Deep Learning and Advanced Neural Network Architectures – Pt. II

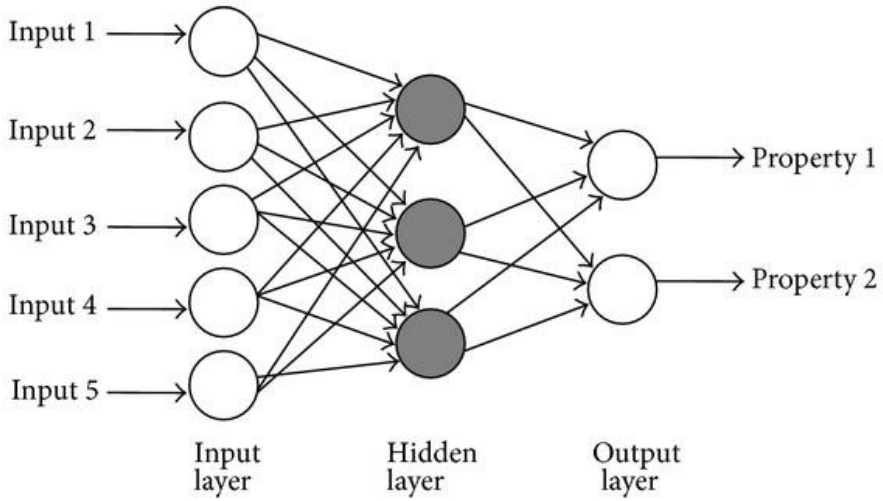
Prepared by Haihan – AQM 2018

Overview

- Learning Goals:
  - Identify and describe the 3 major Machine Learning paradigms
  - Derive and implement Gradient Descent on simple neural net
  - **Understand the usage of different activation functions, loss functions and optimizers**
  - **Understand regularization, dropout and their effects on fitting**
  - **Understand variable selection and feature engineering on input data**
  - **Understand cross validation, bootstrapping, bagging and ensemble models**
  - **Implement Deep MLP network**
  - Implement Convolutional Network
  - Implement LSTM Network
  - Understand the robustness and the pitfalls of these techniques
  - If time permits: SOMs, Word2vec, GANs, Deep Reinforcement Learning, TensorBoard, advanced ML paradigms

Recap

- In the previous slide set we introduced the single hidden layer Multi-layer Perceptron (MLP) and the stochastic/batch gradient descent method with momentum
- We saw how the update equations for the MLP using stochastic gradient descent were derived



Matrix formalism

- The update equations for SGD and the Neural net itself can be expressed more concisely with the matrix formalism. This is the way TensorFlow and other ML Libraries represent neural networks and optimization operations on them.
- The Hadamard or Element-wise product is defined as $\mathbf{x} \circ \mathbf{y} = [\, x_1 y_1, x_2 y_2, \ldots, x_n y_n]$

The neural network has L layers. x_0 is the input vector, x_L is the output vector and t is the truth vector. The weight matrices are $W_1, W_2, .., W_L$ and activation functions are $f_1, f_2, .., f_L$.

Forward Pass:

$$x_i = f_i(W_i x_{i-1})$$
$$E = \|x_L - t\|_2^2$$

Backward Pass:

$$\delta_L = (x_L - t) \circ f_L'(W_L x_{L-1})$$
$$\delta_i = W_{i+1}^T \delta_{i+1} \circ f_i'(W_i x_{i-1})$$

Weight Update:

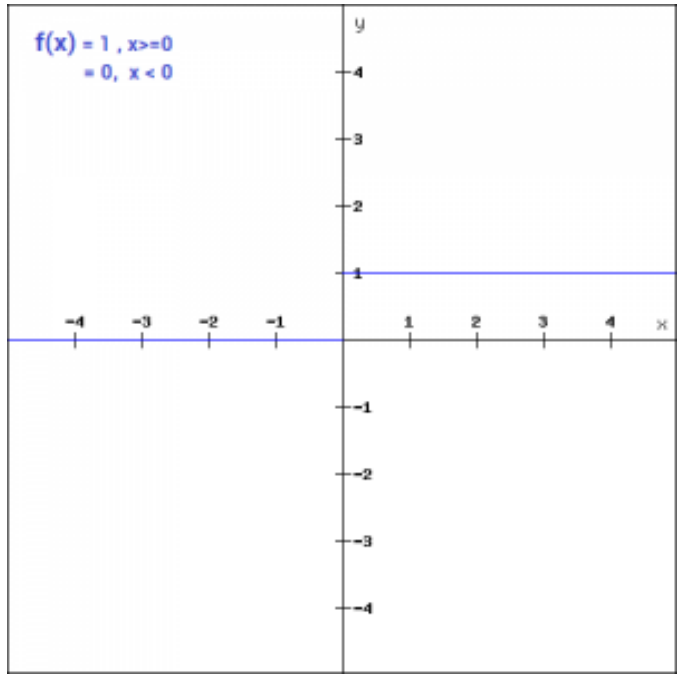
$$\frac{\partial E}{\partial W_i} = \delta_i x_{i-1}^T$$
$$W_i = W_i - \alpha_{W_i} \circ \frac{\partial E}{\partial W_i}$$

https://sudeepraja.github.io/Neural/

Activation Functions

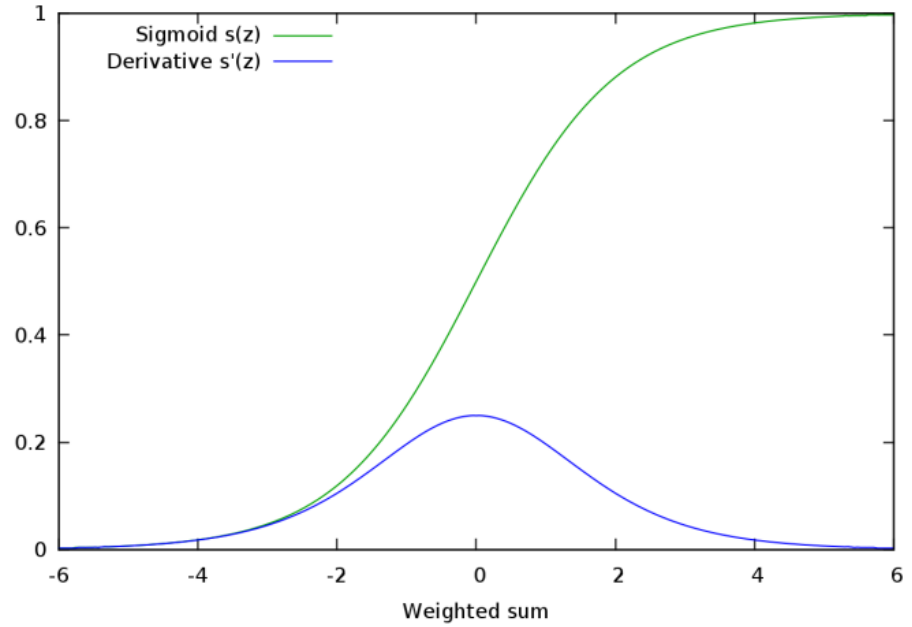
- The purpose of the activation in the neural network is to apply a pointwise non-linear transformation to its input vector (after it has been linearly transformed)
- There are a few different types of activation functions (some times called squashing functions or non-linearities), each with its own appropriate uses.

1. Binary step function:
   - Definition: $f(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$
   - The binary step function was originally used when the first perceptron neural networks were built in the late 1950's. It is no longer practical and only really of theoretical interest.



