Course 1

Neural Networks:

Origins: Algorithms that try to mimic the brain.

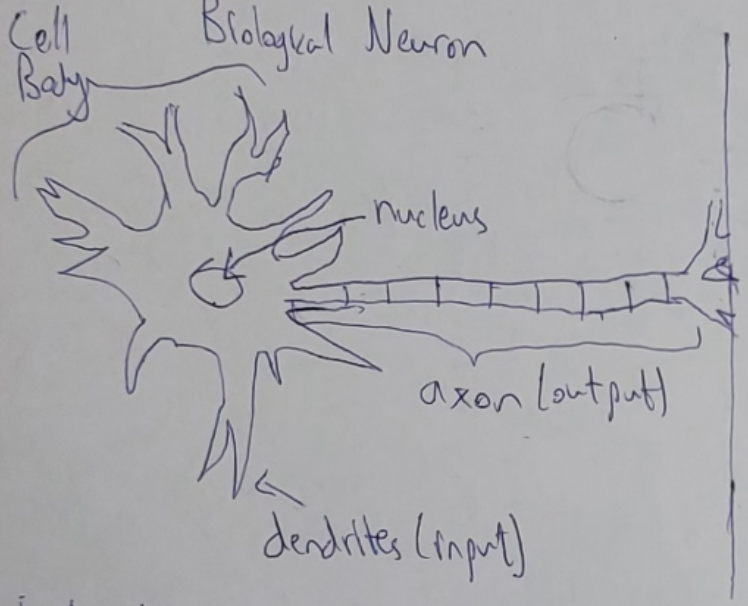
↗ 1980's and early 1990's.

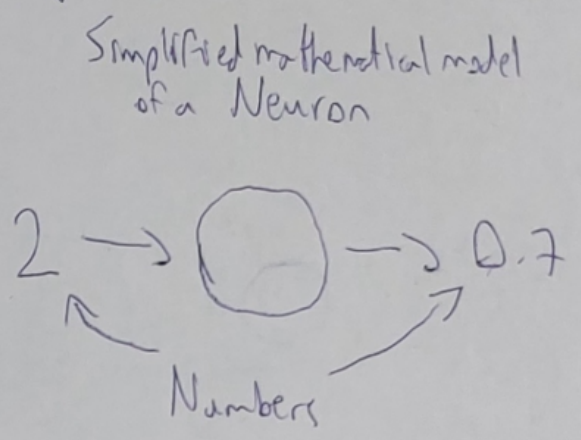
↘ late 1990's

↗ ~2005

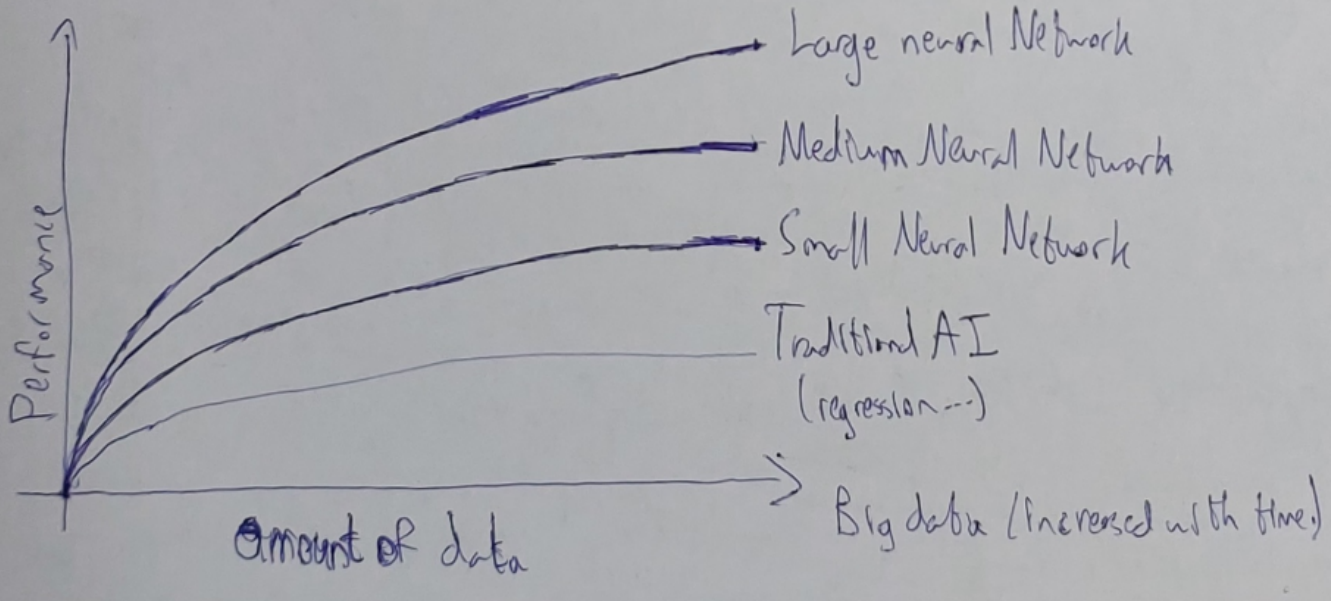
speech → images → text (NLP) → ...

Today's neural networks have almost nothing to do with how the brain learns.



