Neural Networks

Inference = prediction

A neuron is a computational unit that takes a vector of inputs, computes a weighted sum plus a bias, then applies a non-linear activation function to produce a single output.

For inputs $x_1, x_2, \dots, x_n$

$$a = g(w_1x_1 + w_2x_2 + \dots + w_nx_n + b)$$

where w are the weights, b is the bias and g is the activation function.

An activation function is a non-linear function applied to the output of a neuron's linear combination. It allows the network to learn non-linear relationships. Common examples are sigmoid, tanh, ReLU, Softmax.

A layer is a collection of neurons that all receive the same input and compute their outputs in parallel. A layer with n neurons produces a vector of n outputs.

Input layer is the layer that receives the raw input features. It does not perform any computation, it simply represents the input.

Hidden layer is any layer between the input and output layers. It is called hidden because its values are not observed in the training data; they are intermediate computations learned by the network.

Output layer is the final layer of the network. It produces the prediction $\hat{y}$

A neural network is a function composed of multiple layers of neurons, where the output of each layer become the input to the next layer. The function is parameterized by the weights and biases of all layers, which are learned from the data.

A deep neural network is a neural network with more than one hidden layer. The "deep" refers to the depth or number of layers.

Forward propagation is the process of computing the output of a neural network for a given input. It involves applying each layer's transformation in sequence, starting from the input and moving toward the output.

For layer l:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = g^{(l)}(z^{(l)})$$

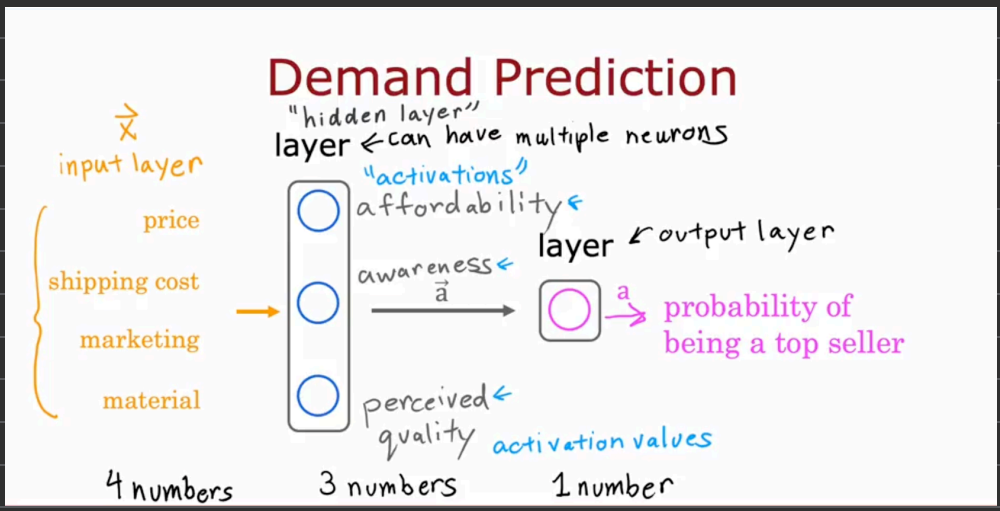
Backward propagation (backprop) is the process of computing the gradients of the loss function with respect to every weight and bias in the network.

Training is the process of adjusting the parameters of the network to minimize the cost function on a training dataset, typically using gradient descent and back propagation.

Inference is the process of using a trained network to make predictions on new data. Parameters are frozen, and only forward propagation is performed.

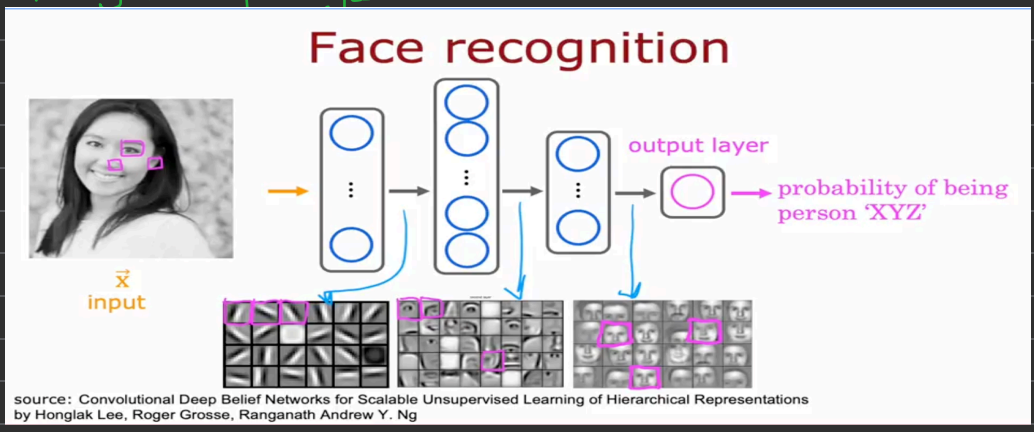
Vectorization is the practice of expressing computations as operations on vectors and matrices rather than explicit loops. It enables use of optimized linear algebra libraries and GPUs.

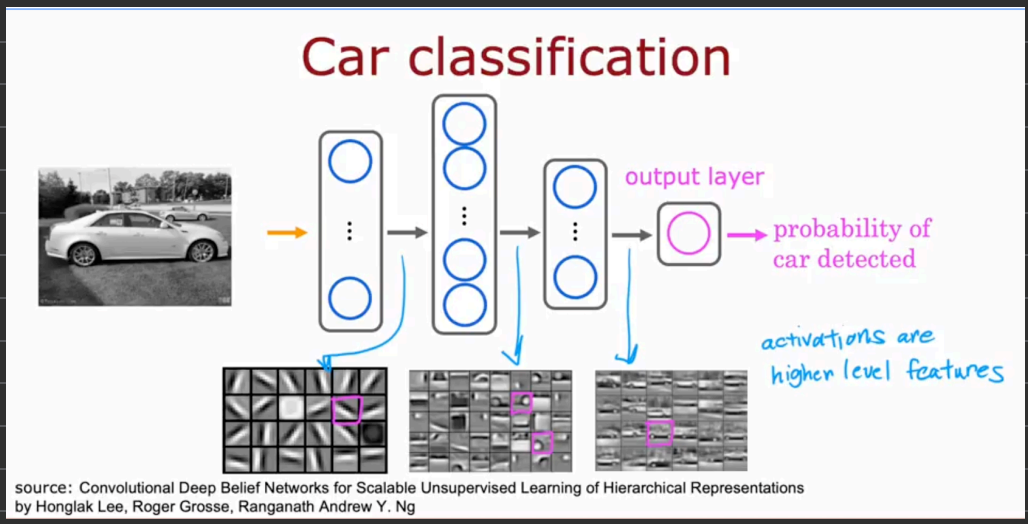
Epoch is one complete pass through the entire training dataset during training.



It is more like the network is doing feature engineering by combining previous layer to produce the next layer containing features that are more representative of the output.