Activation Functions

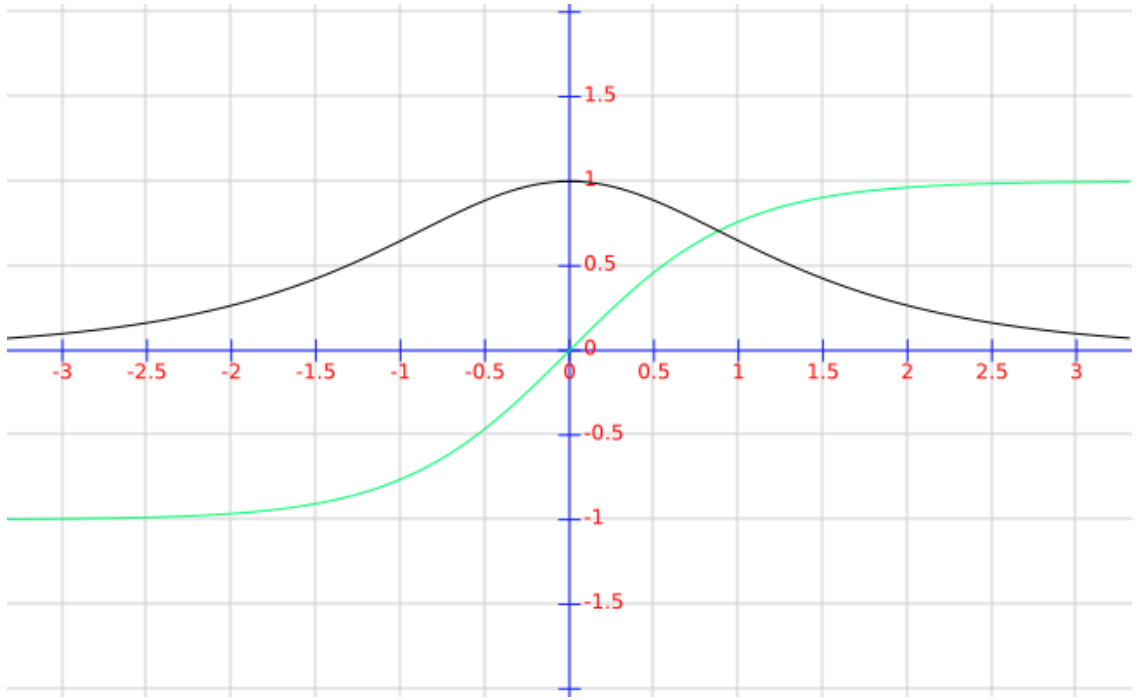
2. Linear:
   - Definition: $f(x) = x,\ x \in \mathbb{R}$
   - The linear activation function is also not very practical for neural networks. Nonlinear transforms are usually required for the neural network to learn effective representations of the data.
3. Unipolar Sigmoid (logistic function):
   - Definition: $f(x) = \frac{1}{1+e^{-x}},\ x \in \mathbb{R}$
   - Derivative: $f'(x) = f(x)(1 - f(x))$
   - The unipolar logistic function is one of the more practical activation functions used in neural networks. It resembles a vertically translated linear function near the origin, but saturates at values greater than +/- 2. The derivative of the sigmoid in the interval [-2, 2] the derivative is relatively large, indicating a small change in x can cause a significant change in the output. However, the unipolar logistic function is not symmetric about the origin and its output is always positive.



Activation Functions

4. Bipolar sigmoid, tanh():
   - Definition: $f(x) = \frac{2}{1+e^{-x}} - 1,\ x \in \mathbb{R}$
   - $f'(x) = f(x)(1 - f(x))$

   The bipolar sigmoid function (tanh() is one case of such) is one of the most useful general-purpose activation functions. It's output is in range [-1, 1] and it's derivative assumes higher values than the logistic function, thereby increasing training speed.