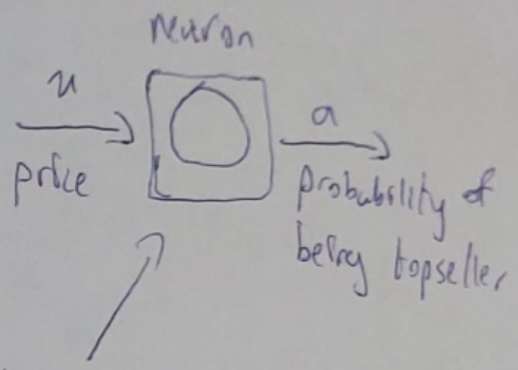


Why Now? 2nd ans: Faster processors, GPUs



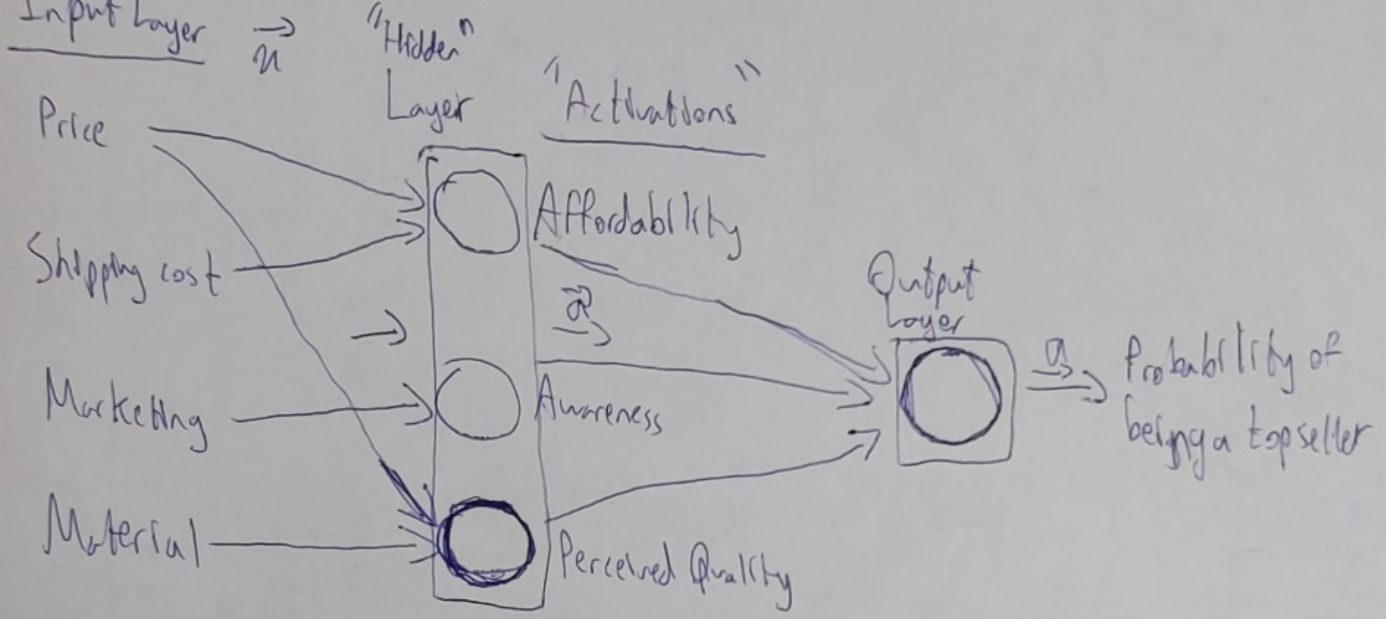
$$a = f(z) = \frac{1}{1 + e^{-(wx+b)}}$$

↑ activation (comes from neuroscience)



Think of it as a tiny little computer that has 1 task

Demand Prediction (4 features)



4 numbers 3 numbers 1 number

Neural Networks don't work like this. Imagine you're building a large neural network and having to manually decide which neuron should take which features as inputs. In practice, each neuron in a certain layer will have access to every feature to every value from the previous layer. It should learn what to ignore and what to consider. Think of it as if the neuron does feature engineering and get numbers that are more predictive of the output.

You do not decide what other features it should compute in the hidden layer.

You only choose the number of Hidden layers and the number of neurons in them.

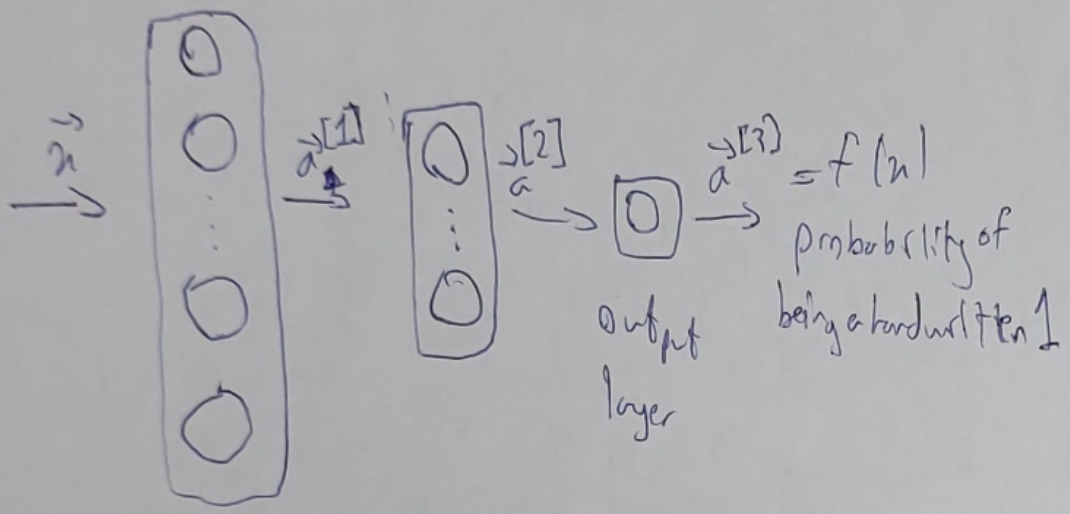
Multilayer Perceptron: Neural Network with more than 1 hidden layer.

Notation for layer numbering: $a^{[k]}$, where a is a quantity associated with layer k.

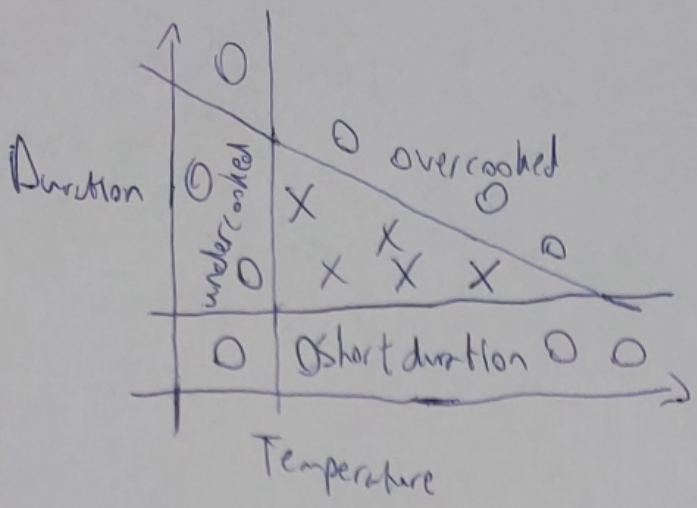
$a_j^{[l]} = g\left(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]}\right)$; where l is the layer.

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

Inferencing: making predictions



Forward Propagation



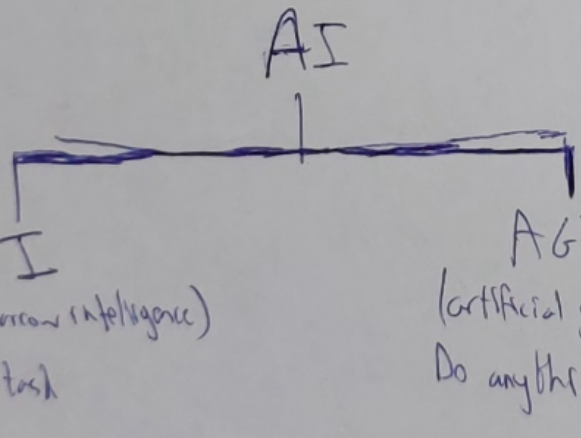

```
x = np.array([[200.0, 17.0]])
layer_1 = Dense(units=3, activation='sigmoid')
a1 = layer_1(x)

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

Using `model = Sequential([layer_1, layer_2])`

we are telling Tensorflow that we want to string the 2 layers sequentially.

Then run
```
model.compile(...)
model.fit(x,y)
model.predict(x_new)
```



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)
```

epochs = # steps in gradient descent

Binary crossentropy is the logistic loss function:

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

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

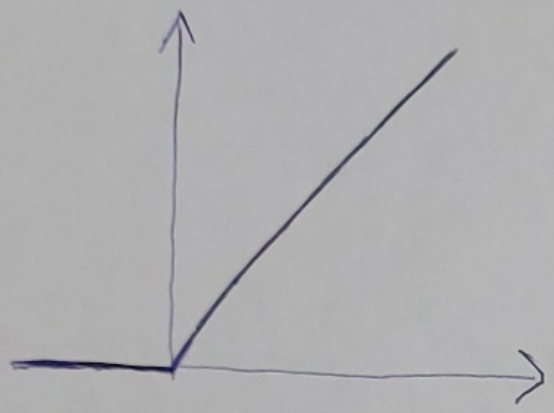
$W^{[1]}$ $W^{[2]}$ $W^{[3]}$ $\vec{b}^{[1]}$ $\vec{b}^{[2]}$ $\vec{b}^{[3]}$