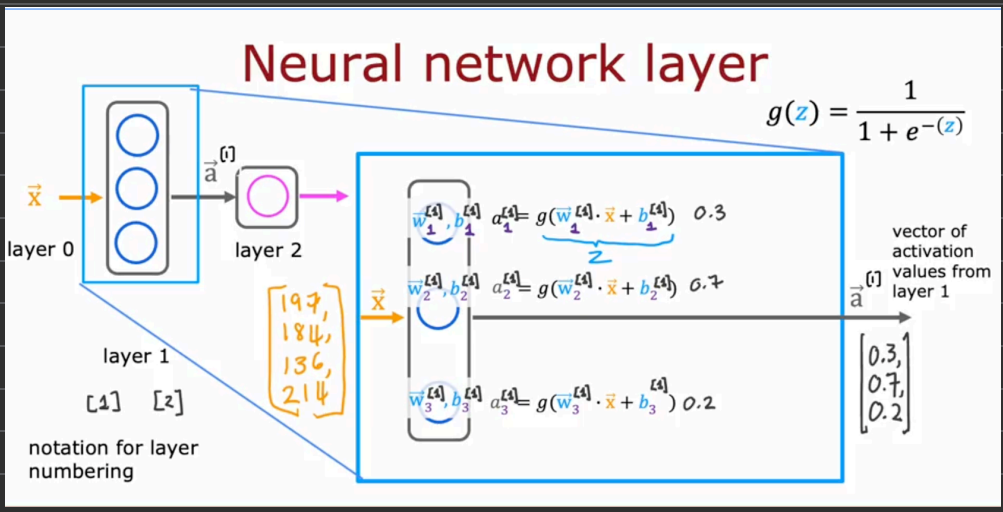
Recognizing images

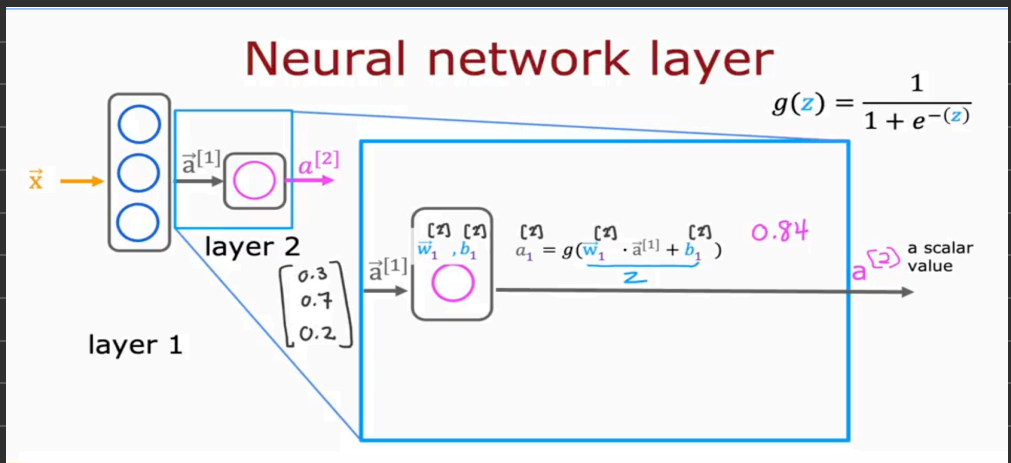


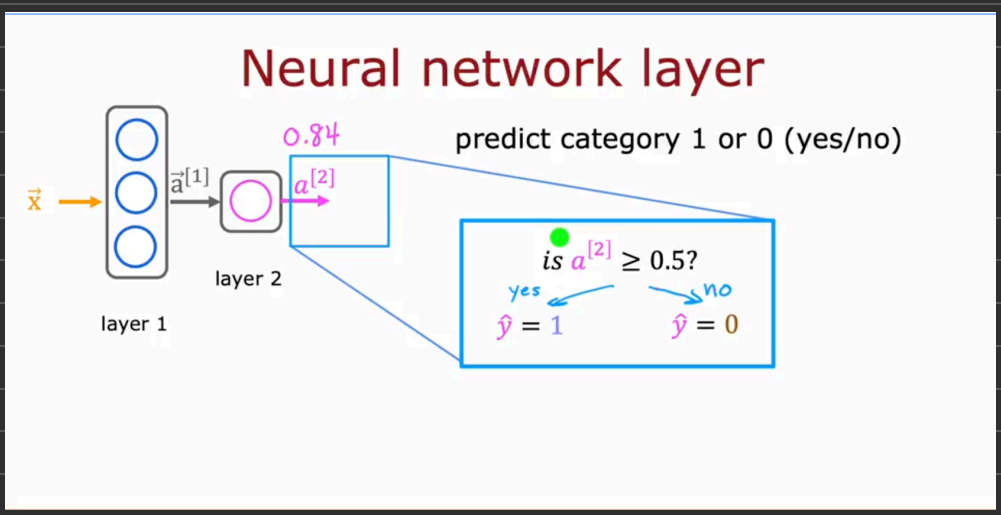


Nobody taught the network what to look for in each layer, it just learns all of that by itself

Neural Network Layer.







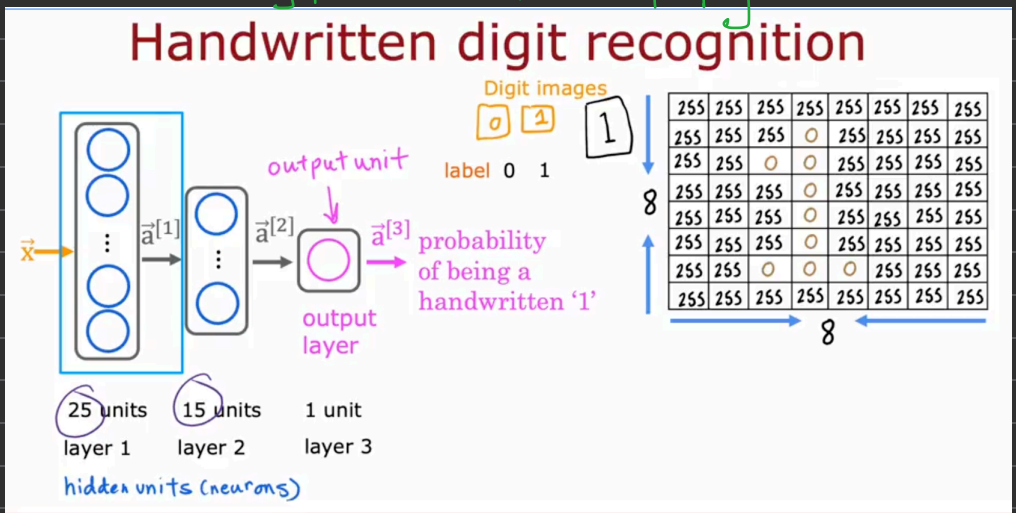
$$a_j^{[l]} = g\left(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]}\right)$$

activation function

parameters w and b of layer l, unit j

$$\vec{x} = \vec{a}^{[0]}$$

Inference: making predictions (forward propagation)



$$\vec{a}^{[1]} = \begin{bmatrix} g(\vec{w}_1^{[1]} \cdot \vec{a}^{[0]} + b_1^{[1]}) \\ \vdots \\ g(\vec{w}_{25}^{[1]} \cdot \vec{x} + b_{25}^{[1]}) \end{bmatrix}$$

$$\vec{a}^{[2]} = \begin{bmatrix} g(\vec{w}_1^{[2]} \cdot a^{[1]} + b_1^{[2]}) \\ \vdots \\ g(\vec{w}_{15}^{[2]} \cdot a^{[1]} + b_{15}^{[2]}) \end{bmatrix}$$

$$\vec{a}^{[3]} = \left[g(\vec{w}_1^{[3]} \cdot a^{[2]} + b_1^{[2]}) \right]$$

is $a_1^{[3]} \geq 0.5$?

yes: $\hat{y} = 1$ — image is digit 1

no: $\hat{y} = 0$ — image is not digit 1

Tensorflow

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import
Dense

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:
    y_hat = 1
else:
    y_hat = 0

print(f"y_hat: {y_hat}")
```

Building a neural network architecture


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

		y
200	17	1
120	5	0
425	20	0
212	18	1

```
x = np.array([[200.0, 17.0],
              [120.0, 5.0],
              [425.0, 20.0],
              [212.0, 18.0]])
```
4 x 2

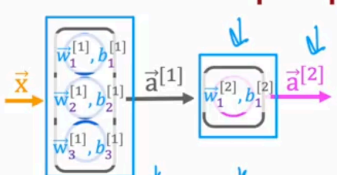
targets `y = np.array([1,0,0,1])`

```
model.compile(...)
model.fit(x,y)
```
← more about this next week!

