Udemy - 3 - Modeling - concepts

https://www.udemy.com/course/aws-machine-learning/learn/lecture/16397704#overview

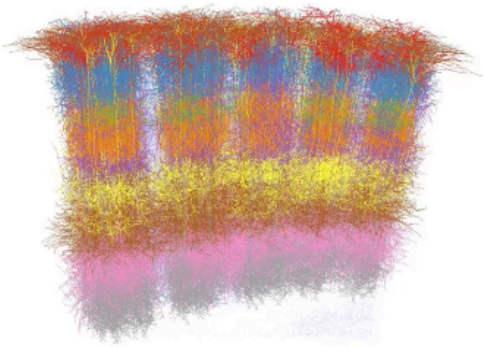
Intro to Deep Learning

The biological inspiration

- Neurons in your cerebral cortex are connected via axons
- A neuron "fires" to the neurons it's connected to, when enough of its input signals are activated.
- Very simple at the individual neuron level – but layers of neurons connected in this way can yield learning behavior.
- Billions of neurons, each with thousands of connections, yields a mind

Cortical columns

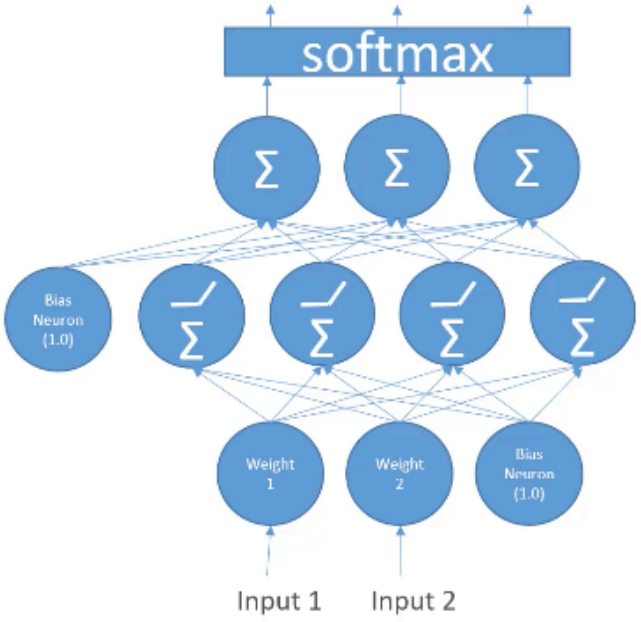
- Neurons in your cortex seem to be arranged into many stacks, or "columns" that process information in parallel
- "mini-columns" of around 100 neurons are organized into larger "hyper-columns". There are 100 million mini-columns in your cortex
- This is coincidentally similar to how GPU's work...



(credit: Marcel Oberlaender et al.)

Deep learning = more than one layer of neurons.
Learn appropriate weights and bias

Deep Neural Networks



Deep Learning Frameworks

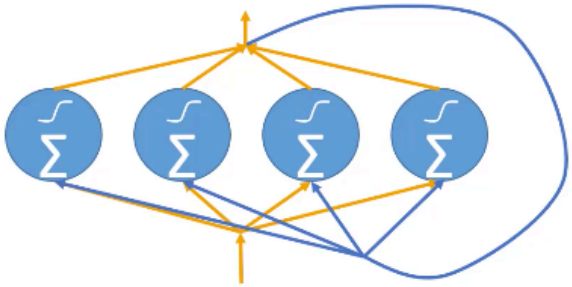
- Tensorflow / Keras
- MXNet

```
model = Sequential()

model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9,
       nesterov=True)
model.compile(loss='categorical_crossentropy',
       optimizer=sgd, metrics=['accuracy'])
```

Types of Neural Networks

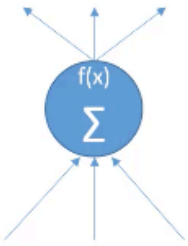
- Feedforward Neural Network
- Convolutional Neural Networks (CNN)
  - Image classification (is there a stop sign in this image?)
- Recurrent Neural Networks (RNNs)
  - Deals with sequences in time (predict stock prices, understand words in a sentence, translation, etc)
  - LSTM, GRU



Activation Functions

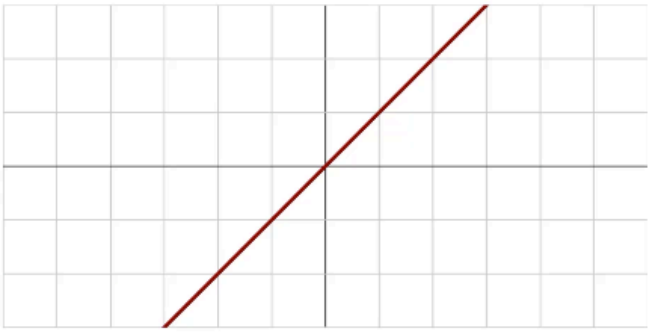
Activation Functions

- Define the output of a node / neuron given its input signals



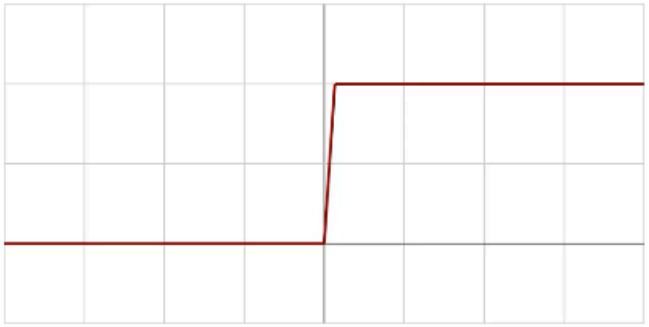
Linear activation function

- It doesn't really *do* anything
- Can't do backpropagation



Binary step function

- It's on or off
- Can't handle multiple classification – it's binary after all
- Vertical slopes don't work well with calculus!