(all W's in all layers)

* Uppercase for matrices

Tensorflow computes the partial derivatives of the cost function using "back propagation" (to minimize J)

Suppose you want an activation to be any non-negative number, not just between 0 and 1 (output of sigmoid function). So we need another activation function:



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

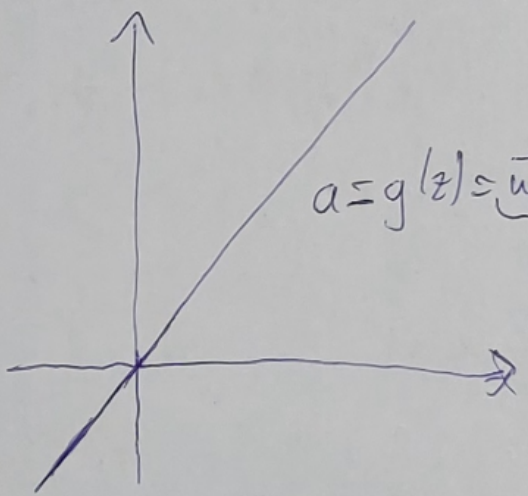
ReLU (Rectified Linear Unit)

Another common example is the linear activation function:

$g(z) = z$

(Ok (come on now do I really have to sketch ~~graph~~ it?))

"No activation function"



Which function should I use?

For the Output Layer:

If you're doing: Binary Classification $\Rightarrow$ sigmoid (0/1)

Regression $\Rightarrow$ Linear (+/-)
(with something like stocks)

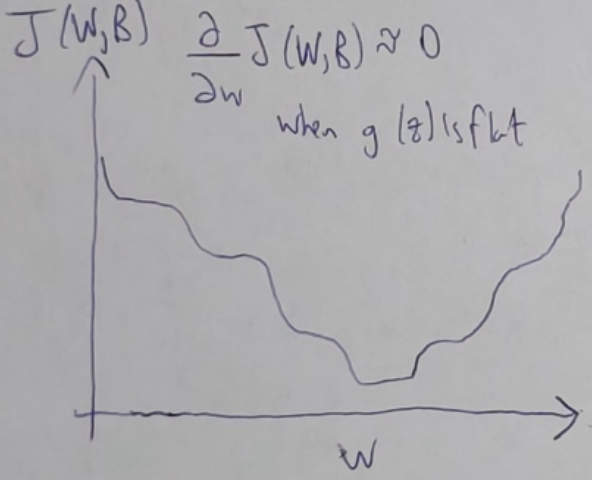
Regression $\Rightarrow$ ReLU
(with non-negative values like house prices)

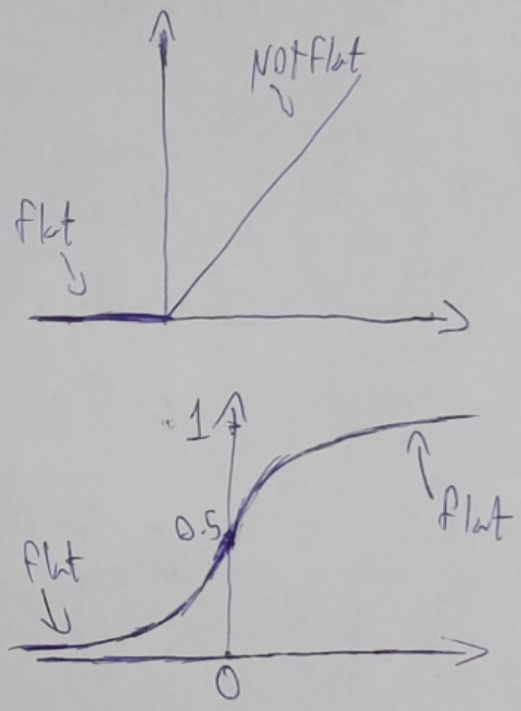
For Hidden Layers:

ReLU is the most common choice. Initially, the sigmoid function was used, but the field has evolved to use ReLU much more often.

Why is that?

- ReLU is faster to compute because you're just computing $\max(0, z)$, while sigmoid requires taking exponentiation, then inverse... so it is a bit less efficient
- ReLU goes "flat" in one part of the graph (left) while the sigmoid function goes flat in 2 places (both directions



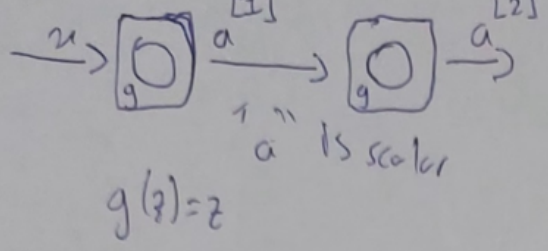


This slows down Gradient Descent

- Researchers have found that using ReLU activation functions can cause neural networks to learn a bit faster.

Other activation functions: tanh, LeakyReLU, swish,

Why do neural networks need activation functions?



$$a^{[1]} = w_1^{[1]} x + b_1^{[1]}$$

$$a^{[2]} = w_1^{[2]} a^{[1]} + b_1^{[2]}$$

$$= w_1^{[2]} \left(w_1^{[1]} x + b_1^{[1]} \right) + b_1^{[2]}$$

$$= \underbrace{\left(w_1^{[2]} w_1^{[1]} \right)}_{w} x + \underbrace{w_1^{[2]} b_1^{[1]} + b_1^{[2]}}_{b}$$

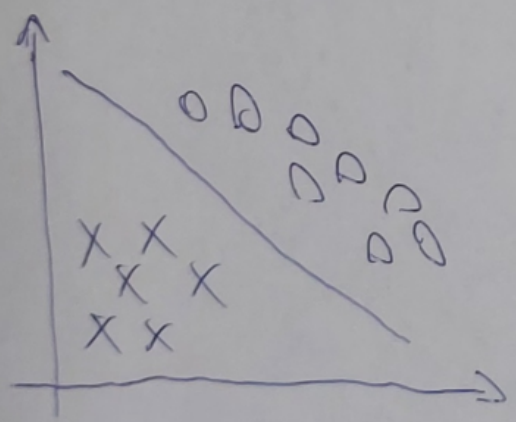
So $a^{[2]} = wx + b$: a linear function.

We could've just used a linear regression model.

That's why in linear algebra, a linear function of a linear function is itself a linear function. This is why having multiple layers in a neural network with linear activation functions doesn't let the neural network compute any more complex features or learn anything more complex than a linear function.

A common rule of thumb: Don't use linear activation functions in hidden layers (use ReLU)

Multi class classification: classification with more than just 2 possible output labels.



Softmax Regression Algorithm: a generalization of logistic regression to the multiclass classification contexts.