```
model.predict(x_new)
```

Forward prop implementation in a single layer:

forward prop (coffee roasting model)



$$a_1^{[2]} = g(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]})$$

```
w2_1 = np.array([-7, 8, 9])
b2_1 = np.array([3])
z2_1 = np.dot(w2_1,a1)+b2_1
a2_1 = sigmoid(z2_1)
```

$w_1^{[2]}$ w2_1

```
x = np.array([200, 17])
```
1D arrays

$$a_1^{[1]} = g(\vec{w}_1^{[1]} \cdot \vec{x} + b_1^{[1]})$$

```
w1_1 = np.array([1, 2])
b1_1 = np.array([-1])
z1_1 = np.dot(w1_1,x)+b1_1
a1_1 = sigmoid(z1_1)
```

$$a_2^{[1]} = g(\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]})$$

```
w1_2 = np.array([-3, 4])
b1_2 = np.array([1])
z1_2 = np.dot(w1_2,x)+b1_2
a1_2 = sigmoid(z1_2)
```

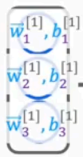
$$a_3^{[1]} = g(\vec{w}_3^{[1]} \cdot \vec{x} + b_3^{[1]})$$

```
w1_3 = np.array([5, -6])
b1_3 = np.array([2])
z1_3 = np.dot(w1_3,x)+b1_3
a1_3 = sigmoid(z1_3)
```

```
a1 = np.array([a1_1, a1_2, a1_3])
```

General Implementation of forward propagation.

Forward prop in NumPy



$\vec{w}_1^{[1]} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ $\vec{w}_2^{[1]} = \begin{bmatrix} -3 \\ 4 \end{bmatrix}$ $\vec{w}_3^{[1]} = \begin{bmatrix} 5 \\ -6 \end{bmatrix}$

```
W = np.array([
    [1, -3, 5]
    [2, 4, -6]])
```
2 by 3

$b_1^{[1]} = -1$ $b_2^{[1]} = 1$ $b_3^{[1]} = 2$

```
b = np.array([-1, 1, 2])
```

$\vec{a}^{[0]} = \vec{x}$

```
a_in = np.array([-2, 4])
```

```
def dense(a_in,W,b):
3 units = W.shape[1]          [0,0,0]   a^[1]
  a_out = np.zeros(units)               a^[1]
  for j in range(units):      0,1,2
    w = W[:,j]
    z = np.dot(w,a_in) + b[j]
    a_out[j] = g(z)
  return a_out
```

```
def sequential(x):
    a1 = dense(x,W1,b1)
    a2 = dense(a1,W2,b2)
    a3 = dense(a2,W3,b3)
    a4 = dense(a3,W4,b4)
    f_x = a4
    return f_x
```

Note: g() is defined outside of dense().
(see optional lab for details)

capital W refers to a matrix

Could be done using dot product for matrices and makes it simpler and more efficient.

Vectorization is the practice of expressing computations as operations on vectors and matrices rather than explicit loops. It enables use of optimized linear algebra libraries and GPUs.

How are neural networks implemented efficiently?

$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$

Non vectorized version:

```
z = 0
for i in range(n):
    z += w[i] * x[i]
z += b
```

This loops n times, doing one multiplication and one addition per iteration

Vectorized version:

```
z = np.dot(w, x) + b
```

This computes the dot product as a single optimized operation.

What vectorization buys us?

* Speed! Modern CPU have SIMD (Single instruction, multiple data) units that can perform the same operation on multiple values simultaneously. GPUs take this further with thousands of parallel cores. Vectorized code uses these capabilities, loop code does not.

* Cleaner code

* Mathematical clarity! Linear algebra notation maps to vectorized code. Reading $z = W@x+b$ reads exactly like the math.

Matrix multiplication

$a \cdot b = a^T b$

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$C = A \cdot B = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

NOTE:

If a network has S_{in} units in a layer and S_{out} units in the next layer, then

W will be of dimension $S_{in} \times S_{out}$

b will be a vector with S_{out} elements.

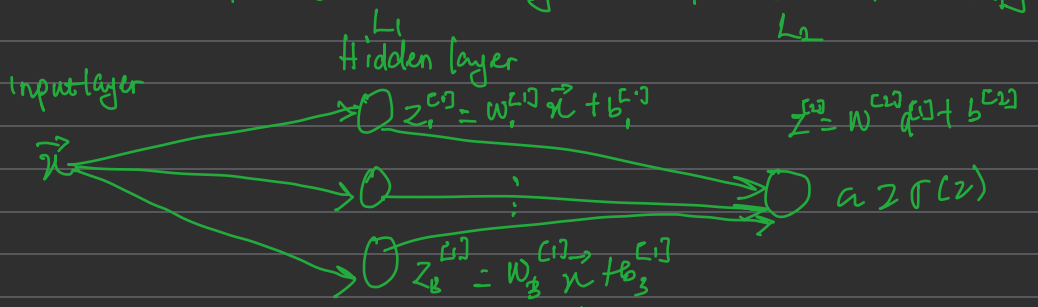
We know $A^{[L]}$ is a vector with S_{out} elements. and $A^{[L-1]}$ is a vector with S_{in} elements.

$$A^{[L]} = W^T A^{[L-1]} + b$$

Let's check the matrix multiplication will go:

$$(S_{in}, S_{out})^T \cdot (S_{in}, 1) = (S_{out}, 1) \longrightarrow \text{matches} \checkmark$$

Task: Implement a 2-layer NN from scratch in Numpy.



$$L = \frac{1}{m}\sum_{i=1}^{m}\left[-y\log\hat{y}^{(i)} - (1-y)\log(1-\hat{y})\right]$$

$z^{[1]} \to z_1$, $z^{[2]} \to z_2$

$$\hat{y} = \sigma(z_2) = \sigma(w_2 a_1 + b_2)$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial w_2} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial w_2}, \quad \frac{\partial \hat{y}}{\partial z^2} = \hat{y}(1-\hat{y})$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(\frac{-y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\cdot\hat{y}(1-\hat{y})\cdot a_1\right]$$

forgive me for not putting all the i's in $\sum$

$$= \frac{1}{m}\sum_{i=1}^{m}(\hat{y}-y)\cdot a_1$$

$$\frac{\partial L}{\partial b_2} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}-y), \quad z_2 = w_2\sigma(z_1) + b_2$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial w_1} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial w_1} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial z_1}\cdot\frac{\partial z_1}{\partial w_1}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(\frac{-y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\cdot\hat{y}(1-\hat{y})\cdot w_2\cdot a_1(1-a_1)\cdot x\right]$$

$$= \frac{1}{m}\sum_{i=1}^{m}\underbrace{\left[(\hat{y}-y)\cdot w_2\cdot a_1(1-a_1)\right]}_{dz_1}\cdot x$$

$$\frac{\partial L}{\partial b_1} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}-y)\cdot w_2\cdot a_1(1-a_1)$$

What if we have multiple layers? Is there a closed form?

Consider the n layered network:

$$\hat{y} = \sigma(z_n) = \sigma(W_n A_{n-1} + b_n)$$

$$A_{n-1} = \sigma(z_{n-1}) = \sigma(W_{n-1} A_{n-2} + b_{n-1})$$

Take the loss function $\mathcal{L} = -y \log \hat{y} - (1-y) \log \hat{y}$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{-y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

$$\frac{\partial \mathcal{L}}{\partial w_n} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_n} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_n} \cdot \frac{\partial z_n}{\partial w_n} = (\hat{y} - y) A_{n-1}$$

$$\frac{\partial \mathcal{L}}{\partial b_n} = \hat{y} - y$$

$$\frac{\partial \mathcal{L}}{\partial w_{n-1}} = (\hat{y} - y) \cdot W_n \cdot A_{n-1}(1 - A_{n-1}) \cdot A_{n-2}$$

$$\frac{\partial \mathcal{L}}{\partial b_{n-1}} = (\hat{y} - y) \cdot W_n \cdot A_{n-1}(1 - A_{n-1})$$

The main thing is finding $\partial \hat{y} / \partial z_k$ for some k

Well we have: $\frac{\partial \hat{y}}{\partial z_{n-1}} = W_n (\hat{y} - y) \cdot A_{n-1}(1 - A_{n-1})$

Now let's find $\frac{\partial \hat{y}}{\partial z_{n-2}}$ and see if there is a pattern.

$$\frac{\partial \hat{y}}{\partial z_{n-2}} = \frac{\partial \hat{y}}{\partial z_{n-1}} \cdot \frac{\partial z_{n-1}}{\partial z_{n-2}} \quad \left(z_{n-1} = W_{n-1} A_{n-2} + b_{n-1} \text{ and } A_{n-1} = \sigma(z_{n-1}) \right)$$

$$= \frac{\partial \hat{y}}{\partial z_{n-1}} \cdot \frac{\partial z_{n-1}}{\partial A_{n-2}} \cdot \frac{\partial A_{n-2}}{\partial z_{n-2}}$$

$$= W_n (\hat{y} - y) \cdot A_{n-1}(1 - A_{n-1}) \cdot W_{n-1} \cdot A_{n-2}(1 - A_{n-2})$$

$$= W_{n-1} W_n (\hat{y} - y) \cdot A_{n-2}(1 - A_{n-2}) \cdot A_{n-1}(1 - A_{n-1})$$

Then we can see a pattern:

$$\frac{\partial \ell}{\partial z_k} = (\hat{y} - y) \cdot \prod_{j=k+1}^{n} W_j \cdot \prod_{j=k}^{n-1} A_j(1 - A_j)$$

and

$$\frac{\partial \ell}{\partial W_k} = \frac{\partial \ell}{\partial z_k} \cdot A_{k-1}$$

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial z_k}$$

We can see the closed form has overlapping terms. Each gradient contains products that earlier gradients also contain. If we always compute from scratch, we will always redo some multiplications many times.

We can compute dz and then reuse it in the next layer.

$$dz_k = (\hat{y} - y) \cdot \prod_{j=k+1}^{n} W_j \cdot \prod_{j=k}^{n-1} A_j(1 - A_j)$$

$$dz_{k+1} = (\hat{y} - y) \cdot \prod_{j=k+2}^{n} W_j \cdot \prod_{j=k+1}^{n-1} A_j(1 - A_j)$$

$$\frac{dz_k}{dz_{k+1}} = W_{k+1} \cdot A_k(1 - A_k) \Rightarrow dz_k = \left(W_{k+1}^T \, dz_{k+1}\right) \odot A_k(1 - A_k)$$

($\odot$: element wise mul, not dot.)

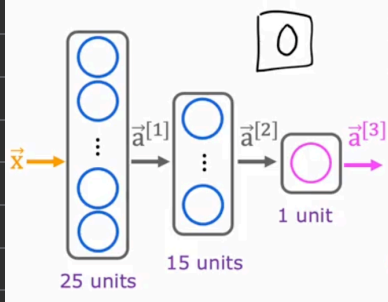
Implementation here:

https://colab.research.google.com/drive/1hHemUJ3s-m8jnZ4jlawQ2C2XKZbwAUKX?usp=sharing

Neural Network Training

Train a neural network in Tensorflow

Train a Neural Network in TensorFlow




