Welcome to Session 01 ¶

This is a Google Colaboratory (https://colab.research.google.com/notebooks/welcome.ipynb) notebook file. Python programs are run directly in the browser—a great way to learn and use TensorFlow. To follow this tutorial, run the notebook in Google Colab by clicking the button at the top of this page.

1. In Colab, connect to a Python runtime: At the top-right of the menu bar, select *CONNECT*.
2. Run all the notebook code cells: Select *Runtime* > *Run all*.

In [0]:
```
#@title ## Mounting Gdrive

USE_G_COLAB = True #@param {type:"boolean"}

if USE_G_COLAB:
    from google.colab import drive


    drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

In [0]:
```
#@title ## Project Root

root_dir = ''

if USE_G_COLAB:
    root_dir = '/content/drive/My Drive/dl_app/' #@param {type:"string"}
```

In [0]:
```
!ls $root_dir
```

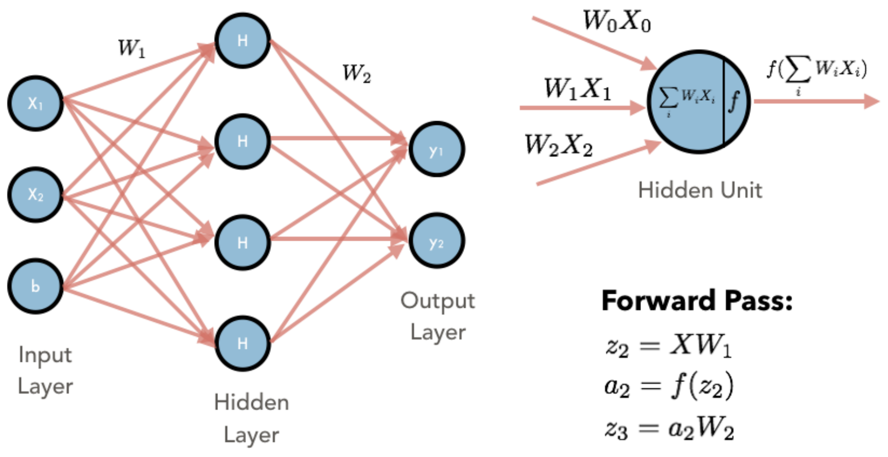
```
ls: cannot access '/content/drive/My': No such file or directory
ls: cannot access 'Drive/workshops/2019_07_21/sessions_01/': No such file or directory
```

In [0]:
```
#@title ## Installing requried packages

#@markdown ---
#@markdown - [TensorFlow 2.0-beta](https://www.tensorflow.org/install/gpu)
#@markdown - [Watermark](https://github.com/rasbt/watermark)
!pip install -q tensorflow-gpu==2.0.0-beta1
!pip install -qU watermark
```

In [0]:
```
#@title ## Custom Matplotlib Style
mpl_style = "https://gist.githubusercontent.com/m3hrdadfi/af8aca01094afb7d3e5b46de9ad8d509/raw/871ec5
d721a3b438c3c896718ea4aafc91ea9744/gadfly.mplstyle" #@param {type:"string"}

!wget -q $mpl_style -O /root/.config/matplotlib/matplotlibrc
```

In [0]:
```
#@title ## General Paramas