Logistic regression (2 possible output values)

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

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

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

Softmax Regression (example with 4 possible outputs):

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

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

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

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

Generalization (N outputs):

$z_j = \vec{w}_j \cdot \vec{x} + b_j \quad j = 1, \dots, N$

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

Note: $a_1 + a_2 + \cdots + a_N = 1$

Logistic Regression

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

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

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

$loss = -y \log a_1 - (1-y) \log(\underbrace{1-a_1}_{a_2})$

so if $y=1$: $loss = -\log a_1$

if $y=0$: $loss = -\log a_2$

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

Softmax regression

$$a_1 = \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}+\cdots+e^{z_N}} = P(y=1|\vec{x})$$

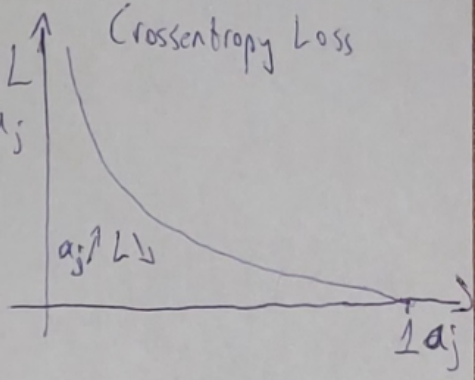
$$\vdots$$

$$a_N = \frac{e^{z_N}}{e^{z_1}+e^{z_2}+e^{z_3}+\cdots+e^{z_N}} = P(y=N|\vec{x})$$

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

$\therefore$ if $y=j$:

$loss = -\log a_j$



To be more accurate: do not compute intermediate terms

In logistic regression: The calculations will be more accurate in code if the loss is:

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

Tensorflow can rearrange terms in this expression and come up with a more numerically accurate way to compute the loss function.

Some procedure goes to softmax.

Another thing to do is to set the output layer to use a linear activation function and add from_logits = True as an argument to the imported loss function.

logit: z

Now the final output layer no longer outputs probabilities $a_1 \cdots a_N$ but is outputting $z_1 \cdots z_N$.

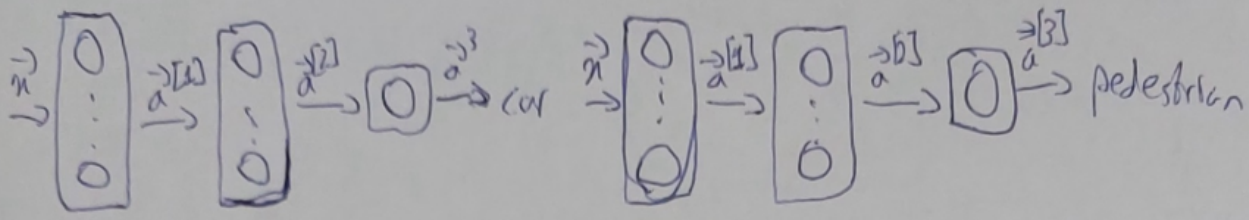
To solve this, write these after model.fit(...), when you're trying to predict:

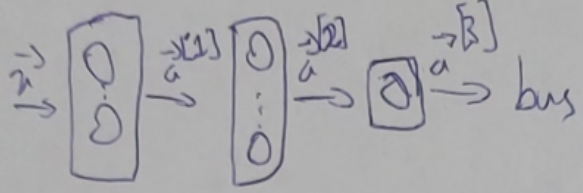
```
logits = model(X)
f_x = tf.nn.softmax(logits)  #softmax can be replaced with sigmoid...
```

Multi-label Classification: There could be multiple labels for one input.

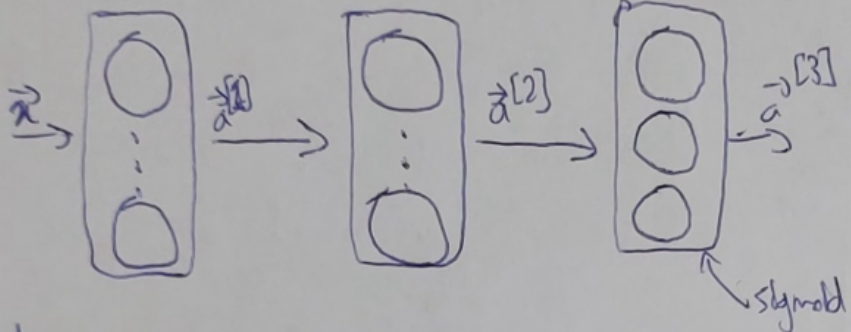
Example: a cat and a dog in one picture (input).

An alternative is to build multiple neural networks to detect your targets.



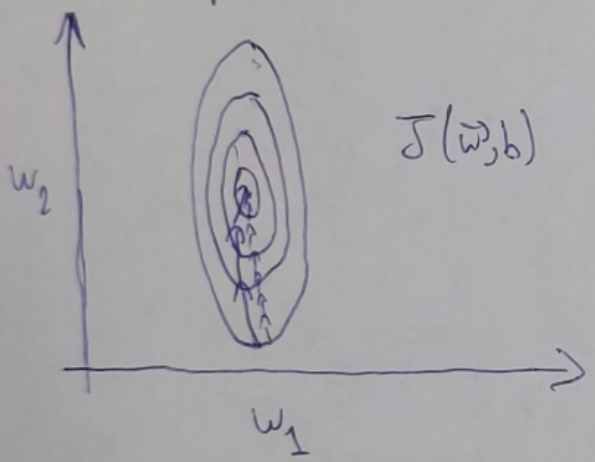


Alternative:

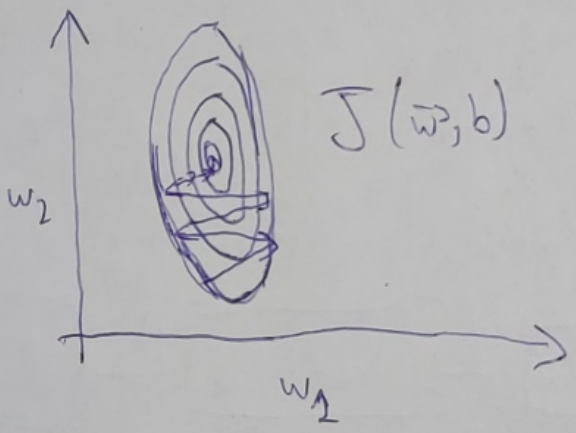


$$\vec{a}^{[3]} = \begin{bmatrix} a_1^{[3]} \\ a_2^{[3]} \\ a_3^{[3]} \end{bmatrix} \begin{matrix} \text{car/no car} \\ \text{bus/no bus} \\ \text{pedestrian/no pedestrian} \end{matrix}$$

Advanced Optimization



If w_j (or b) keeps moving in the same direction, increase α_j



If w_j (or b) keeps oscillating, reduce α_j

This is what the Adam Algorithm does.

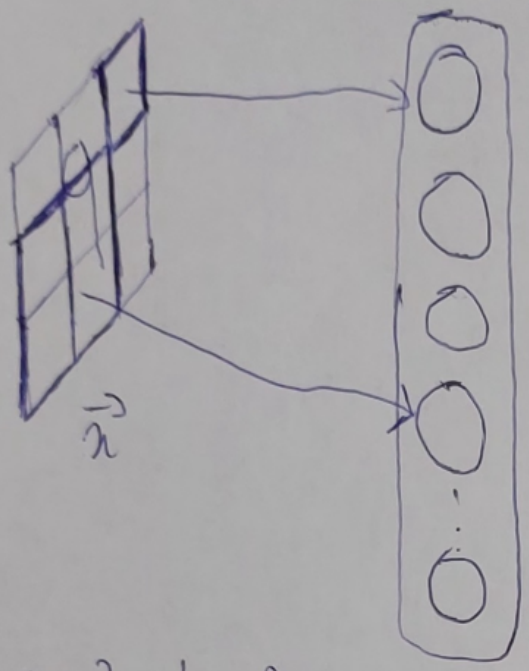
ADAM = Adaptive Moment Estimation

To use it just add:

optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3),

to model.compile(...)