```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
   model = Sequential([
     Dense(units=25, activation='sigmoid'),
     Dense(units=15, activation='sigmoid'),
     Dense(units=1, activation='sigmoid'),
                    ])
from tensorflow.keras.losses import
BinaryCrossentropy
   model.compile(loss=BinaryCrossentropy())
   model.fit(X,Y,epochs=100)
```

① ② ③

epochs: number of steps in gradient descent

Given set of (x,y) examples

How to build and train this in code?

Model training Steps

Model Training Steps

Tensor Flow

① specify how to compute output given input x and parameters w,b (define model)

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

② specify loss and cost

$L(f_{\vec{w},b}(\vec{x}), y)$ 1 example

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

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

logistic regression

```
z = np.dot(w,x)+ b
f_x = 1/(1+np.exp(-z))
```

logistic loss

```
loss = -y * np.log(f_x)
   -(1-y) * np.log(1-f_x)
```

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

neural network

```
model = Sequential([
          Dense(...)
          Dense(...)
          Dense(...)
                ])
```

binary cross entropy

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

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

Training – Definitions and concepts.

Training any neural network follows the same three steps as logistic regression, just at a larger scale:

* Specify the model: define how to compute the output given input and parameters.
* Specify the loss and cost function: define what "good" means
* Train on data to minimize the cost: adjust parameters to reduce loss

```
# Step 1: Specify the model
model = Sequential([
    Dense(units=25, activation='sigmoid'),
    Dense(units=15, activation='sigmoid'),
    Dense(units=1, activation='sigmoid')
])

# Step 2: Specify the loss
model.compile(loss=BinaryCrossentropy())

# Step 3: Train
model.fit(X, y, epochs=100)
```

Loss function: measures how wrong the model is on a single training example.
Cost function: Average of the loss across all training example.

Activation functions: non-linear function applied to the output of each neuron. Without it, stacking layers would collapse into one linear transformation.

Sigmoid

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

* Outputs between 0 and 1
* Used for binary classification at the output layer
* Suffers from vanishing gradients in deep networks.
* Rarely used in hidden layers today.

ReLU (Rectified Linear Unit)

$$g(z) = \max(0, z)$$

* Outputs 0 for negative inputs, the input itself for positive
* Default choice for hidden layer
* Much faster to compute than sigmoid
* Does not suffer from vanishing gradients
* Can have "dead neurons" if many inputs are negative.

Linear (no activation)

$$g(z) = z$$

* Used for regression outputs that can be any real number.
* Equivalent to no activation function.

Softmax

$$g(z)_k = \frac{e^{z_k}}{\sum_i e^{z_i}}$$

* Used for multi-class classification at the output layer.
* Converts a vector of scores into probabilities summing to 1.

Choosing the Output activation

The output activation depends on the type of problem:

Problem type	Output activation
Binary classification	Sigmoid
Multiclass classification	Softmax
Regression (any real number)	Linear
Regression (non-negative)	ReLU

Choosing hidden layer activations

For hidden layers, the default is ReLU. Reasons are:

* faster to compute than sigmoid (just a comparism vs exponential)
* Doesn't saturate on the positive side, so gradients don't vanish.
* In practice, networks with ReLU hidden layers train faster and reach better accuracy.

Why we need activation functions

Without non-linear activation, a deep network reduces to a single linear transformations:

$$Z_2 = W_2 Z_1 + b_2 = W_2(W_1 X + b_1) + b_2 = (W_2 W_1)X + (W_2 b_1 + b_2)$$

Which is just another linear function, a single layer can represent it. The network gains no expressive power from depth.

Non-linear activations break this collapse. They let the network learn arbitrarily complex functions by composing many simple non-linear pieces.

Multiclass classification

When we have more than two classes, we switch from sigmoid binary cross-entropy to softmax + categorical cross-entropy.

Softmax converts logits into a probability distribution over K classes:

$$P(y=k|x) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Categorical cross-entropy measures how far the predicted distribution is from the true (one-hot) distribution.

$$L = -\sum_{k=1}^{K} y_k \log(\hat{y}_k)$$

```
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=10, activation='linear')  # 10 classes
])

model.compile(loss=SparseCategoricalCrossentropy(from_
logits=True))
```

Note: using linear activation at the output and from_logits = True in the loss is numerically more stable than computing softmax separately. The loss function handles both internally.

Less stable (computes softmax then log separately)

```
Dense(units=10, activation='softmax')
loss=SparseCategoricalCrossentropy()
```

The output of the network is now in "logit" form (raw scores), and the loss handles conversions to probabilities internally. To get probabilities at prediction time, apply softmax manually:

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

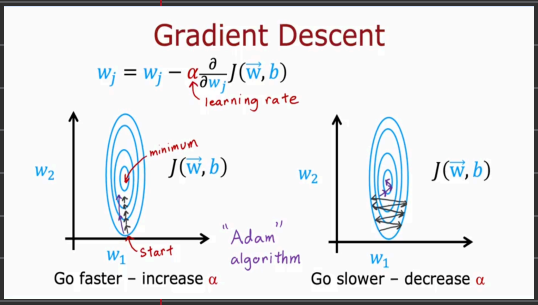
Adam Optimizer (Adaptive moment estimation)

Instead of plain gradient descent with a fixed learning rate, Adam automatically adapts the learning rate per parameter based on past gradients.