#@markdown > A random seed is a number used to initialize a pseudorandom number generator. For a seed
to be used in a pseudorandom number generator, it does not need to be random
RANDOM_SEED = 141 #@param {type:"integer"}
```

Overview



$z_2 = XW_1$

$a_2 = f(z_2)$

$z_3 = a_2 W_2$

$\hat{y} = softmax(z_3)$ # classification

where:

- X = inputs | $\in \mathbb{R}^{NXD}$ (D is the number of features)
- W_1 = 1st layer weights | $\in \mathbb{R}^{DXH}$ (H is the number of hidden units in layer 1)
- z_2 = outputs from first layer's weights $\in \mathbb{R}^{NXH}$
- f = non-linear activation function
- a_2 = activation applied first layer's outputs | $\in \mathbb{R}^{NXH}$
- W_2 = 2nd layer weights | $\in \mathbb{R}^{HXC}$ (C is the number of classes)
- $\hat{y}$ = prediction | $\in \mathbb{R}^{NXC}$ (N is the number of samples)

This is a simple two-layer MLP.

- **Objective:** Predict the probability of class y given the inputs X. Non-linearity is introduced to model the complex, non-linear data.
- **Advantages:**
  - Can model non-linear patterns in the data really well.

- **Disadvantages:**
  - Overfits easily.
  - Computationally intensive as network increases in size.
  - Not easily interpretable.
- **Miscellaneous:** Future neural network architectures that we'll see use the MLP as a modular unit for feed forward operations (affine transformation (XW) followed by a non-linear operation).

7 Common Nonlinear Activation Functions and How to Choose an Activation Function

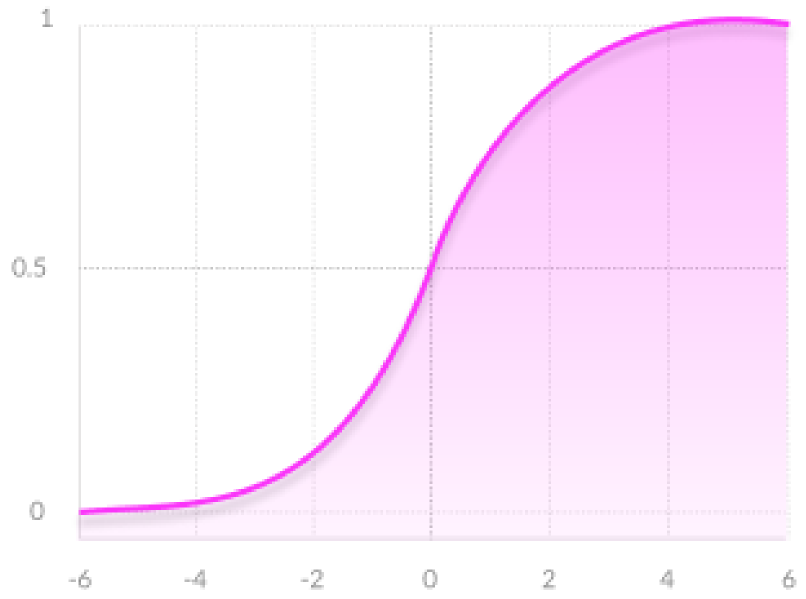
SIGMOID / LOGISTIC

ADVANTAGES

- Smooth gradient, preventing “jumps” in output values.
- Output values bound between 0 and 1, normalizing the output of each neuron.
- Clear predictions—For X above 2 or below -2, tends to bring the Y value (the prediction) to the edge of the curve, very close to 1 or 0. This enables clear predictions.

DISADVANTAGES

- Vanishing gradient—for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
- Outputs not zero centered.
- Computationally expensive



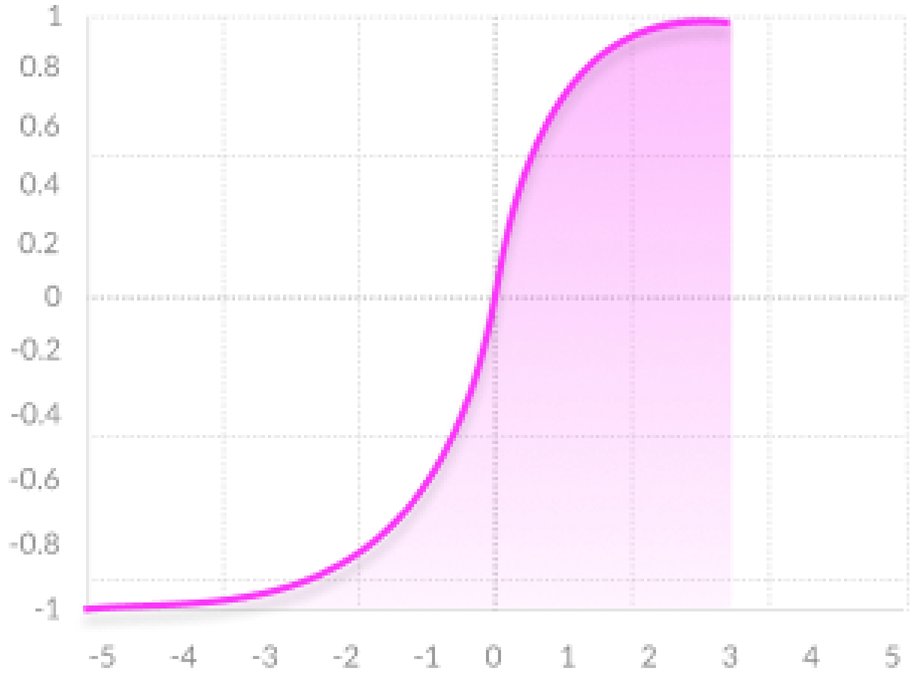
TANH / HYPERBOLIC TANGENT

ADVANTAGES

- Zero centered—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.
- Otherwise like the Sigmoid function.

DISADVANTAGES

- Like the Sigmoid function



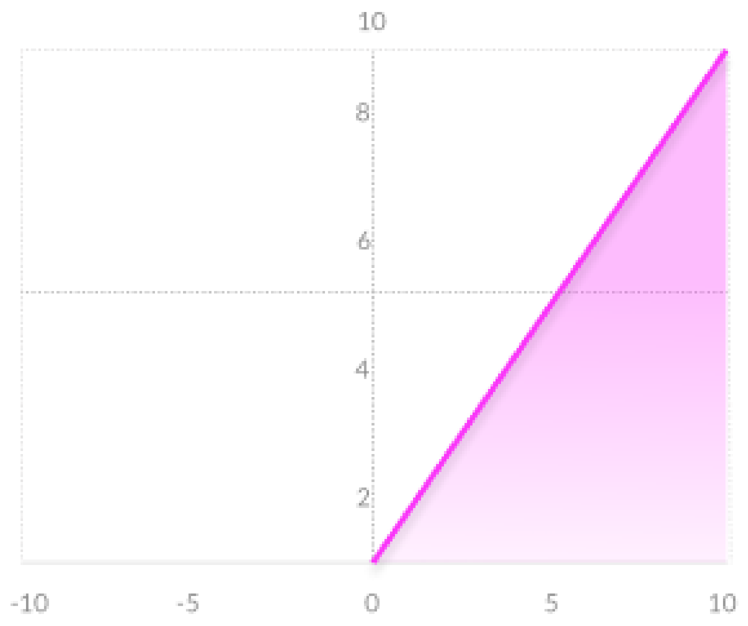
RELU (RECTIFIED LINEAR UNIT)

ADVANTAGES

- Computationally efficient—allows the network to converge very quickly
- Non-linear—although it looks like a linear function, ReLU has a derivative function and allows for backpropagation.

DISADVANTAGES

- The Dying ReLU problem—when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.
- Like the Sigmoid function

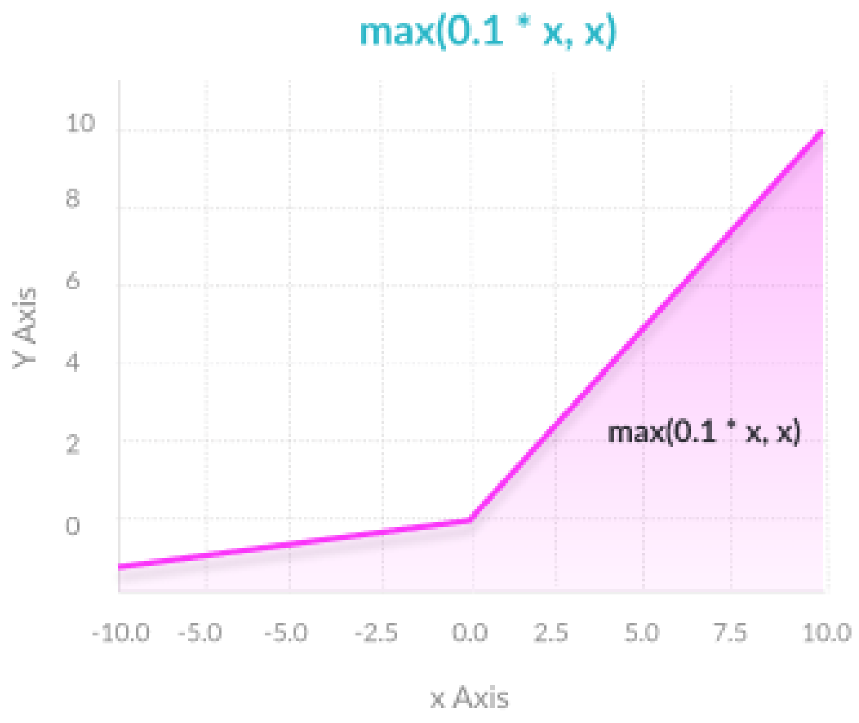


LEAKY RELU

ADVANTAGES

- Prevents dying ReLU problem—this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values Otherwise like ReLU

DISADVANTAGES Results not consistent—leaky ReLU does not provide consistent predictions for negative input values.



PARAMETRIC RELU

ADVANTAGES

- Allows the negative slope to be learned—unlike leaky ReLU, this function provides the slope of the negative part of the function as an argument. It is, therefore, possible to perform backpropagation and learn the most appropriate value of α. Otherwise like ReLU

DISADVANTAGES May perform differently for different problems.

$f(x) = max(\alpha x, x)$

SOFTMAX

ADVANTAGES

- Able to handle multiple classes only one class in other activation functions—normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class.

- Useful for output neurons—typically Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories.

$$\frac{e^{z_j}}{\sum_{k=1}^{c} e^{z_k}}$$

Training

Steps:

1. Randomly initialize the model's weights W (we'll cover more effective initalization strategies in future lessons).
2. Feed inputs X into the model to do the forward pass and receive the probabilities.
3. Compare the predictions $\hat{y}$ (ex. [0.3, 0.3, 0.4]]) with the actual target values y (ex. class 2 would look like [0, 0, 1]) with the objective (cost) function to determine loss J. A common objective function for classification tasks is cross-entropy loss.
   - $z_2 = XW_1$
   - $a_2 = max(0, z_2)$ # ReLU activation
   - $z_3 = a_2 W_2$
   - $\hat{y} = softmax(z_3)$
   - $J(\theta) = - \sum_i y_i ln(\hat{y_i})$
4. Calculate the gradient of loss $J(\theta)$ w.r.t to the model weights.
   - $\frac{\partial J}{\partial W_{2j}} = a_2 \hat{y}, \frac{\partial J}{\partial W_{2y}} = a_2 (\hat{y} - 1)$
   - $\frac{\partial J}{\partial W_1} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_2} \frac{\partial a_2}{\partial z_2} \frac{\partial z_2}{\partial W_1} = W_2 (\partial scores)(\partial ReLU) X$
5. Apply backpropagation to update the weights W using gradient descent. The updates will penalize the probabiltiy for the incorrect classes (j) and encourage a higher probability for the correct class (y).
   - $W_i = W_i - \alpha \frac{\partial J}{\partial W_i}$
6. Repeat steps 2 - 4 until model performs well.

Terminologies

Neural Network Learning as Optimization

A deep learning neural network learns to map a set of inputs to a set of outputs from training data.

We cannot calculate the perfect weights for a neural network; there are too many unknowns. Instead, the problem of learning is cast as a search or optimization problem and an algorithm is used to navigate the space of possible sets of weights the model may use in order to make good or good enough predictions.

Typically, a neural network model is trained using the stochastic gradient descent optimization algorithm and weights are updated using the backpropagation of error algorithm.

The “gradient” in gradient descent refers to an error gradient. The model with a given set of weights is used to make predictions and the error for those predictions is calculated.

The gradient descent algorithm seeks to change the weights so that the next evaluation reduces the error, meaning the optimization algorithm is navigating down the gradient (or slope) of error.

Now that we know that training neural nets solves an optimization problem, we can look at how the error of a given set of weights is calculated.

What Is a Loss Function and Loss?

In the context of an optimization algorithm, the function used to evaluate a candidate solution (i.e. a set of weights) is referred to as the objective function.

We may seek to maximize or minimize the objective function, meaning that we are searching for a candidate solution that has the highest or lowest score respectively.

Typically, with neural networks, we seek to minimize the error. As such, the objective function is often referred to as a cost function or a loss function and the value calculated by the loss function is referred to as simply “loss.” The cost or loss function has an important job in that it must faithfully distill all aspects of the model down into a single number in such a way that improvements in that number are a sign of a better model. We will review best practice or default values for each problem type with regard to the output layer and loss function.

Regression Problem

A problem where you predict a real-value quantity.

Output Layer Configuration: One node with a linear activation unit. Loss Function: Mean Squared Error (MSE).

Binary Classification Problem

A problem where you classify an example as belonging to one of two classes.

The problem is framed as predicting the likelihood of an example belonging to class one, e.g. the class that you assign the integer value 1, whereas the other class is assigned the value 0.

Output Layer Configuration: One node with a sigmoid activation unit. Loss Function: Cross-Entropy, also referred to as Logarithmic loss.

Multi-Class Classification Problem

A problem where you classify an example as belonging to one of more than two classes.

The problem is framed as predicting the likelihood of an example belonging to each class.

Output Layer Configuration: One node for each class using the softmax activation function. Loss Function: Cross-Entropy, also referred to as Logarithmic loss.

How to Implement Loss Functions

In order to make the loss functions concrete, this section explains how each of the main types of loss function works.

Mean Squared Error Loss

Mean Squared Error loss, or MSE for short, is calculated as the average of the squared differences between the predicted and actual values.

The result is always positive regardless of the sign of the predicted and actual values and a perfect value is 0.0. The loss value is minimized, although it can be used in a maximization optimization process by making the score negative.

Cross-Entropy Loss (or Log Loss)

Cross-entropy loss is often simply referred to as “cross-entropy,” “logarithmic loss,” “logistic loss,” or “log loss” for short.

Each predicted probability is compared to the actual class output value (0 or 1) and a score is calculated that penalizes the probability based on the distance from the expected value. The penalty is logarithmic, offering a small score for small differences (0.1 or 0.2) and enormous score for a large difference (0.9 or 1.0).

Cross-entropy loss is minimized, where smaller values represent a better model than larger values. A model that predicts perfect probabilities has a cross entropy or log loss of 0.0.

