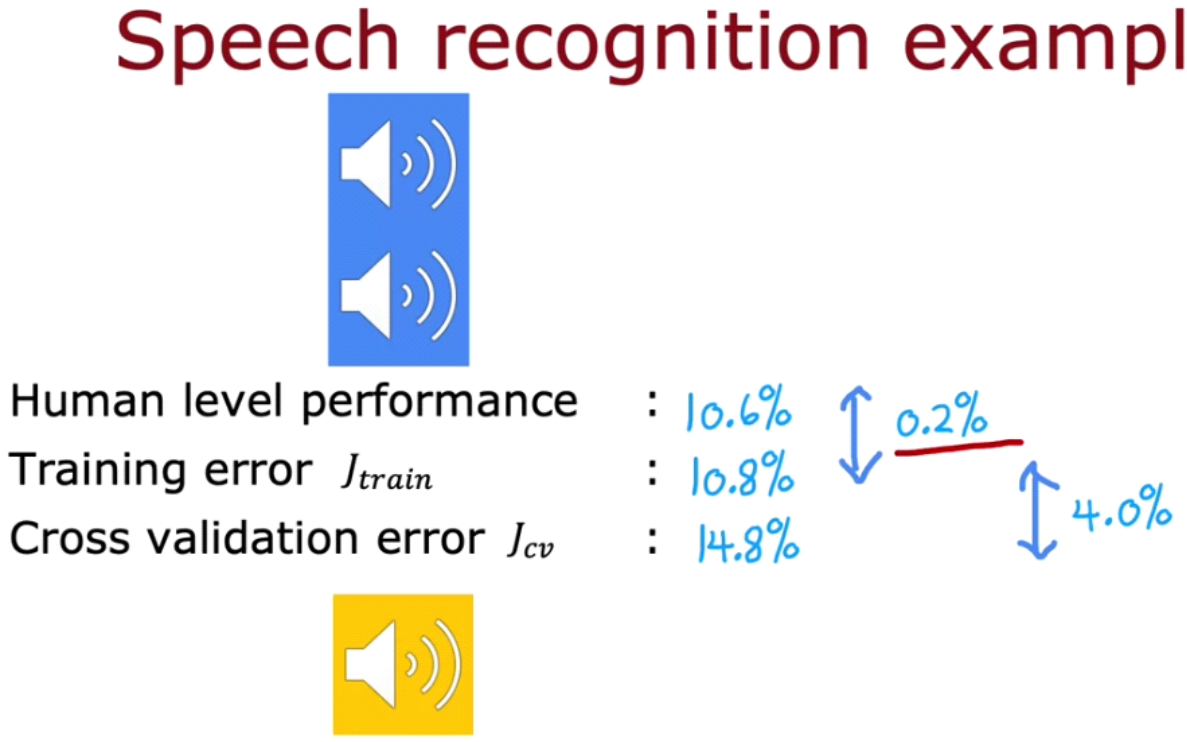
- A speech recognition system aims to transcribe audio clips.
- Training error: 10.8% (incorrect transcriptions)
- Cross-validation error: 14.8%

Evaluating Performance:

- These numbers seem high, but human performance (baseline) is also considered.
- Human error: 10.6% (due to noisy audio)

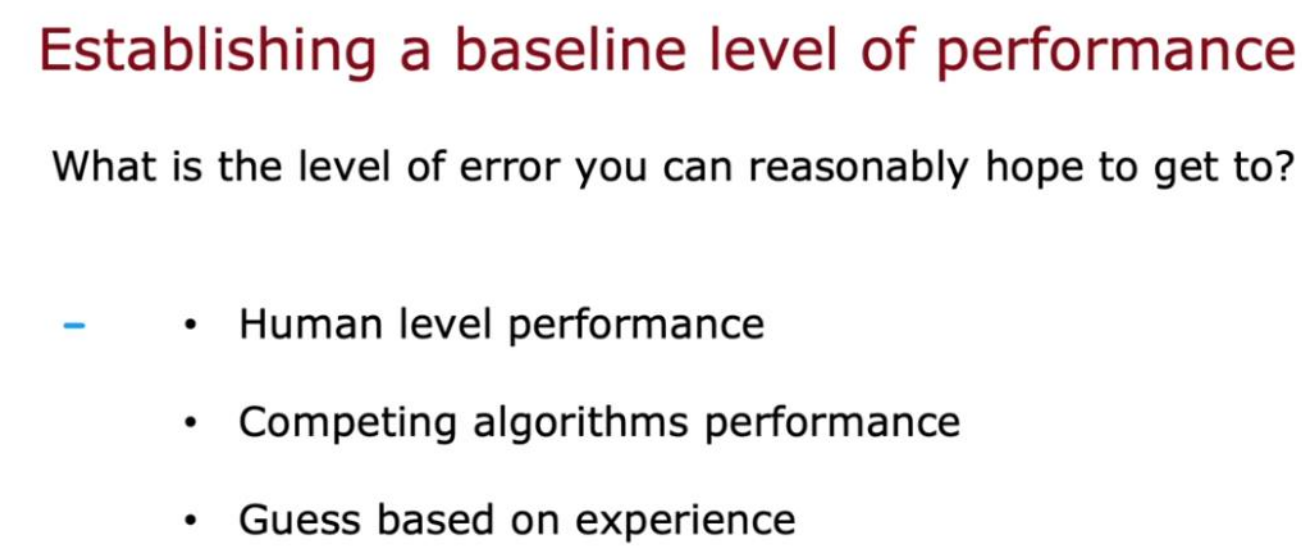
Reasoning:

- The training error (0.2% worse than humans) is acceptable.
- **The larger gap between J-cv and J-train (4%) suggests high variance.**



Establishing Baseline Performance:

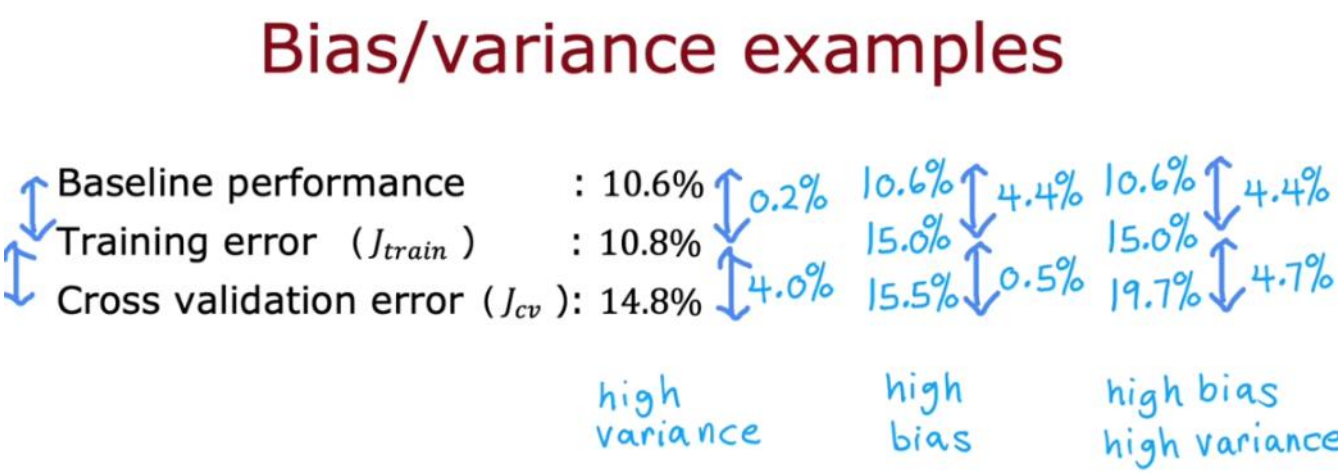
- Crucial for interpreting J-train as "high."
- Options for baselines:
  - Human performance (common for unstructured data like speech, images, text).
  - Existing algorithms (competitors or previous versions).
  - Informed guesses based on experience.



Key Quantities:

- Difference between training error and baseline: Indicates high bias (large gap).
- Difference between training error and cross-validation error: Indicates high variance (large gap).

Second Example:



Summary:

- The gap between training error and baseline indicates high bias.
- The gap between training error and cross-validation error indicates high variance.
- Baseline can be zero (perfect performance) or higher (noisy data).

High Bias and High Variance (Rare Case):

Both gaps (training error vs. baseline and training error vs. J-cv) are large.

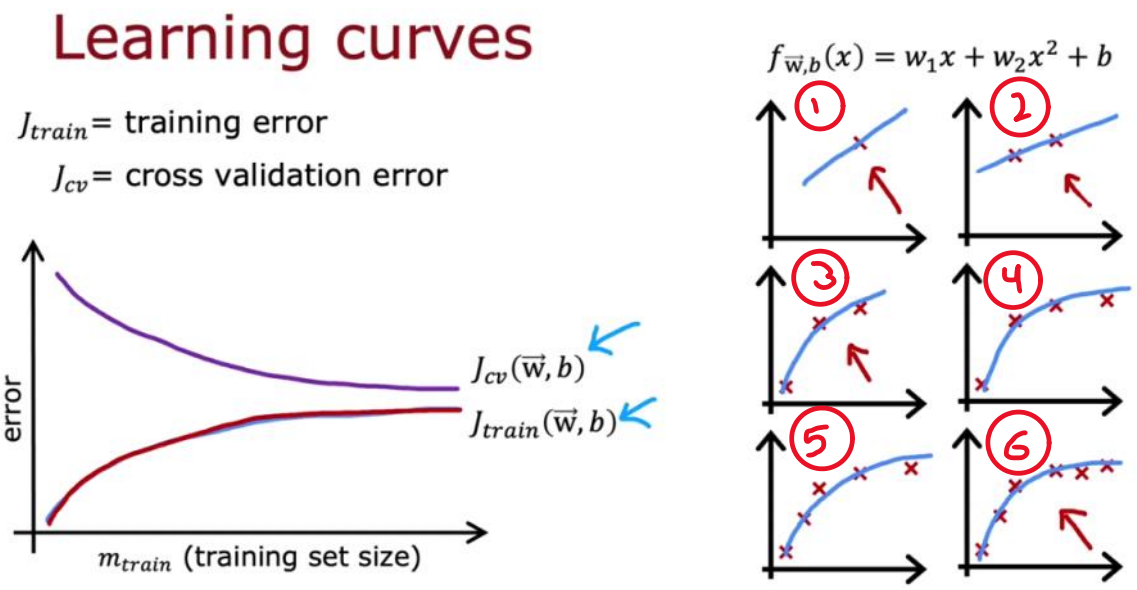
Learning Curves

29 May 2024 21:18

Learning Curve Basics:

- Learning curves are a way to help understand how your learning algorithm is doing as a function of the amount of experience (example: training examples) it has.

Here's plot of the learning curves for a model that fits a second-order polynomial quadratic function like so.



- X-Axis: training set size
  Y-Axis: Cross validation Error (J_{cv}) and Training Error (J_{train})

- As the training size gets bigger, you learn a better model and so J_{cv} goes down.
  On the contrary, J_{train} increases.

Why does J_{train} increase?

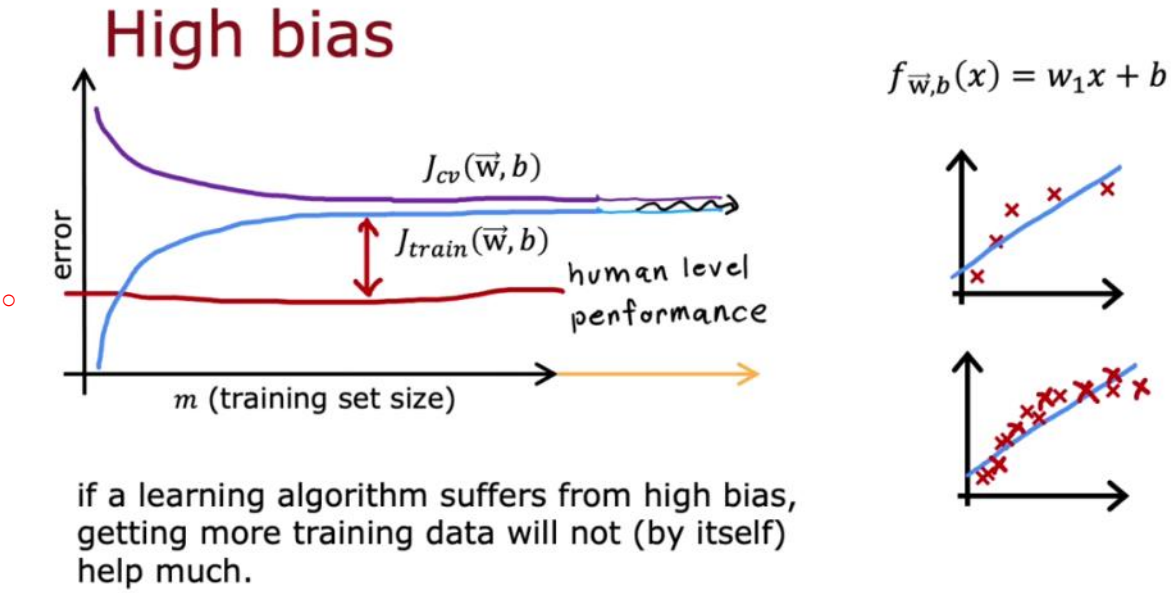
1. We'll start with an example of when you have just a single training example (1). Well, if you were to fit a quadratic model to this, you can fit easiest straight line or a curve and your training error will be zero.
2. How about if you have two training examples like (2)? Well, you can again fit a straight line and achieve zero training error.
3. In fact, if you have three training examples (3), the quadratic function can still fit this very well and get pretty much zero training error
4. If your training set gets a little bit bigger, say you have four training examples (4), then it gets a little bit harder to fit all four examples perfectly. You may get a curve that looks like this, is a pretty well, but you're a little bit off in a few places here and there. When you have increased, the training set size to four the training error has actually gone up a little bit.
5. In examples (5) and (6) you see the line doesn't fit as perfectly as it did earlier with lesser examples.

Which is why as the training set gets bigger, the training error increases because it's harder to fit all of the training examples perfectly. Notice one other thing about these curves, which is the cross-validation error, will be typically higher than the training error because you fit the parameters to the training set. You expect to do at least a little bit better or when m is small, maybe even a lot better on the training set than on the trans validation set.

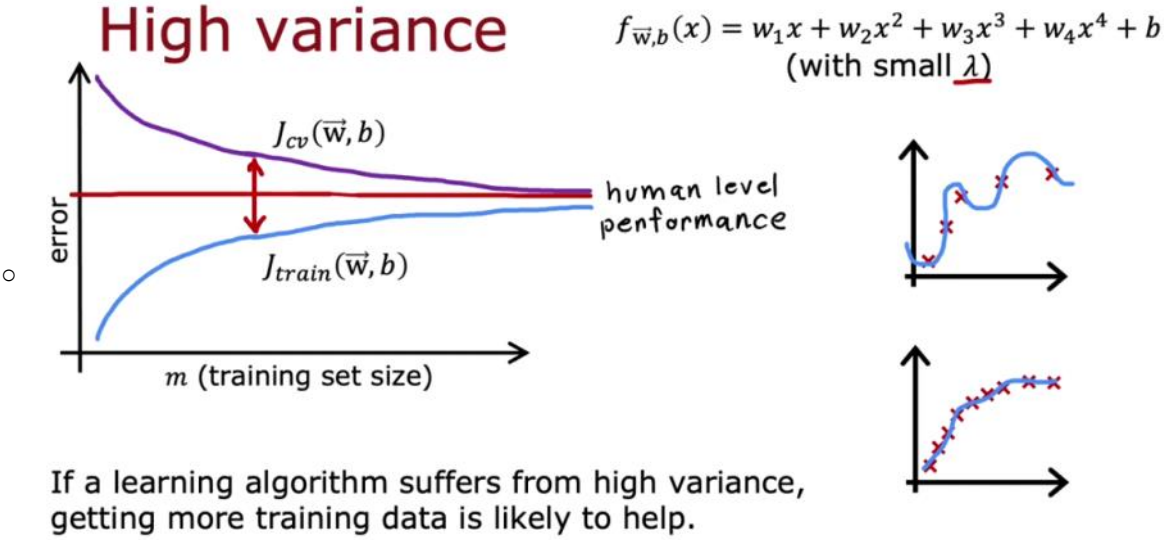
High Bias vs. High Variance in Learning Curves:

- **High Bias (Underfitting):**
  - J_train increases with m_train (flattens out later).

- J_cv also increases (flattens out later).
- Both curves stay above the baseline level (e.g., human performance).
- **Adding more training data won't significantly improve performance.**



- **High Variance (Overfitting):**
  - J_train decreases rapidly with m_train (can even reach zero).
  - J_cv much higher than J_train (large gap).
  - J_train might be lower than the baseline level (unrealistic performance).
  - **Adding more training data can significantly reduce J_cv.**



Key Points:

- A large gap between J_train and the baseline suggests high bias.
- A large gap between J_train and J_cv suggests high variance.
- High bias: More training data won't help much.
- High variance: More training data can significantly improve performance.

Deciding what to try next

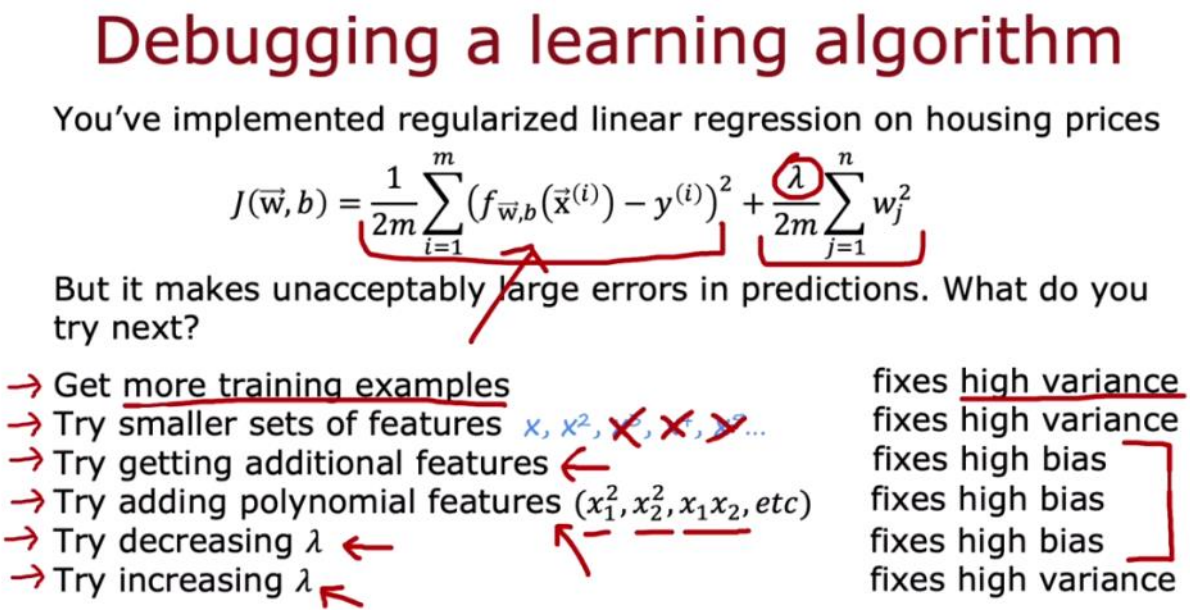
High Bias vs. Solutions:

- **Symptoms:** J_train increases with training data, stays above baseline performance.

- **Problem:** Model underfits the data (too simple).
- **Solutions:**
  - **Get additional features:** Provide more relevant information for predictions.
  - **Add polynomial features:** Increase model complexity to capture non-linear relationships.
  - **Decrease Lambda (regularization parameter):** Allow the model to fit a more complex function.

High Variance vs. Solutions:

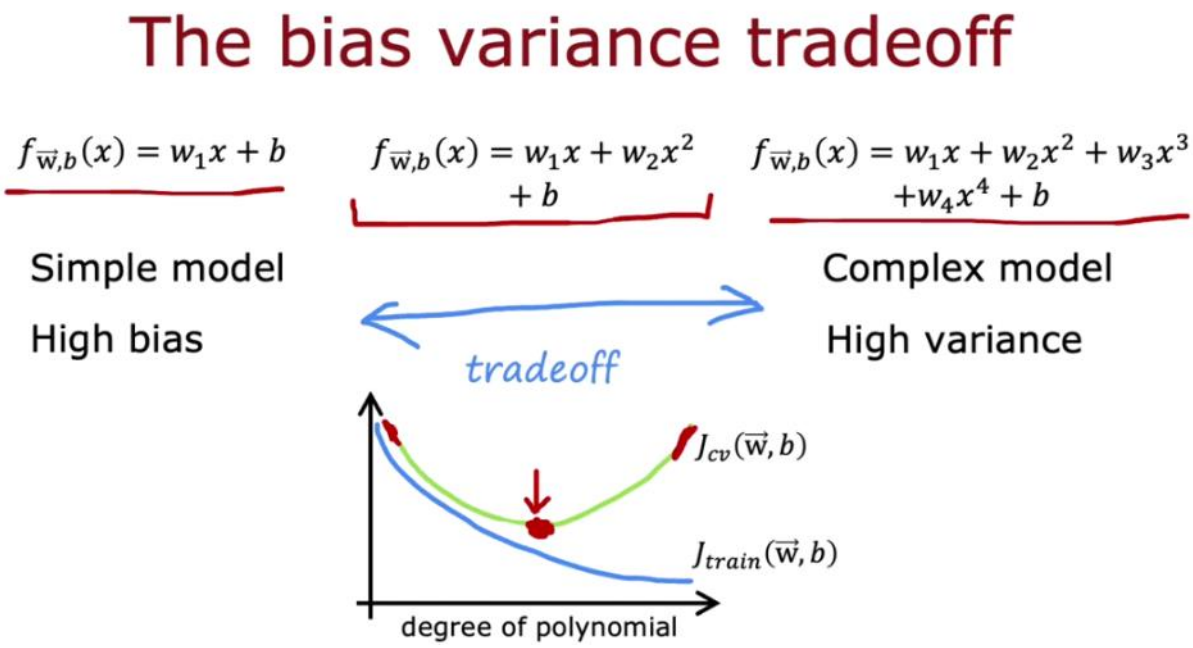
- **Symptoms:** J_train rapidly decreases, J_cv much higher than J_train (large gap).
- **Problem:** Model overfits the training data (too complex).
- **Solutions:**
  - **Get more training data:** Provide more examples to reduce overfitting.
  - **Try a smaller set of features:** Reduce model flexibility to prevent overfitting to noise.
  - **Increase Lambda (regularization parameter):** Reduce model flexibility and focus on generalizability.



High Bias and High Variance

- **High Bias and High Variance**: Both are detrimental as they hurt the performance of the algorithm.
- **Neural Networks' Advantage**: Neural networks, combined with large datasets, can address both high bias and high variance effectively.