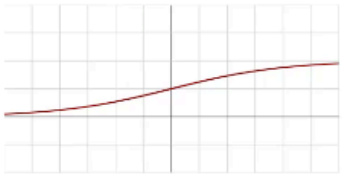


Instead we need non-linear activation functions

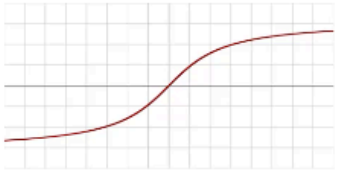
- These can create complex mappings between inputs and outputs
- Allow backpropagation (because they have a useful derivative)
- Allow for multiple layers (linear functions degenerate to a single layer)

Sigmoid / Logistic / TanH

- Nice & smooth
- Scales everything from 0-1 (Sigmoid / Logistic) or -1 to 1 (tanh / hyperbolic tangent)
- But: changes slowly for high or low values
  - The "Vanishing Gradient" problem
- Computationally expensive
- Tanh generally preferred over sigmoid



Sigmoid AKA Logistic



TanH AKA Hyperbolic Tangent

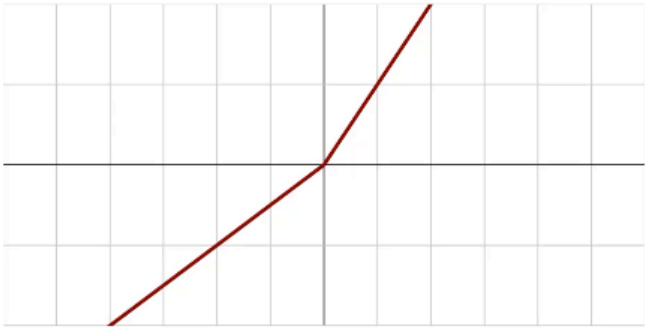
To address above pb, ReLU is a great choice

Rectified Linear Unit (ReLU)

- Now we're talking
- Very popular choice
- Easy & fast to compute
- But, when inputs are zero or negative, we have a linear function and all of its problems
  - The "Dying ReLU problem"

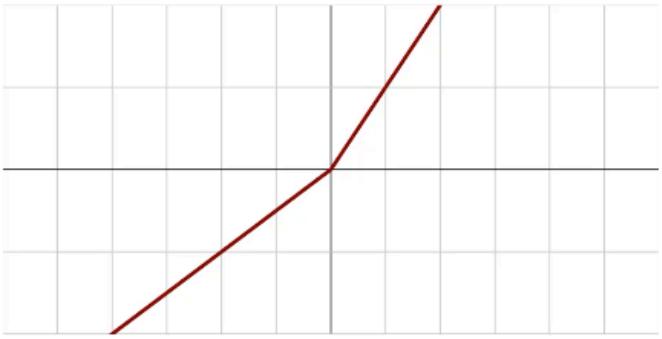
Leaky ReLU

- Solves "dying ReLU" by introducing a negative slope below 0 (usually not as steep as this)



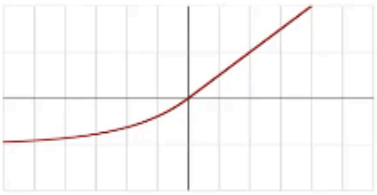
Parametric ReLU (PReLU)

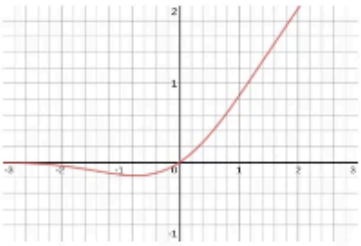
- ReLU, but the slope in the negative part is learned via backpropagation
- Complicated and YMMV



Other ReLU variants

- Exponential Linear Unit (ELU)
- Swish
  - From Google, performs really well
  - But it's from Google, not Amazon…
  - Mostly a benefit with very deep networks (40+ layers)
- Maxout
  - Outputs the max of the inputs
  - Technically ReLU is a special case of maxout
  - But doubles parameters that need to be trained, not often practical.

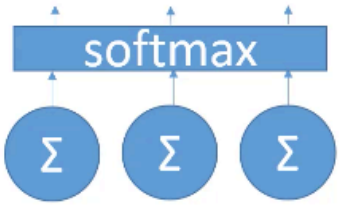




SoftMax: Converts output to probabilities of each classification
To know what classification is best, just take the max from the Softmax output.

Softmax

- Used on the final output layer of a multiple classification problem
- Basically converts outputs to probabilities of each classification
- Can't produce more than one label for something (sigmoid can)
- Don't worry about the actual function for the exam, just know what it's used for.



Choosing an activation function

- For multiple classification, use softmax on the output layer
- RNN's do well with Tanh
- For everything else
  - Start with ReLU
  - If you need to do better, try Leaky ReLU
  - Last resort: PReLU, Maxout
  - Swish for really deep networks

Also Sigmoid is useful for multi-class classification

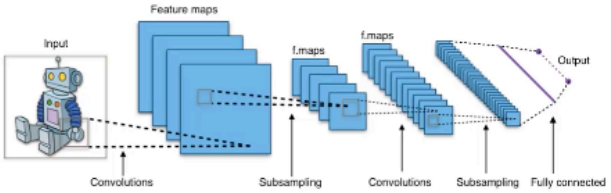
Convolution Neural Network

CNN's: what are they for?

- When you have data that doesn't neatly align into columns
  - Images that you want to find features within
  - Machine translation
  - Sentence classification
  - Sentiment analysis
- They can find features that aren't in a specific spot
  - Like a stop sign in a picture
  - Or words within a sentence
- They are "feature-location invariant"

CNN's: how do they work?

- Inspired by the biology of the visual cortex
  - Local receptive fields are groups of neurons that only respond to a part of what your eyes see (subsampling)
  - They overlap each other to cover the entire visual field (convolutions)
  - They feed into higher layers that identify increasingly complex images
    - Some receptive fields identify horizontal lines, lines at different angles, etc. (filters)
    - These would feed into a layer that identifies shapes
    - Which might feed into a layer that identifies objects
  - For color images, extra layers for red, green, and blue



=> Convolution: break a dataset into chunks

How do we "know" that's a stop sign?