Cross-entropy for a binary or two class prediction problem is actually calculated as the average cross entropy across all examples.

An overview of gradient descent optimization algorithms

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent. These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by.

Gradient descent variants

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

Batch gradient descent

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:

$\theta = \theta - \eta \nabla_{\theta} J(\theta)$
We then update our parameters in the opposite direction of the gradients with the learning rate determining how big of an update we perform. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces. As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Batch gradient descent also doesn't allow us to update our model online, i.e. with new examples on-the-fly. ## Stochastic gradient descent Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example x(i) and label y(i):

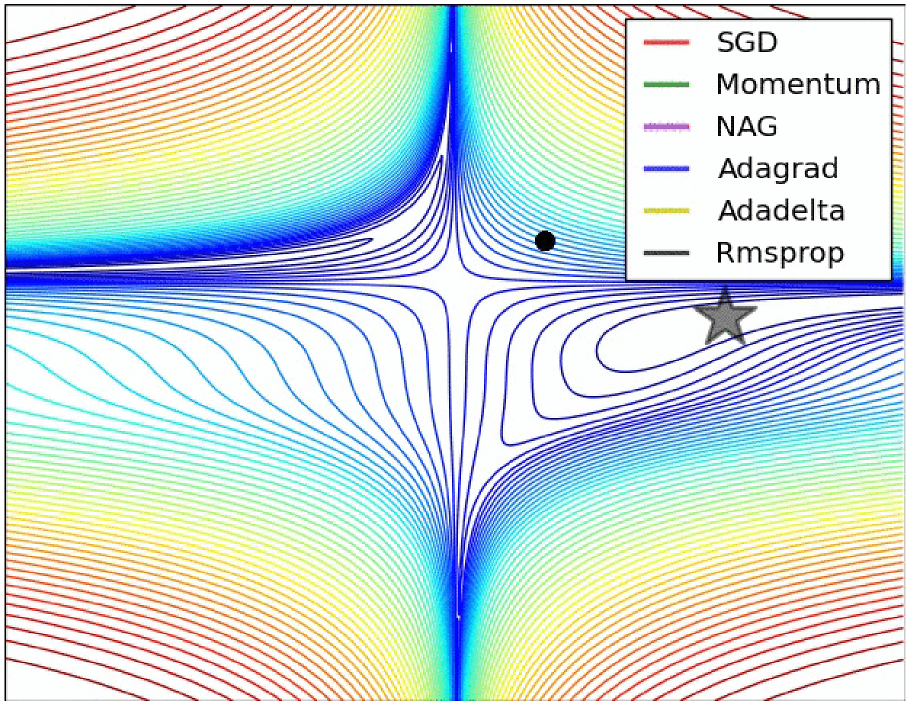
$\theta = \theta - \eta \nabla_{\theta} J(\theta; x^{i}; y^{i})$
Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively. ## Mini-batch gradient descent Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

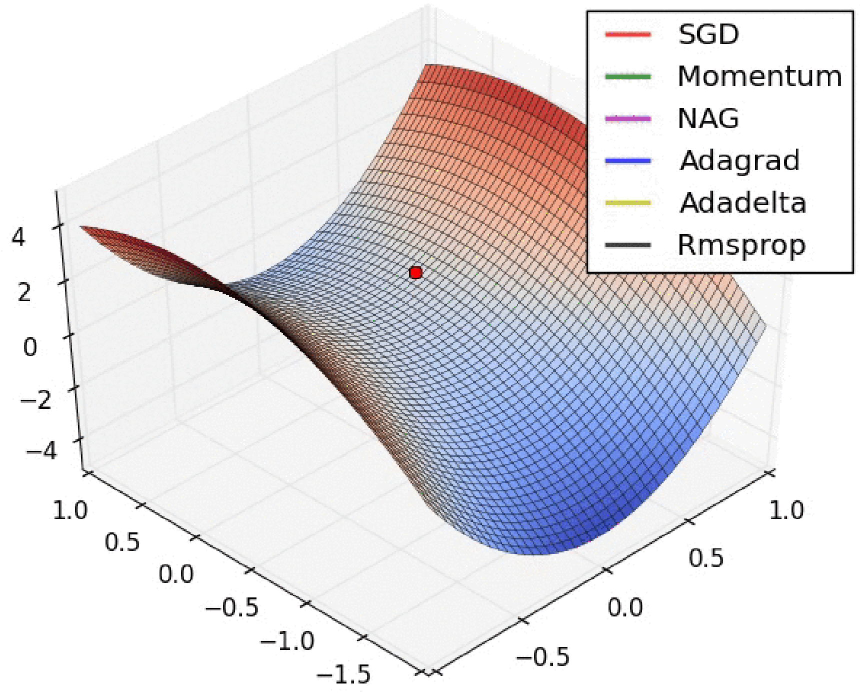
$\theta = \theta - \eta \nabla_\theta J(\theta; x^{i:i+n}; y^{i:i+n})$

This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

Gradient descent optimization algorithms

- Momentum
- Nesterov accelerated gradient (NAG)
- Adagrad
- Adadelta
- RMSprop
- Adam
- AdaMax
- Nadam
- AMSGrad





Which optimizer to use?

So, which optimizer should you now use? If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods. An additional benefit is that you won't need to tune the learning rate but likely achieve the best results with the default value.

In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. It is shown that its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice.

Hyperparameters

Model optimization is one of the toughest challenges in the implementation of machine learning solutions. Entire branches of machine learning and deep learning theory have been dedicated to the optimization of models. Typically, we think about model optimization as a process of regularly modifying the code of the model in order to minimize the testing error. However, deep learning optimization often entails fine tuning elements that live outside the model but that can heavily influence its behavior. Deep learning often refers to those hidden elements as hyperparameters as they are one of the most critical components of any machine learning application.

Some Examples of Hyperparameters

The number and diversity of hyperparameters in machine learning algorithms is very specific to each model. However, there some classic hyperparameters that we should always keep our eyes on and that should help you think about this aspect of machine learning solutions: · Learning Rate: The mother of all hyperparameters, the learning rate quantifies the learning progress of a model in a way that can be used to optimize its capacity. · Number of Hidden Units: A classic hyperparameter in deep learning algorithms, the number of hidden units is key to regulate the representational capacity of a model. · Convolution Kernel Width: In convolutional Neural Networks(CNNs), the Kernel Width influences the number of parameters in a model which, in turns, influences its capacity.

Metrics to Evaluate your Machine Learning Algorithm

Evaluating your machine learning algorithm is an essential part of any project. Your model may give you satisfying results when evaluated using a metric say accuracy_score but may give poor results when evaluated against other metrics such as logarithmic_loss or any other such metric. Most of the times we use classification accuracy to measure the performance of our model, however it is not enough to truly judge our model. In this post, we will cover different types of evaluation metrics available.

Classification Accuracy

Classification Accuracy is what we usually mean, when we use the term accuracy. It is the ratio of number of correct predictions to the total number of input samples.

$$Accuracy = \frac{Number of Correct predictions}{Total Number of Predictions made}$$

It works well only if there are equal number of samples belonging to each class. For example, consider that there are 98% samples of class A and 2% samples of class B in our training set. Then our model can easily get 98% training accuracy by simply predicting every training sample belonging to class A. When the same model is tested on a test set with 60% samples of class A and 40% samples of class B, then the test accuracy would drop down to 60%. Classification Accuracy is great, but gives us the false sense of achieving high accuracy. The real problem arises, when the cost of misclassification of the minor class samples are very high. If we deal with a rare but fatal disease, the cost of failing to diagnose the disease of a sick person is much higher than the cost of sending a healthy person to more tests.

Confusion Matrix

Confusion Matrix as the name suggests gives us a matrix as output and describes the complete performance of the model. Lets assume we have a binary classification problem. We have some samples belonging to two classes : YES or NO. Also, we have our own classifier which predicts a class for a given input sample. On testing our model on 165 samples ,we get the following result.

$$LogLoss = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot log_e(\hat{y_i}) + (1-y_i)\cdot log_e(1-\hat{y_i})]$$

where, y_ij, indicates whether sample i belongs to class j or not p_ij, indicates the probability of sample i belonging to class j Log Loss has no upper bound and it exists on the range Log Loss nearer to 0 indicates higher accuracy, whereas if the Log Loss is away from 0 then it indicates lower accuracy. In general, minimising Log Loss gives greater accuracy for the classifier.

n=165	Predicted: NO	Predicted: YES
Actual: NO	50	10
Actual: YES	5	100

There are 4 important terms :

- True Positives : The cases in which we predicted YES and the actual output was also YES.
- True Negatives : The cases in which we predicted NO and the actual output was NO.
- False Positives : The cases in which we predicted YES and the actual output was NO.
- False Negatives : The cases in which we predicted NO and the actual output was YES. Accuracy for the matrix can be calculated by taking average of the values lying across the “main diagonal” i.e
  $Accuracy = \frac{TruePositive+TrueNegative}{TotalNumberofSamples}$
  $Accuracy = \frac{100+50}{165} = 0.91$

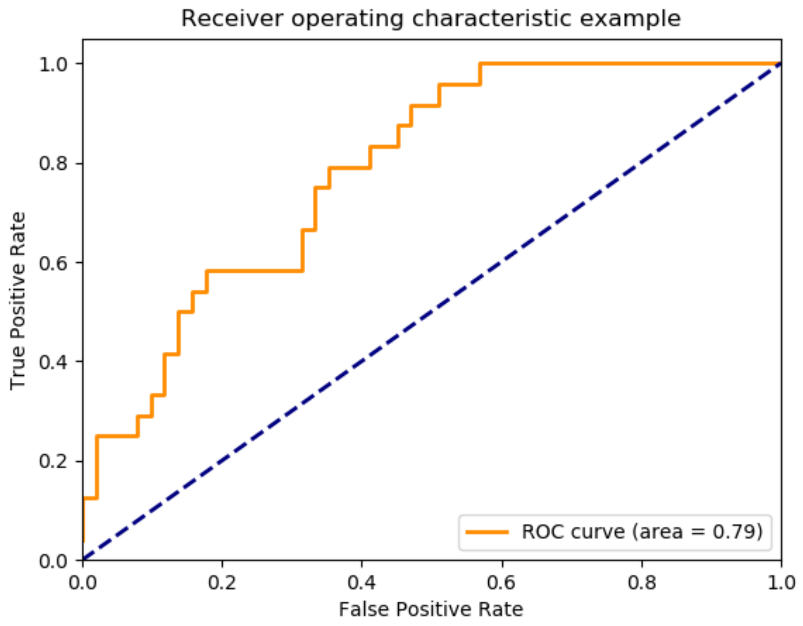
Confusion Matrix forms the basis for the other types of metrics.

Area Under Curve

Area Under Curve(AUC) is one of the most widely used metrics for evaluation. It is used for binary classification problem. AUC of a classifier is equal to the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. Before defining AUC, let us understand two basic terms :

- True Positive Rate (Sensitivity) : True Positive Rate is defined as TP/ (FN+TP). True Positive Rate corresponds to the proportion of positive data points that are correctly considered as positive, with respect to all positive data points.
  $TruePositiveRate = \frac{TruePositive}{FalseNegative+TruePositive}$
- False Positive Rate (Specificity) : False Positive Rate is defined as FP / (FP+TN). False Positive Rate corresponds to the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points.
  $FalsePositiveRate = \frac{FalsePositive}{TrueNegative+FalsePositive}$

False Positive Rate and True Positive Rate both have values in the range [0, 1]. FPR and TPR bot hare computed at threshold values such as (0.00, 0.02, 0.04, …., 1.00) and a graph is drawn. AUC is the area under the curve of plot False Positive Rate vs True Positive Rate at different points in [0, 1].



As evident, AUC has a range of [0, 1]. The greater the value, the better is the performance of our model.

F1 Score

F1 Score is used to measure a test's accuracy F1 Score is the Harmonic Mean between precision and recall. The range for F1 Score is [0, 1]. It tells you how precise your classifier is (how many instances it classifies correctly), as well as how robust it is (it does not miss a significant number of instances). High precision but lower recall, gives you an extremely accurate, but it then misses a large number of instances that are difficult to classify. The greater the F1 Score, the better is the performance of our model. Mathematically, it can be expressed as :

$$F1 = 2 * \frac{1}{\frac{1}{precision} + \frac{1}{recall}}$$

- Precision : It is the number of correct positive results divided by the number of positive results predicted by the classifier.

$$Precision = 2 * \frac{TruePositive}{TruePositive + FalsePositive}$$

Recall : It is the number of correct positive results divided by the number of all relevant samples (all samples that should have been identified as positive).

$$Recall = 2 * \frac{TruePositive}{TruePositive + FalsePositive}$$

Preparation

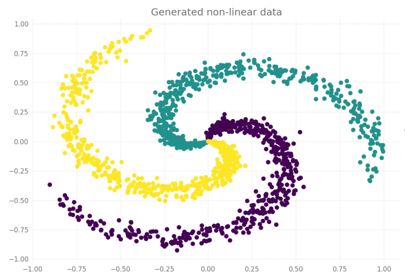
In [0]:
```
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf

import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

import matplotlib as mpl
import matplotlib.pyplot as plt

import itertools
from datetime import datetime
from tqdm import tqdm
import os

%matplotlib inline
mpl.rc_file(mpl.matplotlib_fname())
```

In [0]:
```
%reload_ext watermark
%watermark -m -n -p tensorflow,numpy,matplotlib,sklearn -g
```

```
Sun Jul 21 2019

tensorflow 2.0.0-beta1
numpy 1.16.4
matplotlib 3.0.3
sklearn 0.21.2

compiler   : GCC 8.0.1 20180414 (experimental) [trunk revision 259383
system     : Linux
release    : 4.14.79+
machine    : x86_64
processor  : x86_64
CPU cores  : 2
interpreter: 64bit
Git hash   :
```

In [0]:
```
#@title ## Random seed

np.random.seed(RANDOM_SEED)
```

Data



We're going to first generate some non-linear data for a classification task.

A spiral: is a curve which emanates from a point, moving farther away as it revolves around the point.

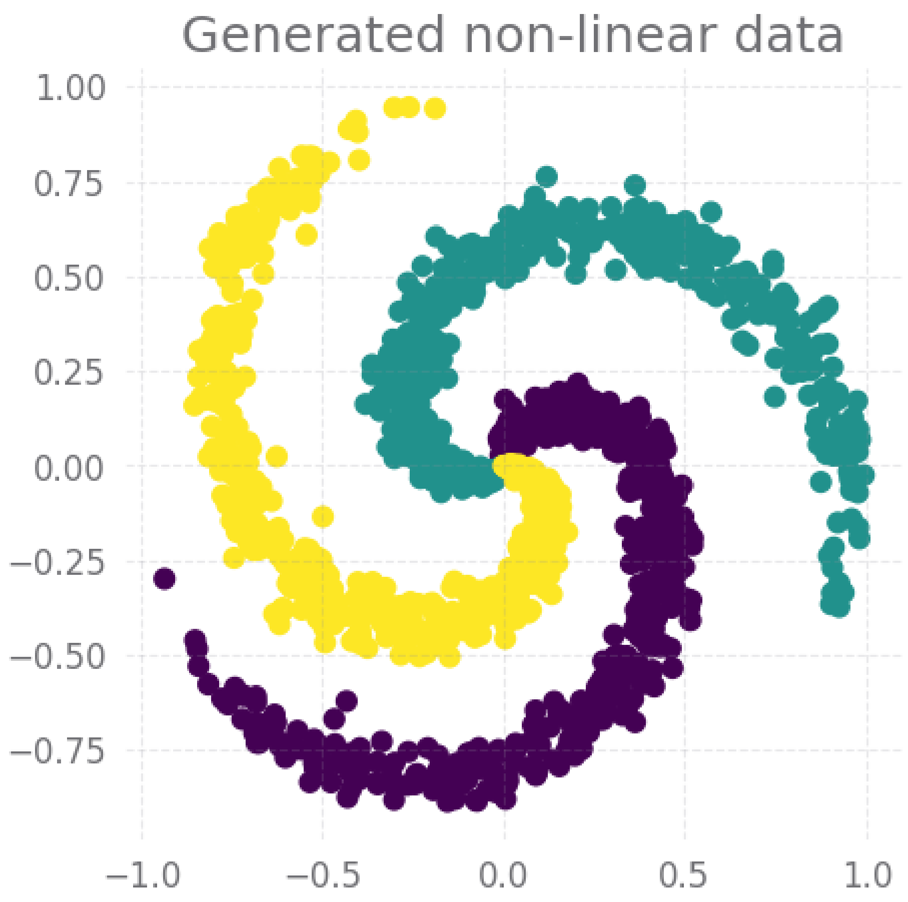
$$Spiral = \begin{cases} X_{1\theta} = r_\theta \cos(\theta) \\ X_{2\theta} = r_\theta \sin(\theta) \end{cases}$$

In [0]:
```
def generate_data(num_samples, dimensions, num_classes):
    """ Generate non-linear dataset.

    Args:
        num_samples (int): number of samples which we want to produce.
        dimensions (int): the dimension of the data which we plan to produce ex. 2d.
        num_classes (int): number of classes which the samples are going to generate ex. 2#.