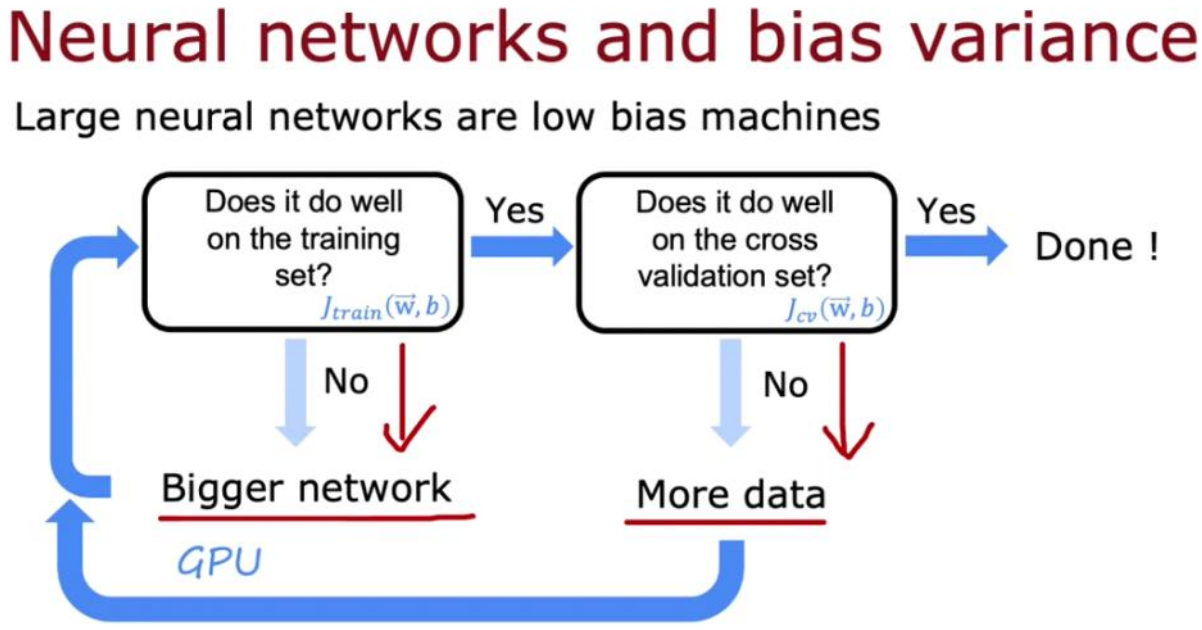
Bias-Variance Tradeoff



- **Bias-Variance Tradeoff**:
  - Simple models (e.g., linear models) can have high bias.
  - Complex models can suffer from high variance.
  - The tradeoff involves choosing a model with the lowest possible cross-validation error.
- **Pre-Neural Networks Era**: Engineers balanced model complexity (polynomial degree or regularization) to manage bias and variance.

Neural Networks' Solution

- **Large Neural Networks**: When trained on small to moderate-sized datasets, they are typically low bias.



- **Recipe for Model Improvement**:
  1. **Train on Training Set**: Measure JtrainJtrain.
  2. **High Bias Problem**: If JtrainJtrain is high, use a larger network (more layers/units).
  3. **Cross-Validation Performance**: Check if it performs well on the cross-validation set.
  4. **High Variance Problem**: If there's a gap between JtrainJtrain and JcvJcv, collect more data and retrain.
  5. **Iterative Process**: Continue adjusting until the model performs well on both training and cross-validation sets.

Limitations

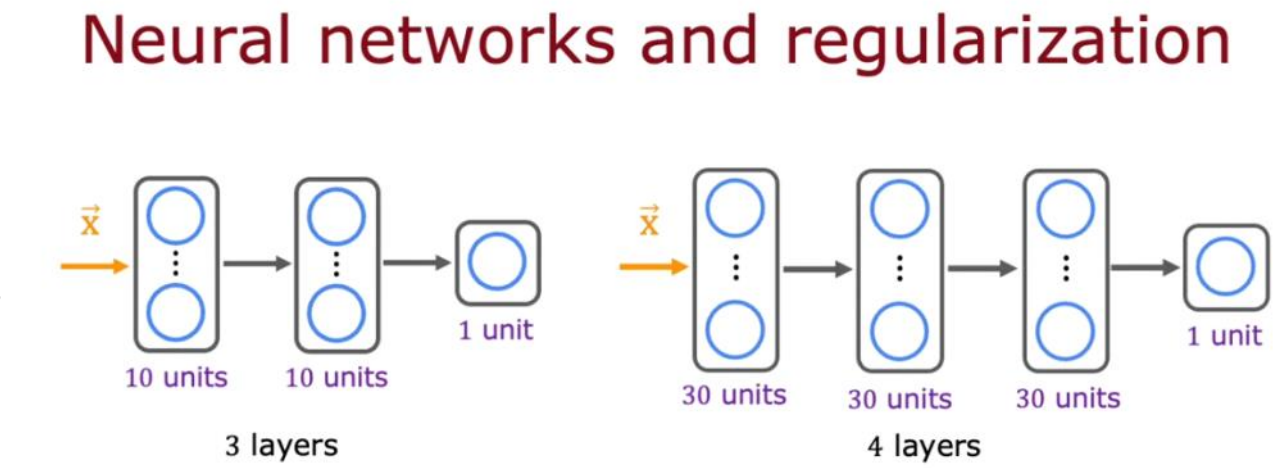
- **Computational Expense**: Training large neural networks can become infeasible due to computational limits.
- **Data Availability**: There are practical limits to how much data can be collected.

Rise of Deep Learning

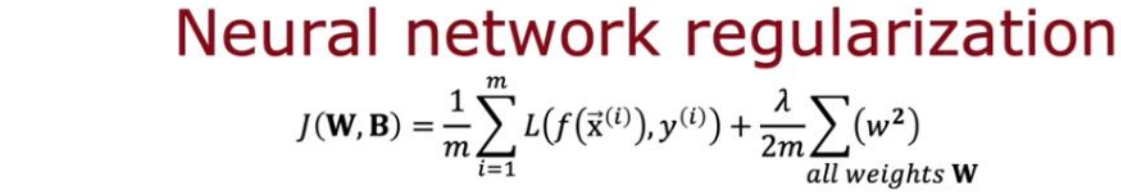
- **Deep Learning Success**:
  - Access to large datasets.
  - Training large neural networks has led to good performance on many applications.
- **Bias and Variance During Development**:
  - Adjustments may need to address high bias or high variance at different stages.

Regularization

- **Large Neural Networks**: With appropriate regularization, larger networks usually perform as well or better than smaller ones.



- **Regularization in TensorFlow**:



Unregularized MNIST model

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

Regularized MNIST model

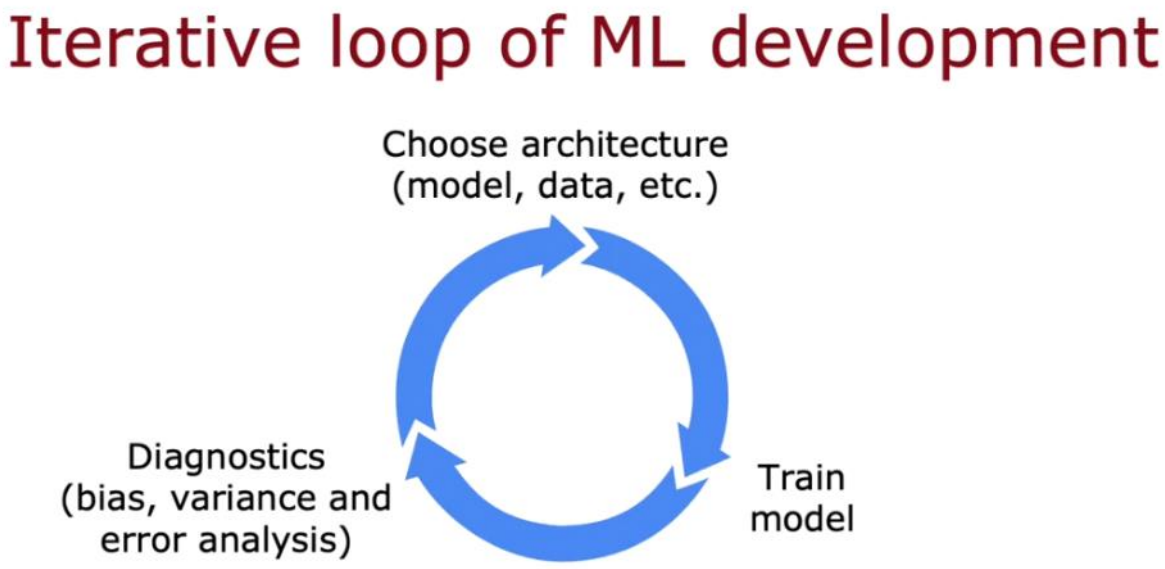
```
layer_1 = Dense(units=25, activation="relu", kernel_regularizer=L2(0.01))
layer_2 = Dense(units=15, activation="relu", kernel_regularizer=L2(0.01))
layer_3 = Dense(units=1, activation="sigmoid", kernel_regularizer=L2(0.01))
model = Sequential([layer_1, layer_2, layer_3])
```

Machine Learning Development

30 May 2024 15:59

Iterative Loop of Development:

1. Decide on system architecture: Choose model, data, hyperparameters, etc.
2. Implement and train the model.
3. Use diagnostics to evaluate performance (bias, variance, and error analysis).
4. Adjust model based on diagnostics (e.g., larger network, different regularization, more data).
5. Repeat until desired performance is achieved.

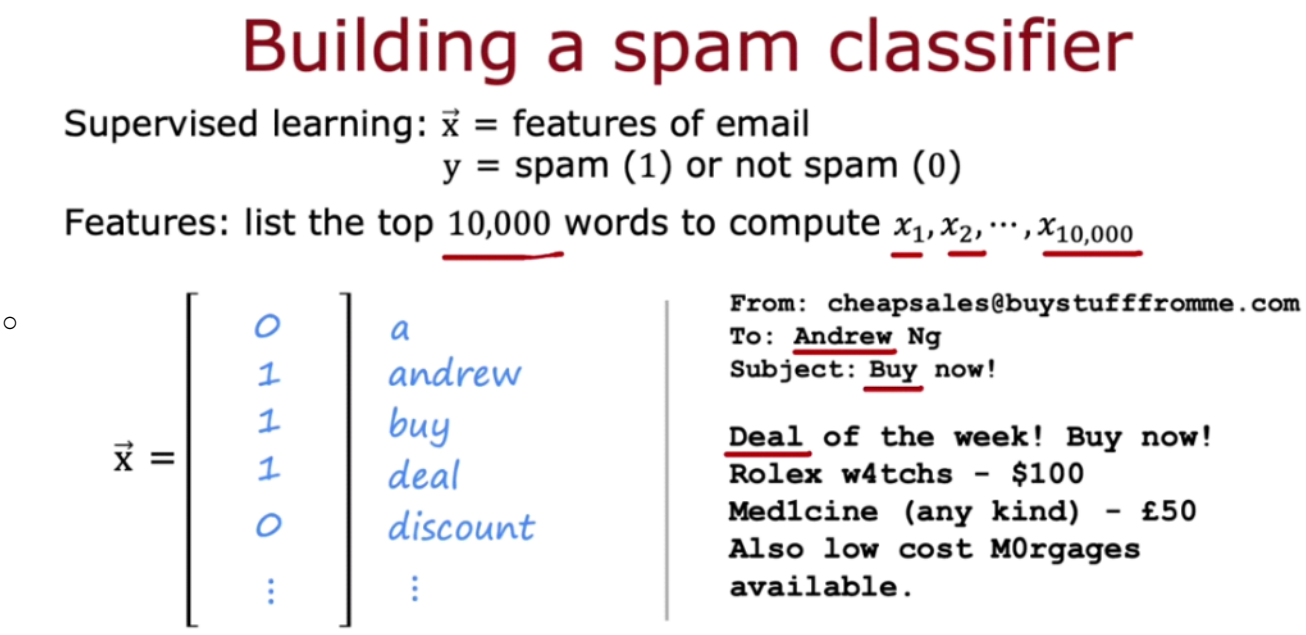


Example: Building an Email Spam Classifier

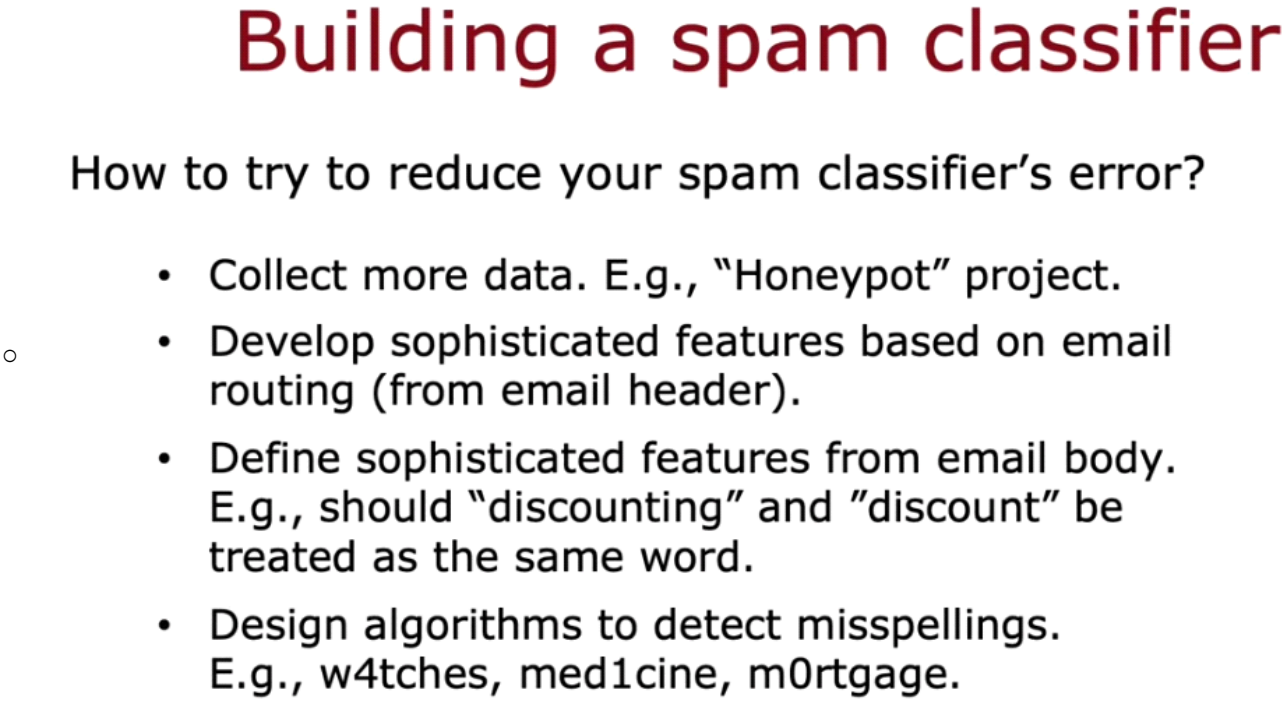
1. **Problem Definition**:
   - Classify emails as spam or non-spam.
   - Input features (x) are the characteristics of an email.

- Output label (y) is 1 for spam and 0 for non-spam.

2. **Feature Construction**:
   - Use top 10,000 words in the English language as features.
   - Features can be binary (0 or 1) indicating the presence of a word.
   - Alternatively, features can be counts of word occurrences.



3. **Model Training**:
   - Train a supervised learning algorithm (e.g., logistic regression, neural network) on the features.
   - Evaluate initial model performance.
4. **Improvement Ideas**:
   - Collect more data (e.g., using honeypot projects to gather spam emails).
   - Develop sophisticated features from email routing (email headers and server paths).
   - Create advanced features from email body text (handle synonyms, detect misspellings).
   - Implement algorithms for deliberate misspellings (e.g., "watches" instead of "watches").



5. **Choosing Promising Improvements**:
   - Diagnostics help determine whether to focus on reducing bias or variance.
   - If high bias: focus on model complexity, features.
   - If high variance: focus on collecting more data.

- Efficiently choosing the right path can significantly speed up project progress.

Key Concepts

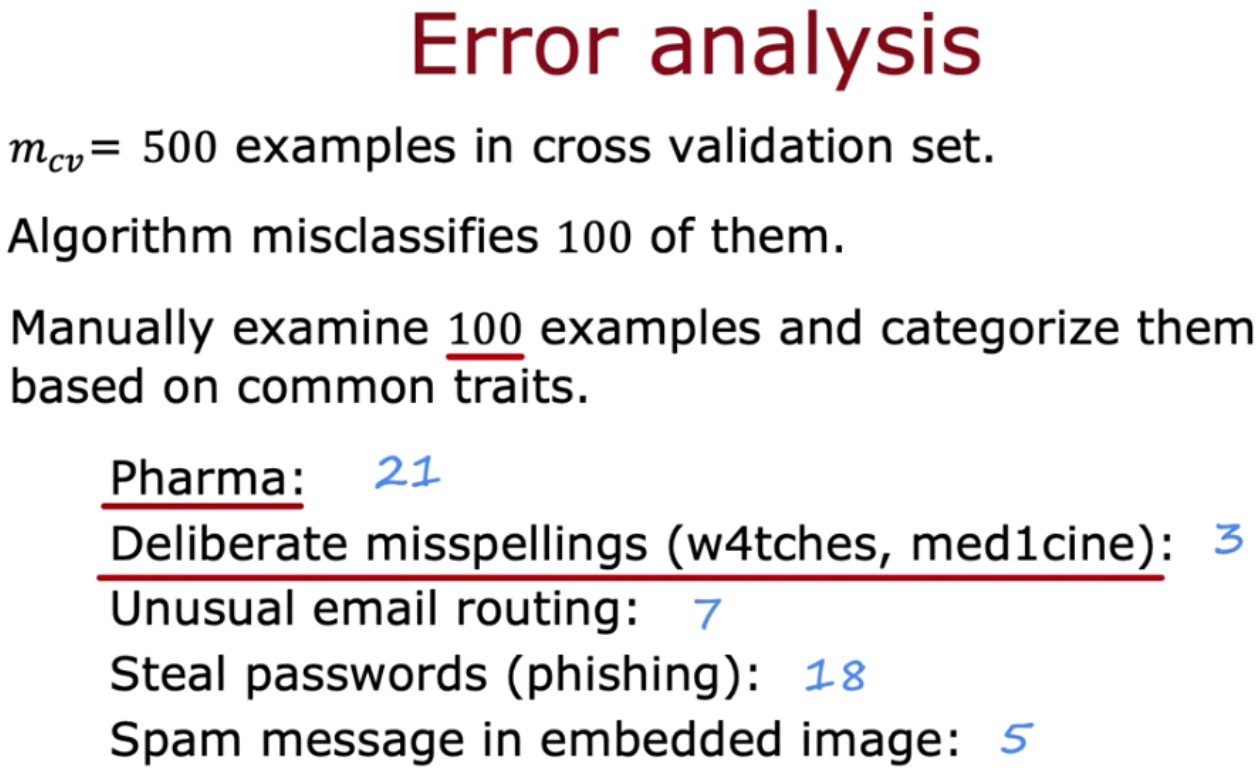
- **Diagnostics**: Tools to evaluate model performance and guide improvements.
- **Bias and Variance**:
  - High Bias: Underfitting, focus on more complex models or better features.
  - High Variance: Overfitting, focus on more data or regularization.
- **Error Analysis**: Process to gain insight into model errors and potential improvements.

Error Analysis

30 May 2024 17:00

Error Analysis Process

1. **Identify Misclassifications**:
   - Example: Out of 500 cross-validation examples, the algorithm misclassifies 100.
2. **Manual Inspection**:
   - Manually review the 100 misclassified examples.
   - Group them by common themes or properties.
3. **Categorization**:
   - Example categories:
     - Pharmaceutical spam: 21 emails.
     - Deliberate misspellings: 3 emails.
     - Unusual email routing: 7 emails.
     - Phishing emails: 18 emails.
     - Embedded image spam: various.
4. **Prioritization**:
   - Focus on categories with the highest frequency of misclassifications.
   - Example: **Prioritize pharmaceutical spam and phishing emails over deliberate misspellings.**
5. **Insight and Action**:
   - Error analysis reveals the most common error types, guiding where to focus efforts.
   - Example: Collect more pharmaceutical spam data, develop new features for phishing detection.



Practical Tips

- **Non-Mutually Exclusive Categories**: Emails can belong to multiple categories.

- **Sampling for Larger Sets**:
    - If too many misclassifications (e.g., 1,000 out of 5,000 examples), sample around 100-200 examples for manageable analysis.
    - Provides enough statistics to understand common error types.

Examples and Inspiration

- **Pharmaceutical Spam**:
    - Collect more specific data.
    - Create features related to drug names.
- **Phishing Emails**:
    - Analyze URLs for suspicious links.
    - Gather more phishing data.

Limitations

- **Human Difficulty**:
    - Error analysis is easier for tasks humans excel at (e.g., identifying spam).
    - Harder for tasks even humans struggle with (e.g., predicting ad clicks).

Tips for Adding Data to Improve Machine Learning Applications

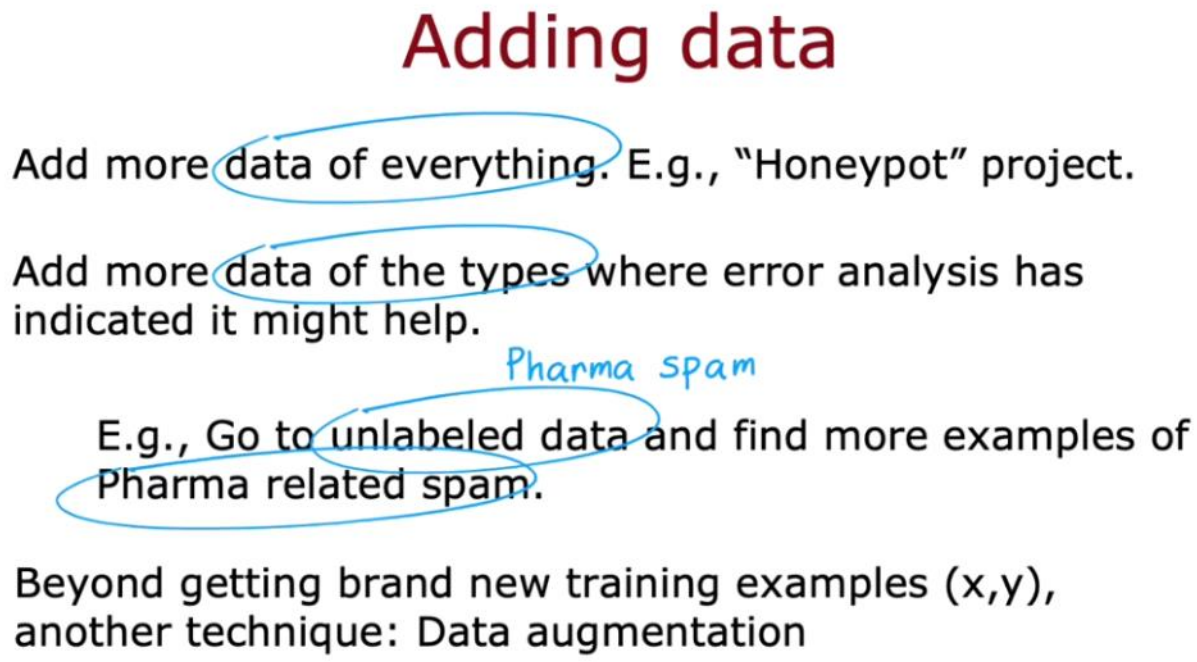
30 May 2024 19:31

Overview

Adding more data to your machine learning application can significantly boost performance. This process can involve collecting, creating, or augmenting data. Different techniques may apply to different applications, and this discussion will cover some of the most effective methods.