```
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss=BinaryCrossentropy()
)
```

Why Adam?

* Adapts the learning rate for each weight individually
* Larger learning rate for weights with consistent gradient direction
* Smaller learning rate for noisy or oscillating gradients.
* Generally converges faster and more reliably than vanilla SGD



Epochs, batches and iterations

* Epoch is one complete pass through the entire training dataset.
* Batch (or mini-batch) is a small subset of the training data used for one gradient descent update. (Common sizes 32, 64, 128.
* Iteration is one gradient update on one batch.

Number of iterations per epoch = samples / batch_size

```
model.fit(X, y, epochs=100, batch_size=32)
```

More epoch = more passes through the data. Larger batches = more stable gradients but slower per epoch.

Forward propagation

Compute the network's output from its input, layer by layer.
For layer l

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}, \quad A^{[l]} = g^{[l]}(Z^{[l]})$$

where:
$A^{[l-1]}$ is the previous layer's activation (or input for layer 1)
$W^{[l]}$ and $b^{[l]}$ are this layer's parameters
$g^{[l]}$ is this layer's activation function.
$Z^{[l]}$ is the pre activation
$A^{[l]}$ is the post activation, passed onto the next layer.

Backpropagation

Computing the gradient of the cost with respect to every weight in the network. We use chain rule from calculus to propagate the error backward from the output to the input.

$$dz^{[l]} = \left(W^{[l+1]T} dz^{[l+1]}\right) \circ g'^{[l]}(z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dz^{[l]} \left(A^{[l-1]}\right)^T$$

$$db^{[l]} = \frac{1}{m} \sum dz^{[l]}$$

If we have N nodes and P parameters, backprop computes derivatives in N+P steps rather than N×P steps.

Weight Initialization

How we set the initial values of weights matters enormously.

If we use zeros, all neurons compute the same thing, so they all update identically.

If we use `W = np.random.randn(...) * 0.01` - this works for shallow networks but can cause vanishing or exploding gradients in deeper ones.

Xavier initialization (for sigmoid/tanh)

$$W = N(0,1) \times \sqrt{\frac{1}{n^{[l-1]}}}$$

He initialization (Best for ReLU)

$$W = N(0,1) \times \sqrt{\frac{2}{n^{[l-1]}}}$$

Building a binary classifier

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import
BinaryCrossentropy
from tensorflow.keras.optimizers import Adam

# 1. Specify the model
model = Sequential([
    Dense(units=25, activation='relu'),
    Dense(units=15, activation='relu'),
    Dense(units=1, activation='sigmoid')
])

# 2. Specify the loss
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss=BinaryCrossentropy(),
    metrics=['accuracy']
)

# 3. Train
model.fit(X_train, y_train, epochs=100,
validation_data=(X_val, y_val))

# 4. Predict
predictions = model.predict(X_test)
```

Multilabel Classification

In multi-label classification, each example can belong to multiple classes at the same time. Each label is independent, we predict yes or no for each one.

Contrast this with multi-class classification, where each example belongs to exactly one of K classes.

* Multi-class uses softmax because outputs must sum to 1, picking one class means rejecting the others.
* Multi-label uses sigmoid on each output independently, saying yes to cat doesn't affect whether we say yes to dog.

For K labels, the output layer has K neurons, each with sigmoid activation:

$$\hat{y}_k = \sigma(z_k) \quad \text{for } k = 1, 2, \ldots, K$$

Each output is interpreted as the probability that label K is present, independent of others.

```
model = Sequential([
    Dense(units=64, activation='relu'),
    Dense(units=32, activation='relu'),
    Dense(units=K, activation='sigmoid')   # K sigmoid
outputs, not softmax
])
```

Binary cross-entropy is applied independently to each label then averaged

$$L = \frac{-1}{K} \sum_{k=1}^{K} \left[y_k \log(\hat{y}_k) + (1 - y_k) \log(1 - \hat{y}_k) \right]$$

```
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss=BinaryCrossentropy(),
    metrics=['accuracy']
)
```

The labels are multi-hot encoded instead of one-hot encoded for multiclass.

Additional Layer Types.

In dense layers, each neuron output is a function of all the activation outputs of the previous layer.

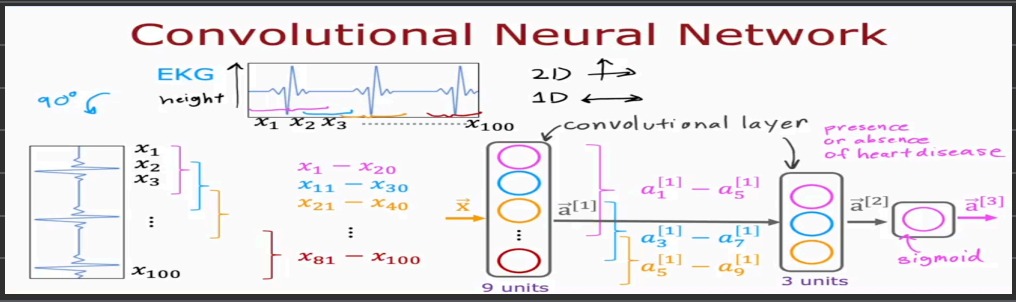
$$\vec{a}_1^{[2]} = g\left(\vec{w}_1^{[2]} \vec{a}^{[1]} + b^{[2]}\right)$$

Convolutional layers, each neuron only looks at part of the previous layer's output.

But why would we want to do this?

* Faster computation
* Need less training data (less prone to overfitting)

If we have multiple convolutional layers in a neural network we call that network a convolutional neural network (CNN).



Advice for Applying Machine Learning.

When the model performs poorly, what should we actually do? There are many possible next steps:

* Collecting more training data.
* Try a smaller set of features.
* Add more features.
* Add polynomial features.
* Decrease the regularization parameter λ
* Increase λ
* Make the model bigger
* Train longer.

Train / validation / test split:
We can split the data into three sets:

* Training set - the data the model learns from
* Validation set (also called dev set or cross-validation set) - used to tune hyperparameters and pick between models.
* Test set - Used only once at the very end to estimate true generalization.

Typical splits:

Dataset size	Train	Val	Test
Small (~10k)	60%	20%	20%
Large (millions)	98%	1%	1%

We use three sets and not two because if we use the test set to pick hyperparameters, we have effectively trained on it. It's accuracy is no longer an honest measurement of generalization. The validation set is the data we are allowed to look at when making decisions.

Bias and Variance

* Bias is how much the model underfits

* Variance is how much the model overfits

High variance means the model has memorized training-set noise and fails on new data. It overfits.

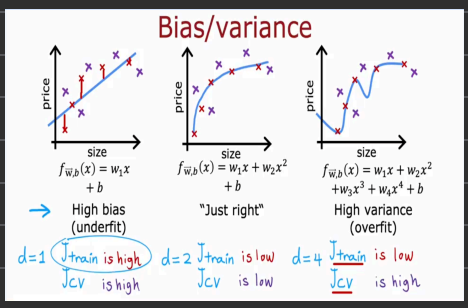
We diagnose these by comparing training error and validation error.

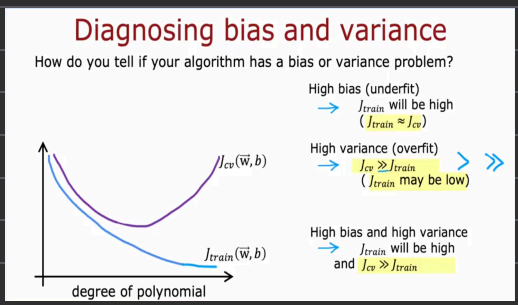
Training error	Validation error	Diagnosis
High	High	High bias (underfitting)
Low	High	High variance (overfitting)
Low	Low	Good fit
High	Even higher	High bias and high variance

Baseline on performance! We can't judge if training error is high or low without context. We compare against:

* Human-level performance — how well do people do this task?
* Simple baseline — What does a basic model achieve
* Best published performance — What's the state of the art (SOTA)?

For example, 10% error on speech recognition sounds bad, but if humans also get 10% on the same audio (because of background noise), the model is fine. The gap that matters is between the model and achievable best.





Diagnostic procedure

Given training error, validation error and a baseline

gap_bias = training_error − baseline
gap_variance = validation_error − training_error

* Large gap_bias → underfitting
* Large gap_variance → overfitting

Both can happen at once.

Fixing high bias (underfitting)

* Add more features
* Add polynomial features
* Use a more complex model (more layers, more units)
* Decrease the regularization parameter λ
* Train longer.

Note that adding more training data does not fix high bias. The model isn't expressive enough to fit even what it has.