- Individual local receptive fields scan the image looking for edges, and pick up the edges of the stop sign in a layer
- Those edges in turn get picked up by a higher level convolution that identifies the stop sign's shape (and letters, too)
- This shape then gets matched against your pattern of what a stop sign looks like, also using the strong red signal coming from your red layers
- That information keeps getting processed upward until your foot hits the brake!
- A CNN works the same way

CNN's with Keras / Tensorflow

- Source data must be of appropriate dimensions
  - ie width x length x color channels
- Conv2D layer type does the actual convolution on a 2D image
  - Conv1D and Conv3D also available – doesn't have to be image data
- MaxPooling2D layers can be used to reduce a 2D layer down by taking the maximum value in a given block
- Flatten layers will convert the 2D layer to a 1D layer for passing into a flat hidden layer of neurons
- Typical usage:
  - Conv2D -> MaxPooling2D -> Dropout -> Flatten -> Dense -> Dropout -> Softmax

Color channels:
- 1 for black and white
- 3 for color: blue, red, green

Conv2D => does convolution on a 2D image
Break image into sub-fields that overlap one with the other for individual processing

MaxPooling2D => reduce the size of the data down. Take the max value in a given block to reduce processing load on CNN

Dropout layer: prevents over fitting

Flatten layer: to be able to feed the data into a perceptron

Dense layer = perceptron = hidden layer of neurons

Softmax => choose final classification coming from neural network

CNN's are hard

- Very resource-intensive (CPU, GPU, and RAM)
- Lots of hyperparameters
  - Kernel sizes, many layers with different numbers of units, amount of pooling... in addition to the usual stuff like number of layers, choice of optimizer
- Getting the training data is often the hardest part! (As well as storing and accessing it)

Specialized CNN architectures

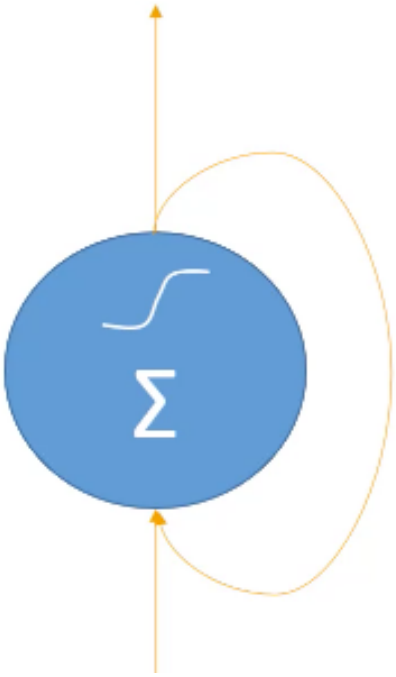
- Defines specific arrangement of layers, padding, and hyperparameters
- LeNet-5
  - Good for handwriting recognition
- AlexNet
  - Image classification, deeper than LeNet
- GoogLeNet
  - Even deeper, but with better performance
  - Introduces *inception modules* (groups of convolution layers)
- ResNet (Residual Network)
  - Even deeper – maintains performance via *skip connections*.

Recurrent Neural Networks

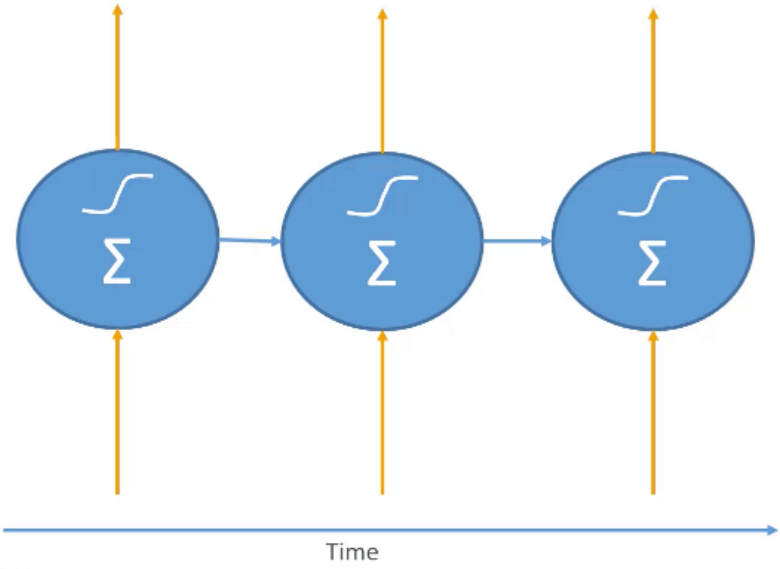
RNN's: what are they for?

- Time-series data
  - When you want to predict future behavior based on past behavior
  - Web logs, sensor logs, stock trades
  - Where to drive your self-driving car based on past trajectories
- Data that consists of sequences of arbitrary length
  - Machine translation
  - Image captions
  - Machine-generated music

A recurrent neuron

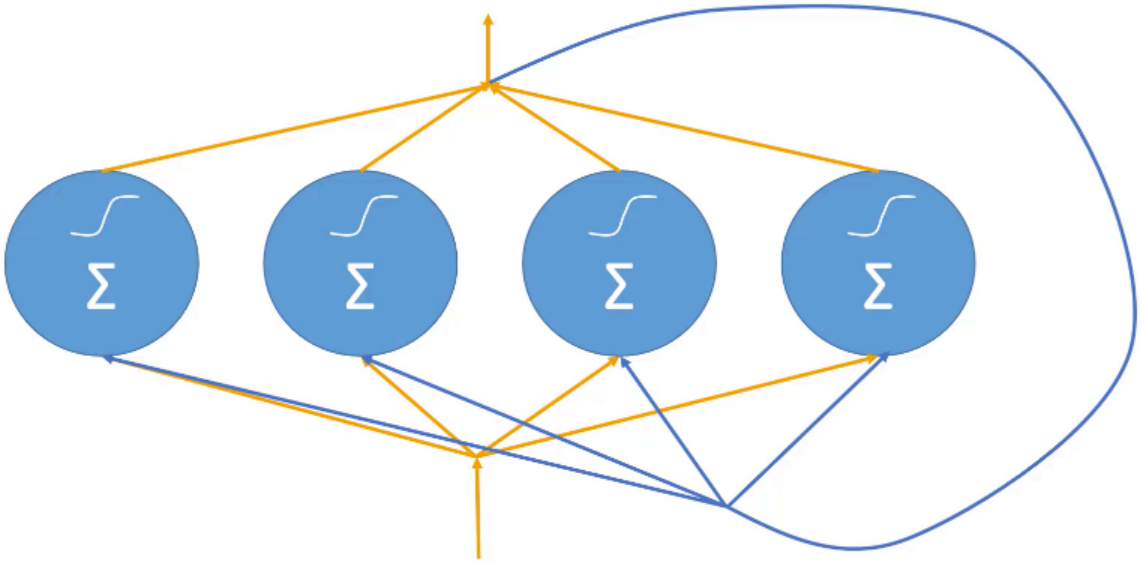


Another way to look at it



A “Memory Cell”