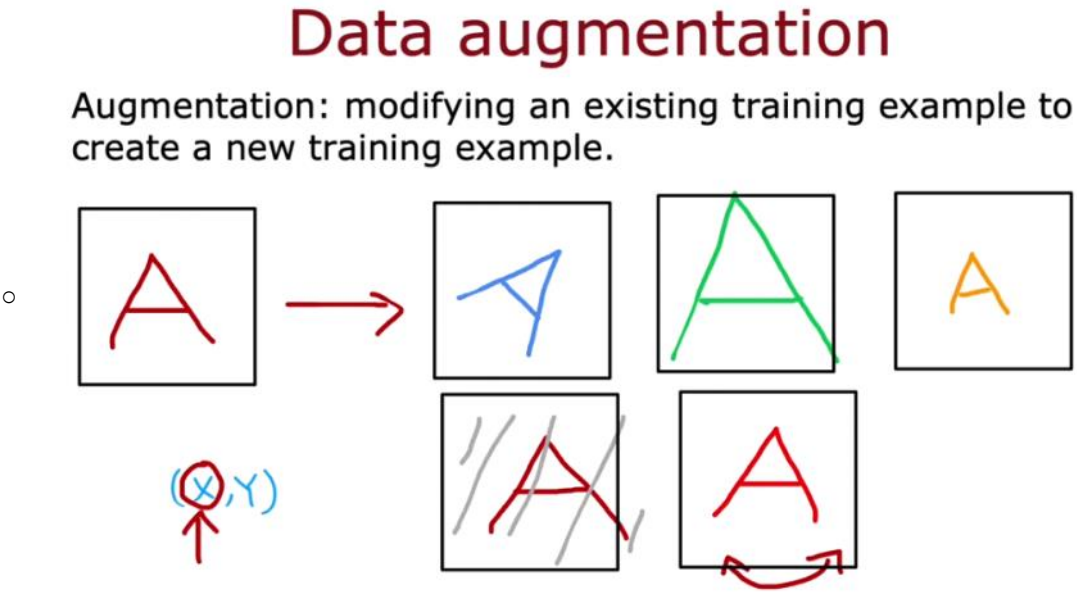
Targeted Data Collection

1. **Focus on Problematic Areas**:
   - Use error analysis to identify specific types of data where the algorithm performs poorly.
   - Collect more examples of these specific types rather than a broad range of data.
2. **Example**:
   - If your spam filter struggles with pharmaceutical spam, focus on collecting more pharmaceutical spam emails.
   - This targeted approach is cost-effective and improves algorithm performance more efficiently than collecting a wide range of data.

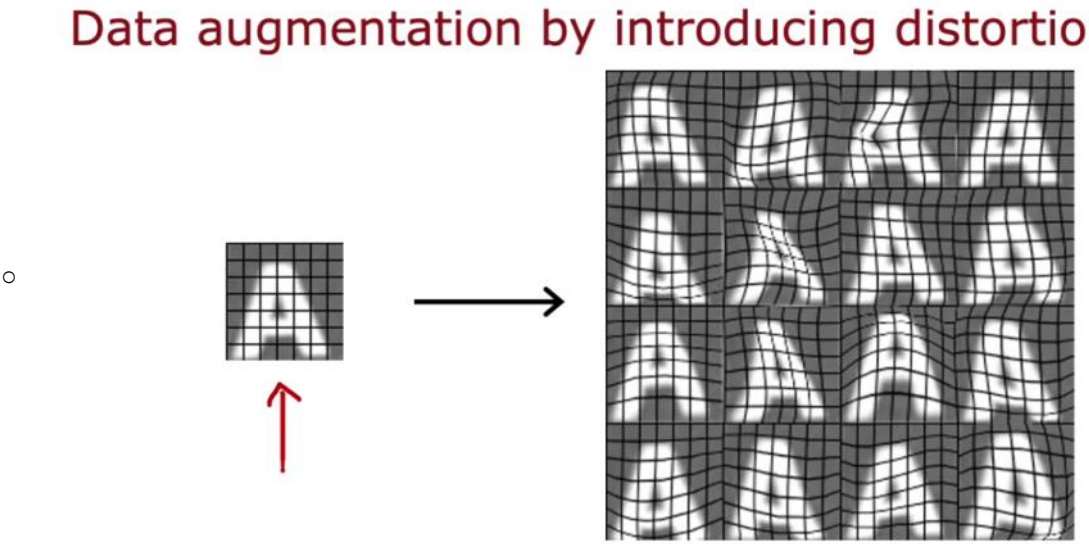


Data Augmentation

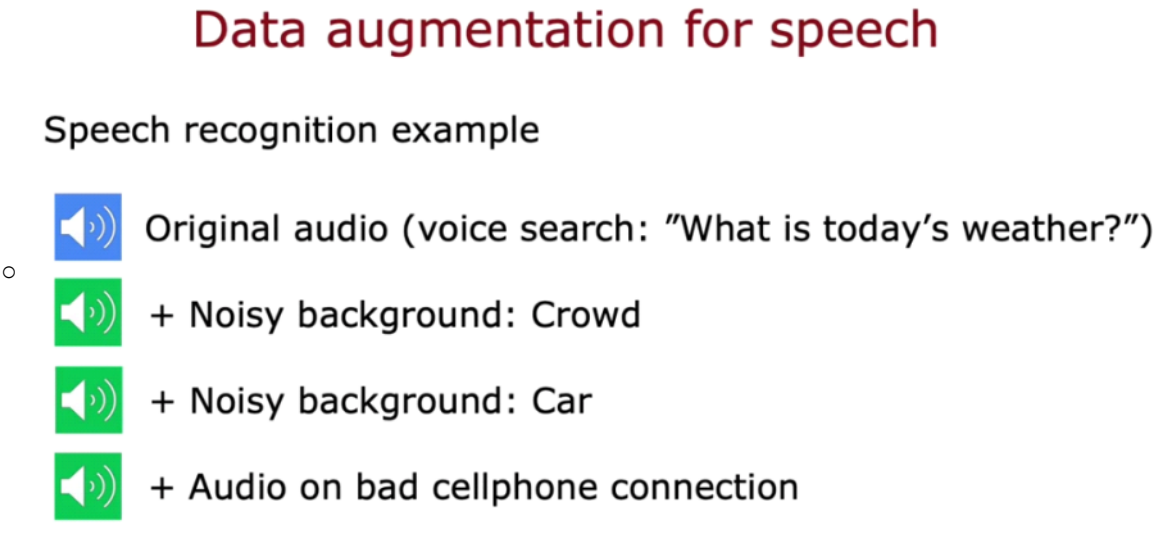
1. **Creating New Training Examples**:
   - Augment existing data by making slight modifications to create new examples.
   - Commonly used in image and audio data.
2. **Image Data Augmentation**:
   - Techniques: Rotating, resizing, changing contrast, or mirroring images.
   - Example: For OCR (Optical Character Recognition), augment an image of the letter 'A' by rotating or changing its size.



- Advanced Example: Apply random warping using a grid overlay to create diverse examples of the letter 'A'.



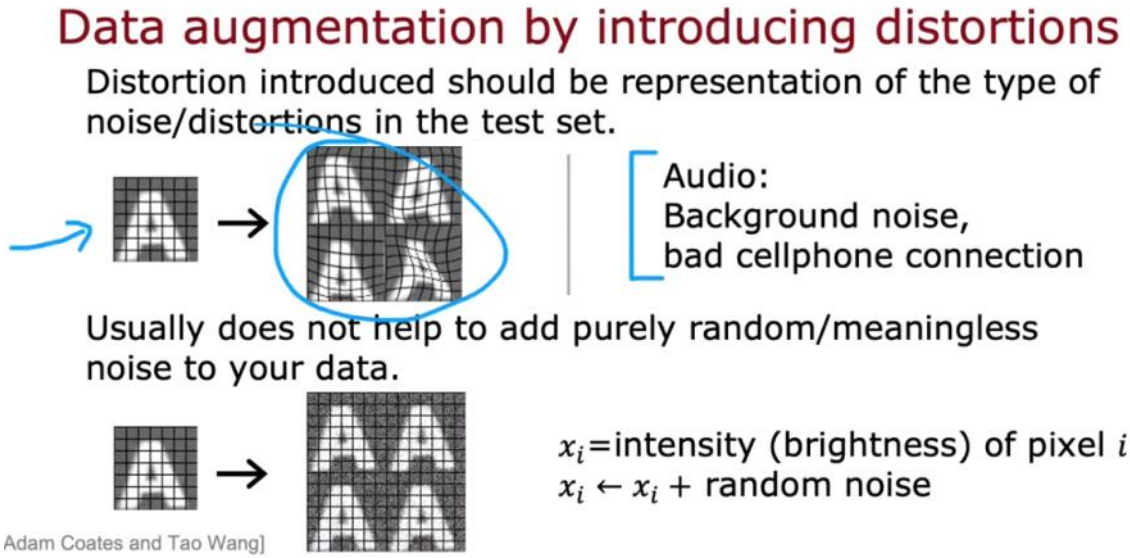
3. **Audio Data Augmentation**:
   - Combine original audio with different background noises to create new training examples.
   - Example: Add crowd noise or car noise to an audio clip of someone speaking to simulate different environments.
   - This technique is crucial for improving speech recognition systems by simulating various real-world scenarios.



4. **Guideline**:
   - Ensure the augmentations reflect realistic variations found in the test set.

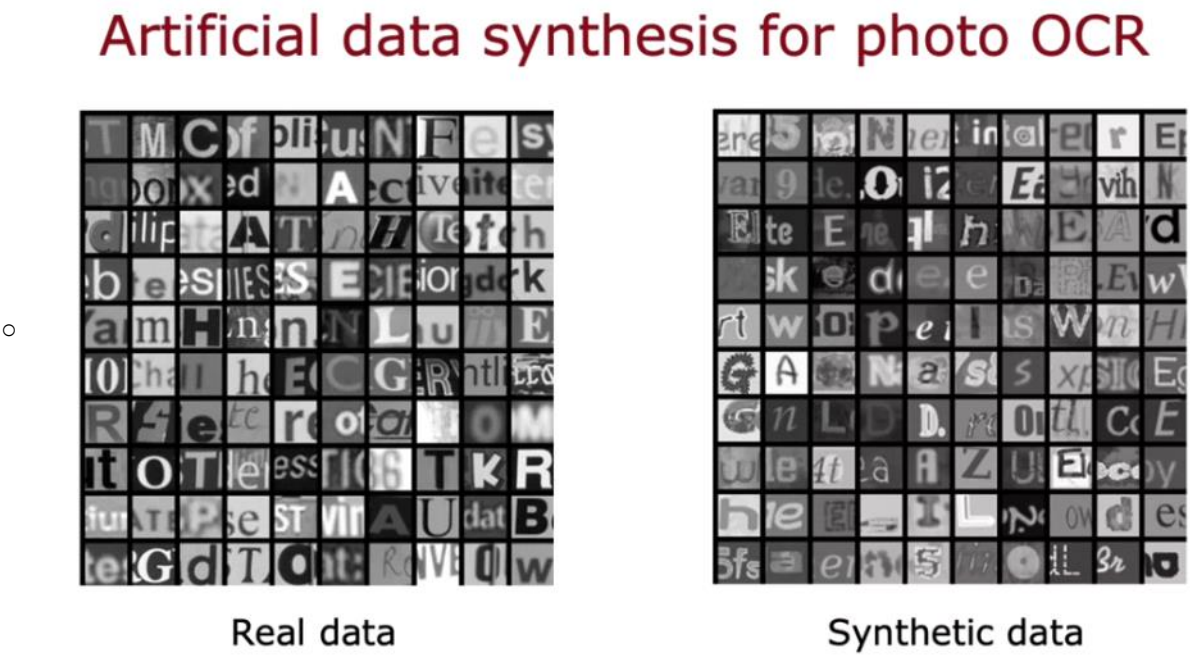
- Avoid purely random noise that doesn't represent real-world conditions.

Data Synthesis

1. **Creating Synthetic Data**:
   - Generate new training examples from scratch, not by modifying existing ones.
   - Most effective in computer vision tasks.
   - 

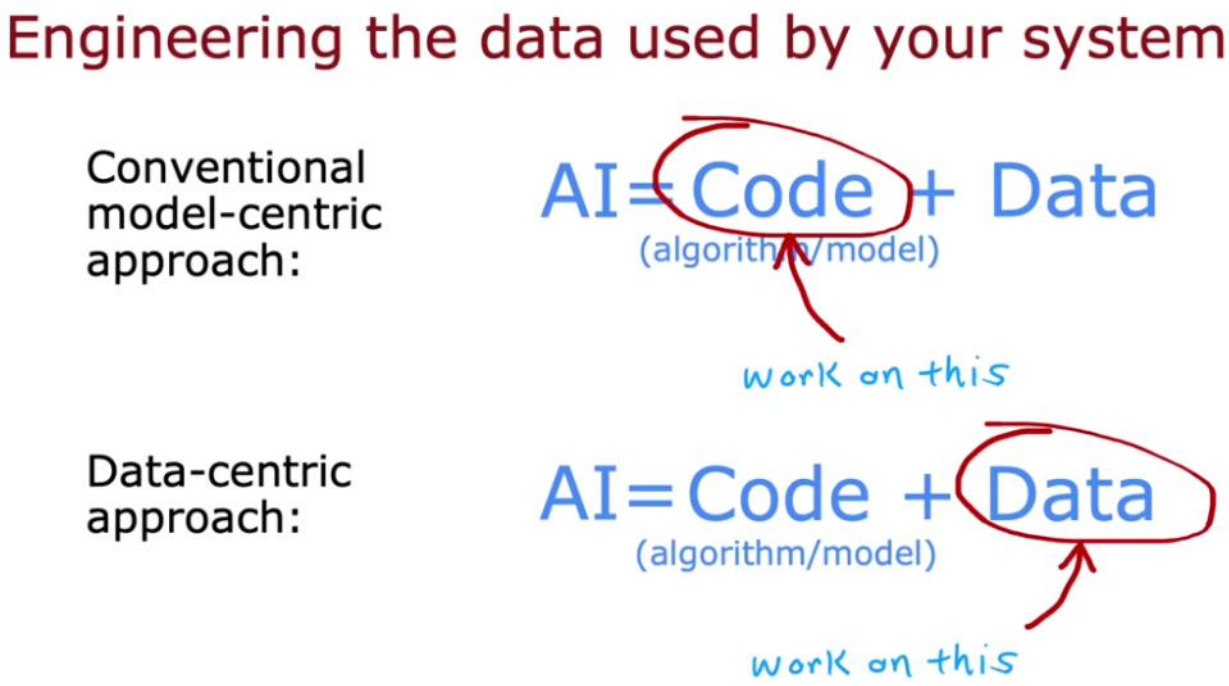
2. **Example: Photo OCR**:
   - Use different fonts and colors in a text editor to create synthetic images of text.
   - 

   - These images simulate real-world text data and significantly enhance training data volume.
   - 



3. **Effort and Reward**:
    - Writing code for synthetic data generation can be labor-intensive.
    - The resulting large dataset can provide a substantial performance boost to the algorithm.

Data-Centric Approach



1. **Shift from Model-Centric to Data-Centric**:
    - Traditional machine learning research focuses on improving the model while keeping the dataset fixed.
    - Modern approach emphasizes engineering and augmenting the data to enhance model performance.
2. **Benefits**:
    - With robust existing algorithms (e.g., linear regression, neural networks), enhancing the dataset can be more fruitful.
    - Techniques like targeted data collection, data augmentation, and data synthesis can efficiently improve algorithm performance.

Transfer Learning

1. **When Data is Scarce**:
    - If obtaining more data is challenging, transfer learning can be a powerful alternative.
    - Leverage data from a different but related task to improve your model.
2. **Next Steps**:
    - The following video will explore how transfer learning works and its application to various tasks.

Transfer Learning

31 May 2024 17:33

1. Overview of Transfer Learning:

- **Purpose:** Used when you don't have much labeled data for your specific task.
- **Technique:** Utilizes data from a different task to help improve performance on your task.

2. How Transfer Learning Works:

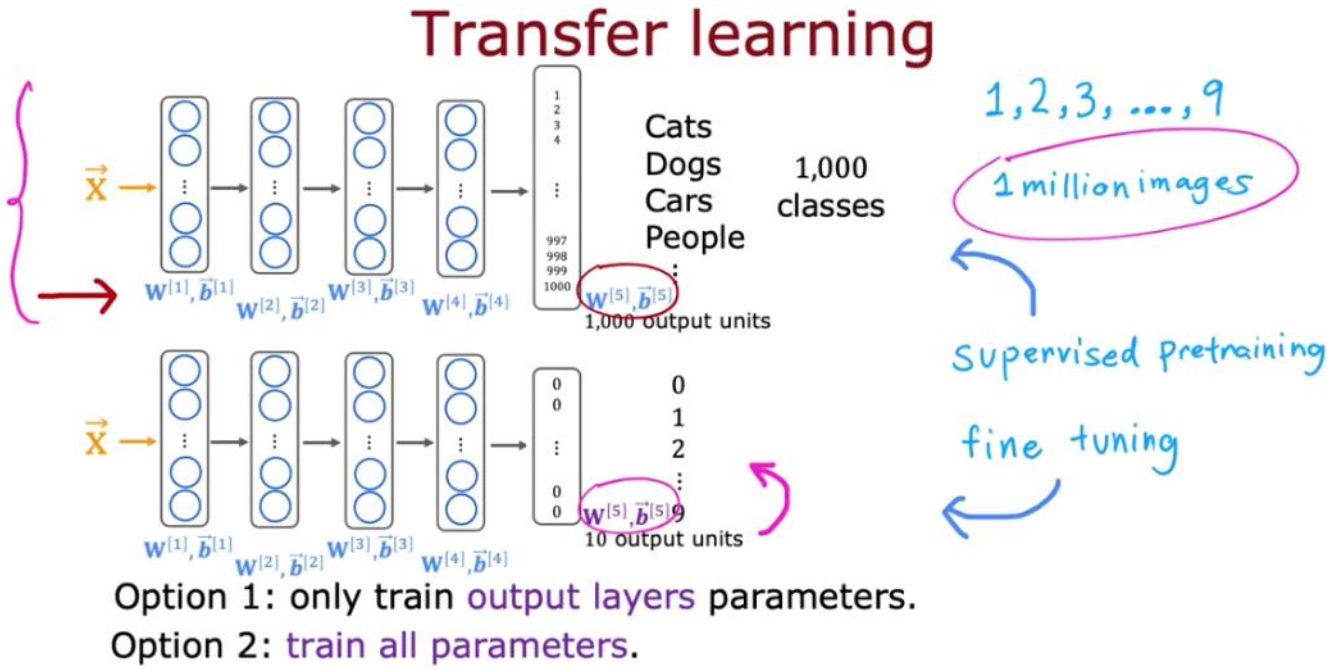
- **Scenario:** Recognizing handwritten digits (0-9) with limited labeled data.
- **Step 1:** Train a neural network on a large dataset (e.g., 1 million images of various objects like cats, dogs, cars, etc.) with many classes (e.g., 1,000 classes).
- **Step 2:** Learn parameters for each layer of the network during this training.

3. Applying Transfer Learning:

- **Layer Parameters:**
  - Keep parameters of the first few layers.
  - Replace the final layer with a new output layer corresponding to your specific classes (e.g., 10 classes for digits 0-9).
- **Training New Parameters:**
  - Train new parameters for the final layer from scratch.

4. Training Options in Transfer Learning:

- **Option 1:**
  - Only train the new output layer.
  - Keep the initial layers' parameters fixed.
- **Option 2:**
  - Train all parameters, but initialize the initial layers with pre-trained values.
- **Choosing an Option:**
  - **Option 1:** Better for very small datasets.
  - **Option 2:** Better for slightly larger datasets.

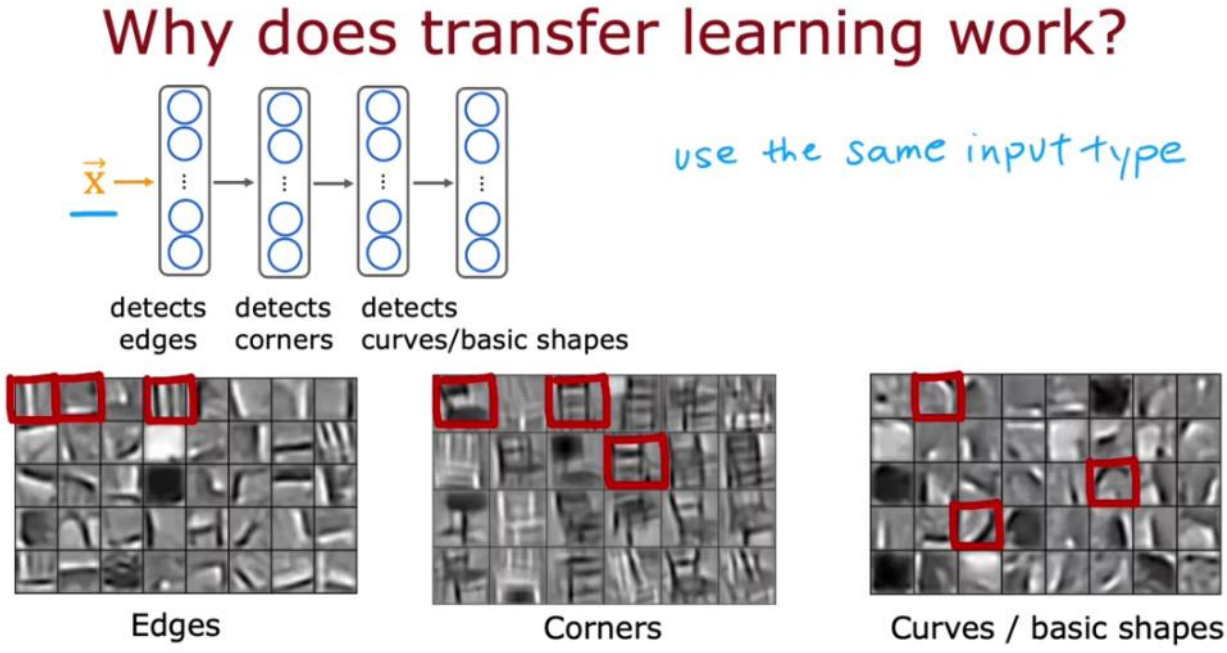


5. Supervised Pre-training and Fine-tuning:

- **Supervised Pre-training:** Train on a large, somewhat related dataset.
- **Fine-tuning:** Further train the network on your specific, smaller dataset.