    Examples:
        >>> x, y = generate_data(2, 2, 2)
        (array([[ 0.        ,  0.        ],
        [-0.712528  , -0.70164368],
        [-0.        , -0.        ],
        [ 0.90006129, -0.43576333]]), array([0, 0, 1, 1], dtype=uint8))

    Returns:
        x (ndarray): a numpy array in a shape like this (num_samples, dimensions).
        y (ndarray): a numpy array in a shape like this (num_samples, num_classes).
    """

    x_origin = np.zeros((num_samples * num_classes, dimensions))
    y = np.zeros(num_samples * num_classes, dtype='uint8')

    for i in tqdm(range(num_classes), position=0):
        idx = range(num_samples * i, num_samples * (i + 1))
        radius = np.linspace(0.0, 1, num_samples)
        theta = np.linspace(i * 4, (i + 1) * 4, num_samples) + np.random.randn(num_samples) * 0.2

        x_origin[idx] = np.c_[radius * np.sin(theta), radius * np.cos(theta)]
        y[idx] = i

    x = np.hstack([x_origin])

    return x, y
```

In [0]:
```
num_samples = 500
dimensions = 2
num_classes = 3

x, y = generate_data(num_samples, dimensions, num_classes)

print()
print('x: %s' % str(x.shape))
print('y: %s' % str(y.shape))
```

```
100%|██████████| 3/3 [00:00<00:00, 1237.62it/s]

x: (1500, 2)
y: (1500,)
```

In [0]:
```
xfig = 5.0
yfig = 5.0

plt.figure(figsize=(xfig, yfig))
plt.title('Generated non-linear data')
plt.scatter(x[:, 0], x[:, 1], c=y)
plt.show()
```



One-hot Encoding

Consider an array of 5 labels out of a set of 3 classes {0, 1, 2}:

```
> labels
array([0, 2, 1, 2, 0])
```

`to_categorical` converts this into a matrix with as many columns as there are classes. The number of rows stays the same.

```
> to_categorical(labels)
array([[ 1.,  0.,  0.],
       [ 0.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.],
       [ 1.,  0.,  0.]], dtype=float32)
```

---

to_categorical

```
tf.keras.utils.to_categorical(y, num_classes=None, dtype='float32')
```

Converts a class vector (integers) to binary class matrix.

Arguments:

- **`y`** : class vector to be converted into a matrix (integers from 0 to num_classes).
- **`num_classes`** : total number of classes.
- **`dtype`** : The data type expected by the input, as a string (float32, float64, int32...)

In [0]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=RANDOM_SEED)

print(x_train.shape, y_train.shape)

y_train_c = tf.keras.utils.to_categorical(y_train, num_classes)
y_test_c = tf.keras.utils.to_categorical(y_test, num_classes)

print(y_train_c.shape)
```

```
(1050, 2) (1050,)
(1050, 3)
```

Linear Model

Before we get to our neural network, we're going to implement a linear model (logistic regression). We want to see why linear models won't suffice for our dataset.

In [0]:
```
def build_linear(n_units, n_features):
    model = tf.keras.Sequential([
        tf.keras.layers.Input(shape=[n_features]),
        tf.keras.layers.Dense(n_units, activation='softmax')
    ])

    opt = tf.keras.optimizers.Adam(lr=0.0001)
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    return model
```

In [0]:
```
linear_model = build_linear(n_units=3, n_features=2)
linear_model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_3 (Dense)              (None, 3)                 9         
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

In [0]:
```
r = linear_model.fit(x_train, y_train_c,
                     validation_split=0.1,
                     epochs=100,
                     verbose=1)


history_dict = r.history
history_list = list(history_dict.keys())
print(history_list)
```

```
Train on 945 samples, validate on 105 samples
Epoch 1/100
945/945 [==============================] - 0s 227us/sample - loss: 0.9801 - accuracy: 0.4381 - val_l
oss: 0.9817 - val_accuracy: 0.4381
Epoch 2/100
945/945 [==============================] - 0s 81us/sample - loss: 0.9793 - accuracy: 0.4392 - val_lo
ss: 0.9810 - val_accuracy: 0.4381
Epoch 3/100
945/945 [==============================] - 0s 75us/sample - loss: 0.9785 - accuracy: 0.4402 - val_lo
ss: 0.9803 - val_accuracy: 0.4381
Epoch 4/100
945/945 [==============================] - 0s 76us/sample - loss: 0.9776 - accuracy: 0.4413 - val_lo
ss: 0.9796 - val_accuracy: 0.4381
Epoch 5/100
945/945 [==============================] - 0s 76us/sample - loss: 0.9768 - accuracy: 0.4413 - val_lo
ss: 0.9790 - val_accuracy: 0.4381
Epoch 6/100
945/945 [==============================] - 0s 75us/sample - loss: 0.9760 - accuracy: 0.4423 - val_lo
ss: 0.9783 - val_accuracy: 0.4381
Epoch 7/100
945/945 [==============================] - 0s 91us/sample - loss: 0.9752 - accuracy: 0.4444 - val_lo
ss: 0.9777 - val_accuracy: 0.4381
Epoch 8/100
945/945 [==============================] - 0s 78us/sample - loss: 0.9744 - accuracy: 0.4476 - val_lo
ss: 0.9770 - val_accuracy: 0.4381
Epoch 9/100
945/945 [==============================] - 0s 74us/sample - loss: 0.9735 - accuracy: 0.4476 - val_lo
ss: 0.9764 - val_accuracy: 0.4381
Epoch 10/100
945/945 [==============================] - 0s 72us/sample - loss: 0.9728 - accuracy: 0.4508 - val_lo
ss: 0.9757 - val_accuracy: 0.4381
Epoch 11/100
945/945 [==============================] - 0s 75us/sample - loss: 0.9720 - accuracy: 0.4519 - val_lo
ss: 0.9751 - val_accuracy: 0.4381
Epoch 12/100
945/945 [==============================] - 0s 71us/sample - loss: 0.9711 - accuracy: 0.4561 - val_lo
ss: 0.9744 - val_accuracy: 0.4381
Epoch 13/100
945/945 [==============================] - 0s 75us/sample - loss: 0.9703 - accuracy: 0.4571 - val_lo
ss: 0.9738 - val_accuracy: 0.4476
Epoch 14/100
945/945 [==============================] - 0s 73us/sample - loss: 0.9696 - accuracy: 0.4593 - val_lo
ss: 0.9731 - val_accuracy: 0.4476
```

```
Epoch 15/100
945/945 [==============================] - 0s 75us/sample - loss: 0.9688 - accuracy: 0.4603 - val_lo
ss: 0.9725 - val_accuracy: 0.4476
Epoch 16/100
945/945 [==============================] - 0s 71us/sample - loss: 0.9680 - accuracy: 0.4614 - val_lo
ss: 0.9719 - val_accuracy: 0.4476
Epoch 17/100
945/945 [==============================] - 0s 72us/sample - loss: 0.9672 - accuracy: 0.4603 - val_lo
ss: 0.9712 - val_accuracy: 0.4476
Epoch 18/100
945/945 [==============================] - 0s 77us/sample - loss: 0.9664 - accuracy: 0.4603 - val_lo
ss: 0.9706 - val_accuracy: 0.4476
Epoch 19/100
945/945 [==============================] - 0s 80us/sample - loss: 0.9656 - accuracy: 0.4603 - val_lo
ss: 0.9700 - val_accuracy: 0.4476
Epoch 20/100
945/945 [==============================] - 0s 77us/sample - loss: 0.9649 - accuracy: 0.4614 - val_lo
ss: 0.9693 - val_accuracy: 0.4476
Epoch 21/100
945/945 [==============================] - 0s 88us/sample - loss: 0.9641 - accuracy: 0.4614 - val_lo
ss: 0.9687 - val_accuracy: 0.4571
Epoch 22/100
945/945 [==============================] - 0s 71us/sample - loss: 0.9633 - accuracy: 0.4635 - val_lo
ss: 0.9681 - val_accuracy: 0.4571
Epoch 23/100
945/945 [==============================] - 0s 71us/sample - loss: 0.9626 - accuracy: 0.4667 - val_lo
ss: 0.9675 - val_accuracy: 0.4571
Epoch 24/100
945/945 [==============================] - 0s 77us/sample - loss: 0.9618 - accuracy: 0.4667 - val_lo
ss: 0.9669 - val_accuracy: 0.4571
Epoch 25/100
945/945 [==============================] - 0s 78us/sample - loss: 0.9610 - accuracy: 0.4677 - val_lo
ss: 0.9663 - val_accuracy: 0.4571
Epoch 26/100
945/945 [==============================] - 0s 76us/sample - loss: 0.9603 - accuracy: 0.4698 - val_lo
ss: 0.9657 - val_accuracy: 0.4571
Epoch 27/100
945/945 [==============================] - 0s 75us/sample - loss: 0.9595 - accuracy: 0.4709 - val_lo
ss: 0.9650 - val_accuracy: 0.4571
Epoch 28/100
945/945 [==============================] - 0s 76us/sample - loss: 0.9588 - accuracy: 0.4709 - val_lo
ss: 0.9645 - val_accuracy: 0.4667
Epoch 29/100
```