Convolutional Layer:



Each neuron only looks at a part of the previous layer's outputs.

Why?

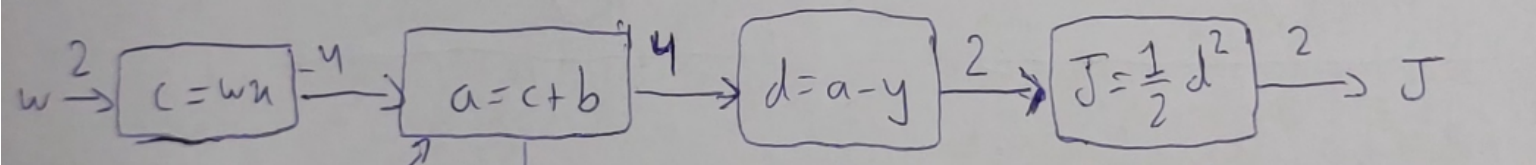
- Faster Computation
- Need less training data (less prone to overfitting)

Let: $w=2$ $b=8$

$x=-2$ $y=2$

$a = wx+b$ linear activation: $a = g(z) = z$

$J(w,b) = \frac{1}{2}(a-y)^2$



computation graph: forward prop

$w \uparrow 0.001$ $c \downarrow 0.002$

$c = -2 \times 0.001$

so $J \uparrow -4 \times 0.001$

so $\frac{\partial J}{\partial w} = -4$

$c \uparrow 0.001$ $a \uparrow 0.001$ then $J \uparrow 0.002$

$b \uparrow 0.001$ $a \uparrow 0.002$

so $J \uparrow 0.002$

$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial c} = 2$

$a \uparrow 0.001$ $d \uparrow 0.001$

so $J \uparrow 0.002$

$\frac{\partial J}{\partial a} = 2$

$d \uparrow \underbrace{0.001}_{\varepsilon}$ $J \uparrow \underbrace{0.002}_{2\varepsilon}$

then $\frac{\partial J}{\partial d} = 2$

What we basically did was chain rule:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial d} \cdot \frac{\partial d}{\partial a} \cdot \frac{\partial a}{\partial c} \cdot \frac{\partial c}{\partial w}$$

Backprop computes derivatives in roughly $N+P$ steps rather than $N \times P$, where N is the number of nodes and P is the number of parameters.

Debugging:

- Get more training examples	high variance
- Try smaller sets of features	high variance
- Try getting additional features	high bias
- Try adding polynomial features ($x_1^2, x_2^2, x_1 x_2$, etc)	high bias
- Try decreasing λ	high bias
- Try increasing λ	high variance

Evaluating your model:

Divide your data into a training set (70%-80%) and a test set (30%-20%)

Compare between $J_{test}(\vec{w}, b)$ and $J_{train}(\vec{w}, b)$ which are basically the cost function without the regularization term.

If $J_{train}(\vec{w}, b)$ is much smaller than $J_{test}(\vec{w}, b)$, you have overfitting. (not so good at generalizing)

When working with classification, measure the fraction of the test set and the fraction of the training set that the algorithm has misclassified and compare them.

How to automatically choose a good model?

If your model is overfit, the training error $J_{train}(\vec{w}, b)$ is likely lower than the actual generalization error.

$J_{test}(\vec{w}, b)$ is a better estimate of how well the model will generalize to new data than $J_{train}(\vec{w}, b)$

Divide up the dataset into 60% training set, 20% cross-validation and 20% test set.

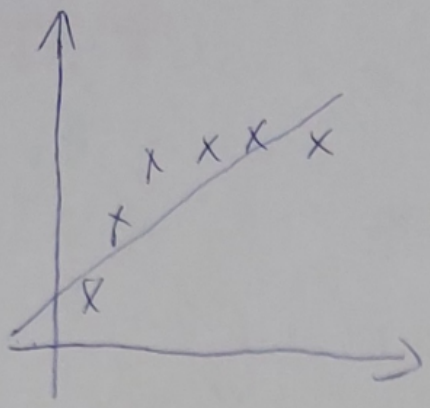
The cross validation set is an extra dataset that we're going to use to cross check the validity or accuracy of different models. Other names: validation set, development set, dev set.

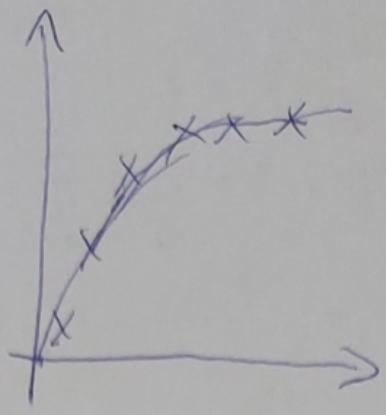
Compare $J_{cv}(\dots)$ of different models (degree = 1, 2, 3 ... N) and choose the model with the lowest error.

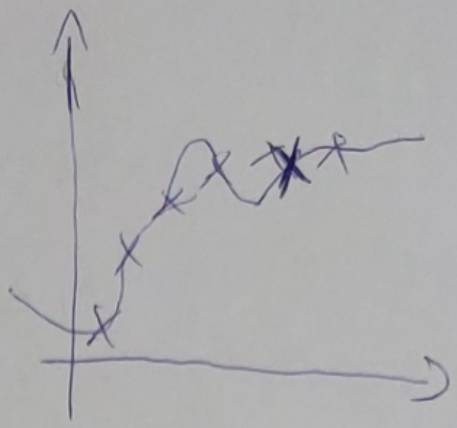
Estimate generalization error using $J_{test}(\vec{w}, b$ chosen by lowest $J_{cv}(\dots))$

You can also use this method on Neural Network's layers and units/layer (architecture)

Diagnostic: A test you run to gain insight into what is/isn't working with a learning algorithm to gain ~~insight~~ guidance into improving its performance.