6. Why Transfer Learning Works:

- **Generic Features:** Initial layers learn to detect generic features like edges, corners, and shapes, useful for many tasks.
- **Same Input Type:** Pre-training and fine-tuning must be on the same type of input (e.g., images for both).

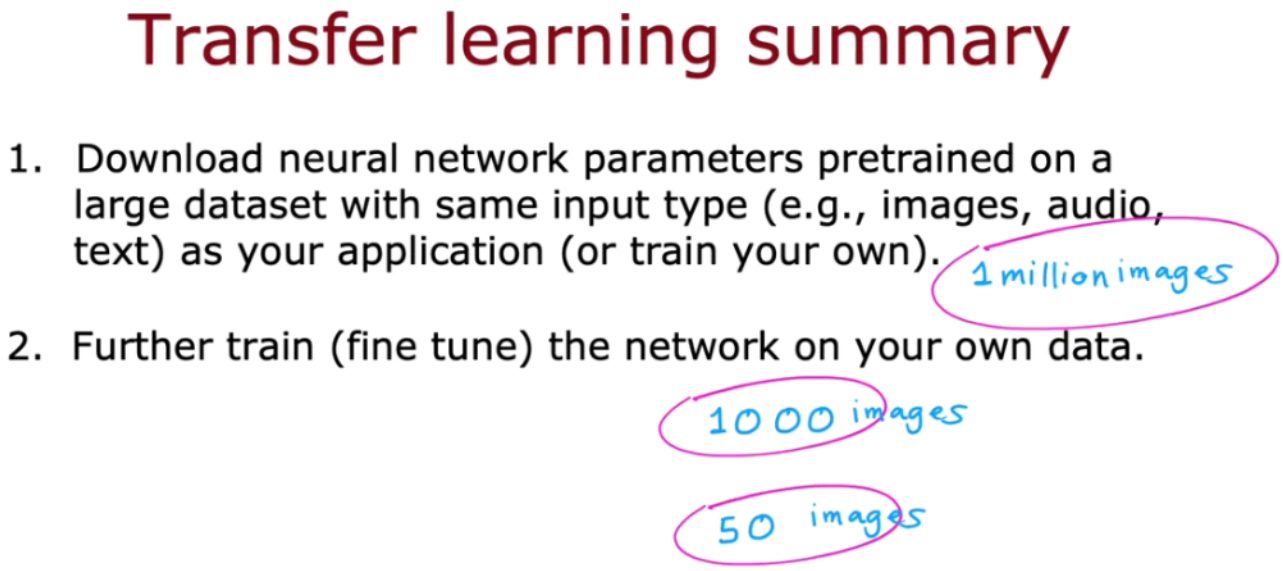


7. Practical Steps for Transfer Learning:

- **Step 1:** Download a pre-trained neural network with parameters trained on a large dataset of the same input type as your application.
- **Step 2:** Fine-tune the network on your smaller dataset.

8. Advantages of Transfer Learning:

- **Efficiency:** Can achieve good performance with a much smaller dataset for the specific task.
- **Community Contribution:** Benefit from pre-trained models available online.
- **Use Cases:** Often used in advanced techniques (e.g., GPT-3, BERT, ImageNet models).



Full cycle of machine learning project

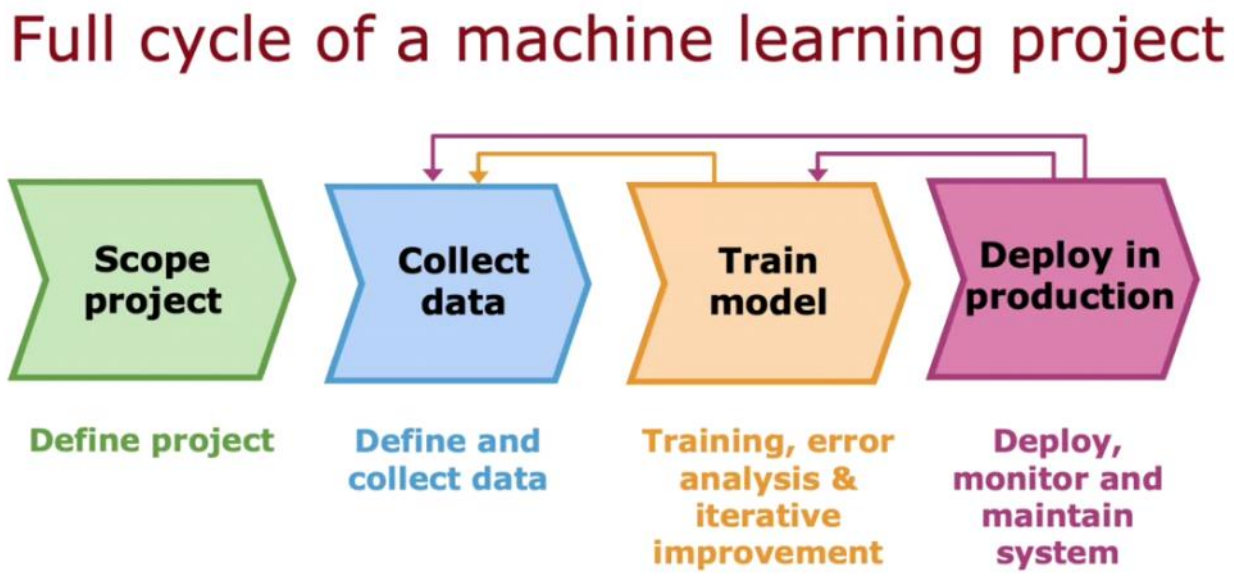
31 May 2024 20:51

1. Introduction:

- Training a model and collecting data are crucial, but only parts of the process.
- Understanding the full cycle of a machine learning project is essential.

2. Steps in a Machine Learning Project:

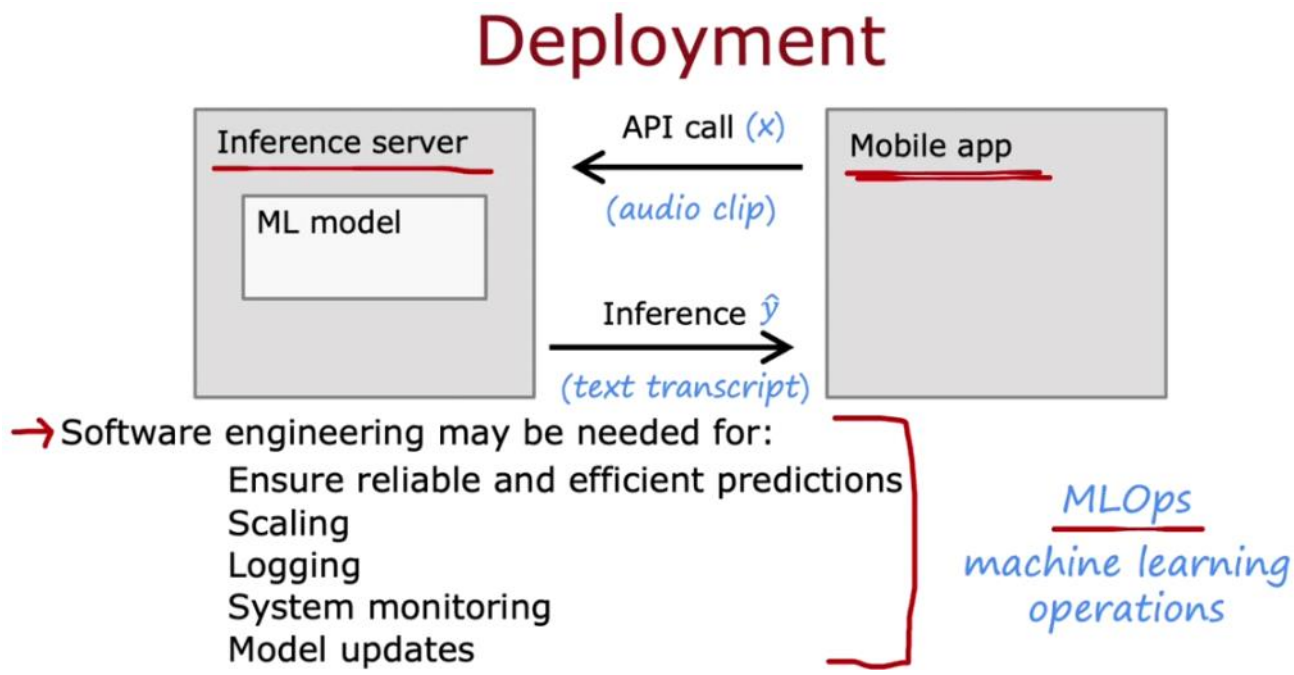
- **Project Scoping:**
  - Define the project and its goals.
  - Example: Developing speech recognition for voice search.
- **Data Collection:**
  - Determine and gather the necessary data (e.g., audio recordings and transcripts).
- **Training the Model:**
  - Train the machine learning model with the collected data.
  - Perform error analysis and iteratively improve the model.
- **Deploy in production**
  - Make it available for users to use.
  - When you deploy a system you have to also make sure that you continue to monitor the performance of the system and to maintain the system in case the performance gets worse to bring its performance back up instead of just hosting your machine learning model on a server.



3. Iterative Improvement:

- Error analysis and bias-variance analysis may indicate the need for more data.
- Collect additional data based on specific needs (e.g., noisy environments like car noise).
- Repeat the cycle of training, error analysis, and data collection until satisfactory performance is achieved.

4. Deployment and Monitoring:



- **Deploying the Model:**
  - Make the model available for users, often by implementing it on an inference server.
  - Example: A mobile app makes API calls to the inference server to get predictions (e.g., transcriptions).
- **Monitoring and Maintenance:**
  - Continuously monitor the system's performance and maintain it to ensure consistent accuracy.
  - Update the model as needed based on new data or performance issues.

5. Detailed Steps for Deployment:

- **Inference Server:**
  - Host the trained model to make predictions based on input data.
- **API Calls:**
  - The application sends input data to the inference server, which returns predictions.
- **Software Engineering:**
  - Implement necessary code to handle predictions, scaling, logging, and monitoring.
  - Ensure efficient and reliable predictions, particularly for large-scale applications.

6. Logging and System Monitoring:

- Log inputs and predictions (x and ŷ), with user consent and privacy considerations.
- Use logged data to identify performance issues and retrain the model as needed.

7. MLOps (Machine Learning Operations):

- Practice of systematically building, deploying, and maintaining machine learning systems.
- Ensure the model is reliable, scalable, monitored, and updateable.
- Optimize implementations for cost-effective and efficient large-scale deployment.

8. Ethical Considerations:

- Importance of ethics in building machine learning systems.
- Next video will cover the ethics of machine learning development.

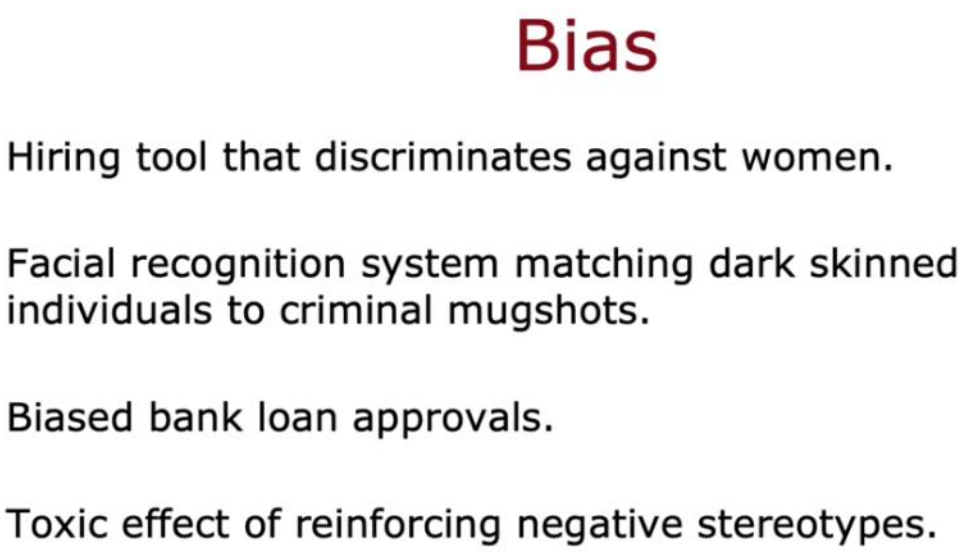
Fairness, Bias, and Ethics in Machine Learning

31 May 2024 22:16

1. Importance of Ethics in Machine Learning:

- Machine learning algorithms impact billions of people.
- Ensuring fairness, freedom from bias, and ethical integrity is crucial.
- Systems should be reasonably fair and free from bias.

2. Examples of Bias and Ethical Issues:



- **Hiring Tools:** Discriminated against women.
- **Face Recognition:** Higher error rates for dark-skinned individuals.
- **Loan Approvals:** Biased against certain subgroups.
- **Reinforcing Stereotypes:** Discouraging underrepresented groups from certain professions.
- **Adverse Use Cases:**
  - Deepfakes used unethically.
  - Social media spreading toxic speech.
  - Bots generating fake content.
  - Machine learning used for harmful products or fraud.

3. Addressing Ethical Concerns:

Adverse use cases

Deepfakes

Spreading toxic/incendiary speech through optimizing for engagement.

Generating fake content for commercial or political purposes.

Using ML to build harmful products, commit fraud etc.

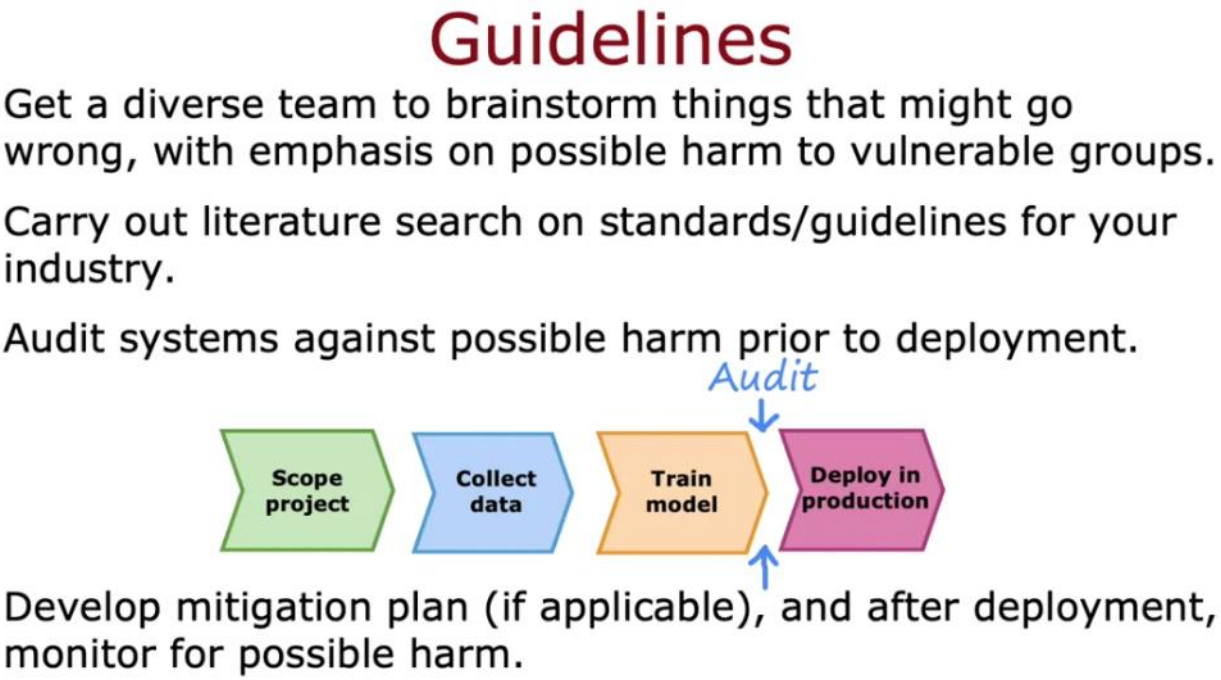
Spam vs anti-spam : fraud vs anti-fraud.

- **Avoiding Negative Impact:**
  - Do not build systems that harm society.
  - Walk away from unethical projects.

- **Ethical Decision-Making:**
  - Ethics is a complex field with no simple checklists.
  - General guidance and suggestions can help.

4. Suggestions for Ethical Machine Learning:

- **Assemble a Diverse Team:**
  - Diversity in gender, ethnicity, culture, and other traits.
  - Brainstorm potential harms, especially to vulnerable groups.
- **Literature Search:**
  - Research standards and guidelines for your industry.
  - Financial industry example: Standards for loan approval fairness.
- **Audit the System:**
  - Identify possible problems and audit the system before deployment.
  - Measure performance for bias against subgroups.
- **Develop a Mitigation Plan:**
  - Plan for addressing issues post-deployment.
  - Example: Self-driving cars having accident mitigation plans.
- **Monitor and Maintain:**
  - Continue monitoring after deployment.
  - Act quickly to address any issues that arise.



5. The Importance of Addressing Bias:

- Bias and fairness issues should be taken seriously.
- Ethical implications vary by project type.
- Aim to avoid past mistakes in the machine learning field.

6. Additional Learning:

- Optional videos on handling skewed datasets.
- Techniques for addressing datasets with an imbalanced ratio of positive to negative examples.

Decision Trees

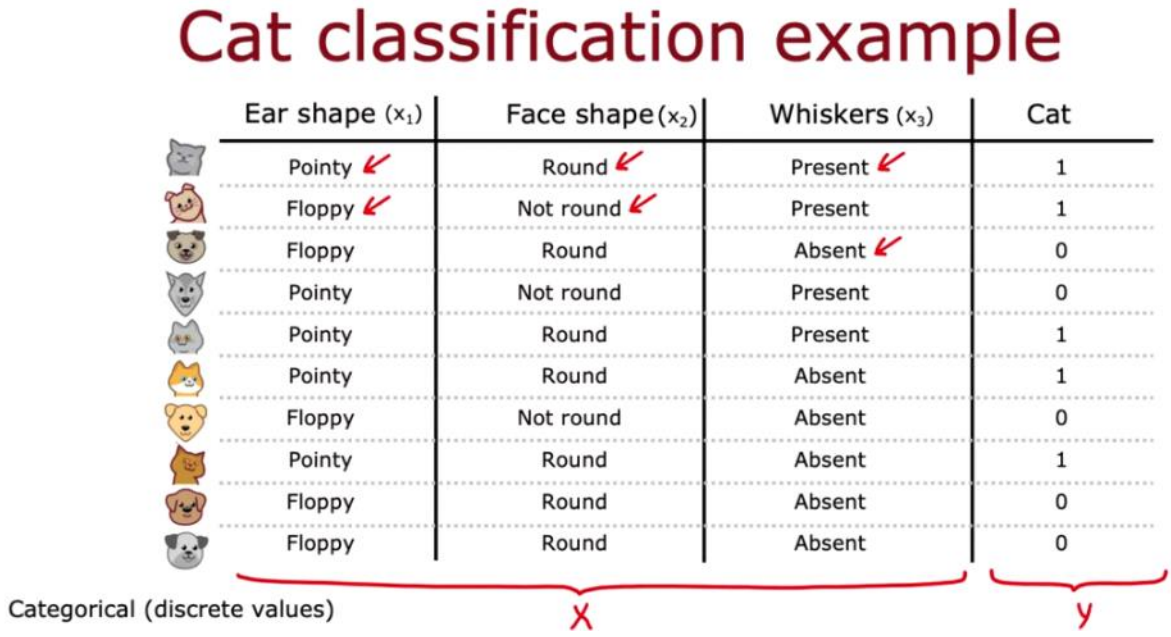
01 June 2024 11:33

Introduction to Decision Trees and Tree Ensembles:

- Decision trees are powerful learning algorithms widely used in many applications.
- They are often used in machine learning competitions.
- Despite their success, decision trees haven't received much attention in academia.
- This week, we'll learn about decision trees and how to get them to work for you.

Running Example: Cat Classification:

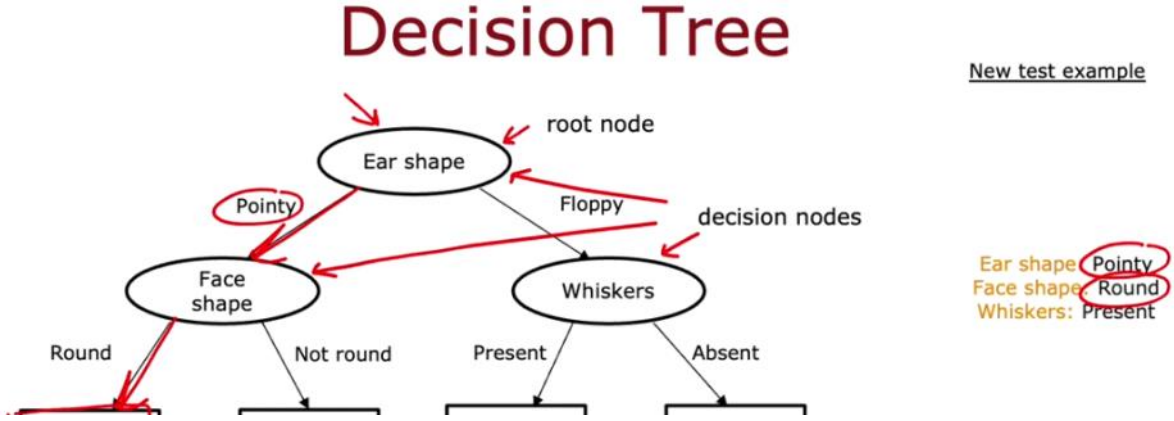
- Scenario: Running a cat adoption center and training a classifier to identify cats.
- Features: Ear shape, face shape, whiskers.
- Labels: Whether the animal is a cat or not.
- Dataset: 10 examples (5 cats, 5 dogs).