```
945/945 [==============================] - 0s 71us/sample - loss: 0.9580 - accuracy: 0.4698 - val_lo
ss: 0.9639 - val_accuracy: 0.4667
Epoch 30/100
945/945 [==============================] - 0s 64us/sample - loss: 0.9573 - accuracy: 0.4709 - val_lo
ss: 0.9632 - val_accuracy: 0.4571
Epoch 31/100
945/945 [==============================] - 0s 72us/sample - loss: 0.9565 - accuracy: 0.4720 - val_lo
ss: 0.9626 - val_accuracy: 0.4571
Epoch 32/100
945/945 [==============================] - 0s 63us/sample - loss: 0.9558 - accuracy: 0.4741 - val_lo
ss: 0.9620 - val_accuracy: 0.4571
Epoch 33/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9550 - accuracy: 0.4741 - val_lo
ss: 0.9614 - val_accuracy: 0.4571
Epoch 34/100
945/945 [==============================] - 0s 64us/sample - loss: 0.9543 - accuracy: 0.4772 - val_lo
ss: 0.9608 - val_accuracy: 0.4571
Epoch 35/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9535 - accuracy: 0.4783 - val_lo
ss: 0.9602 - val_accuracy: 0.4571
Epoch 36/100
945/945 [==============================] - 0s 85us/sample - loss: 0.9528 - accuracy: 0.4783 - val_lo
ss: 0.9596 - val_accuracy: 0.4571
Epoch 37/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9521 - accuracy: 0.4783 - val_lo
ss: 0.9591 - val_accuracy: 0.4667
Epoch 38/100
945/945 [==============================] - 0s 67us/sample - loss: 0.9513 - accuracy: 0.4783 - val_lo
ss: 0.9585 - val_accuracy: 0.4667
Epoch 39/100
945/945 [==============================] - 0s 62us/sample - loss: 0.9506 - accuracy: 0.4804 - val_lo
ss: 0.9579 - val_accuracy: 0.4667
Epoch 40/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9499 - accuracy: 0.4804 - val_lo
ss: 0.9573 - val_accuracy: 0.4667
Epoch 41/100
945/945 [==============================] - 0s 67us/sample - loss: 0.9492 - accuracy: 0.4804 - val_lo
ss: 0.9567 - val_accuracy: 0.4667
Epoch 42/100
945/945 [==============================] - 0s 72us/sample - loss: 0.9484 - accuracy: 0.4794 - val_lo
ss: 0.9561 - val_accuracy: 0.4762
Epoch 43/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9477 - accuracy: 0.4794 - val_lo
```

```
ss: 0.9556 - val_accuracy: 0.4667
Epoch 44/100
945/945 [==============================] - 0s 65us/sample - loss: 0.9470 - accuracy: 0.4794 - val_lo
ss: 0.9550 - val_accuracy: 0.4667
Epoch 45/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9463 - accuracy: 0.4794 - val_lo
ss: 0.9544 - val_accuracy: 0.4762
Epoch 46/100
945/945 [==============================] - 0s 66us/sample - loss: 0.9456 - accuracy: 0.4794 - val_lo
ss: 0.9538 - val_accuracy: 0.4667
Epoch 47/100
945/945 [==============================] - 0s 65us/sample - loss: 0.9449 - accuracy: 0.4804 - val_lo
ss: 0.9533 - val_accuracy: 0.4667
Epoch 48/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9442 - accuracy: 0.4836 - val_lo
ss: 0.9527 - val_accuracy: 0.4667
Epoch 49/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9435 - accuracy: 0.4847 - val_lo
ss: 0.9521 - val_accuracy: 0.4667
Epoch 50/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9427 - accuracy: 0.4857 - val_lo
ss: 0.9516 - val_accuracy: 0.4667
Epoch 51/100
945/945 [==============================] - 0s 86us/sample - loss: 0.9420 - accuracy: 0.4857 - val_lo
ss: 0.9510 - val_accuracy: 0.4762
Epoch 52/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9413 - accuracy: 0.4857 - val_lo
ss: 0.9504 - val_accuracy: 0.4762
Epoch 53/100
945/945 [==============================] - 0s 63us/sample - loss: 0.9406 - accuracy: 0.4899 - val_lo
ss: 0.9499 - val_accuracy: 0.4762
Epoch 54/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9399 - accuracy: 0.4921 - val_lo
ss: 0.9493 - val_accuracy: 0.4857
Epoch 55/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9392 - accuracy: 0.4931 - val_lo
ss: 0.9487 - val_accuracy: 0.4857
Epoch 56/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9386 - accuracy: 0.4931 - val_lo
ss: 0.9482 - val_accuracy: 0.4857
Epoch 57/100
945/945 [==============================] - 0s 73us/sample - loss: 0.9379 - accuracy: 0.4963 - val_lo
ss: 0.9476 - val_accuracy: 0.4857
```

```
Epoch 58/100
945/945 [==============================] - 0s 67us/sample - loss: 0.9372 - accuracy: 0.4952 - val_lo
ss: 0.9471 - val_accuracy: 0.4857
Epoch 59/100
945/945 [==============================] - 0s 63us/sample - loss: 0.9365 - accuracy: 0.4974 - val_lo
ss: 0.9465 - val_accuracy: 0.4857
Epoch 60/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9358 - accuracy: 0.4995 - val_lo
ss: 0.9460 - val_accuracy: 0.4857
Epoch 61/100
945/945 [==============================] - 0s 72us/sample - loss: 0.9351 - accuracy: 0.4995 - val_lo
ss: 0.9454 - val_accuracy: 0.4857
Epoch 62/100
945/945 [==============================] - 0s 64us/sample - loss: 0.9344 - accuracy: 0.4995 - val_lo
ss: 0.9448 - val_accuracy: 0.4952
Epoch 63/100
945/945 [==============================] - 0s 71us/sample - loss: 0.9338 - accuracy: 0.5026 - val_lo
ss: 0.9443 - val_accuracy: 0.4952
Epoch 64/100
945/945 [==============================] - 0s 63us/sample - loss: 0.9331 - accuracy: 0.5037 - val_lo
ss: 0.9437 - val_accuracy: 0.4952
Epoch 65/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9324 - accuracy: 0.5037 - val_lo
ss: 0.9432 - val_accuracy: 0.4952
Epoch 66/100
945/945 [==============================] - 0s 82us/sample - loss: 0.9318 - accuracy: 0.5058 - val_lo
ss: 0.9427 - val_accuracy: 0.4952
Epoch 67/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9311 - accuracy: 0.5037 - val_lo
ss: 0.9421 - val_accuracy: 0.4952
Epoch 68/100
945/945 [==============================] - 0s 75us/sample - loss: 0.9304 - accuracy: 0.5058 - val_lo
ss: 0.9416 - val_accuracy: 0.4952
Epoch 69/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9298 - accuracy: 0.5058 - val_lo
ss: 0.9411 - val_accuracy: 0.4952
Epoch 70/100
945/945 [==============================] - 0s 63us/sample - loss: 0.9291 - accuracy: 0.5058 - val_lo
ss: 0.9405 - val_accuracy: 0.4952
Epoch 71/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9284 - accuracy: 0.5048 - val_lo
ss: 0.9400 - val_accuracy: 0.4952
Epoch 72/100
```