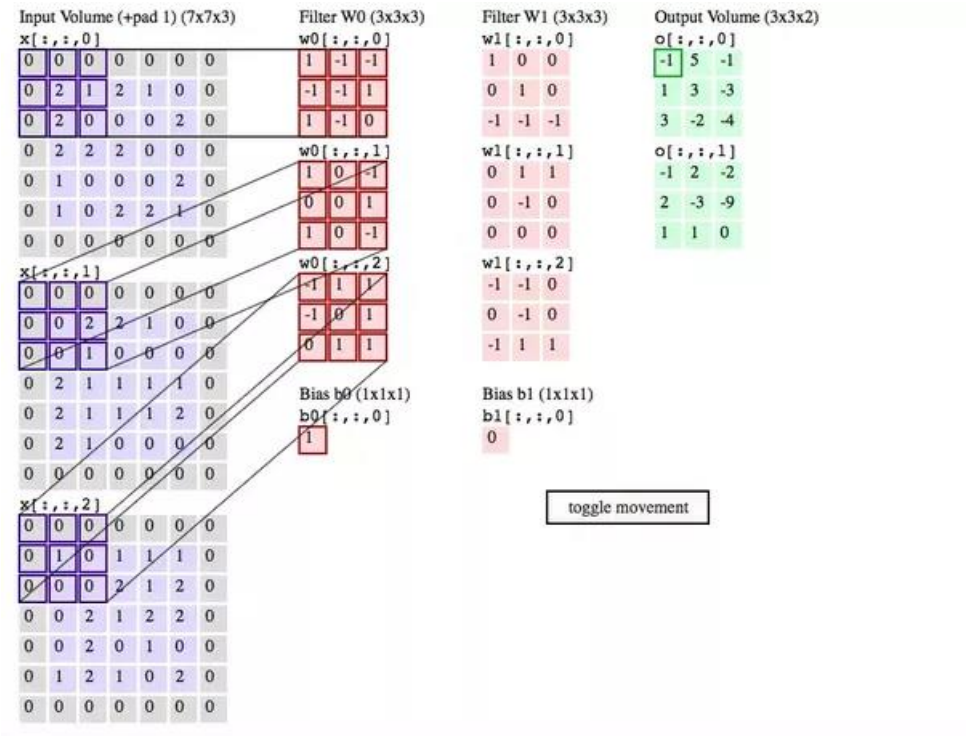


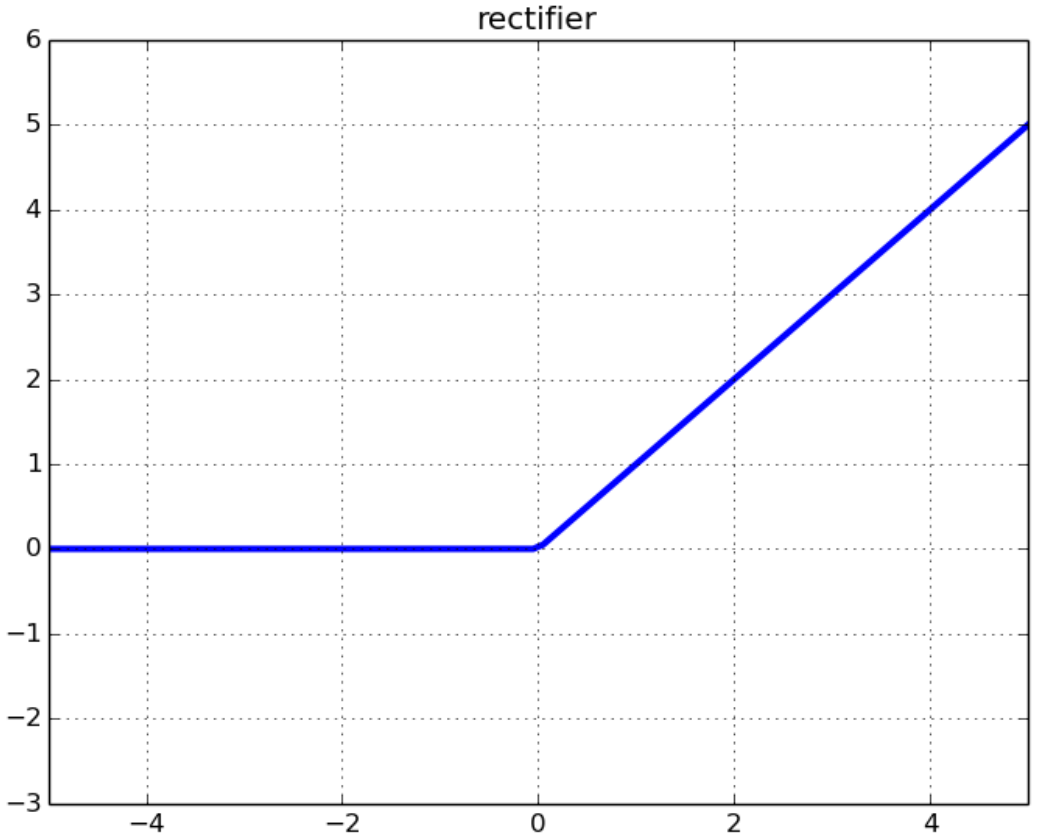
Activation Functions

5. Rectified Linear unit (ReLu):

- Definition: $f(x) = x, \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$

The ReLu activation resembles the linear activation but has a kink at $x = 0$, and is Therefore non-linear. ReLu's are used extensively in convolutional networks, where simple neural *sub-networks* are repeated or tiled.