A layer of recurrent neurons



RNN topologies

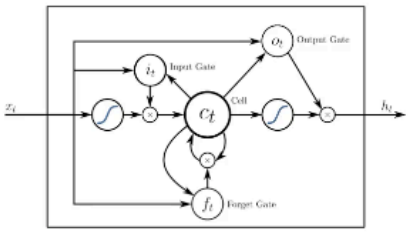
- Sequence to sequence
  - i.e., predict stock prices based on series of historical data
- Sequence to vector
  - i.e., words in a sentence to sentiment
- Vector to sequence
  - i.e., create captions from an image
- Encoder -> Decoder
  - Sequence -> vector -> sequence
  - i.e., machine translation

Training rnn's

- Backpropagation through time
  - Just like backpropagation on MLP's, but applied to each time step.
- All those time steps add up fast
  - Ends up looking like a really, really deep neural network.
  - Can limit backpropagation to a limited number of time steps (truncated backpropagation through time)

Training rnn's

- State from earlier time steps get diluted over time
  - This can be a problem, for example when learning sentence structures
- LSTM Cell
  - Long Short-Term Memory Cell
  - Maintains separate short-term and long-term states
- GRU Cell
  - Gated Recurrent Unit
  - Simplified LSTM Cell that performs about as well



GRU Cell are very popular in practice

Training rnn's

- It's really hard
  - Very sensitive to topologies, choice of hyperparameters
  - Very resource intensive
  - A wrong choice can lead to a RNN that doesn't converge at all.

Deep Learning on EC2 / EMR
Deep learning is a great fit for GPU: P2 / P3 / G3

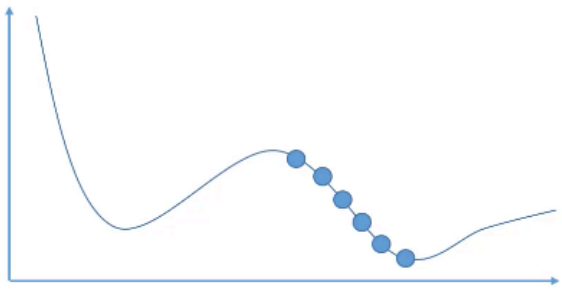
Deep Learning on EC2 / EMR

- EMR supports Apache MXNet and GPU instance types
- Appropriate instance types for deep learning:
  - P3: 8 Tesla V100 GPU's
  - P2: 16 K80 GPU's
  - G3: 4 M60 GPU's (all Nvidia chips)
- Deep Learning AMI's

Tuning Neural Networks

Learning Rate

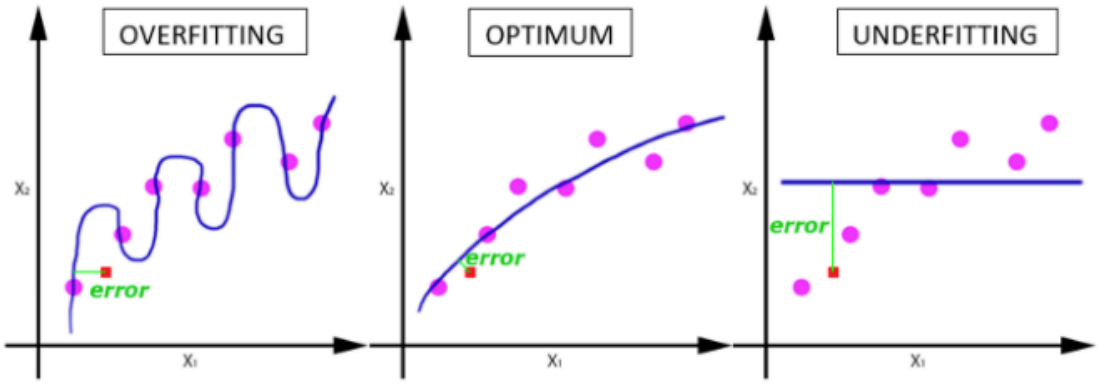
- Neural networks are trained by gradient descent (or similar means)
- We start at some random point, and sample different solutions (weights) seeking to minimize some cost function, over many *epochs*
- How far apart these samples are is the *learning rate*



Epochs: iterations over the entire training dataset
One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE.

Since **one epoch** is too big to feed to the computer at once we divide it in several smaller **batches**

> *One epoch leads to underfitting of the curve in the graph (below).*



As the number of epochs increases, more number of times the weight are changed in the neural network and the curve goes from **underfitting** to **optimal** to **overfitting** curve.

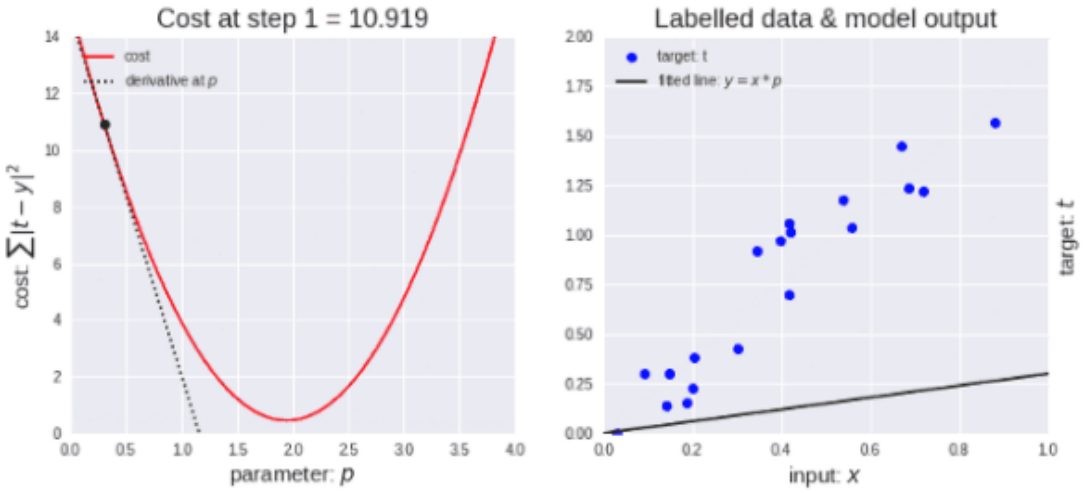
Batch size: Total number of training examples present in a single batch.
As I said, you can't pass the entire dataset into the neural net at once. So, you **divide dataset into Number of Batches or sets or parts.**