```
945/945 [==============================] - 0s 64us/sample - loss: 0.9278 - accuracy: 0.5058 - val_lo
ss: 0.9394 - val_accuracy: 0.4952
Epoch 73/100
945/945 [==============================] - 0s 73us/sample - loss: 0.9271 - accuracy: 0.5048 - val_lo
ss: 0.9389 - val_accuracy: 0.4952
Epoch 74/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9264 - accuracy: 0.5058 - val_lo
ss: 0.9384 - val_accuracy: 0.4952
Epoch 75/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9258 - accuracy: 0.5058 - val_lo
ss: 0.9379 - val_accuracy: 0.4952
Epoch 76/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9251 - accuracy: 0.5069 - val_lo
ss: 0.9373 - val_accuracy: 0.4952
Epoch 77/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9245 - accuracy: 0.5079 - val_lo
ss: 0.9368 - val_accuracy: 0.4952
Epoch 78/100
945/945 [==============================] - 0s 63us/sample - loss: 0.9239 - accuracy: 0.5101 - val_lo
ss: 0.9363 - val_accuracy: 0.4952
Epoch 79/100
945/945 [==============================] - 0s 73us/sample - loss: 0.9232 - accuracy: 0.5111 - val_lo
ss: 0.9358 - val_accuracy: 0.4952
Epoch 80/100
945/945 [==============================] - 0s 62us/sample - loss: 0.9225 - accuracy: 0.5111 - val_lo
ss: 0.9352 - val_accuracy: 0.4952
Epoch 81/100
945/945 [==============================] - 0s 86us/sample - loss: 0.9219 - accuracy: 0.5090 - val_lo
ss: 0.9347 - val_accuracy: 0.4952
Epoch 82/100
945/945 [==============================] - 0s 72us/sample - loss: 0.9213 - accuracy: 0.5101 - val_lo
ss: 0.9342 - val_accuracy: 0.4952
Epoch 83/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9206 - accuracy: 0.5101 - val_lo
ss: 0.9337 - val_accuracy: 0.4857
Epoch 84/100
945/945 [==============================] - 0s 64us/sample - loss: 0.9200 - accuracy: 0.5111 - val_lo
ss: 0.9332 - val_accuracy: 0.4857
Epoch 85/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9193 - accuracy: 0.5111 - val_lo
ss: 0.9327 - val_accuracy: 0.4857
Epoch 86/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9187 - accuracy: 0.5122 - val_lo
```

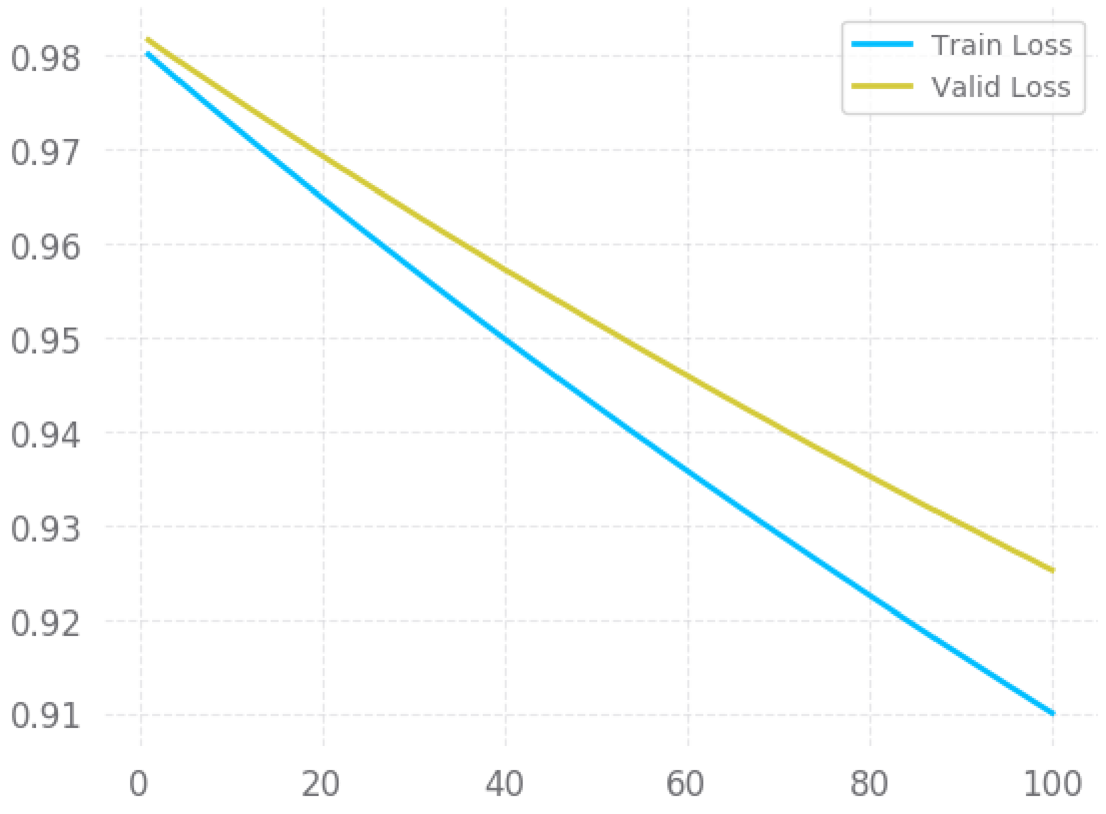
```
ss: 0.9321 - val_accuracy: 0.4857
Epoch 87/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9181 - accuracy: 0.5122 - val_lo
ss: 0.9316 - val_accuracy: 0.4857
Epoch 88/100
945/945 [==============================] - 0s 66us/sample - loss: 0.9174 - accuracy: 0.5122 - val_lo
ss: 0.9312 - val_accuracy: 0.4857
Epoch 89/100
945/945 [==============================] - 0s 70us/sample - loss: 0.9168 - accuracy: 0.5143 - val_lo
ss: 0.9306 - val_accuracy: 0.4857
Epoch 90/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9162 - accuracy: 0.5143 - val_lo
ss: 0.9302 - val_accuracy: 0.4857
Epoch 91/100
945/945 [==============================] - 0s 63us/sample - loss: 0.9156 - accuracy: 0.5143 - val_lo
ss: 0.9297 - val_accuracy: 0.4857
Epoch 92/100
945/945 [==============================] - 0s 67us/sample - loss: 0.9150 - accuracy: 0.5153 - val_lo
ss: 0.9292 - val_accuracy: 0.4857
Epoch 93/100
945/945 [==============================] - 0s 72us/sample - loss: 0.9143 - accuracy: 0.5153 - val_lo
ss: 0.9287 - val_accuracy: 0.4857
Epoch 94/100
945/945 [==============================] - 0s 62us/sample - loss: 0.9137 - accuracy: 0.5153 - val_lo
ss: 0.9282 - val_accuracy: 0.4857
Epoch 95/100
945/945 [==============================] - 0s 76us/sample - loss: 0.9131 - accuracy: 0.5175 - val_lo
ss: 0.9277 - val_accuracy: 0.4762
Epoch 96/100
945/945 [==============================] - 0s 89us/sample - loss: 0.9125 - accuracy: 0.5164 - val_lo
ss: 0.9272 - val_accuracy: 0.4762
Epoch 97/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9119 - accuracy: 0.5175 - val_lo
ss: 0.9267 - val_accuracy: 0.4762
Epoch 98/100
945/945 [==============================] - 0s 69us/sample - loss: 0.9113 - accuracy: 0.5175 - val_lo
ss: 0.9262 - val_accuracy: 0.4762
Epoch 99/100
945/945 [==============================] - 0s 65us/sample - loss: 0.9106 - accuracy: 0.5153 - val_lo
ss: 0.9257 - val_accuracy: 0.4762
Epoch 100/100
945/945 [==============================] - 0s 68us/sample - loss: 0.9100 - accuracy: 0.5175 - val_lo
```

```
ss: 0.9252 - val_accuracy: 0.4762
['loss', 'accuracy', 'val_loss', 'val_accuracy']
```

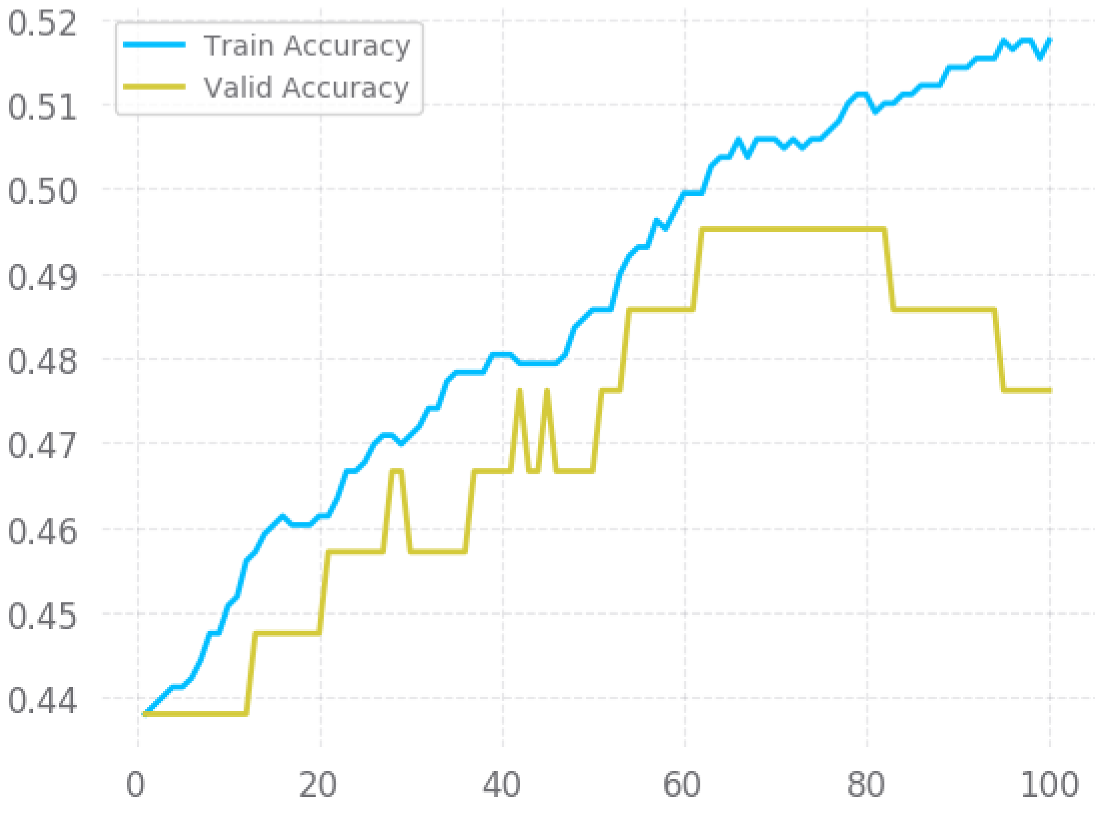
In [0]:
```
evaluation = linear_model.evaluate(x_test, y_test_c, verbose=0)
print(evaluation)
```

```
[0.9198821216159396, 0.5311111]
```

In [0]:
```
loss = history_dict['loss']
val_loss = history_dict['val_loss']
epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, label='Train Loss')
plt.plot(epochs, val_loss, label='Valid Loss')
plt.legend()
plt.show()
```



In [0]:
```
accuracy = history_dict['accuracy']
val_accuracy = history_dict['val_accuracy']
epochs = range(1, len(accuracy) + 1)

plt.plot(epochs, accuracy, label='Train Accuracy')
plt.plot(epochs, val_accuracy, label='Valid Accuracy')
plt.legend()
plt.show()
```



In [0]:
```
#@title ## Multiclass decision boundary

def plot_mc_decision_boundary(model, x, y, steps=1000):
    """ Plot multiclass decision boundary,

    Args:
        model (keras.Model): a keras model.
        x (ndarray): a numpy array.
        y (ndarray): a numpy array.
        steps (integer): number of linear spaces.