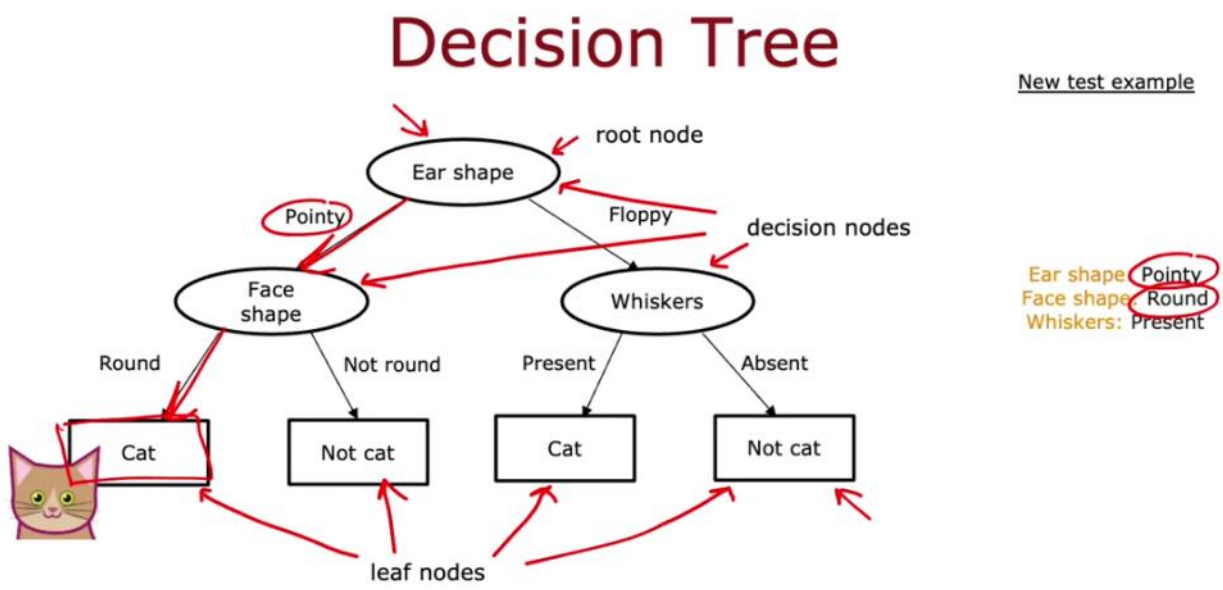


Features and Labels:

- Input features (X): Ear shape, face shape, whiskers.
- Target output (Y): Is this a cat or not? (binary classification)
- Features take on categorical values: pointy/floppy ears, round/not round face, whiskers present/absent.

What is a Decision Tree?





- A decision tree model looks like a tree (in computer science terms).
- Nodes:
  - Root node: The top-most node.
  - Decision nodes: Ovals that look at a feature and decide to go left or right.
  - Leaf nodes: Rectangular boxes that make a prediction.

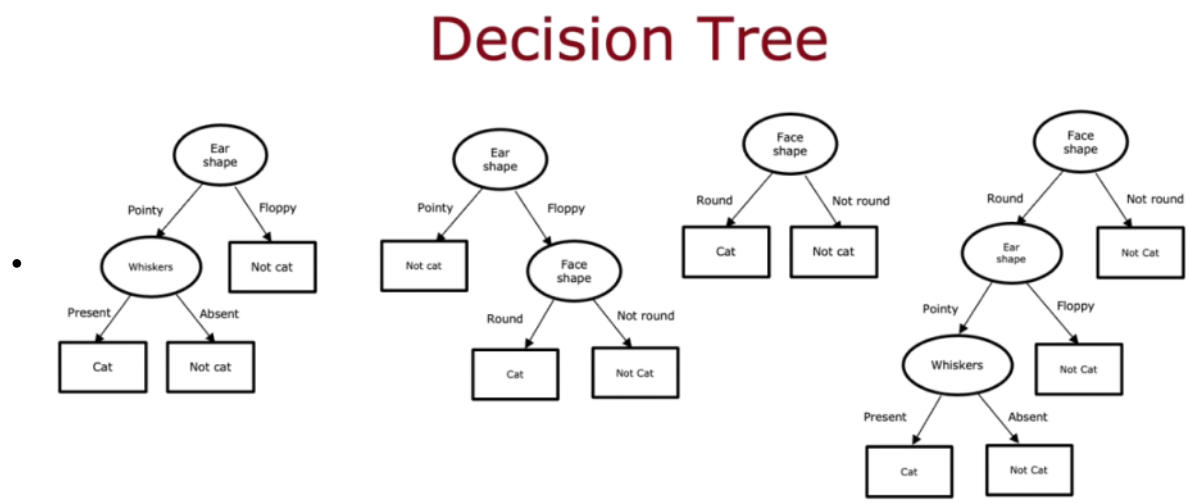
Example Decision Tree:

- To classify an example (e.g., cat with pointy ears, round face, whiskers present):
  - Start at the root node.
  - Based on ear shape, move left or right.
  - Continue based on face shape and whiskers.
  - Reach a leaf node to make the final prediction (cat or not).

Terminology:

- Root node: Top-most node of the tree.
- Decision nodes: Nodes that make decisions based on feature values.
- Leaf nodes: Nodes at the bottom that make predictions.

Multiple Decision Trees:

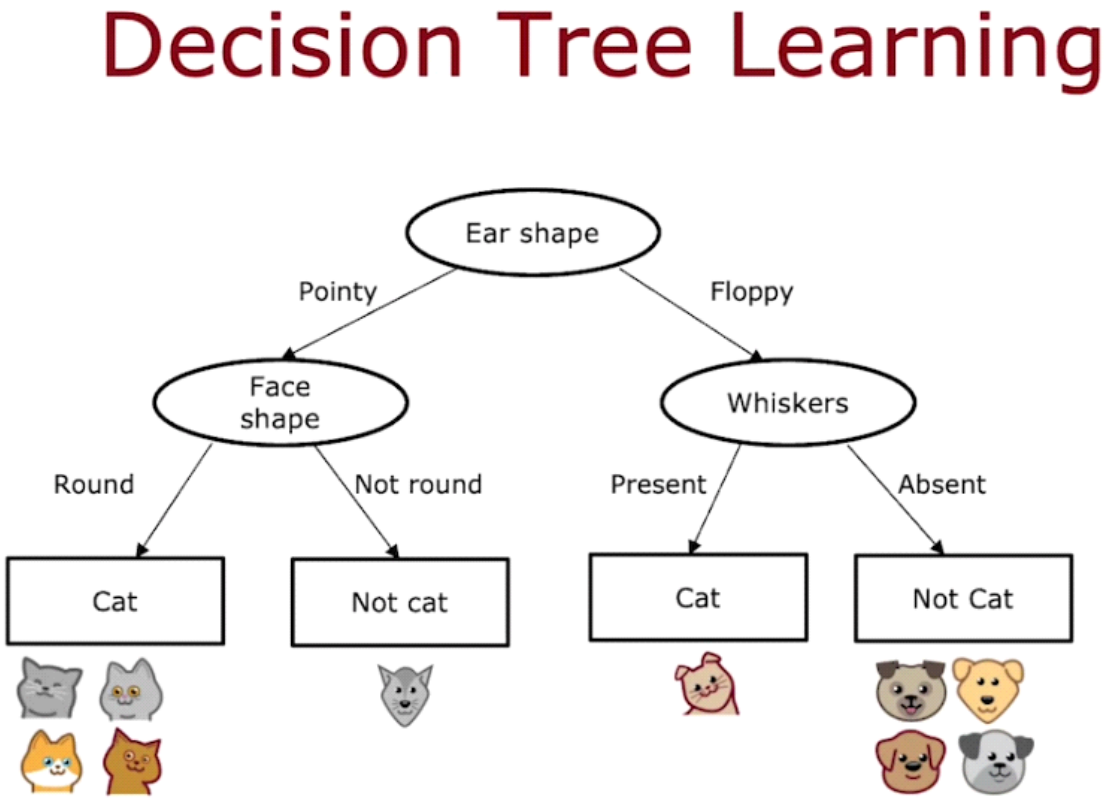
- Different decision trees can classify the same dataset.
- 

- Examples of different trees for classifying cats vs. not cats:
  - Tree 1: Starts with ear shape, then whiskers.

    - Tree 2: Starts with whiskers, then ear shape.
- The goal of the decision tree learning algorithm is to pick a tree that performs well on the training set and generalizes to new data.

Learning Process

- The learning algorithm selects a specific decision tree based on the training set.
- The chosen tree should ideally perform well on cross-validation and test sets.



Step 1: Choosing the Root Node Feature

- The first step is to decide what feature to use at the root node (the topmost node of the decision tree).
- An algorithm, which will be explained in later videos, helps determine this feature.
- For example, if we choose "ear shape" as the root node feature, we split the training examples based on ear shape:
    - Five examples with pointy ears go to the left.
    - Five examples with floppy ears go to the right.

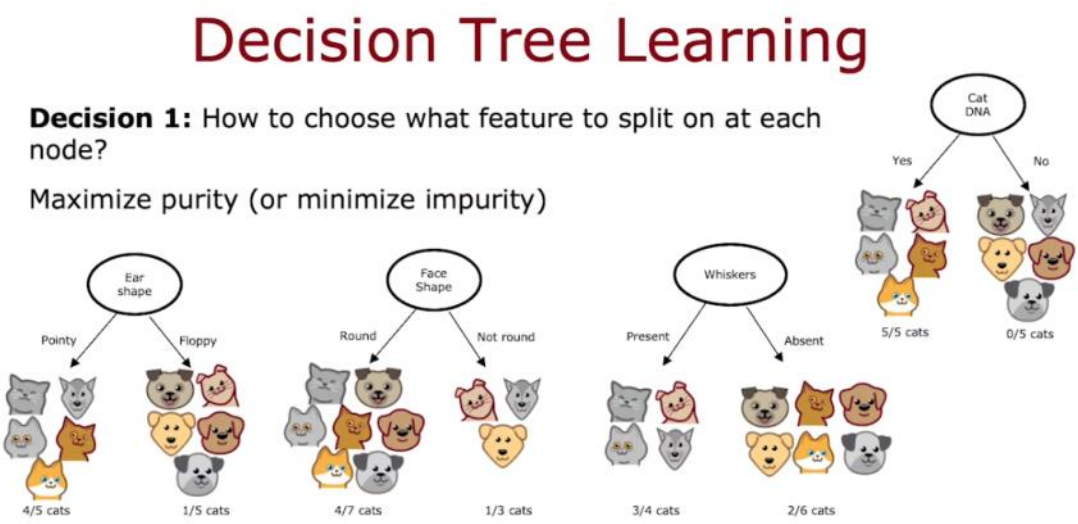
Step 2: Splitting Further Nodes

- Focus on the left branch (examples with pointy ears) and decide the next feature to split on.
- Suppose we choose "face shape" for the next split:
    - Four examples with a round face move to the left.
    - One example with a not-round face moves to the right.
- Create leaf nodes:
    - Left node (all round faces): All examples are cats, so this leaf node predicts "cat."
    - Right node (not-round face): The example is a dog, so this leaf node predicts "not cat."

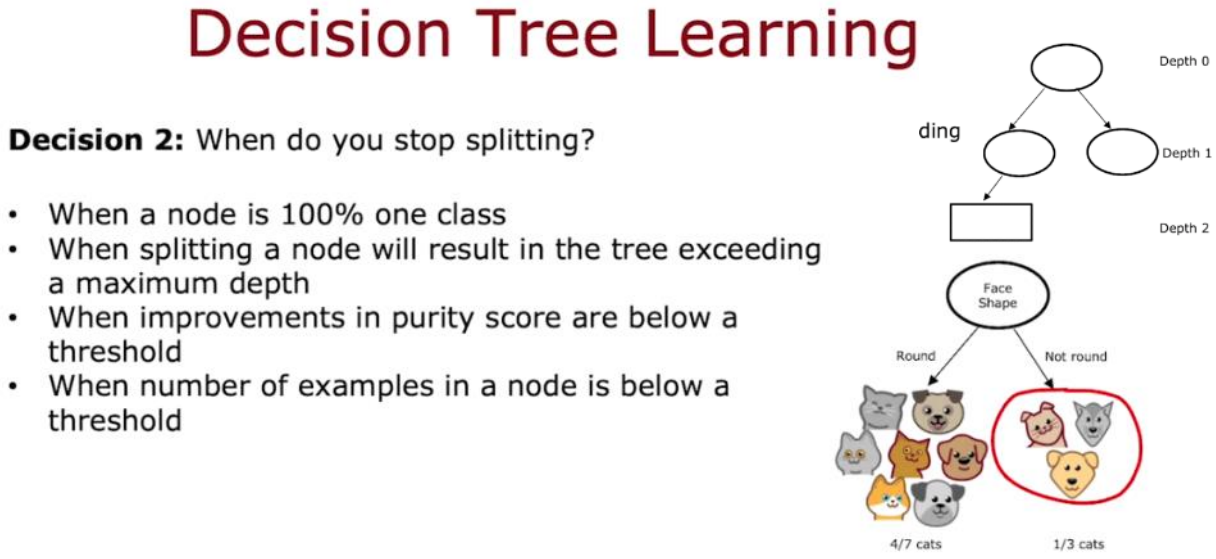
Step 3: Repeating the Process on the Right Branch

- Now focus on the right branch (examples with floppy ears).
- Suppose we choose "whiskers" as the feature to split on:
    - Split examples based on whether whiskers are present or absent.
    - Left node (whiskers present): Contains one cat, so this leaf node predicts "cat."
    - Right node (whiskers absent): Contains four dogs, so this leaf node predicts "not cat."

Key Decisions in Building a Decision Tree:



- **Choosing Features to Split On:**
  - At each node, decide which feature to split on (ear shape, face shape, or whiskers).
  - The goal is to maximize the purity of the resulting subsets (i.e., make them as homogeneous as possible).
- **When to Stop Splitting:**
  - Stop splitting when all examples in a node are of the same class.
  - Alternatively, stop splitting when the tree reaches a maximum depth or when further splitting does not significantly improve purity.
  - Other criteria include a minimum improvement threshold in purity or a minimum number of examples at a node.



Example of Feature Selection for Splitting:

- If we had a feature "cat DNA" (not available in this example), it would result in perfect splits:
  - Left branch: All cats.
  - Right branch: All not-cats.
- For available features, the decision tree algorithm evaluates:
  - Ear shape: Four out of five cats on the left.
  - Face shape: Four out of seven cats on the left.
  - Whiskers: Three out of four cats on the left.

Understanding Purity and Impurity:

- The algorithm chooses features that result in the greatest purity (least impurity).
- For example, a perfect feature would result in subsets that are either all cats or all not-cats.

Reflection on the Algorithm's Complexity:

- Decision trees may seem complex due to various refinements over time by different researchers.
- The key ideas and most important concepts will be covered in this course.
- Practical use of open-source packages will simplify the process of making decisions like when to stop splitting.

Measuring Purity

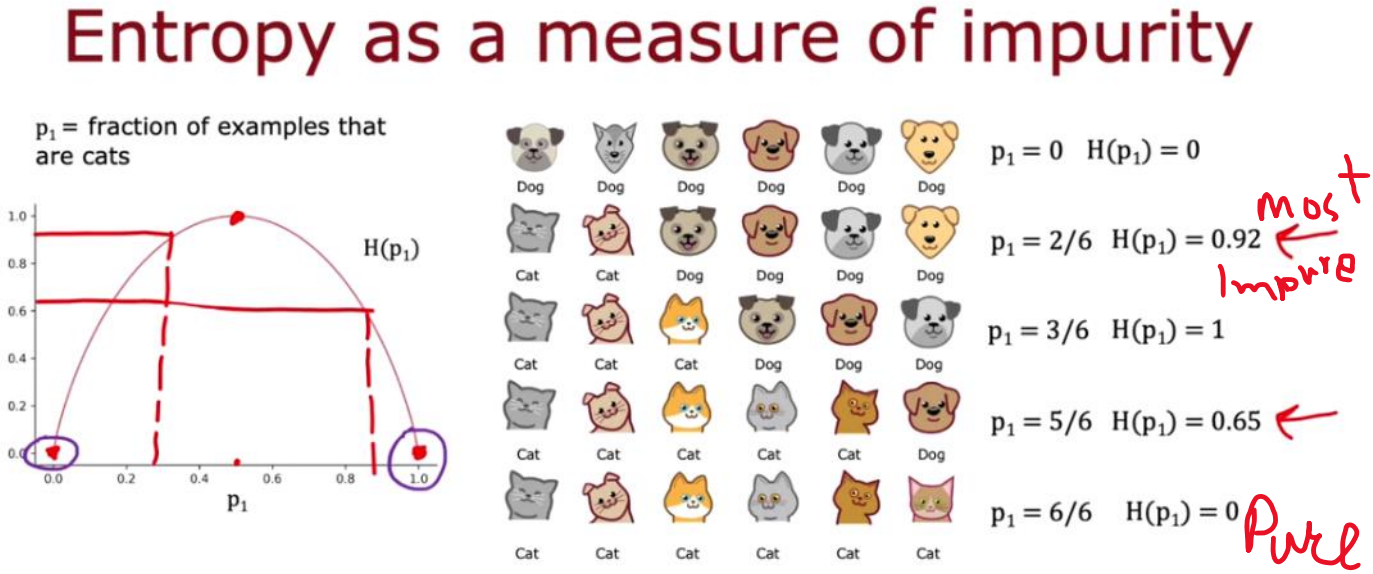
02 June 2024 23:12

Definition of Entropy:

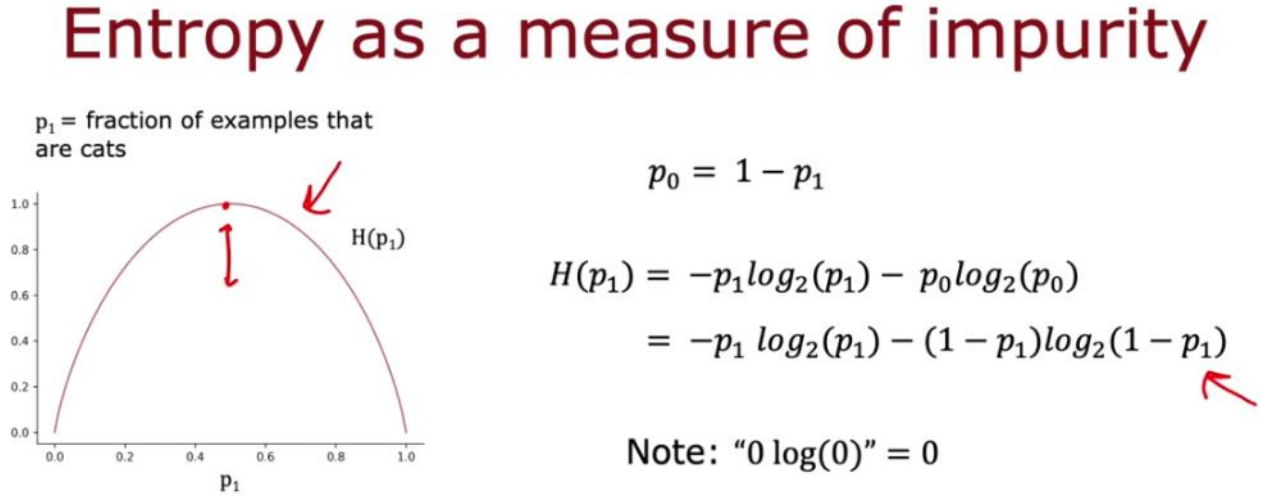
- Entropy is a measure of the impurity of a set of data.
- For example, given a set of six examples with three cats and three dogs:
  - Let p1p1 be the fraction of examples that are cats.
  - p1=36=0.5p1=63=0.5.

Entropy Function:

- The entropy function, denoted as H(p1)H(p1), measures the impurity.
- It is plotted with p1p1 (fraction of cats) on the horizontal axis and entropy on the vertical axis.
- At p1=0.5p1=0.5 (equal mix of cats and dogs), entropy is 1 (maximum impurity).
- If a set is all cats or all dogs, entropy is 0 (complete purity).



Entropy Equation:



- The entropy function is defined as:
  H(p1)=−p1log 2(p1)−p0log 2(p0)H(p1)=−p1log2(p1)−p0log2(p0)
  where p0p0 is the fraction of examples that are not cats (p0=1−p1p0=1−p1).
- This can also be written as:

$H(p1)=-p1\log\ 2(p1)-(1-p1)\log\ 2(1-p1)H(p1)=-p1\log2(p1)-(1-p1)\log2(1-p1)$

- When plotted, this function creates a curve with a peak at p1=0.5p1=0.5.

Special Cases:

- If p1p1 or p0p0 is 0, the term 0log 2(0)0log2(0) is conventionally taken as 0.
- This ensures the entropy calculation remains valid and results in zero impurity for completely pure sets.

Similar Measures:

- Other measures, like the Gini index, also assess impurity and are similar to entropy.
- For simplicity, we focus on entropy in this course, but Gini can also be used effectively in decision trees.

Choosing Features in Decision Trees with Information Gain

- In decision tree learning, we decide which feature to split on by reducing entropy the most.
- The reduction in entropy is called information gain.

Example Scenario:

- We are building a decision tree to recognize cats versus not cats.
- We have three potential features to split on: ear shape, face shape, and whiskers.