Fixing high variance (overfitting)

* Get more training data
* Reduce the number of features
* Increase the regularization parameter
* Use a simpler model
* Use dropout (for neural networks)
* Use early stopping

Quick reference table

Symptom	Diagnosis	What to try
High train error, high val error	High bias	Bigger model, more features, less regularization
Low train error, high val error	High variance	More data, fewer features, more regularization
Both low	Healthy	Done, possibly try a smaller model for efficiency

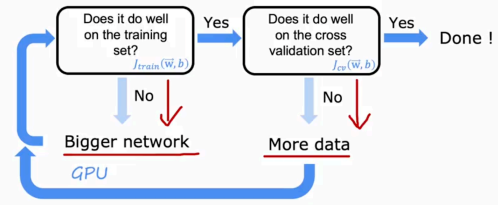
Bias/Variance and neural networks

For classical ML, bias and variance are in tension. We trade one against the other by adjusting model complexity.

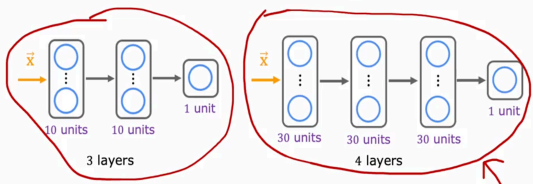
Neural networks change this trade-off, with enough data and proper regularization, a large network can reduce both bias and variance simultaneously

Neural networks and bias variance

Large neural networks are low bias machines



Neural networks and regularization



A large neural network will usually do as well or better than a smaller one so long as regularization is chosen appropriately.