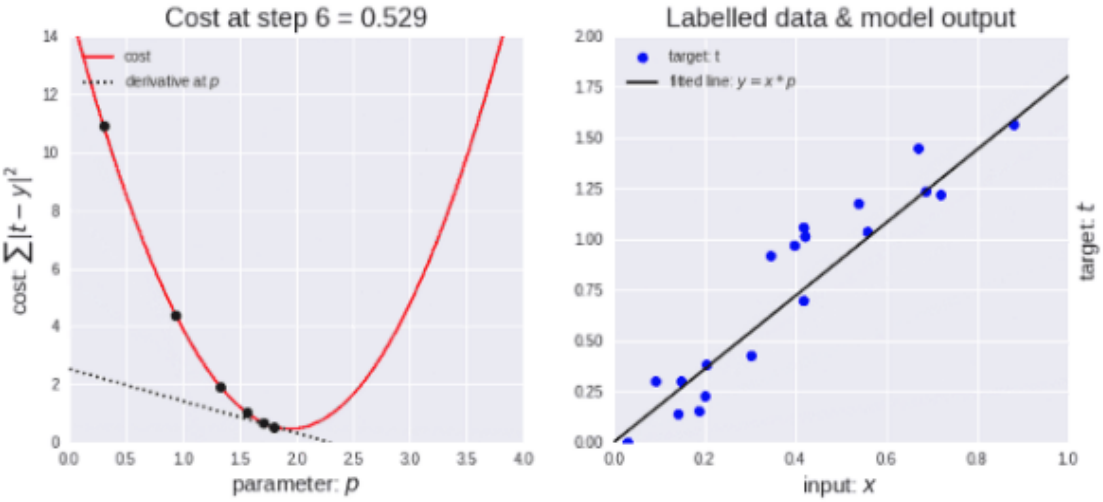
Iterations: Iterations is the number of batches needed to complete one epoch.
Note: *The number of batches is equal to number of iterations for one epoch.*

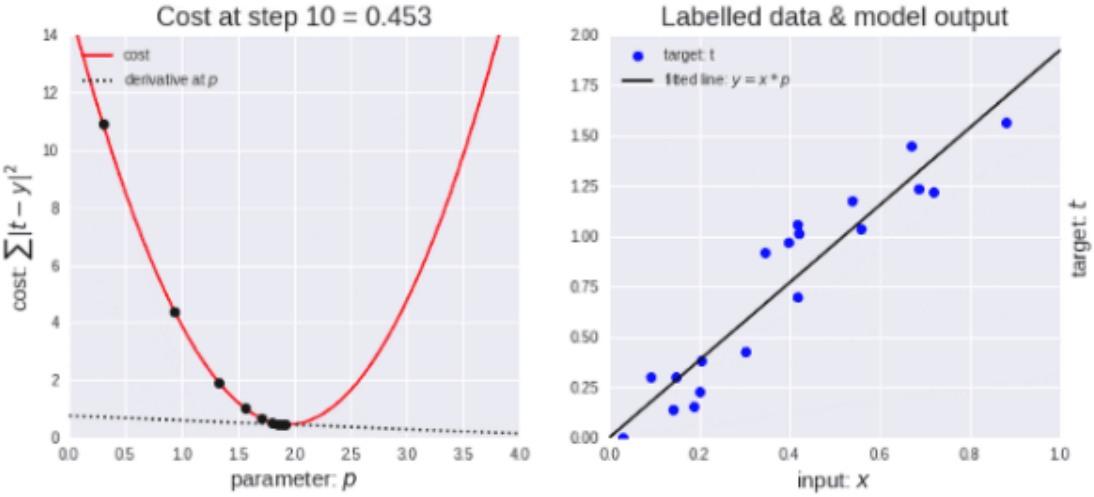
Let's say we have 2000 training examples that we are going to use .
We can divide the dataset of 2000 examples into batches of 500 then it will take 4 iterations to complete 1 epoch.

Goal: minimize the **cost** function, and get to the minimum => **gradient** descent
Learning rate: how far apart are the samples, or how large is the "step" down.
If too high learning rate, we might miss some minima. But if too small, we will be sampling a lot of datapoint and it will take a lot of epochs to find the optimal solution

Gradient descent illustrated:
https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9
*both **Cost** and **Loss** represent same thing*

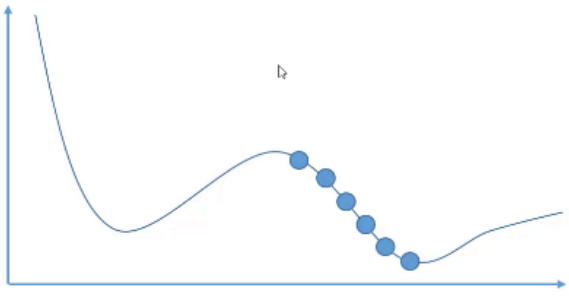






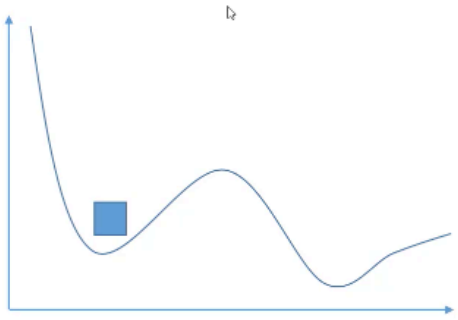
Effect of learning rate

- Too high a learning rate means you might overshoot the optimal solution!
- Too small a learning rate will take too long to find the optimal solution
- Learning rate is an example of a *hyperparameter*



Batch Size

- How many training samples are used within each epoch
- Somewhat counter-intuitively:
  - Smaller batch sizes can work their way out of "local minima" more easily
  - Batch sizes that are too large can end up getting stuck in the wrong solution
  - Random shuffling at each epoch can make this look like very inconsistent results from run to run



Important: A small batch size can "wiggle" its way out of local minima

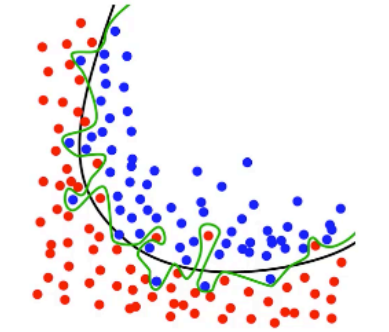
To Recap (this is important!)