Splitting by Ear Shape:

1. **Left Subset (4/5 Cats):**
   - p1=45=0.8p1=54=0.8
   - Entropy: H(0.8)≈0.72H(0.8)≈0.72
2. **Right Subset (1/5 Cats):**
   - p1=15=0.2p1=51=0.2
   - Entropy: H(0.2)≈0.72H(0.2)≈0.72

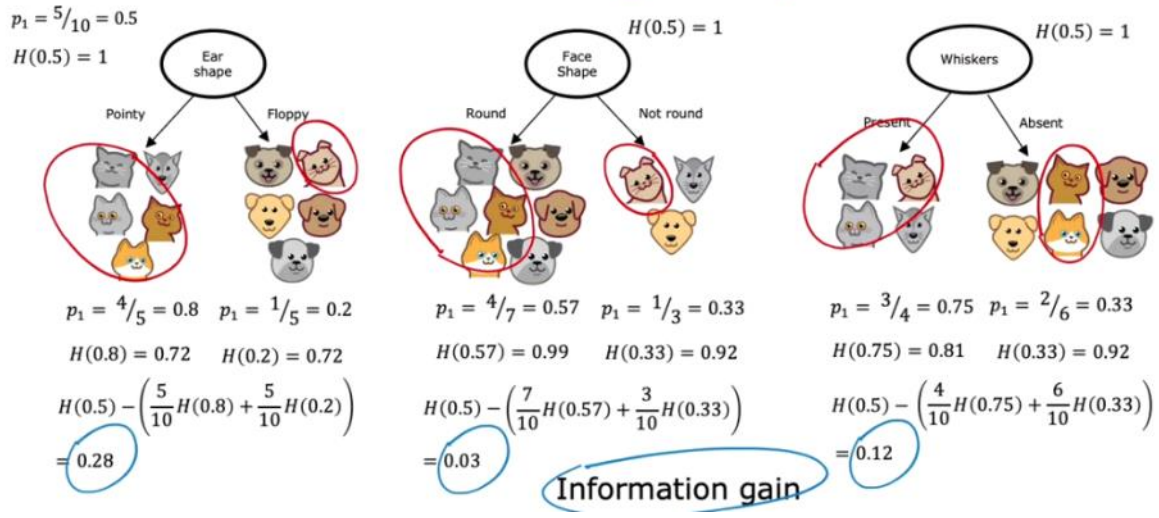
Splitting by Face Shape:

3. **Left Subset (4/7 Cats):**
   - p1=47≈0.57p1=74≈0.57
   - Entropy: H(0.57)≈0.99H(0.57)≈0.99
4. **Right Subset (1/3 Cats):**
   - p1=13≈0.33p1=31≈0.33
   - Entropy: H(0.33)≈0.92H(0.33)≈0.92

Splitting by Whiskers:

5. **Left Subset (3/4 Cats):**
   - p1=34=0.75p1=43=0.75
   - Entropy: H(0.75)≈0.81H(0.75)≈0.81
6. **Right Subset (2/6 Cats):**
   - p1=26=0.33p1=62=0.33
   - Entropy: H(0.33)≈0.92H(0.33)≈0.92

Weighted Average Entropy:



- **Ear Shape:**
  - 510×H(0.8)+510×H(0.2)=0.5×0.72+0.5×0.72=0.72105×H(0.8)+105×H(0.2)=0.5×0.72+0.5×0.72=0.72
- **Face Shape:**
  - 710×H(0.57)+310×H(0.33)=0.7×0.99+0.3×0.92=0.963107×H(0.57)+103×H(0.33)=0.7×0.99+0.3×0.92=0.963
- **Whiskers:**
  - 410×H(0.75)+610×H(0.33)=0.4×0.81+0.6×0.92=0.87104×H(0.75)+106×H(0.33)=0.4×0.81+0.6×0.92=0.87

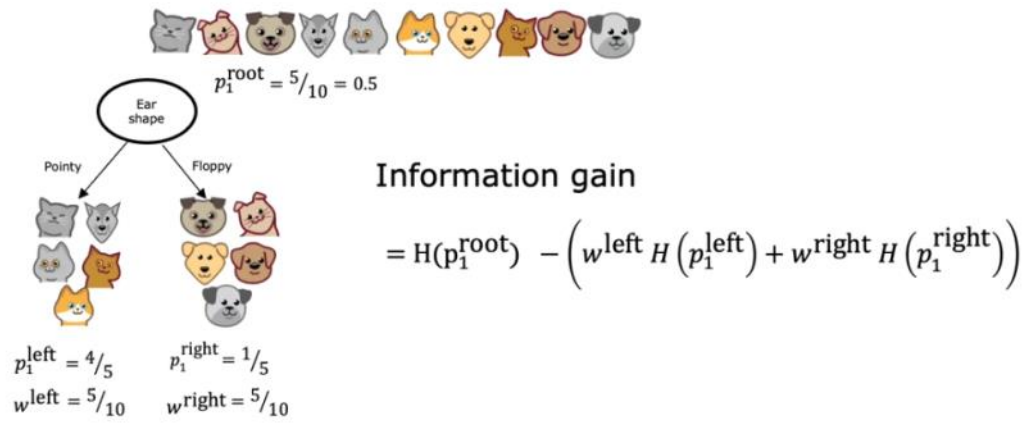
Choosing the Best Split:

- We choose the split that gives the lowest weighted average entropy.

Information Gain Calculation:

- Entropy at the root node: H(0.5)=1H(0.5)=1
- **Ear Shape Information Gain:**
  - IG=1−0.72=0.28IG=1−0.72=0.28
- **Face Shape Information Gain:**
  - IG=1−0.963=0.037IG=1−0.963=0.037
- **Whiskers Information Gain:**
  - IG=1−0.87=0.13IG=1−0.87=0.13
- The split with the highest information gain is chosen.
  Formal Definition of Information Gain:

7. **Define:**
   - p1left: fraction of positive examples in the left subset
   - p1right: fraction of positive examples in the right subset
   - wleft: fraction of examples in the left subset
   - wright: fraction of examples in the right subset
   - p1root: fraction of positive examples at the root



8. Information Gain (IG) formula:
   IG=H(p1root)−[wleft·H(p1left)+wright·H(p1right)]IG=H(p1root)−[wleft·H(p1left)+wright·H(p1right)]

Building a Decision Tree with Information Gain

03 June 2024 19:33

Step-by-Step Process:

1. **Start with All Training Examples at the Root Node:**
   - Calculate the information gain for all possible features.
   - Pick the feature with the highest information gain for the split.
2. **Split the Data into Subsets:**
   - Create left and right branches based on the selected feature.
   - Distribute the training examples accordingly.
3. **Repeat the Splitting Process:**
   - Continue splitting on the left and right branches recursively.
   - Check if stopping criteria are met at each node:
     - All examples in a node belong to a single class (entropy is zero).
     - The tree has reached the maximum depth.
     - The information gain from an additional split is below a threshold.
     - The number of examples in a node is below a threshold.

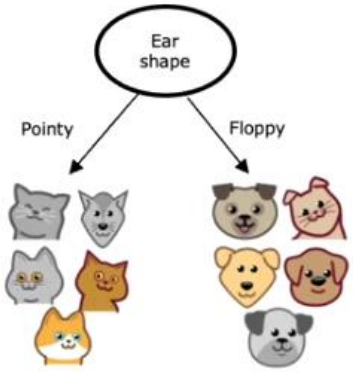
Decision Tree Learning

- Start with all examples at the root node
- Calculate information gain for all possible features, and pick the one with the highest information gain
- Split dataset according to selected feature, and create left and right branches of the tree
- Keep repeating splitting process until stopping criteria is met:
  - When a node is 100% one class
  - When splitting a node will result in the tree exceeding a maximum depth
  - Information gain from additional splits is less than threshold
  - When number of examples in a node is below a threshold

Example:

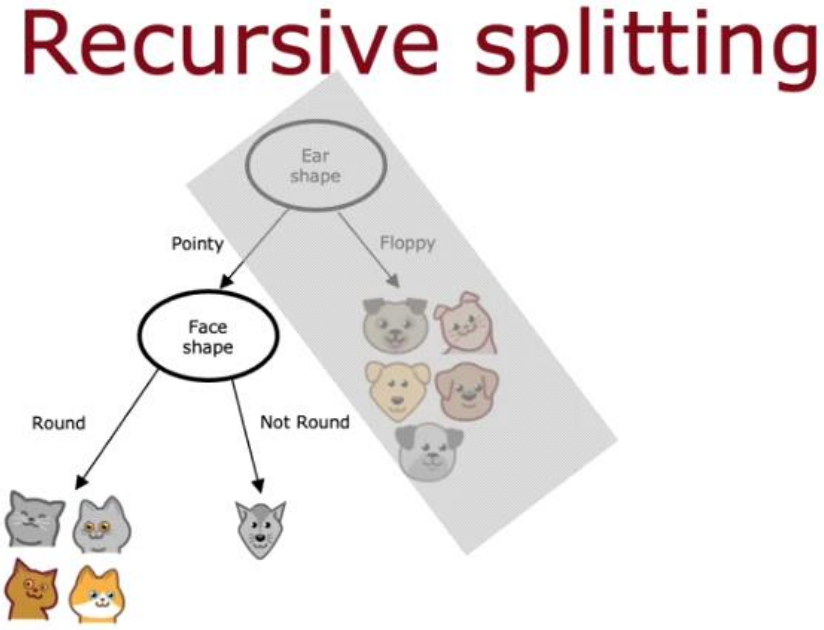
1. **Root Node Split:**
   - Calculate information gain for features: ear shape, face shape, whiskers.
   - Choose ear shape (highest information gain).

Recursive splitting

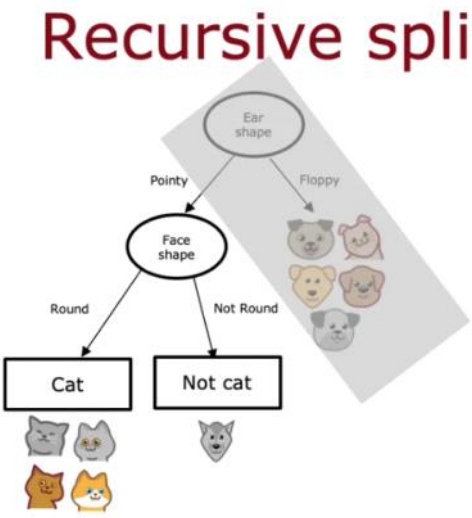


- Split into left (pointy ears) and right (floppy ears) branches.

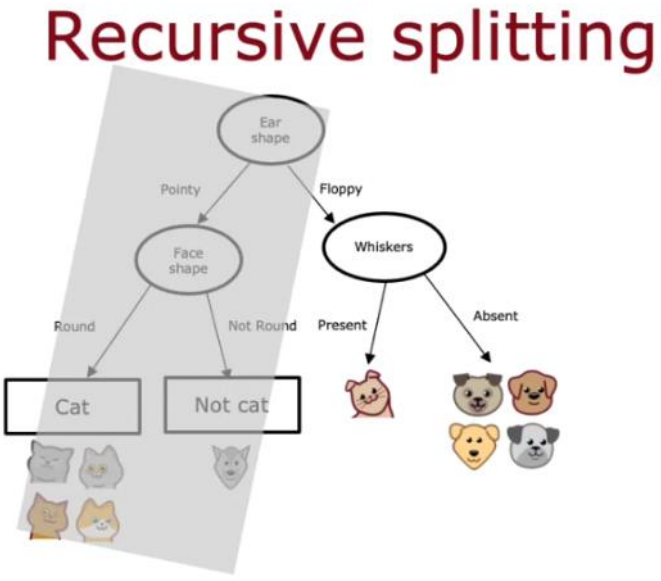
2. **Left Sub-Branch (Pointy Ears):**
   - Examples: [Cat, Cat, Cat, Cat, Dog]
   - Compute information gain for face shape and whiskers.
   - Face shape gives the highest information gain.
   - Split on face shape:
     - Left sub-branch: [Cat, Cat, Cat, Cat] (pure, stop splitting)



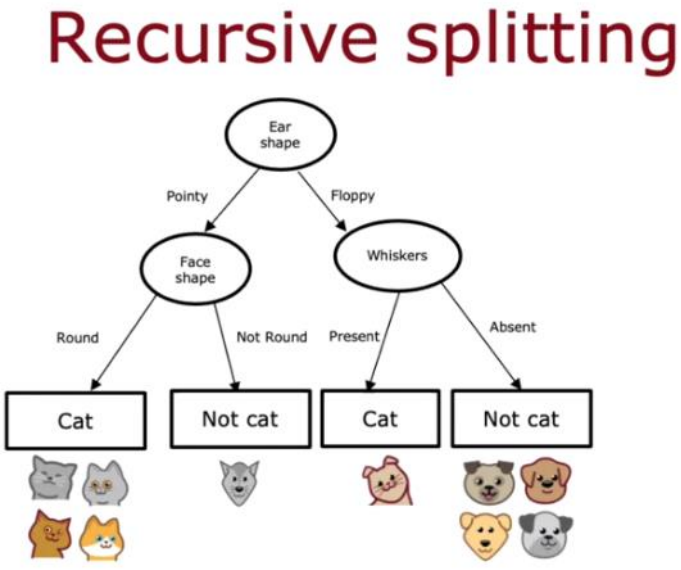
     - Right sub-branch: [Dog] (pure, stop splitting)



3. **Right Sub-Branch (Floppy Ears):**
    - Examples: [Dog, Dog, Dog, Dog, Cat]
    - Compute information gain for face shape and whiskers.
    - Whiskers give the highest information gain.
    - Split on whiskers:
        - Left sub-branch: [Dog, Dog, Dog] (pure, stop splitting)



        - Right sub-branch: [Dog, Cat] (impure, continue splitting if needed)



Recursive Algorithm:

- **Recursive Definition:**
  - At each node, treat it as a new root node.
  - Build the decision tree by recursively creating sub-trees for the left and right branches.

Stopping Criteria:

- **Pure Node:**
  - Stop splitting when all examples are of a single class.
- **Maximum Depth:**
  - Stop splitting when the maximum depth of the tree is reached.
- **Information Gain Threshold:**
  - Stop splitting when the information gain is below a set threshold.
- **Minimum Number of Examples:**
  - Stop splitting when the number of examples in a node is below a threshold.

Choosing Maximum Depth:

- **Default Choices:**
  - Open-source libraries often have good default settings.
- **Cross-Validation:**
  - Experiment with different depths and choose based on cross-validation results.
- **Risk of Overfitting:**
  - Larger depth allows learning more complex models but increases the risk of overfitting.

Handling Small Information Gain:

- **Threshold-Based Stopping:**
  - Stop splitting if the gain from any feature is small.

Handling Small Nodes:

Minimum Examples Threshold: Stop splitting if the number of examples in a node is below a certain threshold.

Prediction with Decision Tree:

- **Follow the Path:**
  - To make a prediction, follow the path from the root to a leaf node based on the features of the test example.
  - The leaf node provides the prediction.

Refinements:

- **Handling Multiple Values:**
  - Decision trees can handle features with more than two values. In upcoming discussions, we can explore how to manage such features.

Handling Features with More than Two Discrete Values Using One-Hot Encoding

Scenario:

- The initial training set for our pet adoption center application included features with only two possible values.
- **Example:**

- Ear shape: pointy or floppy
- Face shape: round or not round
- Whiskers: present or absent

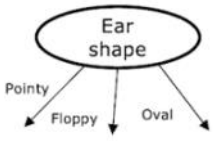
New Scenario:

- Now, ear shape can take on three possible values: pointy, floppy, and oval.
- **Challenge:**
  - When splitting on this feature, we need to create three subsets and build three sub-branches.

Features with three possible values

Ear shape (x_1)	Face shape (x_2)	Whiskers (x_3)	Cat (y)
Pointy	Round	Present	1
Oval	Not round	Present	1
Oval	Round	Absent	0
Pointy	Not round	Present	0
Oval	Round	Present	1
Pointy	Round	Absent	1
Floppy	Not round	Absent	0
Oval	Round	Absent	1
Floppy	Round	Absent	0
Floppy	Round	Absent	0

3 possible values



Solution: One-Hot Encoding:

Step-by-Step Process:

1. **Convert Categorical Feature into Binary Features:**
   - Instead of using a single feature (ear shape) with three possible values, create three new binary features:
     - Does the animal have pointy ears? (1 for yes, 0 for no)
     - Does the animal have floppy ears? (1 for yes, 0 for no)
     - Does the animal have oval ears? (1 for yes, 0 for no)

2. **Example Transformation:**
   - **Original Feature:**
     - First example: ear shape = pointy
     - Second example: ear shape = oval
   - **One-Hot Encoded Features:**
     - First example: [pointy_ear=1, floppy_ear=0, oval_ear=0]
     - Second example: [pointy_ear=0, floppy_ear=0, oval_ear=1]

One hot encoding

Ear shape	Pointy ears	Floppy ears	Oval ears	Face shape	Whiskers	Cat
Pointy	1	0	0	Round	Present	1
Oval	0	0	1	Not round	Present	1
Oval	0	0	1	Round	Absent	0
Pointy	1	0	0	Not round	Present	0
Oval	0	0	1	Round	Present	1
Pointy	1	0	0	Round	Absent	1
Floppy	0	1	0	Not round	Absent	0
Oval	0	0	1	Round	Absent	1
Floppy	0	1	0	Round	Absent	0
Floppy	0	1	0	Round	Absent	0

3. **General Rule:**
   - If a categorical feature can take on kk possible values, replace it with kk binary features.
   - Each binary feature can only take the value 0 or 1.
   - In any row, exactly one of these binary features will be 1 (the "hot" feature), giving the method its name: one-hot encoding.

You will notice that among all of these three features, if you look at any role here, **exactly 1 of the values is equal to 1.** And because one of these features will always take on the value 1 that's the hot feature and hence the name one-hot encoding.

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

4. **Advantages:**
   - Transforms the dataset so that each feature only takes on one of two possible values (0 or 1).
   - Makes it compatible with decision tree learning algorithms without further modifications.

Application Beyond Decision Trees:

- **Neural Networks:**
  - One-hot encoding is also used for training neural networks.
  - Neural networks expect numerical inputs, so encoding categorical features using ones and zeros is necessary.
- **Other Models:**
  - Logistic regression
  - Linear regression

Example with Multiple Features:

- **Original Categorical Features:**
  - Ear shape: pointy, floppy, oval
  - Face shape: round, not round
  - Whiskers: present, absent

- **One-Hot Encoded Features:**
  - Ear shape: [pointy_ear, floppy_ear, oval_ear]
  - Face shape: [round_face (1 if round, 0 if not round)]
  - Whiskers: [whiskers_present (1 if present, 0 if absent)]

- **Transformed Feature List:**
  - [pointy_ear, floppy_ear, oval_ear, round_face, whiskers_present]

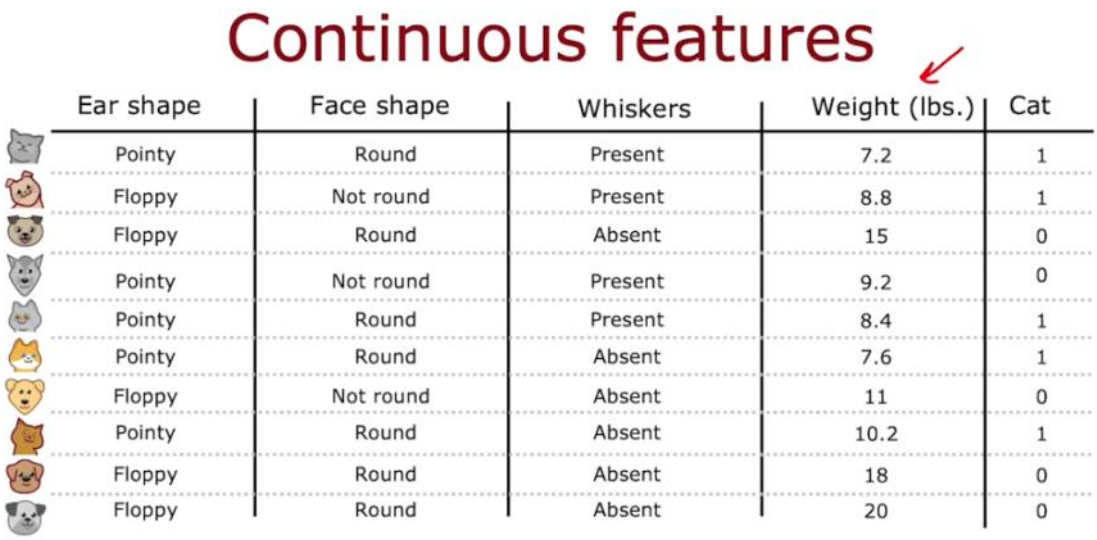
This approach ensures that decision tree algorithms can handle categorical features with more than two values effectively. It also makes the data suitable for other machine learning algorithms that require numerical inputs.

Handling Continuous Value Features in Decision Trees

04 June 2024 14:43

Scenario:

- Adding a continuous feature to the dataset, such as the weight of an animal in pounds.
- Unlike categorical features, continuous features can take any numerical value.



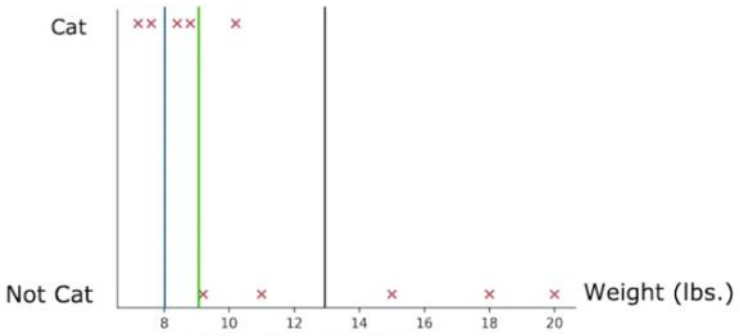
Ear shape	Face shape	Whiskers	Weight (lbs.)	Cat
Pointy	Round	Present	7.2	1
Floppy	Not round	Present	8.8	1
Floppy	Round	Absent	15	0
Pointy	Not round	Present	9.2	0
Pointy	Round	Present	8.4	1
Pointy	Round	Absent	7.6	1
Floppy	Not round	Absent	11	0
Pointy	Round	Absent	10.2	1
Floppy	Round	Absent	18	0
Floppy	Round	Absent	20	0

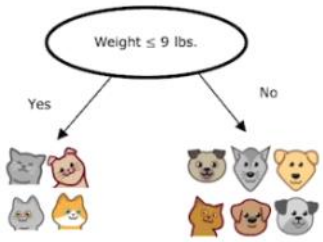
Process for Splitting on Continuous Features:

1. **Consider Splitting on Weight Feature:**
   - Split the data based on whether the weight is less than or equal to a certain threshold value.
   - The learning algorithm will determine the optimal threshold value that maximizes information gain.
2. **Example:**
   - Plot data with weight on the horizontal axis and the label (cat or not cat) on the vertical axis.
   - Determine possible threshold values (e.g., weight ≤ 8, weight ≤ 9, etc.).
3. **Calculating Information Gain:**
   - For each potential threshold, calculate the information gain.
   - **Information Gain Calculation:**
     - Compute the entropy at the root node.
     - Calculate the weighted average entropy of the subsets created by the split.
     - Information gain is the difference between the root entropy and the weighted average entropy of the subsets.