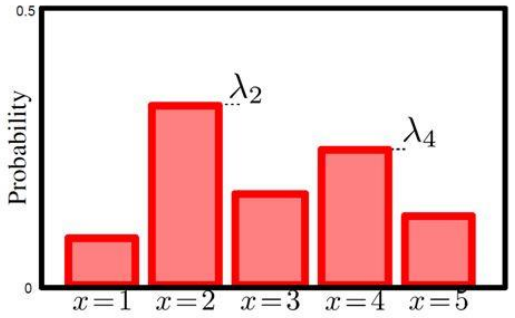
Activation Functions

6. Softmax:

- Definition: $f(\mathbf{x}, x_j) = \frac{\exp(x_j)}{\sum \exp(x_i)}$

The softmax function is a special activation function used at the output layer of classifiers. When trained with the *Cross Entropy* loss function, the softmax represents a categorically distributed output (the multi-class generalization of a Bernoulli distributed output). The values that the network outputs can be interpreted as the predicted probabilities of a certain item belonging to a class.

Categorical Distribution



$$Pr(x = k) = \lambda_k$$

or can think of data as vector with all elements zero except k[th] e.g. [0,0,0,1 0]

$$Pr(\mathbf{x} = \mathbf{e}_k) = \prod_{j=1}^{K} \lambda_j^{x_j} = \lambda_k$$

For short we write:

$$Pr(x) = \mathrm{Cat}_x[\boldsymbol{\lambda}]$$

Categorical distribution describes situation where K possible outcomes y=1… y=k.

Takes K parameters $\lambda_k \in [0, 1]$ where $\sum_k \lambda_k = 1$

Loss Functions revisited:

Mean Squared Error (MSE) loss:

Minimize $MSE(y_t, o_t) = \frac{1}{2}(y_t - o_t)^2$ where y_t is the neural network output at time t and o_t is the supervised training target at time t. MSE losses are most popular for regression tasks and neural network coupled deep reinforcement learning (see equation (2) in this reference note the expected value operator denotes batch update).

Assumes Gaussian distributed noise.

Cross Entropy loss:

Minimize

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

Used to train softmax output layer classifiers. Related to the loss function in logistic regression but in general is multiclass. See here for a more detailed explanation.

Other types of losses:

Hinge (max-margin for SVMs), Root Mean Squared Logarithmic Error (used when there's large differences between target and output), Poisson (count variables), Negative Log-likelihood (Maximum Likelihood estimation). See here for full review of the most popular loss functions.

Custom Losses:

Implementable in Keras using the Backend API or directly in Tensorflow or Theano. See here.

Optimizers:

Material mostly from http://ruder.io/optimizing-gradient-descent/

SGD (including Batch/Mini Batch SGD) update with momentum function $M(\theta_{old}, \Delta_\theta)$:

$$\theta_{new} = M(\theta_{old}, \eta \cdot \sum_{i=0}^{N} \nabla_\theta J(\theta_{old}, o_i, y_i))$$

Without momentum batch SGD looks something like:

$$\theta_{new} = \theta_{old} - \eta \cdot \sum_{i=0}^{N} \nabla_\theta J(\theta_{old}, o_i, y_i))$$

- Fastest and simplest but not necessarily most robust. Good for basic regression tasks. Usually the first one you should try (benchmarking) or the last one you should try (if nothing else works).

Adagrad

- Adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data.
- Applications in Image Classification and Natural Language processing
- In the following, $g_{t,i}$ is the gradient of the loss w.r.t the parameters at iteration t for some data row i

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t [25], while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Interestingly, without the square root operation, the algorithm performs much worse.