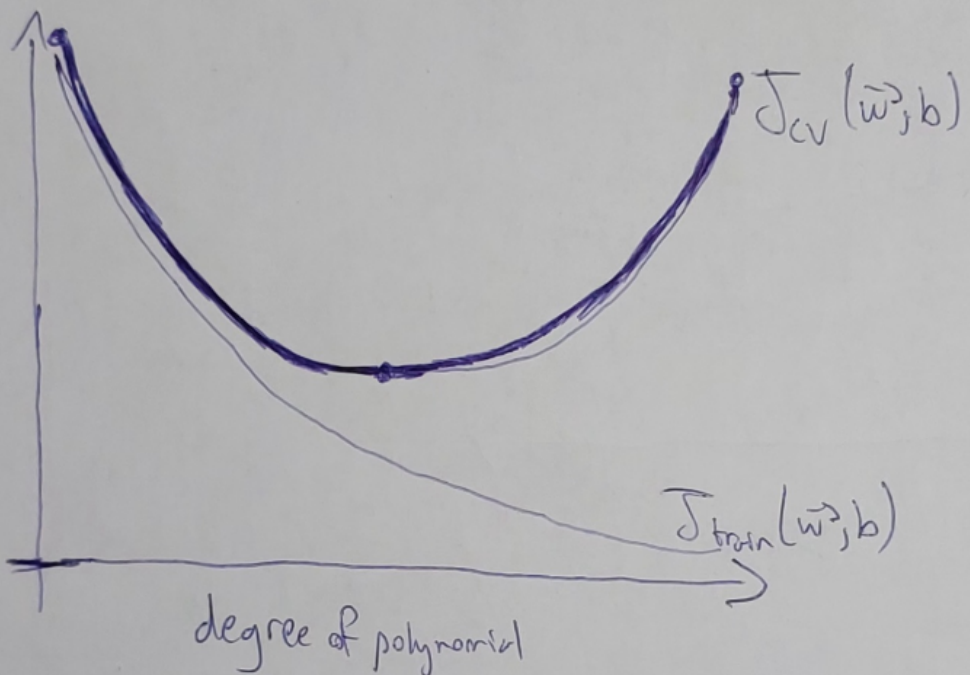






$d=1$: $f_{\vec{w},b}(x) = w_1 x + b$ $d=2$: $f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$ $d=3$: $f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

High bias/underfit	"Just Right"	High Variance/Overfit
J_{train} is high	J_{train} is low	J_{train} is low
J_{cv} is high	J_{cv} is low	J_{cv} is high



High bias: J_{train} will be high

$J_{train} \approx J_{cv}$

High Variance: $J_{cv} \gg J_{train}$

J_{train} may be low

High bias & High Variance:

J_{train} will be high

$J_{cv} \gg J_{train}$

While doing regularization, high λ would increase bias and low λ would increase variance. The graph of $J_{cv}(\vec{w},b)$ and $J_{train}(\vec{w},b)$ resemble the mirror image of the graph on the previous page (reflection in vertical axis)

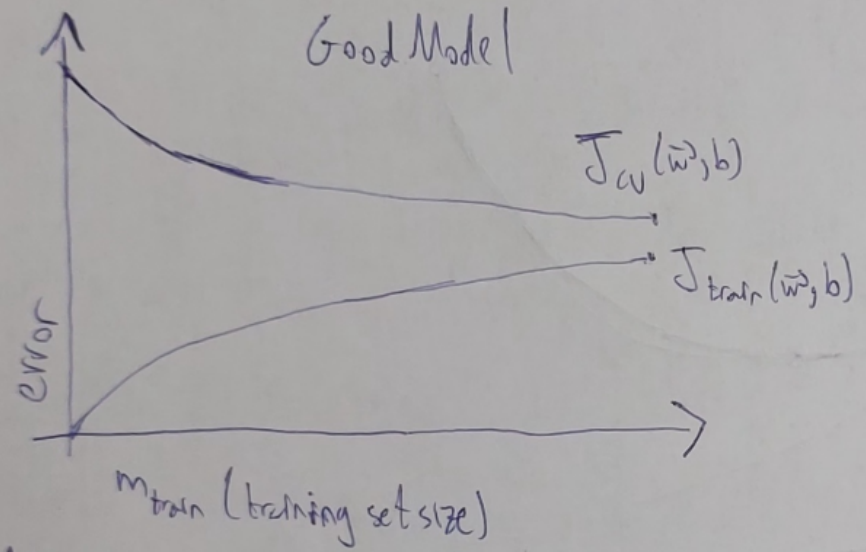
Establishing a baseline level of performance:

- Human level performance
- Competing algorithms performance
- Guess based on experience

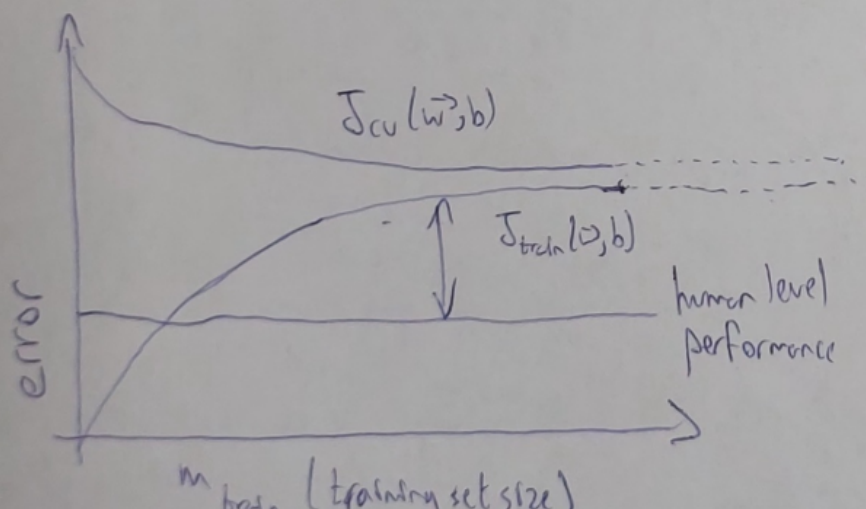
high/low bias → Baseline performance ↕ J_{train}

low/high variance → J_{cv}

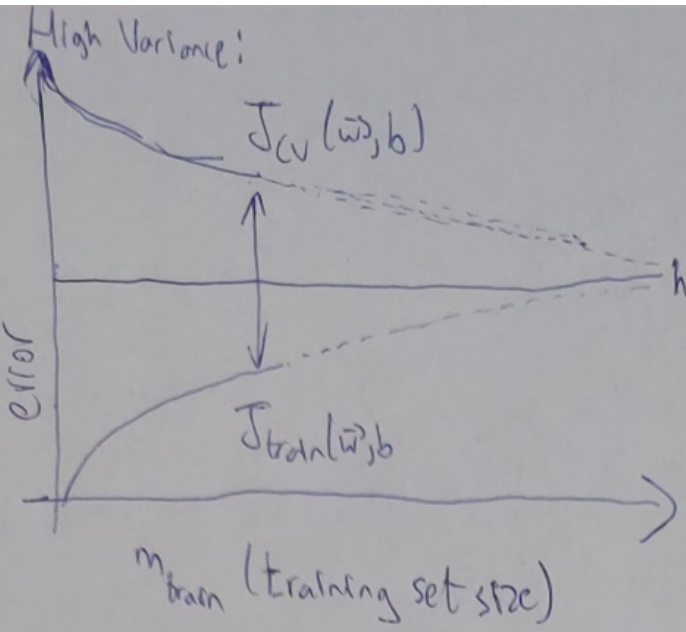
Learning Curves:



High Bias:

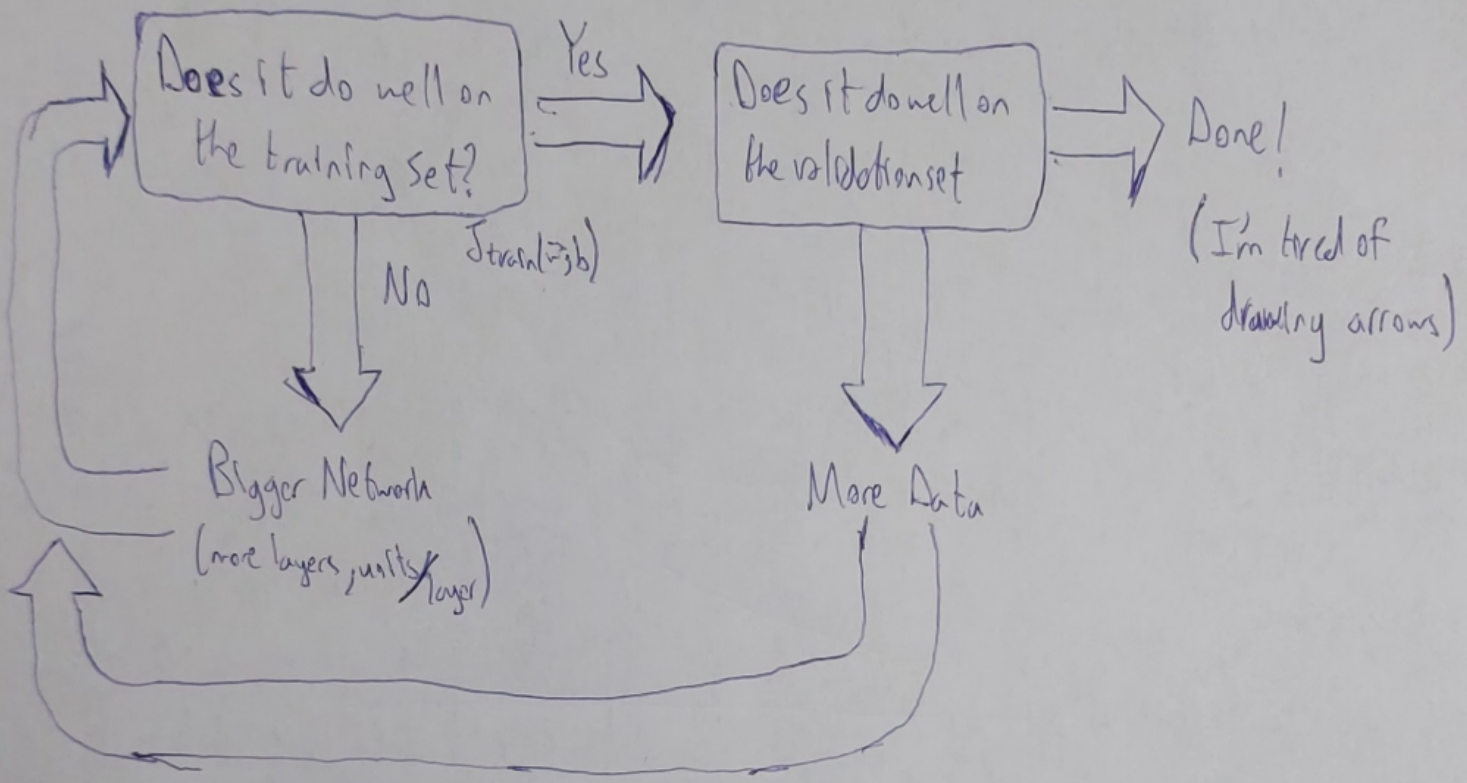


So if you have high bias, adding more training data won't help.



So adding more training data might probably help in a high variance situation

Large Neural Networks are low bias machines.



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

To regularize a neural network, add `kernel_regularizer = L2(0.01)` to the layer

(λ → 0.01)

Iterative loop of ML Development:

- Choose Architecture (model, data, etc...)
- Train model
- Diagnostics (bias, variance and error analysis)

Error analysis: 2nd to bias and variance in improving learning algorithm performance.

Augmentation: modifying/distorting/warping an existing training example to create a new training example. Examples: rotating the image of a sign, enlarging it, shrinking it, adding noise to an audio...

Take a Neural Network with 1000 output units trained on 1 million images of 1000 classes.

Only train the output layer (say you put 10 output units because that is what you want).

This process is called supervised pretraining or fine tuning.

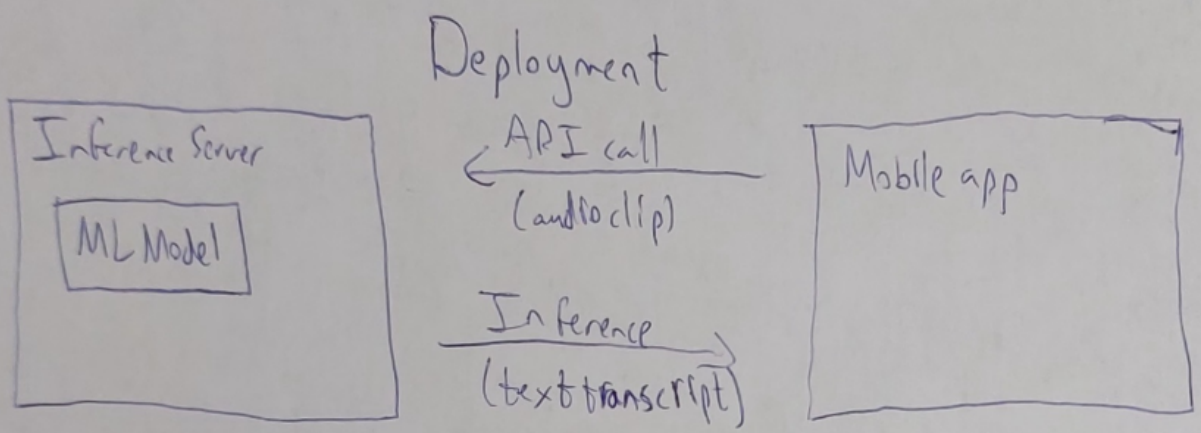
Transfer Learning Steps:

1- Download Neural Network Parameters pretrained on a large dataset with the same input type (images, audio, text...) as your application (or train your own).

2- Further train (fine tune) the network on your own data.

Full cycle of a machine learning project

- Scope project: Define it
- Collect Data
- Train model: training, error analysis & iterative improvement. Might need to go back to step #2
- Deploy in production, monitor, and maintain. Might need to go back to steps 2 and 3.