Detailed Calculation:

Splitting on a continuous variable





$$H(0.5) - \left(\frac{2}{10}H\left(\frac{2}{2}\right) + \frac{8}{10}H\left(\frac{3}{8}\right)\right) = 0.24$$

$$H(0.5) - \left(\frac{4}{10}H\left(\frac{4}{4}\right) + \frac{6}{10}H\left(\frac{1}{6}\right)\right) = 0.61$$

$$H(0.5) - \left(\frac{7}{10}H\left(\frac{5}{7}\right) + \frac{3}{10}H\left(\frac{0}{3}\right)\right) = 0.40$$

- **Example Calculation for Weight ≤ 8:**
  - **Entropy at the Root Node:**
    $H(0.5) = -0.5\log \ 2(0.5) - 0.5\log \ 2(0.5) = 1 H(0.5) = -0.5\log 2(0.5) - 0.5\log 2(0.5) = 1$
  - **Left Subset (Weight ≤ 8):** 2 cats
    $2 10 \times H(1) = 2 10 \times 0 = 0 10 2 \times H(1) = 102 \times 0 = 0$
  - **Right Subset (Weight > 8):** 3 cats, 5 dogs
    $8 10 \times H(38) = 810 \times (-38\log \ 2(38) - 58\log \ 2(58)) 108 \times H(83) = 108 \times (-83\log 2(83) - 85\log 2(85))$
  - **Information Gain:**
    $1 - (0 + 810 \times H(38)) = 1 - 0.76 = 0.24 1 - (0 + 108 \times H(83)) = 1 - 0.76 = 0.24$

- **Example Calculation for Weight ≤ 9:**
  - **Left Subset (Weight ≤ 9):** 4 cats
    $4 10 \times H(1) = 410 \times 0 = 0 104 \times H(1) = 104 \times 0 = 0$
  - **Right Subset (Weight > 9):** 1 cat, 5 dogs
    $6 10 \times H(16) = 610 \times (-16\log \ 2(16) - 56\log \ 2(56)) 106 \times H(61) = 106 \times (-61\log 2(61) - 65\log 2(65))$
  - **Information Gain:**
    $1 - (0 + 610 \times H(16)) = 1 - 0.39 = 0.61 1 - (0 + 106 \times H(61)) = 1 - 0.39 = 0.61$

- **Example Calculation for Weight ≤ 13:**
  - **Left Subset (Weight ≤ 13):** 5 cats
    $5 10 \times H(1) = 510 \times 0 = 0 105 \times H(1) = 105 \times 0 = 0$
  - **Right Subset (Weight > 13):** 1 cat, 4 dogs
    $5 10 \times H(15) = 510 \times (-15\log \ 2(15) - 45\log \ 2(45)) 105 \times H(51) = 105 \times (-51\log 2(51) - 54\log 2(54))$
  - **Information Gain:**
    $1 - (0 + 510 \times H(15)) = 1 - 0.6 = 0.401 - (0 + 105 \times H(51)) = 1 - 0.6 = 0.40$

4. **Selecting the Best Threshold:**
   - Among all potential threshold values, select the one that gives the highest information gain.
   - **Example:**
     - If the threshold of weight ≤ 9 provides the highest information gain (0.61), then split the data at this threshold.

5. **Building the Tree Recursively:**

   After selecting the optimal threshold and splitting the data, apply the same process recursively to the resulting subsets.

Decision trees for regression

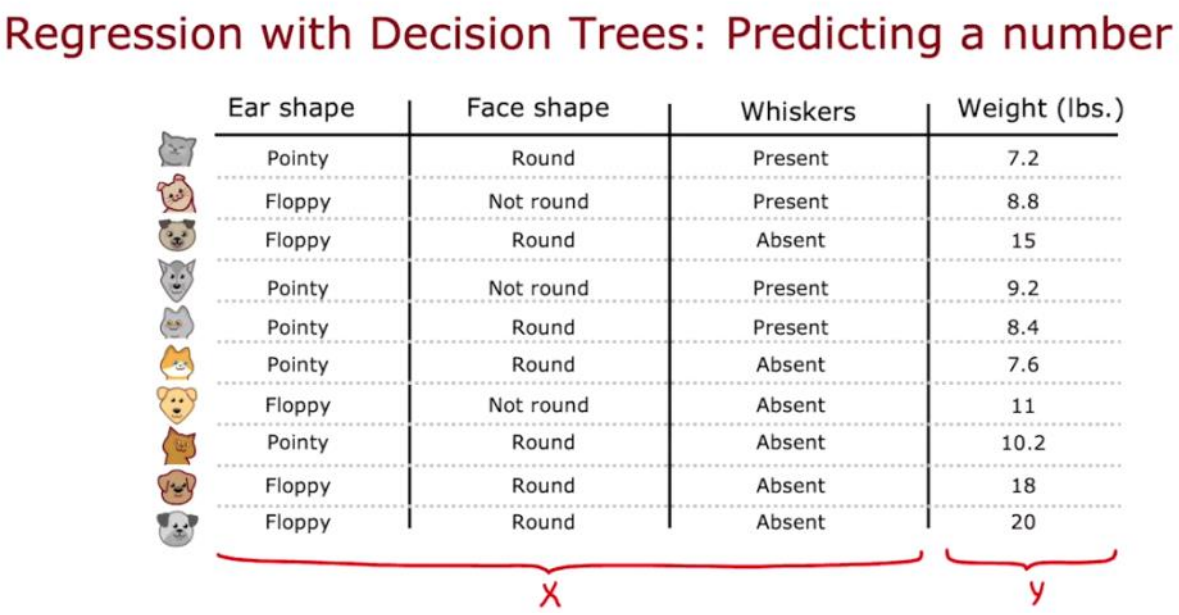
04 June 2024 23:11

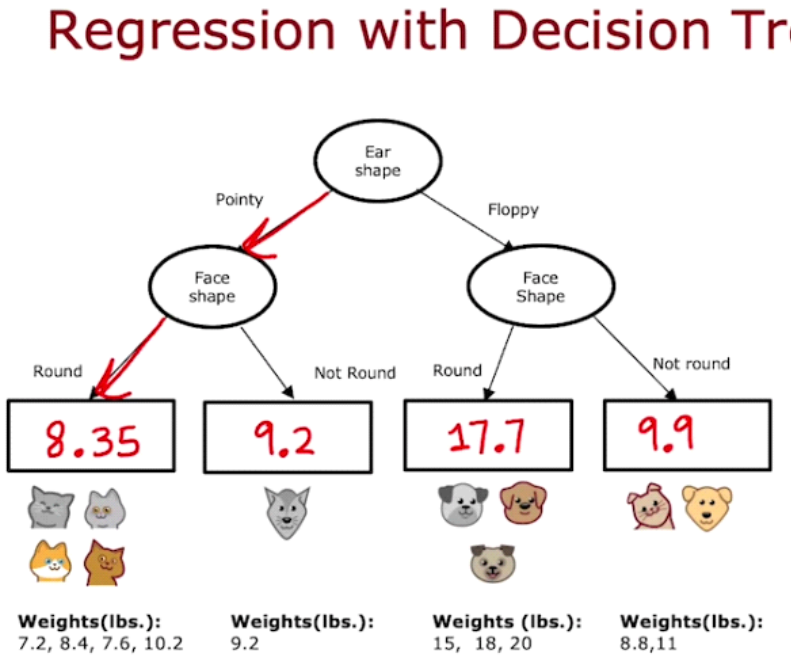
Overview:

Decision trees can be extended to handle regression tasks, predicting a continuous numerical value instead of a categorical label.

Example Setup:

- Features (X) used previously are now used to predict the weight of an animal (Y), which is the target output.
- Constructing a regression tree for predicting weight involves similar steps to building a classification tree but with a focus on minimizing variance.



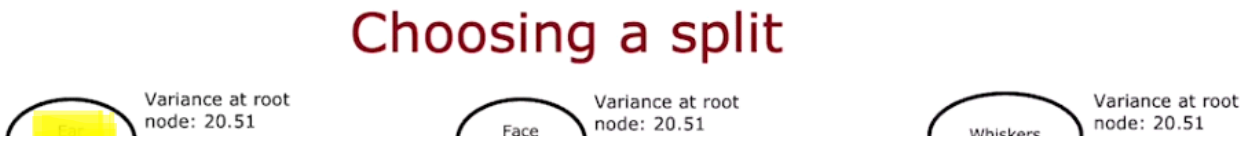


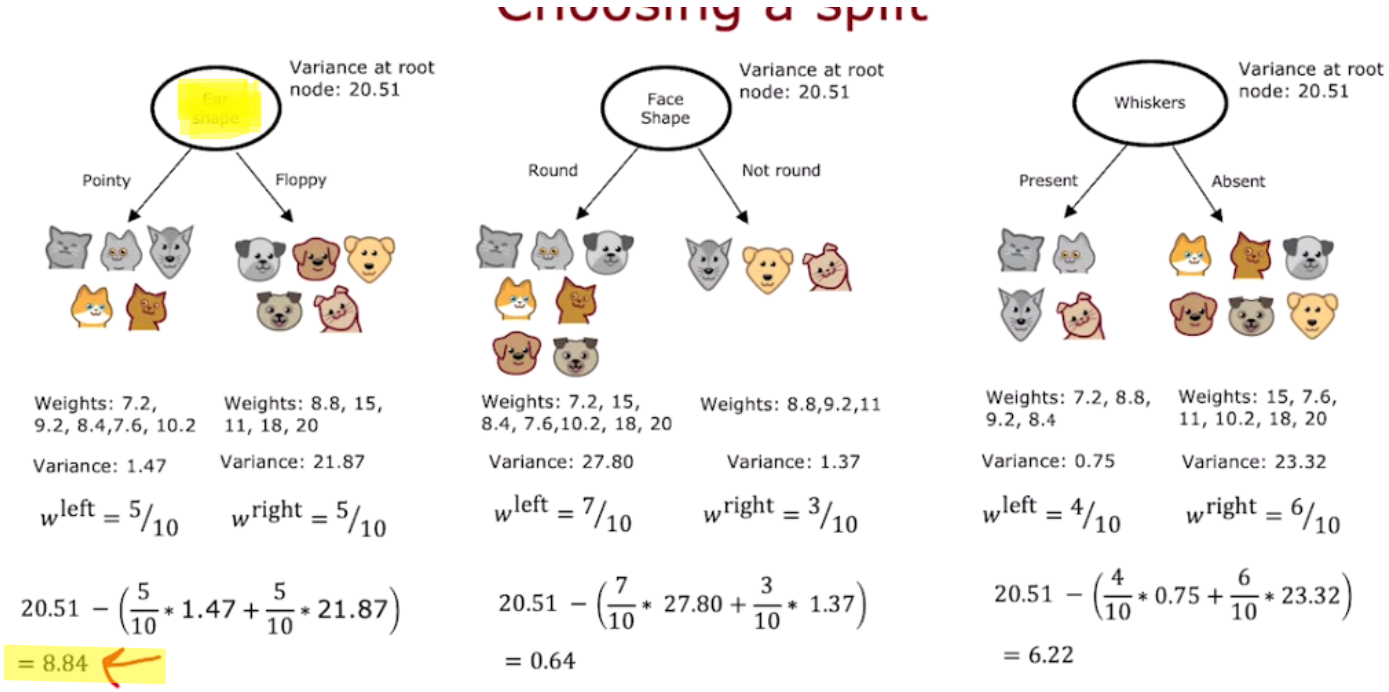
Regression Tree Construction:

1. **Splitting the Data:**
   - Split the data based on features like ear shape, face shape, or whiskers.
   - At each split, determine the subsets of data that result from the split.
2. **Variance Reduction:**
   - Instead of reducing entropy (used in classification), regression trees aim to reduce the variance of the target values (Y) in the subsets.
   - **Variance Calculation:**
     - Variance measures how much the target values (weights) vary within a subset.
     - For example, a set of weights [7.2, 9.2, 10.2] has a variance of 1.47, indicating low variability.
     - A set of weights [8.8, 15, 11, 18, 20] has a variance of 21.87, indicating high variability.
3. **Evaluating Splits:**
   - Calculate the weighted average variance after the split.
   - **Weighted Average Variance:**
     Weighted Average Variance=Wleft×Varianceleft+Wright×VariancerightWeighted Average Variance=Wleft ×Varianceleft+Wright×Varianceright
     - WleftWleft and WrightWright are the fractions of examples going to the left and right branches, respectively.
     - For instance, splitting on ear shape might yield a weighted average variance of 8.84.
4. **Reduction in Variance:**
   - Compute the reduction in variance as the difference between the variance at the root node and the weighted average variance after the split.
   - **Reduction in Variance Calculation:**
     Reduction in Variance=Varianceroot−Weighted Average VarianceReduction in Variance=Varianceroot −Weighted Average Variance
     - For example, if the variance at the root is 20.51 and the weighted average variance after splitting on ear shape is 8.84, the reduction in variance is:
       20.51−8.84=11.6720.51−8.84=11.67
5. **Selecting the Best Split:**
   - Choose the feature and threshold that result in the largest reduction in variance.
   - Continue splitting the data recursively on the resulting subsets until stopping criteria are met (e.g., minimum number of examples in a node or maximum tree depth).

Example:

- **Initial Splits:**
  - **Ear Shape:**
    - Left subset variance: 1.47
    - Right subset variance: 21.87
    - Weighted average variance: 510×1.47+510×21.87=11.67105×1.47+105×21.87=11.67
    - Reduction in variance: 20.51−11.67=8.8420.51−11.67=8.84
  - **Face Shape:**
    - Left subset variance: 27.80
    - Right subset variance: 1.37
    - Weighted average variance: 710×27.80+310×1.37=20.51107×27.80+103×1.37=20.51
    - Reduction in variance: 20.51−20.51=020.51−20.51=0
  - **Whiskers:**
    - Left subset variance: 3.26
    - Right subset variance: 14.23
    - Weighted average variance: 610×3.26+410×14.23=6.22106×3.26+104×14.23=6.22
    - Reduction in variance: 20.51−6.22=14.2920.51−6.22=14.29
- **Best Split:**
  - Choose the feature with the highest reduction in variance. In this example, splitting on whiskers gives the largest reduction (14.29), so it is selected.





Recursive Construction:

After the initial split, recursively apply the same process to the resulting subsets to further partition the data until the stopping criteria are met.

Using Multiple Decision Trees

05 June 2024 16:10

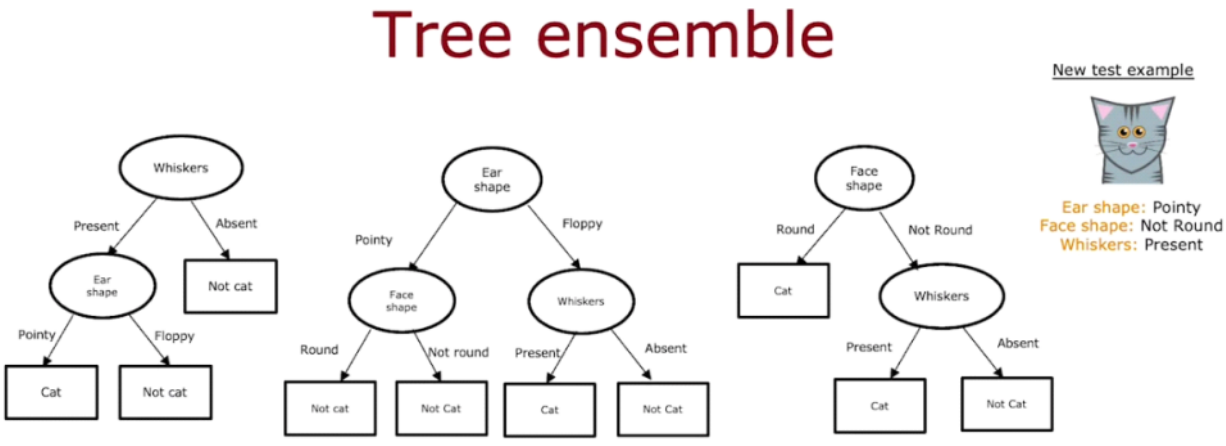
Tree Ensembles and Their Robustness

Weakness of a Single Decision Tree:

- Highly sensitive to small changes in the data.
- A single change in the training example can lead to different splits and thus a completely different tree.

Solution: Tree Ensembles:

- Build multiple decision trees instead of just one.
- Combine the predictions of these trees to improve accuracy and robustness.



Example:

- In a cat vs. not cat classification, the best feature to split on at the root could change if a single training example is modified.
- This sensitivity can cause significant changes in the decision tree structure.

Tree Ensemble:

- **Concept:**
  - Train multiple decision trees.

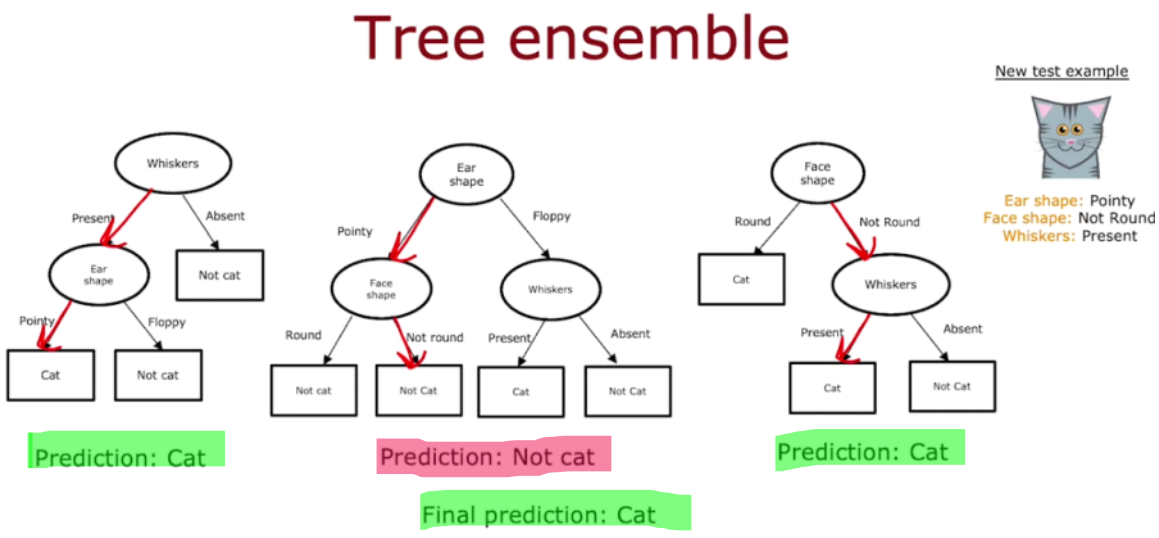
- Aggregate their predictions by voting or averaging.
- **Voting Example:**
  - Given three decision trees:
    - Tree 1: Predicts "cat"
    - Tree 2: Predicts "not cat"
    - Tree 3: Predicts "cat"
  - Final prediction is based on the majority vote: "cat".

Advantages of Tree Ensembles:

- **Robustness:**
  - Less sensitive to individual variations in the training data.
  - A single tree's prediction has less influence on the final outcome.
- **Accuracy:**
  - Combining multiple trees generally leads to more accurate predictions.

Building the Ensemble:

- To create an ensemble of decision trees, each tree must be slightly different.
- This can be achieved using techniques like "sampling with replacement".



Sampling with Replacement

Concept:

- Randomly sample the training data multiple times to create different training sets for each tree.
- **Procedure:**
  - Given a dataset with nnn examples, create a new training set by randomly selecting nnn examples from the original dataset, with replacement.
  - This means some examples may appear multiple times in the new set, while others may not appear at all.

Importance in Tree Ensembles:

- **Diversity:**
  - Each tree is trained on a different subset of the data, introducing variability in the trees.
- **Reducing Overfitting:**
  - The trees are less likely to overfit to any specific part of the data since they each see a different version of it.

Sampling with Replacement: Key to Building Tree Ensembles

Concept:

- Sampling with replacement is a technique where each selected sample is put back into the pool before the next draw, allowing the same sample to be picked more than once.

Demonstration with Tokens:

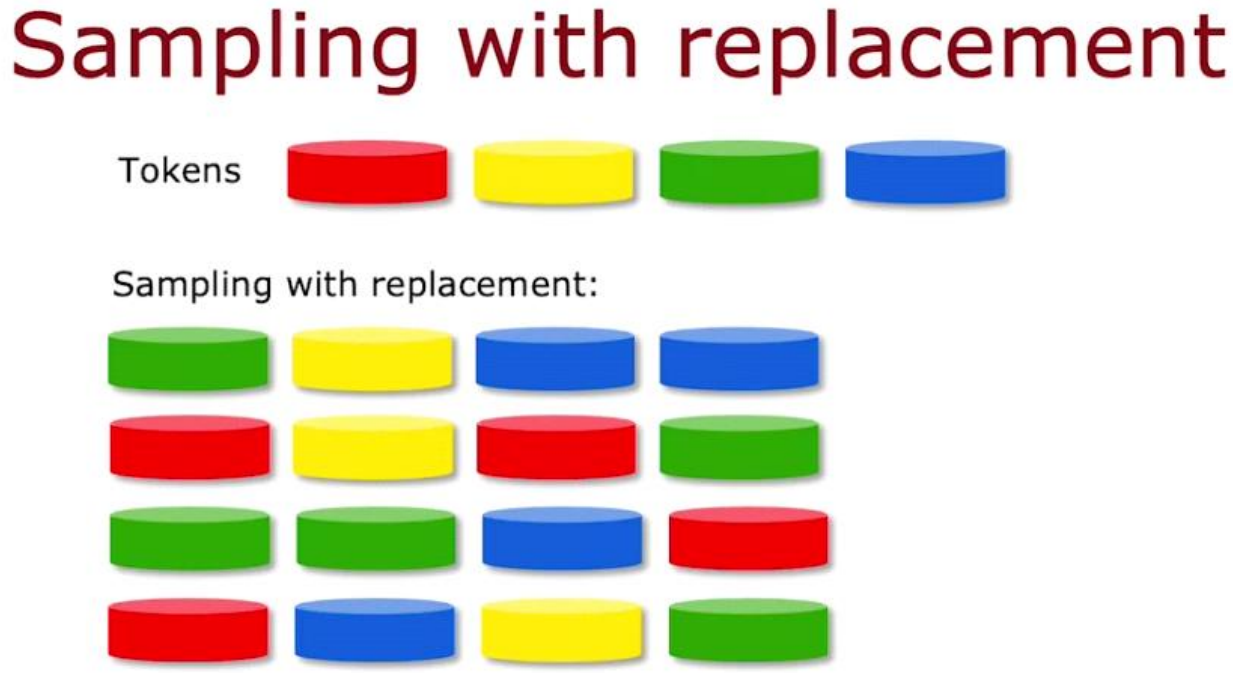
- Imagine you have four tokens of different colors: red, yellow, green, and blue.
- Place these tokens in a bag and shake it up.
- Draw one token, note its color, and then put it back into the bag.
- Repeat this process multiple times.

Example Process:

1. Shake the bag and pick a token: Green.
2. Replace the green token in the bag.
3. Shake the bag and pick another token: Yellow.
4. Replace the yellow token in the bag.
5. Shake the bag and pick another token: Blue.
6. Replace the blue token in the bag.
7. Shake the bag and pick another token: Blue again.

Possible Outcomes:

- Green, Yellow, Blue, Blue
- Red, Yellow, Red, Green
- Green, Green, Blue, Red
- Red, Blue, Yellow, Green



Key Point:

- With replacement, it's possible to draw the same token multiple times and some tokens might not be drawn at all.