Optimizers:

RMSProp

- Resolves Adagrad's radically diminishing learning rates.
- Popular choice for Recurrent neural networks such as LSTMs (Long Short Term Memory)

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Adam

Adadelta

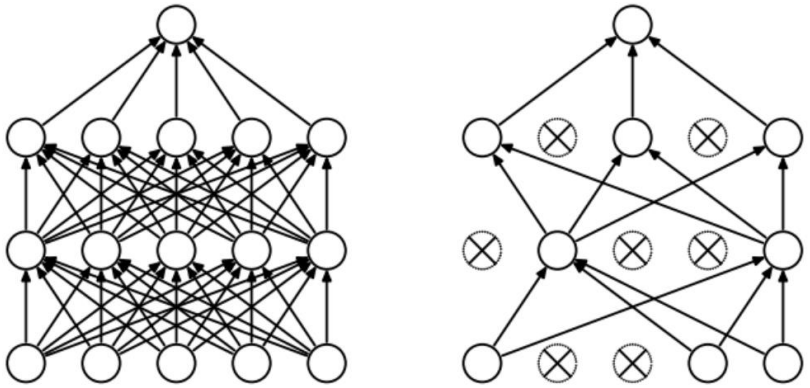
Genetic Algorithms (heuristic)

Particle Swarm Optimization (heuristic)

For a comparison of a few of these optimizers see https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/

Regularization and dropout

- Recall that regularization (L1, L2) mitigates overfitting and increases robustness of your models on the test set
- In many Neural net packages, L1 and L2 regularization (soft clipping), and hard weight and gradient clipping are available
- Another technique called **dropout** can be used in your neural nets



(a) Standard Neural Net

(b) After applying dropout.

- Dropout forces a neural network to learn fewer parameters, and can in many cases increases generalization and robustness, at the expense of increased training time (iterations and epochs), especially in deep networks (2 or more hidden layers)
- Typically applied at the *Dense* layers near the output of a classifier or regressor

Variable selection and Feature engineering

- Variable preprocessing and scaling:
  - In general inputs to neural networks need to be **normalized**, also detrended if possible (for example with time series)
  - Normalization depends on the activation of the input layer: eg. [-1, 1] for tanh, [0,1] for logistic (sigmoid) and ReLu
  - If input ranges are too large for a certain numerical variable, consider log transforms
  - Categorical and ordinal inputs should be one-hot encoded

Original data:			One-hot encoding format:					
id	Color		id	White	Red	Black	Purple	Gold
1	White		1	1	0	0	0	0
2	Red		2	0	1	0	0	0
3	Black		3	0	0	1	0	0
4	Purple		4	0	0	0	1	0
5	Gold		5	0	0	0	0	1

- Variable Selection:
  - Consider the techniques we have covered before:
    - PCA
    - Decision trees and Random Forests
    - Forward and Backward selection (can be very slow with neural networks)

- Feature engineering:
  - Selecting which variables and applying the right transforms requires domain knowledge of specific problem
    - For example, in Natural Language Processing whether to use IF-IDF tokenization or Word2vec embedding
    - Or selecting which bins in a Fourier transform spectrum to use for a spectral classifier
  - Spend some time to think ahead, and research what works and what doesn't for your particular problem!