- Small batch sizes tend to not get stuck in local minima
- Large batch sizes can converge on the wrong solution at random
- Large learning rates can overshoot the correct solution
- Small learning rates increase training time

Regularization Techniques for Neural Networks
=> prevents overfitting

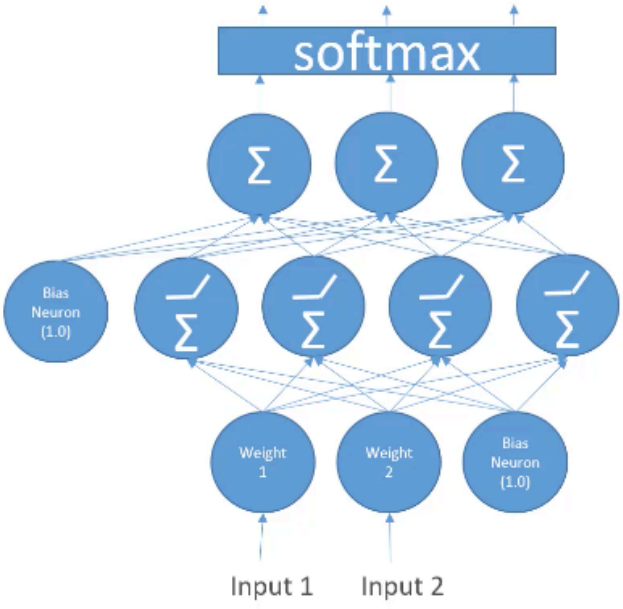
What is regularization?

- Preventing *overfitting*
  - Models that are good at making predictions on the data they were trained on, but not on new data it hasn't seen before
  - Overfitted models have learned patterns in the training data that don't generalize to the real world
  - Often seen as high accuracy on training data set, but lower accuracy on test or evaluation data set.
    - When training and evaluating a model, we use *training*, *evaluation*, and *testing* data sets.
- Regularization techniques are intended to prevent overfitting.



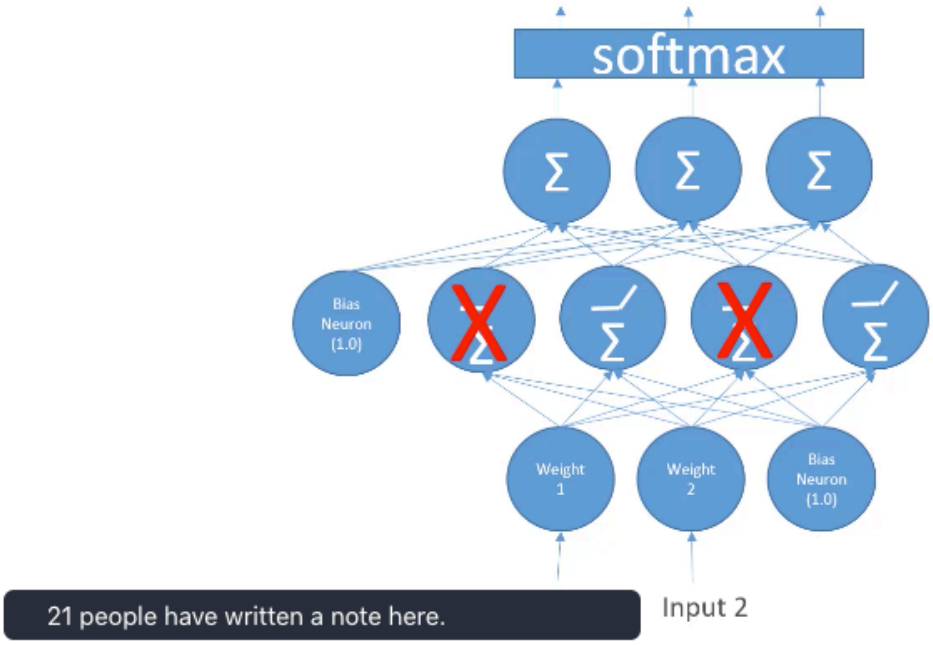
When are we potentially over-fitting? When we have complex models
Idea => start simplifying

Too many layers? Too many neurons?



Dropout: remove neurons at each epoch => spread out learning across neurons and layers
=> helps prevent particular neurons from over fitting to some data points

Dropout



Early stopping: automatically detect when validation accuracy has leveled out (while training accuracy may still grow
and stop the training.
=> allows us to not learn the network further than it needs.

Loss below gets better, then we oscillate after epoch 5.