Application to Building Tree Ensembles

Creating Random Training Sets:

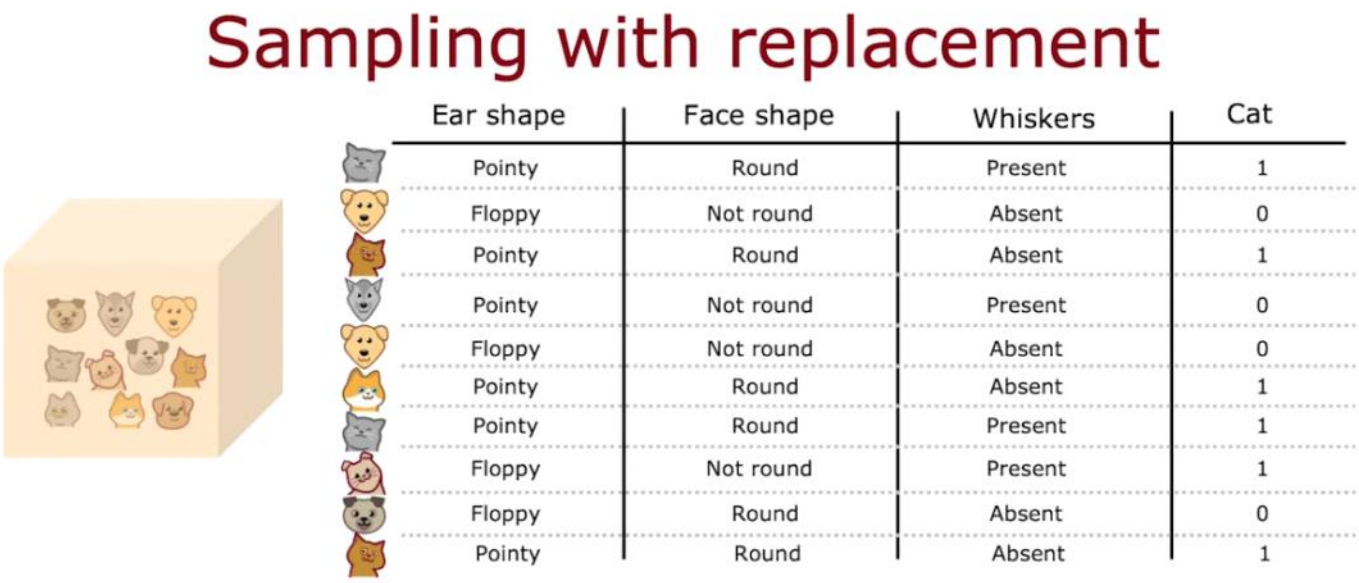
- Original training set: 10 examples of cats and dogs.
- Simulate placing these examples in a theoretical bag (don't actually use real animals).
- Create a new training set by randomly sampling 10 examples from the bag, with replacement.

Procedure:

1. Place the 10 training examples in the theoretical bag.
2. Randomly pick one training example and record it.
3. Replace the example in the bag.
4. Repeat until you have 10 training examples in your new set.

Example Result:

- Training set might contain: [Example 1, Example 2, Example 2, Example 3, Example 4, Example 5, Example 5, Example 6, Example 7, Example 8]
- Note: Some examples are repeated, and some may be missing.



Importance in Tree Ensembles

Diversity:

- Each tree in the ensemble is trained on a slightly different training set.
- This diversity in training sets helps create varied trees.

Robustness:

- Ensemble of trees is less sensitive to any single training example.
- Aggregating predictions from multiple trees leads to a more stable and accurate model.

Random Forest Algorithm

05 June 2024 17:03

Building a Tree Ensemble with Random Forest

The random forest algorithm is a powerful tree ensemble method that significantly improves prediction accuracy over using a single decision tree. Here's how it works:

1. Creating Multiple Training Sets

- Given a training set of size MMM, generate BBB different training sets, each also of size MMM, using sampling with replacement.
- Each new training set is created by randomly sampling MMM examples from the original set with replacement. This means some examples may be repeated, and some may not appear in the new set at all.

2. Training Multiple Decision Trees

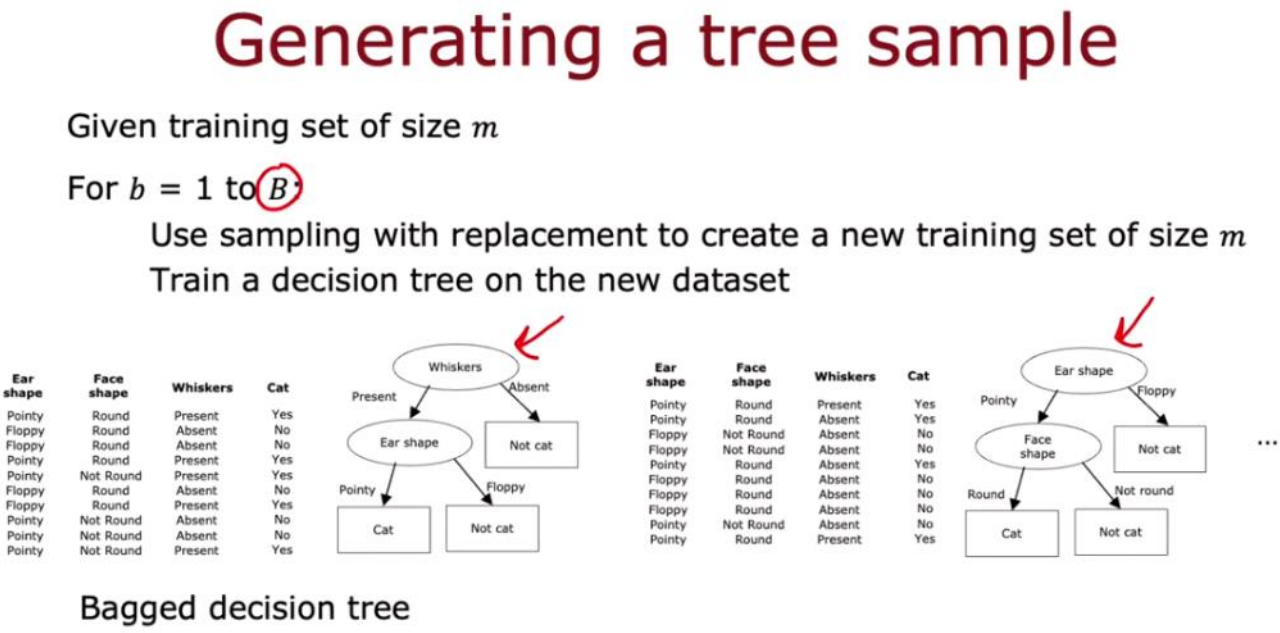
- For each of the BBB training sets, train a decision tree.
- Since each training set is slightly different due to the sampling process, the resulting decision trees will also be different.

3. Making Predictions with the Ensemble

- When a new test example needs to be classified, run it through all BBB decision trees.
- Each tree makes its own prediction, and the final prediction is made by taking a majority vote (for classification) or averaging the predictions (for regression).

4. Bagged Decision Trees

- This ensemble method, where multiple trees are trained on different samples generated with replacement, is called "bagging" (Bootstrap Aggregating).
- Bagged decision trees help reduce the variance of the model, making it more robust.



Enhancing Random Forest

A random forest is an enhanced version of bagged decision trees. The key modification is to increase the diversity among the trees, especially at the root and near-root levels:

1. Random Feature Selection

- At each node in the decision tree, instead of considering all NNN features, a random subset of KKK features is selected.
- The best feature for the split is chosen only from this subset.
- Typically, KKK is chosen as the square root of NNN. For example, if there are 100 features, KKK would be around 10.

2. Increasing Model Robustness

- By randomly selecting features at each node, the trees become more varied and less correlated.
- This further reduces the variance of the final model, making it more robust and accurate than bagged decision trees alone.

Benefits of Random Forest

- **Accuracy:** Random forests usually have high accuracy due to the ensemble approach.
- **Robustness:** They are less sensitive to changes in the training data.
- **Versatility:** Can be used for both classification and regression tasks.

Example: Building a Random Forest

- Assume we have 10 training examples with 3 features.
- We generate 100 new training sets using sampling with replacement.
- For each training set, a decision tree is trained.
- At each node of each tree, a random subset of features is selected, and the best feature from this subset is used for the split.
- For a new test example, all 100 trees are used to make predictions, and the final prediction is made based on majority voting.

Randomizing the feature choice

At each node, when choosing a feature to use to split, if n features are available, pick a random subset of $k < n$ features and allow the algorithm to only choose from that subset of features.

$K = \sqrt{n}$

Random forest algorithm

XGBoost

05 June 2024 17:08

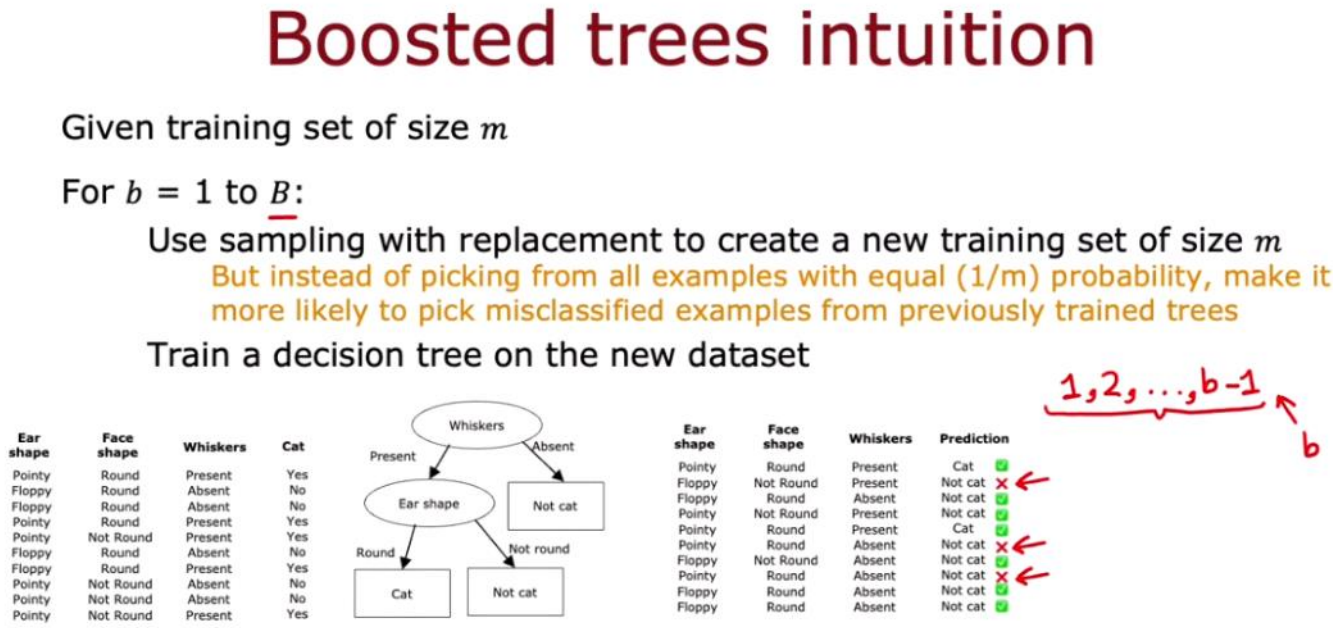
XGBoost, or e**X**treme **G**radient **Boost**ing, is one of the most powerful and widely-used algorithms for building decision tree ensembles. It has been instrumental in winning numerous machine learning competitions and is favored in commercial applications due to its efficiency and performance. Here's a breakdown of how XGBoost works and why it is so effective.

The Boosting Concept

Boosting is a method to convert weak learners (like shallow decision trees) into strong learners. The key idea is to sequentially train new models that focus on correcting the errors made by previous models.

Steps in Boosting:

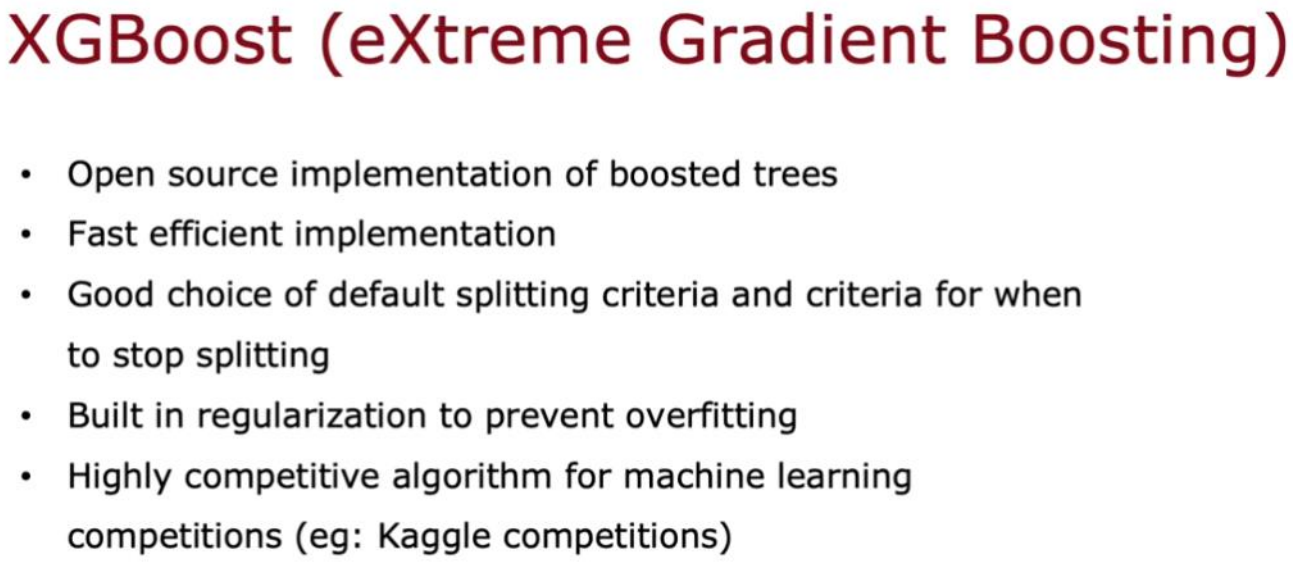
1. **Initial Model Training**:
   - Train an initial decision tree on the training set.
2. **Focus on Errors**:
   - For subsequent trees, increase the focus on examples that the previous trees misclassified. This is akin to deliberate practice, where you concentrate on parts of a task that need improvement.
3. **Weighted Sampling**:
   - Instead of uniformly sampling examples, boosting assigns higher weights to misclassified examples, making it more likely that these examples will be used to train the next tree.
4. **Combine Models**:
   - The final prediction is a weighted combination of the predictions from all the trees.



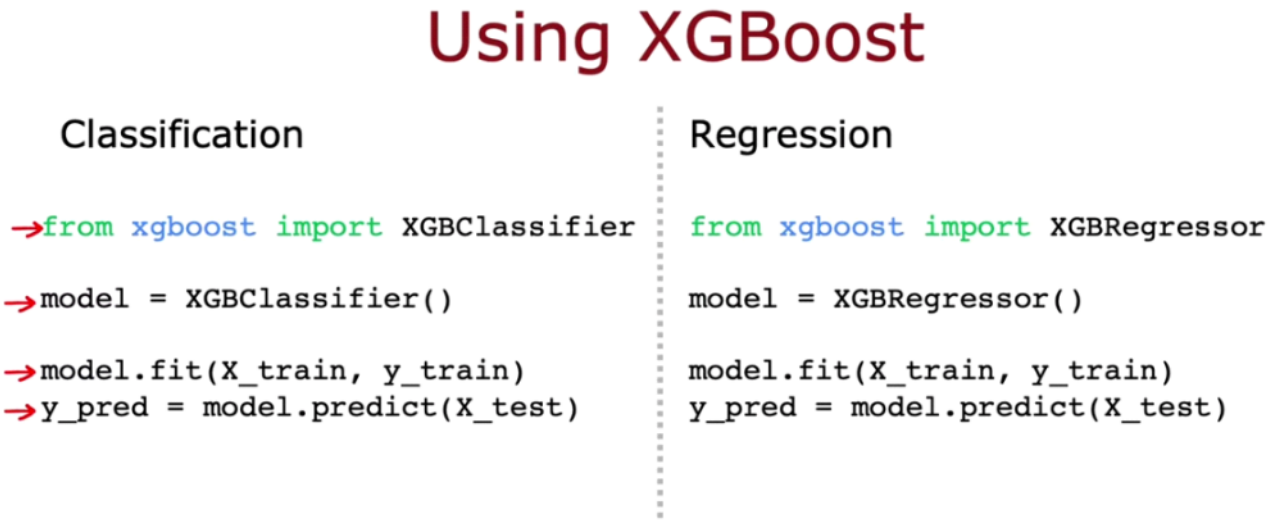
How XGBoost Enhances Boosting

XGBoost takes the concept of boosting and incorporates several optimizations to improve performance and efficiency:

1. **Gradient Boosting**:
   - XGBoost uses gradient boosting, where new models are trained to correct the residual errors (gradients) of the existing models.
2. **Weighted Examples**:
   - Instead of sampling with replacement, XGBoost assigns different weights to training examples based on their difficulty.
3. **Regularization**:
   - To prevent overfitting, XGBoost includes regularization terms that penalize model complexity. This ensures the models generalize well to new data.
4. **Efficient Computation**:
   - XGBoost is designed to be highly efficient in terms of both memory usage and computation speed. It uses advanced data structures and parallel processing to handle large datasets quickly.



Implementing XGBoost



Using XGBoost in Python is straightforward due to the availability of the xgboost library. Here's a basic implementation for both classification and regression tasks:

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# Load your data
X, y = load_data()
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=
0.2, random_state=42)
# Initialize the XGBoost classifier
model = xgb.XGBClassifier()
# Train the model
model.fit(X_train, y_train)
# Make predictions
y_pred = model.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

Regression Example:

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
# Load your data
X, y = load_data()
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=
0.2, random_state=42)
# Initialize the XGBoost regressor
model = xgb.XGBRegressor()
# Train the model
model.fit(X_train, y_train)
# Make predictions
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
```

Choosing Between Decision Trees and Neural Networks

Both decision trees/tree ensembles and neural networks are powerful learning algorithms, but they excel in different contexts. Here's a comparison of when you might choose one over the other:

Decision Trees/Tree Ensembles

Pros:

1. **Effective on Tabular Data**: Works well on structured data, such as datasets in spreadsheet format.
2. **Speed**: Fast to train, allowing for quicker iteration and model improvement.
3. **Interpretability**: Small decision trees can be human-interpretable, helping to understand how decisions are made.
4. **XGBoost**: Highly effective tree ensemble algorithm that improves performance.

Cons:

1. **Limited Use on Unstructured Data**: Not suitable for unstructured data like images, audio, and text.
2. **Ensemble Complexity**: Interpreting large ensembles can be challenging.

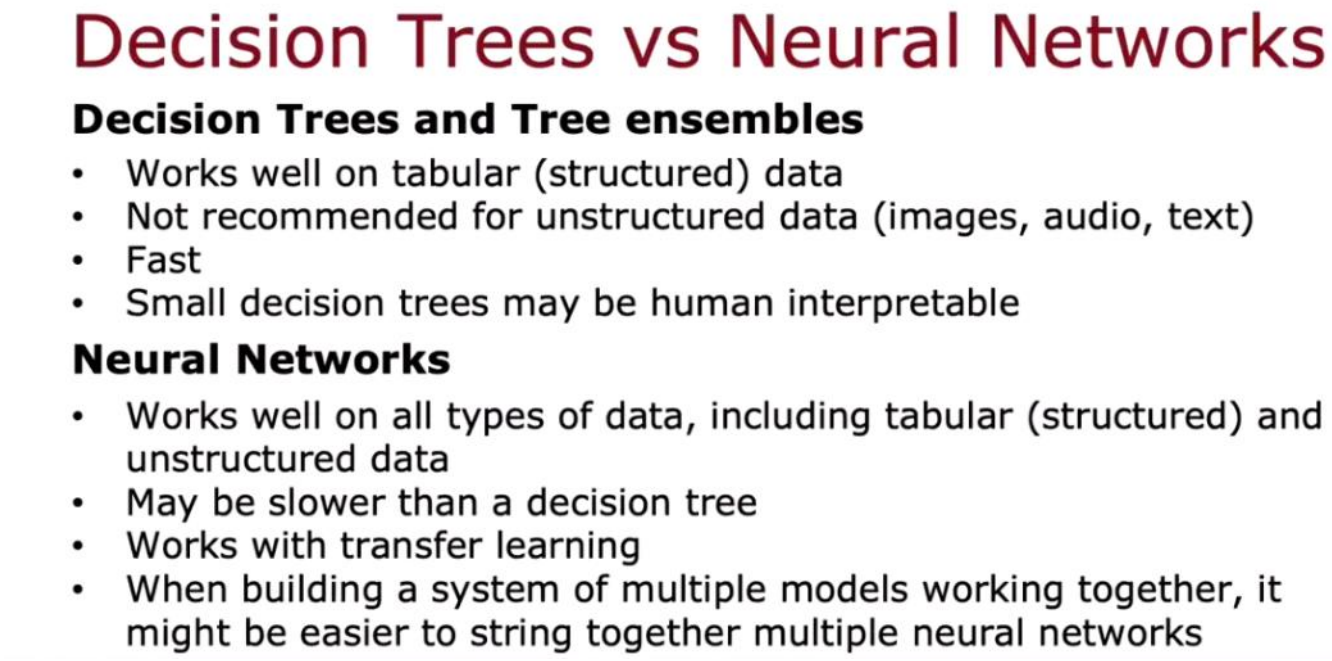
Neural Networks

Pros:

1. **Versatile**: Works well on all types of data, including structured, unstructured, and mixed data.
2. **Performance**: Can achieve state-of-the-art performance on complex tasks like image classification and natural language processing.
3. **Transfer Learning**: Ability to leverage pre-trained models for improved performance on small datasets.
4. **Flexibility in Model Composition**: Easier to integrate and train multiple neural networks together.

Cons:

1. **Training Time**: Larger neural networks can be slow to train.
2. **Complexity**: Understanding and optimizing neural networks can be challenging.



Choosing Between Them

- **Tabular Data (Structured)**:
  - **Decision Trees**: Often competitive, especially with tree ensembles like XGBoost.
- **Unstructured Data (Images, Audio, Text)**:
  - **Neural Networks**: Preferred due to their ability to learn complex patterns in data.
- **Mixed Data (Structured and Unstructured)**:
  - **Neural Networks**: Ideal for handling both structured and unstructured components together.
- **Training Time Considerations**:
  - **Decision Trees**: Faster to train, making them suitable for quick iterations and explorations.
  - **Neural Networks**: Slower to train, but can achieve superior performance on complex tasks with sufficient computational resources.