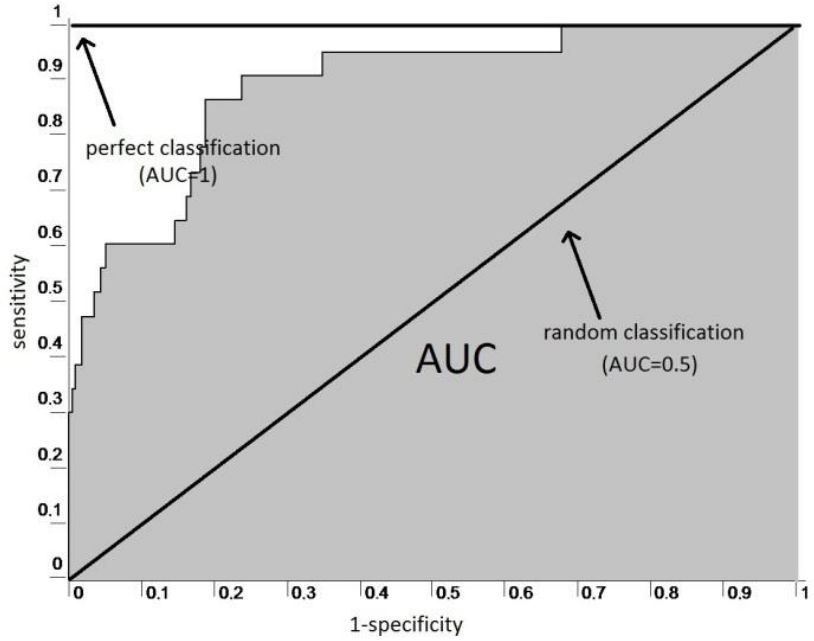
Cross-validation and accuracy

- Simple hold-out (1 training and 1 test set, typically 80-20 or 90 -10)
- K-fold: *k* non-overlapping training sets with *k* non-overlapping training sets, aggregate training results:

```
k = 10
for i in 1:k
 # Select unique training and testing datasets
KFoldTraining <-- subset(Data)
KFoldTesting <-- subset(Data) # Train and record performance KFoldPerformance[i] <-- SmartTrain(KFoldTraining, KFoldTesting)

# Compute overall performance TotalPerformance <-- ComputePerformance(KFoldPerformance)
```

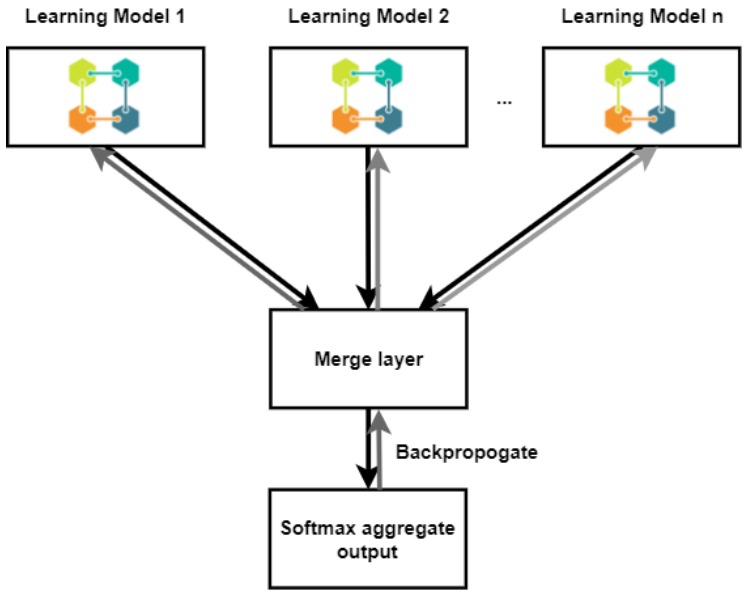
From: https://stackoverflow.com/questions/25889637/how-to-use-k-fold-cross-validation-in-a-neural-network

- Previously discussed accuracy measures for classification such as ROC, F1 scores apply
- For regression: aggregate the errors or losses given by your loss function



Bagging and Ensemble models

- Bagging and ensemble methods apply
- Most straightforward way with bagging: train an ensemble of neural net models with same architecture on bootstrap sample
- Aggregate the results (vote, 'softmax within a softmax', average, etc.)
- Can also stack, but more complex and can take more time
- A more advanced way of ensembling and stacking using Merge layers in Keras: https://towardsdatascience.com/the-softmax-function-neural-net-outputs-as-probabilities-and-ensemble-classifiers-9bd94d75932 (the dataset is not bootstrapped, but this is still an ensemble model, the fact that the neural net weights for different sub-models are initially randomized is good enough for 'pseudo bootstrapping')