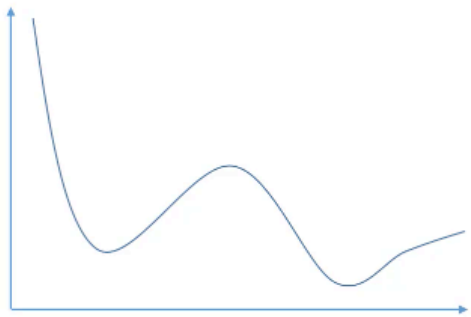
Early Stopping

```
Epoch 1/10
 - 4s - loss: 0.2406 - acc: 0.9302 - val_loss: 0.1437 - val_acc: 0.9557
Epoch 2/10
 - 2s - loss: 0.0971 - acc: 0.9712 - val_loss: 0.0900 - val_acc: 0.9725
Epoch 3/10
 - 2s - loss: 0.0653 - acc: 0.9803 - val_loss: 0.0725 - val_acc: 0.9786
Epoch 4/10
 - 2s - loss: 0.0471 - acc: 0.9860 - val_loss: 0.0689 - val_acc: 0.9795
Epoch 5/10
 - 2s - loss: 0.0367 - acc: 0.9890 - val_loss: 0.0675 - val_acc: 0.9808
Epoch 6/10
 - 2s - loss: 0.0266 - acc: 0.9919 - val_loss: 0.0680 - val_acc: 0.9796
Epoch 7/10
 - 2s - loss: 0.0208 - acc: 0.9937 - val_loss: 0.0678 - val_acc: 0.9811
Epoch 8/10
 - 2s - loss: 0.0157 - acc: 0.9953 - val_loss: 0.0719 - val_acc: 0.9810
Epoch 9/10
 - 2s - loss: 0.0130 - acc: 0.9960 - val_loss: 0.0707 - val_acc: 0.9825
Epoch 10/10
 - 2s - loss: 0.0097 - acc: 0.9972 - val_loss: 0.0807 - val_acc: 0.9805
```

Grief with Gradients: vanishing gradient

The Vanishing Gradient Problem

- When the slope of the learning curve approaches zero, things can get stuck
- We end up working with very small numbers that slow down training, or even introduce numerical errors
- Becomes a problem with deeper networks and RNN's as these "vanishing gradients" propagate to deeper layers
- Opposite problem: "exploding gradients"



Gradient is approaching 0 => slope approaching 0
Could be local minima or global.
That can be an issue.

https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/

How to Fix the Vanishing Gradients Problem Using the ReLU

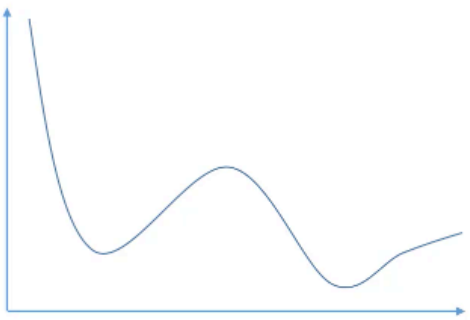
It describes the situation where a deep multilayer feed-forward network or a recurrent neural network is unable to propagate useful gradient information from the output end of the model back to the layers near the input end of the model. The result is the general inability of models with many layers to learn on a given dataset, or for models with many layers to prematurely converge to a poor solution.

Vanishing Gradients Problem

Neural networks are trained using stochastic gradient descent.
This involves first calculating the prediction error made by the model and using the error to estimate a gradient used to update each weight in the network so that less error is made next time. This error gradient is propagated backward through the network from the output layer to the input layer.
It is desirable to train neural networks with many layers, as the addition of more layers increases the capacity of the network, making it capable of learning a large training dataset and efficiently representing more complex mapping functions from inputs to outputs.
A problem with training networks with many layers (e.g. deep neural networks) is that the gradient diminishes dramatically as it is propagated backward through the network. The error may be so small by the time it reaches layers close to the input of the model that it may have very little effect. As such, this problem is referred to as the "*vanishing gradients*" problem.
Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function …

Fixing the Vanishing Gradient Problem

- Multi-level heirarchy
  - Break up levels into their own sub-networks trained individually
- Long short-term memory (LSTM)
- Residual Networks
  - i.e., ResNet
  - Ensemble of shorter networks
- Better choice of activation function
  - ReLU is a good choice



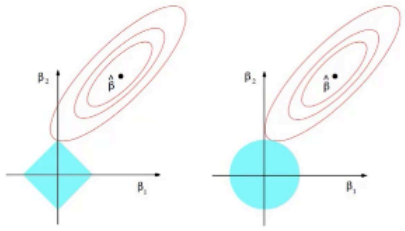
Gradient Checking

- A debugging technique
- Numerically check the derivatives computed during training
- Useful for validating code of neural network training
  - But you're probably not going to be writing this code...

L1 and L2 Regularization

L1 and L2 Regularization

- Preventing overfitting in ML in general
- A regularization term is added as weights are learned
- L1 term is the sum of the weights
  - $\lambda \sum_{i=1}^{k} |w_i|$
- L2 term is the sum of the square of the weights
  - $\lambda \sum_{i=1}^{k} w_i^2$
- Same idea can be applied to loss functions



What's the difference?

- L1: sum of weights
  - Performs *feature selection* – entire features go to 0
  - Computationally inefficient
  - Sparse output
- L2: sum of square of weights
  - All features remain considered, just weighted
  - Computationally efficient
  - Dense output

Why would you want L1?

- Feature selection can reduce dimensionality
  - Out of 100 features, maybe only 10 end up with non-zero coefficients!
  - The resulting sparsity can make up for its computational inefficiency
- But, if you think all of your features are important, L2 is probably a better choice.

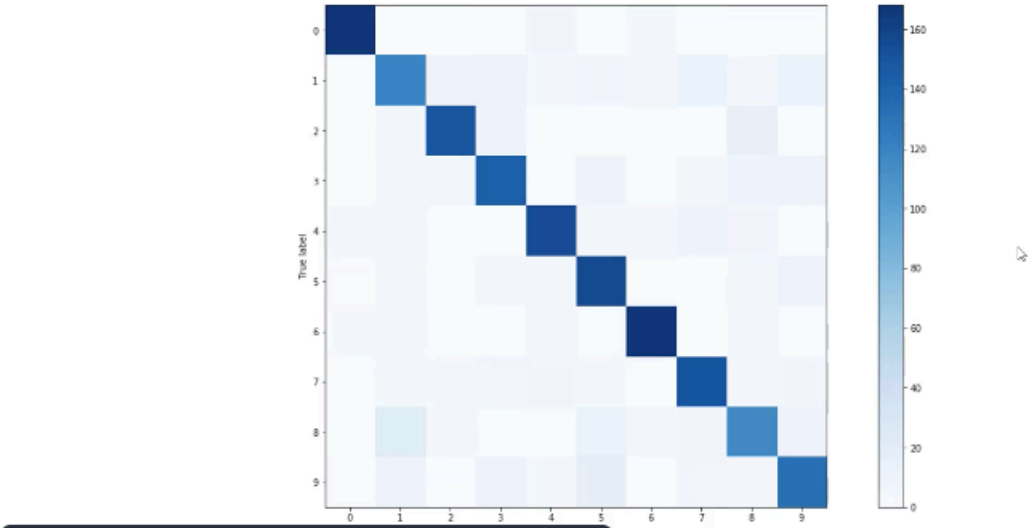
Confusion Matrix

Sometimes accuracy doesn't tell the whole story

- A test for a rare disease can be 99.9% accurate by just guessing "no" all the time
- We need to understand true positives and true negative, as well as false positives and false negatives.
- A confusion matrix shows this.