Neural network regularization

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

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])
```

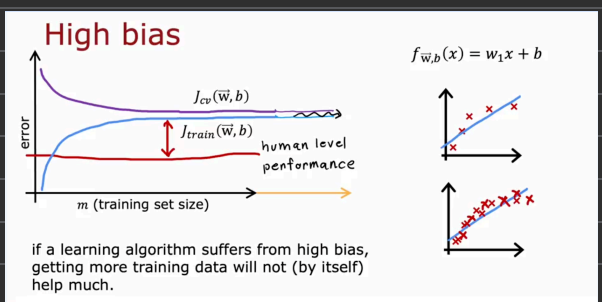
Learning Curves

One downside is that it is computationally expensive to train so many different models using different size subset of training set. It isn't done that much in practice

We can plot the training error and validation error as a function of training set size.

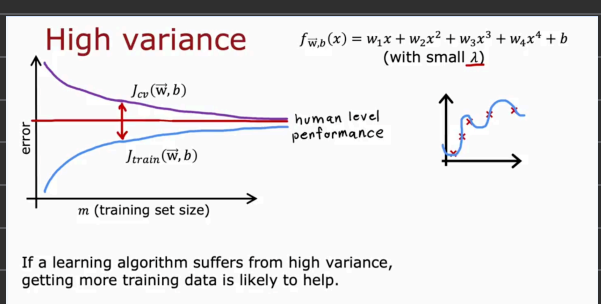
High bias pattern!

* Both curves plateau at a high error
* The gap between them is small
* Adding more dat won't help, both curves are already flat.



High variance pattern!

* Training error is low and stays low
* Validation error decreases as you add data.
* The gap closes as data grows, more data will help.



The curves are diagnostic and we should run them before deciding to collect more data.

Regularization

Adding a penalty to large weights to discourage overfitting.
Cost function with L2 regularization

$$J(w,b) = \frac{1}{m}\sum L(y,\hat{y}) + \frac{\lambda}{2m}\sum w^2$$

Tuning λ:

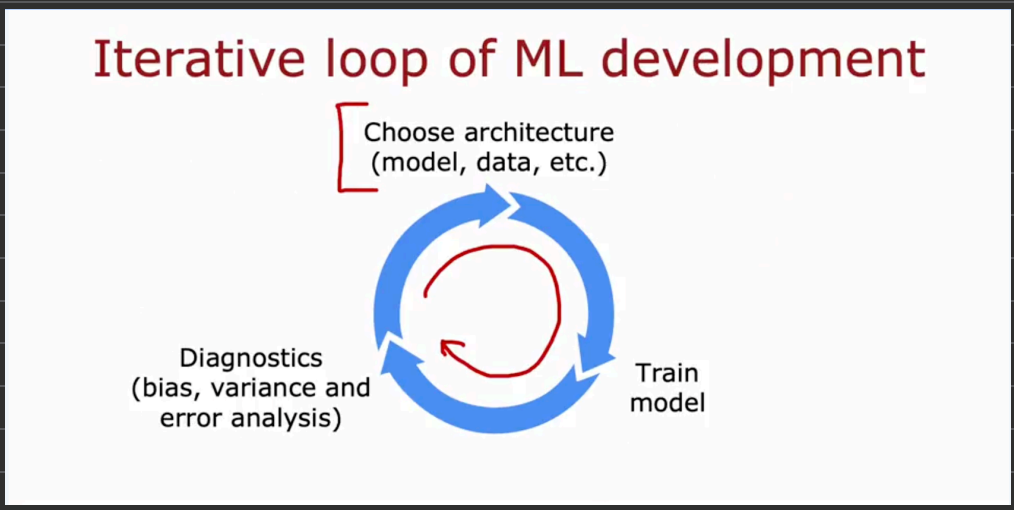
* $\lambda = 0$ → no regularization. Model can overfit freely. High variance.
* λ small → Slight regularization. Doesn't restrict the model much.
* λ moderate → Sweet spot. Model fits data without overfitting noise.
* λ very large → over regularized. All weights forced near zero. High bias

Iterative loop of ML development.

The whole development process is iterative:

* Choose architecture (model type, features, hyperparameters)
* Train the model
* Diagnose errors (high bias? high variance? specific failure modes?)
* Use diagnostics to decide what to change.
* Go back to the first step.

We don't get everything right on the first try. The bias/variance make iteration efficient.



Error Analysis

After training, we can manually look at examples the model gets wrong. We can categorize them, for instance, if we are building a spam filter and we have 500 misclassified examples in the validation set:

* 50 are spam but classified as not spam (false negatives)
* 200 are not spam but classified as spam (false positives)

Then we can look at false positives:

* 30% are pharmaceutical ads
* 20% are stolen passwords/phishing
* 15% are unusual punctuation patterns
* 10% are deliberately misspelled words.

We can see phising detection is more impactful than mispellings. Then we know the problem and we can try fixing the biggest categories.

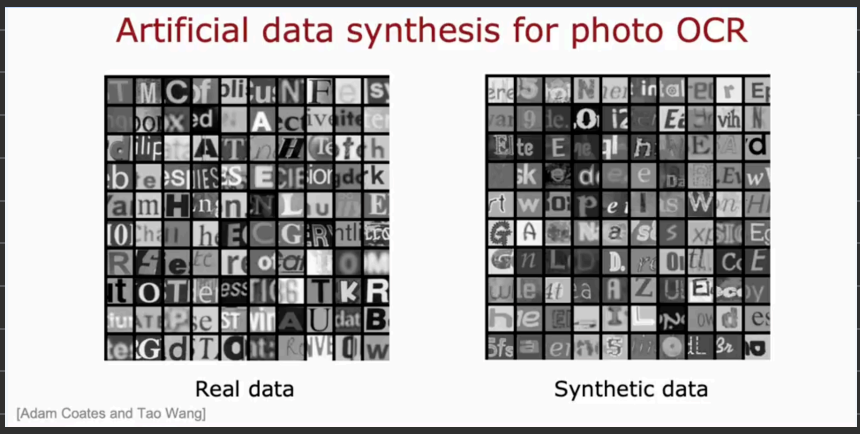
Error analysis is hard for tasks humans are not good at, for example when we try to predict clicks on a website.

Adding more data - Selectively.
If error analysis reveals one category dominates errors, we can target data collection at that category instead of just collecting more of everything.

Data augmentation (for image and audio) creates new training examples from existing ones:

* Rotate, crop or flip images
* Add noise to audio
* Apply distortions that don't change the label

Data synthesis uses artificial data inputs to create new training data (synthetic data).
Let's say for OCR we need synthetic data, we can use different fonts to create random texts and screenshot it using different colors, contrast and fonts and we get synthetic data like that.



Data synthesis is mostly used for computer vision tasks.

This is much cheaper than collecting genuinely new labeled data

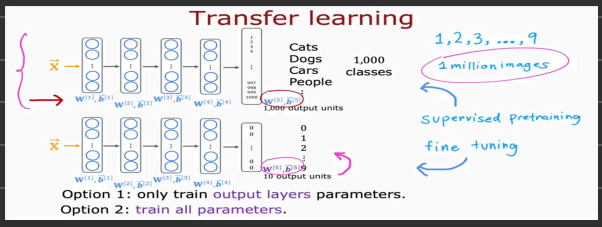
Transfer Learning

When we have a small dataset, we can train on a related large dataset first (supervised pretraining) then fine-tune on our smaller dataset.

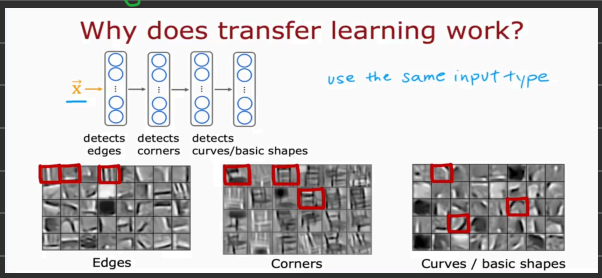
Two approaches:

* Use a pretrained model as a feature extractor. We replace just the output layer. We train only the new output layer's weights. Useful when we have very little data.

* Fine-tune the whole network. We initialized with pretrained weights, then train all weights on our data with a small learning rate. Useful when we have more data.



Why this works: Lower layer learns general features (edges in image, common words in text) that transfer across tasks. Higher layer learns task-specific things. By starting with good lower layers, we only need data for the task-specific learning.

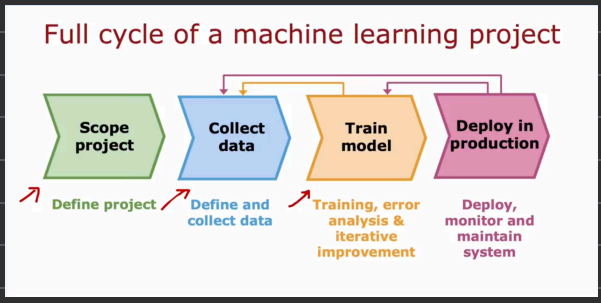


This is one of the most impactful techniques in practical deep learning.

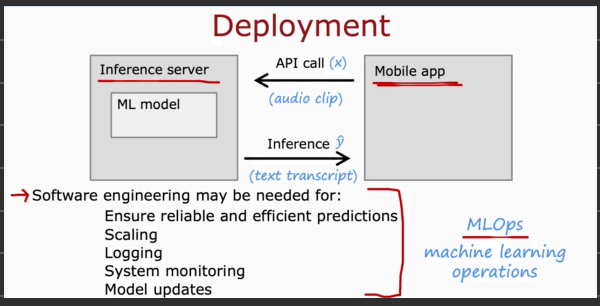
Full ML workflow:

* Scope the project. What's the problem? What does success look like?
* Collect data. Gather and label training data.
* Train a model. Pick architecture, train, evaluate.
* Deploy and monitor. Get the model into production, watch for drift and failures.

We iterate between these. We can go back to collect more dat after a model fails in production.



Deployment



Fairness, Bias and Ethics.

Bias

Hiring tool that discriminates against women.

Facial recognition system matching dark skinned individuals to criminal mugshots.

Biased bank loan approvals.

Toxic effect of reinforcing negative stereotypes.

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.

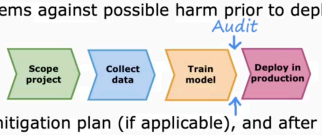
Spam vs anti-spam : fraud vs anti-fraud.

Guidelines

Get a diverse team to brainstorm things that might go wrong, with emphasis on possible harm to vulnerable groups.

Carry out literature search on standards/guidelines for your industry.

Audit systems against possible harm prior to deployment.



Develop mitigation plan (if applicable), and after deployment, monitor for possible harm.

Skewed Datasets.

When one class is rare (for example, 1% of patients have a disease), accuracy is misleading. A model that predicts "no disease" for everyone gets 99% accuracy but it is useless.

We should:

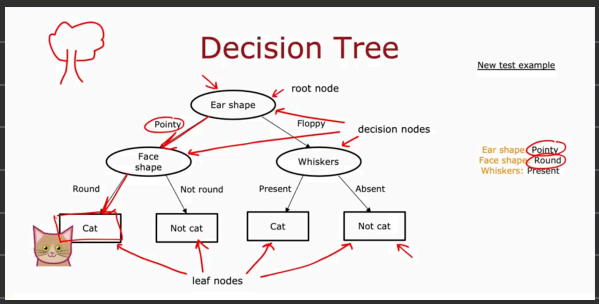
* Precision = TP/(TP+FP) - when model predicts positive, how often is it right?
* Recall = TP/(TP+FN) - of all actual positives, how many did we catch?
* F1 score - harmonic mean of precision and recall.

The right trade-off depends on the cost of errors:

* Medical screening → optimize recall (don't miss real cases)
* Spam filter → optimize precision (don't block real emails)

Decision Trees.

A decision tree is a model that makes prediction by asking a series of yes/no questions about the input feature. Each internal node tests one feature; each branch is a possible answer; each leaf is a final prediction.



Learning process:
- the tree is built greedily - top down, one split at a time.

1. Start at the root with all training examples
2. Pick the best feature to split on (to max purity or min impurity)
3. Partition the data based on that feature
4. Repeat the same process on each child node.
5. Stop when a criterion is met (pure node, max depth, etc).

The key question is which feature should we split on at each node?
The answer comes down to pick the split that produces the purest child nodes.

When a node is 100% one class
When splitting a node will result in tree exceeding a maximum depth
When improvements in purity score are below a threshold
When number of examples in a node is below a threshold.

Measuring Purity (Entropy)

A node is pure when all its examples belong to the same class. A node is impure when classes are mixed.

Entropy measures impurity:

$$H(p_1) = -p_1 \log_2(p_1) - (1-p_1)\log_2(1-p_1)$$

Where p_1 is the fraction of positive examples in the node.

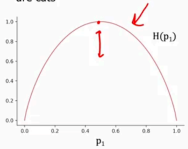
Key properties:

* $H(0.5) = 1$, maximum entropy, 50/50 split, most impure
* $H(0) = 0$ and $H(1) = 0$, all examples in one class, pure, zero entropy
* The curve looks like an inverted U peaking at $p_1 = 0.5$

By convention $0 \log_2 0 = 0$

Entropy as a measure of impurity

p_1 = fraction of examples that are cats



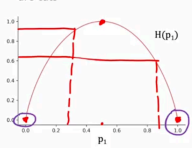
$$p_0 = 1 - p_1$$

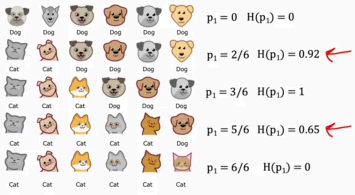
$$H(p_1) = -p_1 log_2(p_1) - p_0 log_2(p_0)$$
$$= -p_1 \log_2(p_1) - (1-p_1) log_2(1-p_1)$$

Note: "0 log(0)" = 0

Entropy as a measure of impurity

p_1 = fraction of examples that are cats





Choosing a split: Information gain

When we split a node, we reduce entropy. The amount of reduction is information gain.

For a split that produces a node with weights w^{left} and a right node with weight w^{right}:

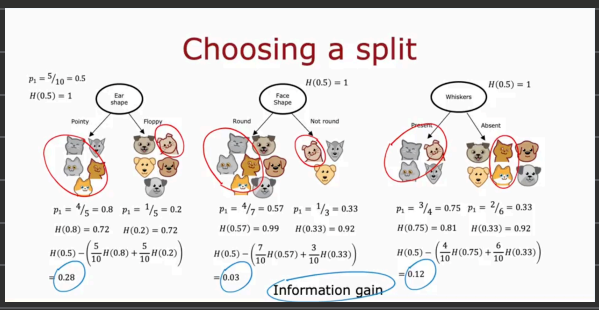
$$\text{Information Gain} = H(p_1^{node}) - \left[w^{left} H(p_1^{left}) + w^{right} H(p_1^{right})\right]$$

Where:

* p_1^{root} is the fraction of positive examples at the parent node.
* w^{left} and w^{right} are the fractions of examples going to each child
* p_1^{left} and p_1^{right} are the fraction of positives in each child

The split with the highest positive gain is chosen. This is the splitting criterion.

Why did we compute reduction in entropy instead of entropy? The stopping criteria for deciding when to not bother to split any further is if the reduction in entropy is too small. (one of the criteria)



Stopping criteria

Without a stopping rule, the tree keeps splitting until every node is pure, which overfits.

Common stopping conditions:

* A node is fully pure (all examples in one class)
* Reaching a maximum depth
* The information gain from any further split is below a threshold
* The number of examples in a node is below a minimum.

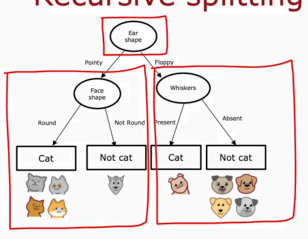
These act as regularization, they prevent the tree from memorizing training data.

Whole learning process

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

Recursive splitting



Recursive algorithm