The Keras package

- Keras is a high level neural network API (application programming interface - roughly speaking a package with a bunch of functions) that can run on top of Tensorflow or Theano.
- Writing neural networks in Tensorflow or Theano can be slow, lots of extra setup in building the network and training it
- Keras abstracts the whole process of building a network and training it
- But if you wanted to write custom layers or custom loss functions it's a good idea to use Tensorflow or Theano
- Install and read the get started portion here: https://keras.io/

Building the network:

We will start with the **Sequential model** which is just a linear stack of layers from input to output

- See here https://keras.io/getting-started/sequential-model-guide/
- In the sequential model, the input shape is specified and then layers are added.
- A loss function and optimizer are selected and then the model is compiled
  - You can have a list of models in Python as part of an ensemble
- Training and validation can be run with one function: model.fit()

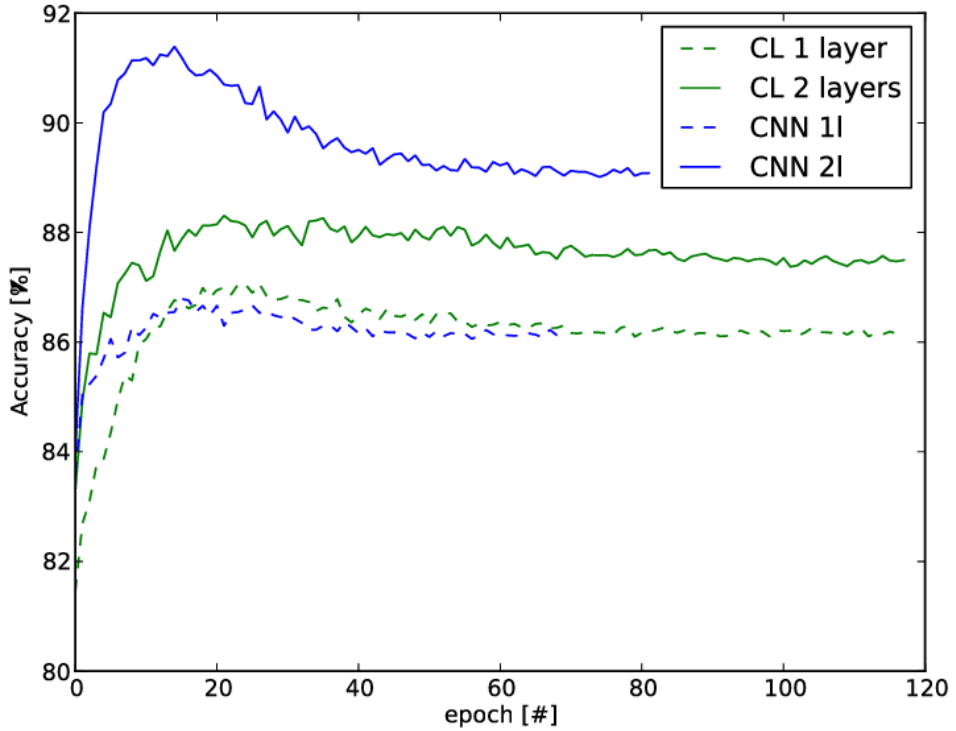
The Keras package

- Basic Keras example:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation from keras.optimizers
import SGD # Generate dummy data
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)),
num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)),
num_classes=10)

model = Sequential()

# Dense(64) is a fully-connected layer with 64 hidden
units. # in the first layer, you must specify the expected input data
shape: # here, 20-dimensional vectors.
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’])
model.fit(x_train, y_train, epochs=20, batch_size=128) score =
model.evaluate(x_test, y_test, batch_size=128)
```

Benchmarking and keeping track

- In many practical data science and ML applications, using neural nets is a time consuming trail-and-error process
- Keep your hyper parameters and other program constants in a neat collection somewhere in your file
- After you finish the run, you can copy and save the entire script file in a backup directory with the results and brief description in the filename or whatever helps you remember Think of this as version control combined with benchmarking
- The you can spreadsheet your results for visual comparison



Take-home exercise

Additional reading

- A basic multilayer perceptron in Tensorflow: http://www.jessicayung.com/explaining-tensorflow-code-for-a-multilayer-perceptron/