	Predicted NO	Predicted YES	
Actual NO	50	5	**55**
Actual YES	10	100	**110**
	60	**105**	

Multi-class confusion matrix + heat map



Recall

- $$\frac{TRUE\ POSITIVES}{TRUE\ POSITIVES+FALSE\ NEGATIVES}$$

- AKA Sensitivity, True Positive rate, Completeness
- Percent of positives rightly predicted
- Good choice of metric when you care a lot about false negatives
  - i.e., fraud detection

Recall example

	Actual fraud	Actual not fraud
Predicted fraud	5	20
Predicted not fraud	10	100

Recall = TP/(TP+FN)

Recall = 5/(5+10) = 5/15 = 1/3 = 33%

Precision

- $$\frac{TRUE\ POSITIVES}{TRUE\ POSITIVES+FALSE\ POSITIVES}$$

- AKA Correct Positives
- Percent of relevant results
- Good choice of metric when you care a lot about false positives
  - i.e., medical screening, drug testing

Precision example

	Actual fraud	Actual not fraud
Predicted fraud	5	20
Predicted not fraud	10	100

Precision = TP/(TP+FP)

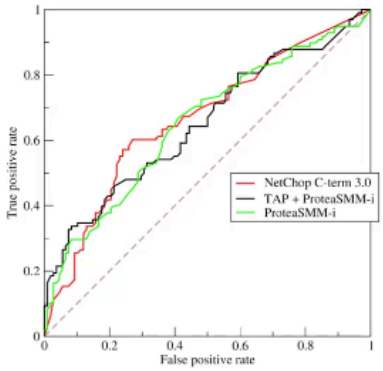
Precision = 5/(5+20) = 5/25 = 1/5 = 20%

Other metrics

- Specificity = $\frac{TN}{TN+FP}$ = "True negative rate"
- F1 Score
  - $\frac{2TP}{2TP+FP+FN}$
  - $2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$
    - Harmonic mean of precision and sensitivity
    - When you care about precision AND recall
- RMSE
  - Root mean squared error, exactly what it sounds like
  - Accuracy measurement
  - Only cares about right & wrong answers

ROC Curve

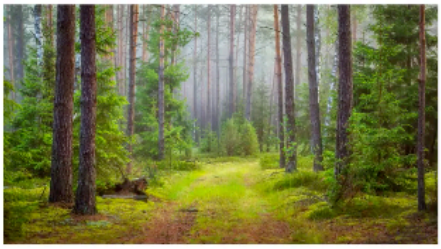
- Receiver Operating Characteristic Curve
- Plot of true positive rate (recall) vs. false positive rate at various threshold settings.
- Points above the diagonal represent good classification (better than random)
- Ideal curve would just be a point in the upper-left corner
- The more it's "bent" toward the upper-left, the better

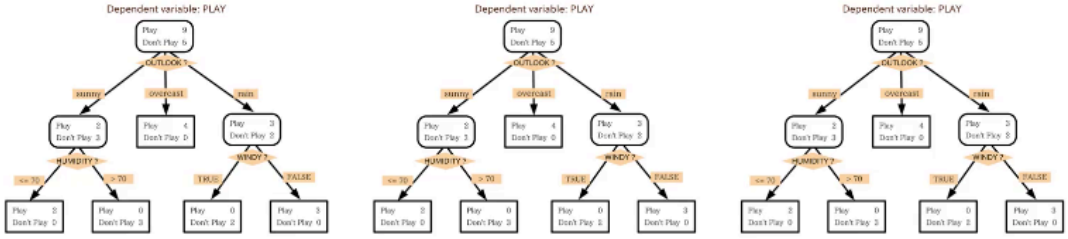


Ensemble Methods: Bagging and Boosting

Ensemble methods

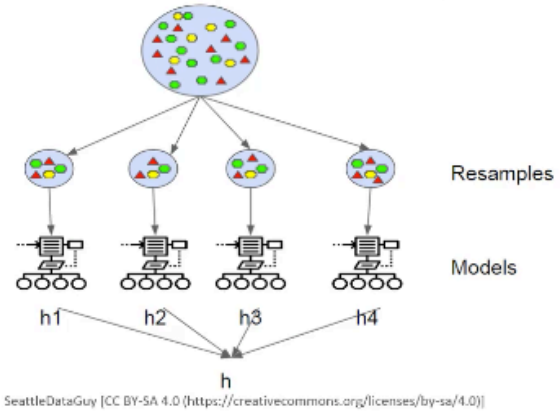
- Common example: random forest
  - Decision trees are prone to overfitting
  - So, make lots of decision trees and let them all vote on the result
  - This is a random forest
  - How do they differ?





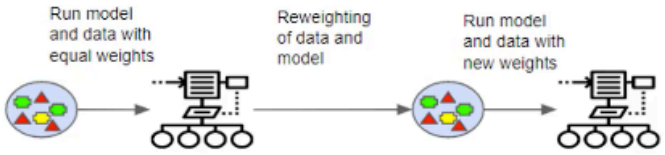
Bagging

- Generate N new training sets by **random sampling with replacement**
- Each resampled model can be trained in parallel



Boosting

- Observations are weighted
- Some will take part in new training sets more often
- Training is sequential; each classifier takes into account the previous one's success.



Bagging vs. Boosting

- XGBoost is the latest hotness
- Boosting generally yields better accuracy
- But bagging avoids overfitting
- Bagging is easier to parallelize
- So, depends on your goal

Boosting strength is Accuracy => XGboost is the new hot thing in SM and many Kaggle competition challenges are won with it
Bagging avoids overfitting (as we split the data into separate samples) and can work faster (parallel)