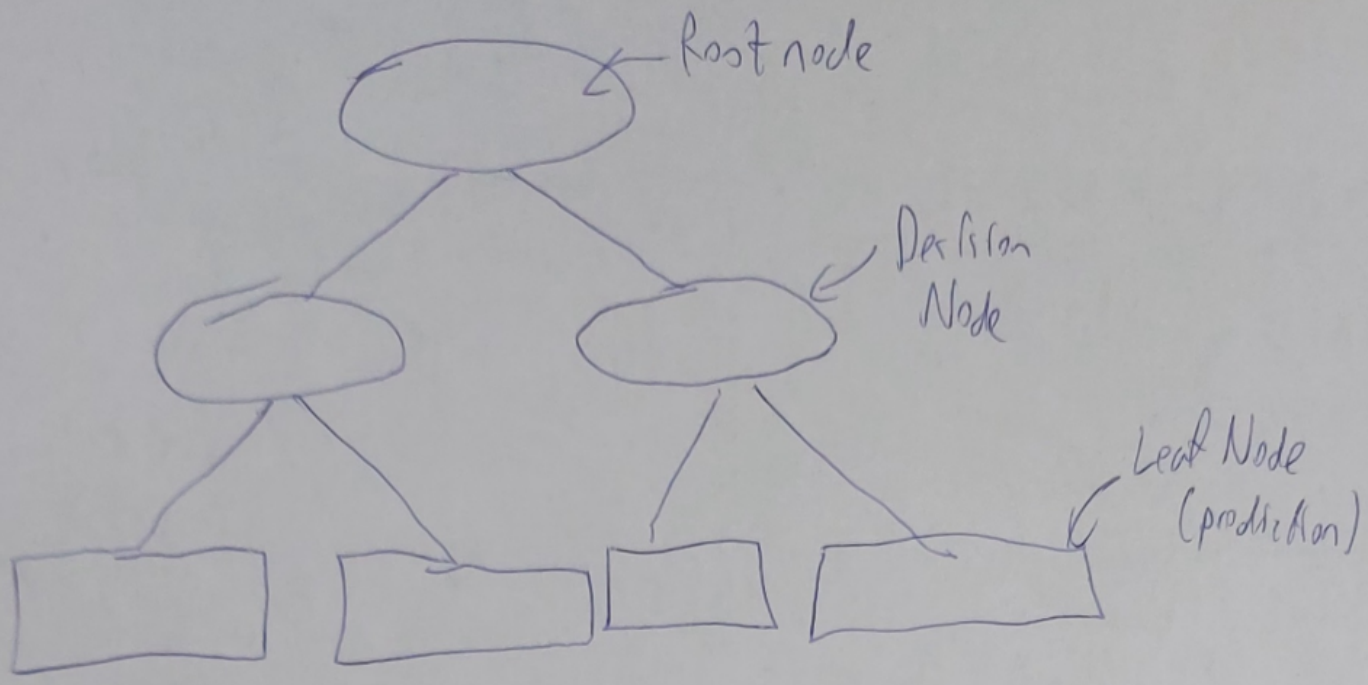


Software engineering needed for: Ensuring reliable and efficient predictions

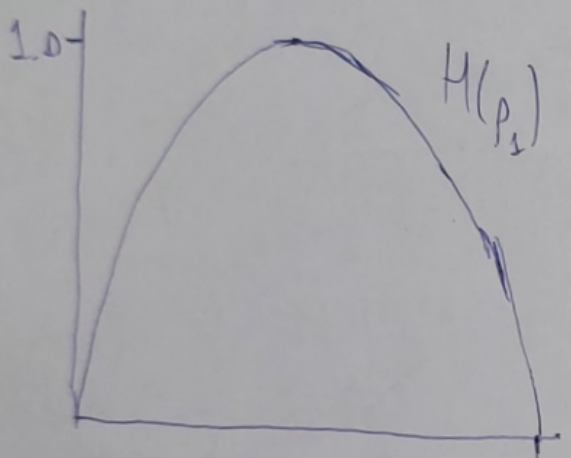
- Scaling
- Logging
- System Monitoring
- Model updates

ML Ops: Machine Learning Operations (practice of how to systematically build and deploy and maintain ML systems)

Decision Trees



Entropy as a measure of impurity:



$p_0 = 1 - p_1$

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

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

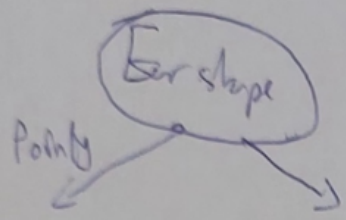
similar to logistic loss

$\log(0)$ is not defined but for the purpose of computing entropy $0 \log 0 = 0$

What feature to split at a node will be based on what choice of feature reduces entropy the most.

The reduction of entropy is called information gain.

$P_1 = \frac{5}{10} = 0.5$

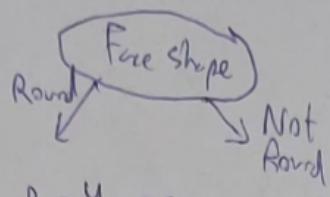


$P_2 = \frac{4}{5} = 0.8$ $\quad P_1 = \frac{1}{5} = 0.2$

$H(0.8) = H(0.2) = 0.72$

$H(0.5) - \left(\frac{5}{10} H(0.8) + \frac{5}{10} H(0.2)\right)$

$= 0.28$

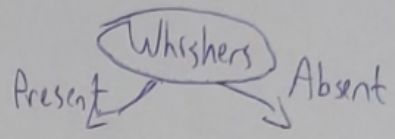


$P_1 = \frac{4}{7} = 0.57$ $\quad P_1 = \frac{1}{3} = 0.33$

$H(0.57) = 0.99$ $\quad H(0.33) = 0.92$

$H(0.5) - \left(\frac{7}{10} H(0.57) + \frac{3}{10} H(0.33)\right)$

$= 0.03$



$P_2 = \frac{3}{4} = 0.75$ $\quad P_1 = \frac{2}{8} = 0.33$

$H(0.75) = 0.81$ $\quad H(0.33) = 0.92$

$H(0.5) - \left(\frac{4}{10} H(0.75) + \frac{6}{10} H(0.33)\right)$

$= 0.12$

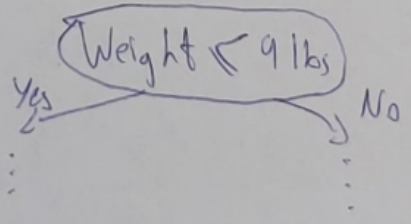
Information Gain

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

Use recursive splitting (common sense)

One hot encoding: If a categorical feature can take on k values, create k binary features (0 or 1 valued)

Splitting on a continuous variable: Just select different value for a "boundary"/limit based on the highest information gain. Then splits



Tree Ensemble (An-sambl): a collection of multiple trees.

Generating a tree sample:

Given training size m: 64-128 (recommended)/stands for bag

For b=1 to B: Use sampling with replacement to create a new training set of size m.

Train a Decision tree on the new dataset.

- This specific instantiation of a tree ensemble is also called a bagged decision tree.
- One modification would change this bagged decision tree into the random forest algorithm. This modification is randomizing the feature choice (so that we don't get similar splits at the root or near it).
- 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
- This will make the trees different from each other.
- Common value for k: $k = \sqrt{n}$

"Where does a machine learning engineer go camping?
In a random forest"

* XGBoost or Boosted decision trees perform better than random forest.
- When doing sampling with replacement to create a new training set of size m, make it more likely to pick misclassified examples from previously trained trees (instead of making all examples having $\frac{1}{m}$ probability of being chosen.)

Idea: Deliberate practice!

If you're learning to play the piano and you're trying to master a piece on the piano, rather than practicing the entire piece over and over, which is time consuming, instead play the piece then focus your attention on just the parts of the piece that you aren't yet playing that well in practice. It is a more efficient way.

XGBoost (eXtreme Gradient Boosting):

- Open source implementation of Boosted trees
- Fast & Efficient
- Good choice of default splitting criteria and criteria for when to stop splitting.
- Built in regularization to prevent overfitting
- Highly competitive algorithm for ML competitions (Kaggle competitions...)
- Assigns different weights to different training examples, so it doesn't need to generate a lot of randomly chosen training sets, making it more efficient.

Implementing it:

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

Decision Trees VS Neural Networks

Decision Trees and Tree Ensembles:

- Tabular (structured / in a spreadsheet) data
- Not recommended for unstructured data (images, audio, text)
- Fast to train
- Small Decision Trees may be Human interpretable.

Neural Networks:

- Works well on all types of data.
- Slower to train than Decision Trees
- Work with Transfer Learning
- When building a system of multiple models working together, it might be easier to string together multiple neural networks

May the forest be with you.