```
build_tree(node):
    if stopping condition is met:
        return leaf with majority class

    best_feature = feature with highest information gain
    split the node on best_feature

    for each child node:
        build_tree(child)
```

Recursive Algorithm

One-hot encoding for categorical features

If a feature has more than two possible values (e.g. ear shape: pointy, floppy, oval), the tree can't split into 3 branches. We can convert it to multiple binary features instead.

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

One hot encoding →

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

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

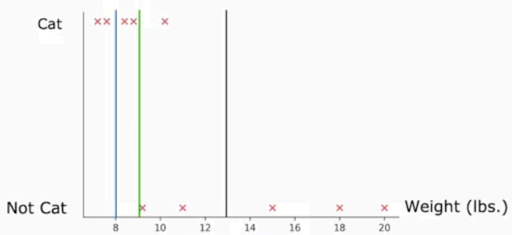
Continous valued features

For continous features (e.g weight = 8.5 kg), the tree tries different threshold values and picks the one with highest information gain. The split becomes weight < threshold → yes/no.

Procedure:

1. Sort examples by the feature value
2. Try each midpoint between consecutive distinct values as a candidate threshold
3. Compute information gain for each
4. Pick the threshold with the best gain.

Splitting on a continuous variable



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

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

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

Regression Trees

Decision trees can predict continous values too and not just classes. The differences:

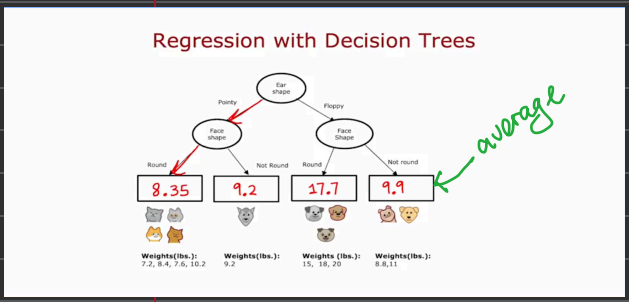
* Each leaf predicts the average of the target values of examples in that leaf.

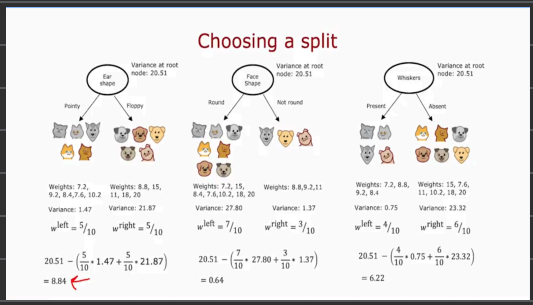
* The splitting criterion is the variance reduction instead of information gain.

For a split:

$$\text{Variance Reduction} = \text{Var}(\text{parent}) - \left[w^{\text{left}} \cdot \text{Var}(\text{left}) + w^{\text{right}} \cdot \text{Var}(\text{right}) \right]$$

The split that reduces the variance the most is chosen.

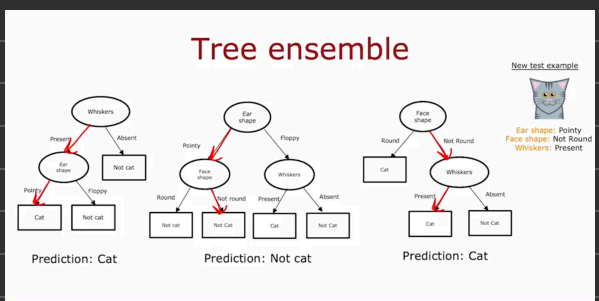




Tree ensembles

Decision trees are sensitive, small changes in training data can produce very different trees. This high variance is a problem.

Tree ensembles combine many trees to reduce variance. It is basically a collection of trees



We vote and majority carries the vote.

Two main approaches:

* Random forest: bagging + random feature selection
* Boosted trees (XGBoost): sequential trees focused on misclassified examples.

Sampling with replacement (bootstrap)

To build an ensemble, we need different trees. We create different training sets by sampling with replacement from the original data, pick a random example, put it back, pick again. Some examples might be picked multiple times, some won't be picked at all.

Each tree in the ensemble gets its own bootstrap sample this gives the forest diverse training data without needing more data overall

Random Forest

A random forest is an ensemble of decision tree with two sources of randomness:

* Bootstrap sampling: Each tree trained on a different bootstrap sample.
* Random feature: At each node, we only consider a random subset of features (typically $\sqrt{n}$ for classification)

For prediction:

* Classification → majority vote across trees
* Regression → average of trees predictions.

The random feature subset is what distinguishes random from plain bagging. It forces trees to be more diverse, reducing correlation between them. More diversity → better variance reduction when averaging.

Bagging

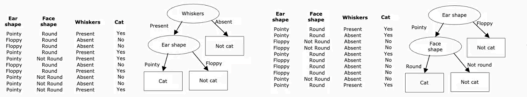
Generating a tree sample

Given training set of size m

For b = 1 to B ⇒ 64 ~ 128 for B

Use sampling with replacement to create a new training set of size m

Train a decision tree on the new dataset



Bagged decision tree

Boosted Trees (XGBoost)

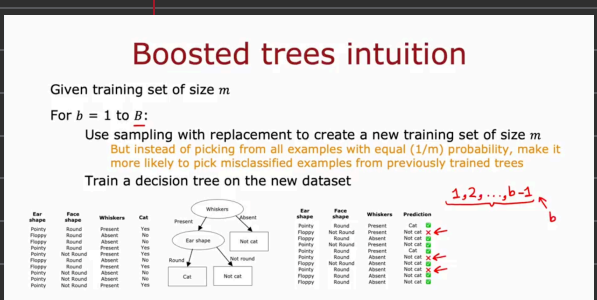
Unlike random forest where trees are trained independenty in parallel, boosting trains trees sequentially. Each new tree focuses on the example that previous trees got wrong.

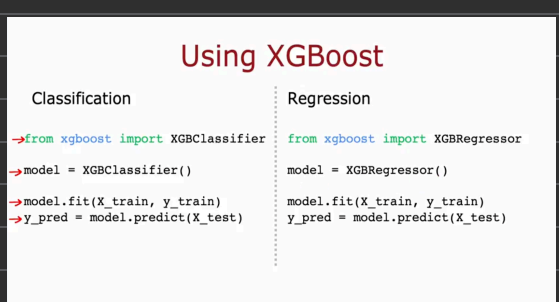
Process:

* Train a tree on the data
* Identify examples the current ensemble misclassifies
* Train the next tree with more weight on those misclassified examples.
* Add new tree to the ensemble
* Repeat.

XGBoost (Extreme Gradient Boosting) is the most popular implementation. It's fast, regularized and dominates many ML competitions on tabular data.

```
from xgboost import XGBClassifier
model = XGBClassifier()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```





When to use decision trees?

Decision trees (and ensembles) work well when:

* Data is tabular / structured (rows and columns)
* We need interpretability
* Training time matters (they train faster)
* Small to moderate dataset sizes
* Mixed feature types (numerical + categorical)

Neural networks

* Works on all sorts of data (whether structured or not)
* May be slower than a decision tree
* Works with transfer learning
* When building a system of multiple models working together, it might be easier to string together multiple neural networks