    Returns:
        None
    """
    x_min, x_max = x[:, 0].min() - 0.1, x[:, 0].max() + 0.1
    y_min, y_max = x[:, 1].min() - 0.1, x[:, 1].max() + 0.1

    x_span = np.linspace(x_min, x_max, steps)
    y_span = np.linspace(y_min, y_max, steps)
    xx, yy = np.meshgrid(x_span, y_span)

    y_pred = model.predict(np.c_[xx.ravel(), yy.ravel()])
    y_pred = np.argmax(y_pred, axis=1)
    y_pred = y_pred.reshape(xx.shape)

    plt.contourf(xx, yy, y_pred, alpha=0.5)
    plt.scatter(x[:, 0], x[:, 1], c=y)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())

    return None
```

In [0]:
```
#@title ## Visualize the decision boundary
```

In [0]:
```
#@title ## Plot confusion matrix

def plot_confusion_matrix(cm, classes):
    """ Plot confusion matrix

    Args:
        cm (ndarray): the input of confusion matrix.
        classes (integer): number of classes.

    Returns:
        None
    """
    cmap=plt.cm.Blues
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title("Confusion Matrix")
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    plt.grid(False)

    fmt = 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()

    return None
```

Classification Report

`sklearn.metrics.classification_report`

Build a text report showing the main classification metrics

Params:

- `y_true` : 1d array-like, or label indicator array / sparse matrix
- `y_pred` : 1d array-like, or label indicator array / sparse matrix

Output:

```
precision    recall  f1-score   support

           1       1.00      0.67      0.80         3
           2       0.00      0.00      0.00         0
           3       0.00      0.00      0.00         0

   micro avg       1.00      0.67      0.80         3
   macro avg       0.33      0.22      0.27         3
weighted avg       1.00      0.67      0.80         3
```

In [0]:
```
#@title ## Confusion matrix
```

Non-linear Model

Now let's see how the MLP performs on the data. Note that the only difference is the addition of the non-linear activation function (we use $ReLU$ which is just $max(0, z)$).

In [0]:
```
#@title ## Build non-linear model [MLP Model](https://en.wikipedia.org/wiki/Multilayer_perceptron)

def build_non_linear(n_classes, n_units, n_features):
    model = None

    return model
```

In [0]:
```
#@title ## Create non-linear model
```

In [0]:
```
#@title ## Fit non-linear model.
```

In [0]:
```
#@title ## Evaluating
```

In [0]:
```
#@title ## Plotting
```

In [0]:
```
#@title ## Visualize the decision boundary
```

In [0]:
```
#@title ## Confusion matrix
```

Overfitting

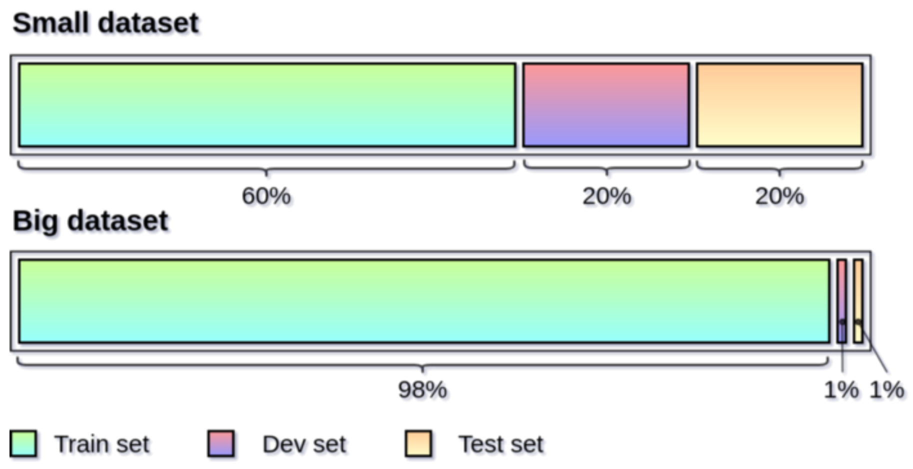
Though neural networks are great at capturing non-linear relationships they are highly susceptible to overfitting to the training data and failing to generalize on test data. Just take a look at the example below where we generate completely random data and are able to fit a model with $2 * N * C + D$ (https://arxiv.org/abs/1611.03530) hidden units. The training performance is great but the overfitting leads to very poor test performance. We'll be covering strategies to tackle overfitting in future lessons.

Let me define overfitting more formally. Overfitting refers to a model that models the "training data" too well. Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data.

How do you know your NN is overfitting?

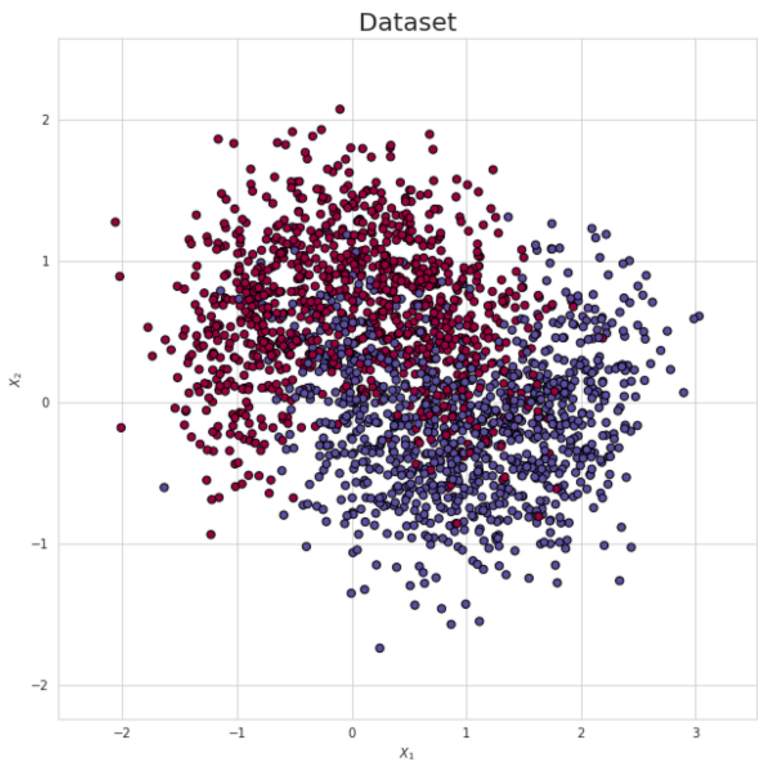
In practice, detecting that our model is overfitting is difficult. It's not uncommon that our trained model is already in production and then we start to realize that something is wrong. In fact, it is only by confronting new data that you can make sure that everything is working properly. However, during the training we should try to reproduce the real conditions as much as possible. For this reason, it is good practice to divide our dataset into three parts - training set, dev set (also known as cross-validation or hold-out) and test set. Our model learns by seeing only the first of these parts. Hold-out is used to track our progress and draw conclusions to optimise the model. While, we use a test set at the end of the training process to evaluate the performance of our model. Using completely new data allows us to get an unbiased opinion on how well our algorithm works.

It is very important to make sure that your cross-validation and test set come from the same distribution as well as that they accurately reflect data that we expect to receive in the future. Only then we can be sure that the decisions we make during the learning process bring us closer to a better solution. I know what you are thinking about… "How should I divide my dataset?" Until recently, one of the most frequently recommended splits was 60/20/20, but in the era of big data, when our dataset can count millions of entries, those fixed proportions are no longer appropriate. In short, everything depends on the size of the dataset we work with. If we have millions of entries at our disposal, perhaps it would be better idea to divide them in 98/1/1 ratio. Our dev and test sets should be simply large enough to give us high confidence in the performance of our model.

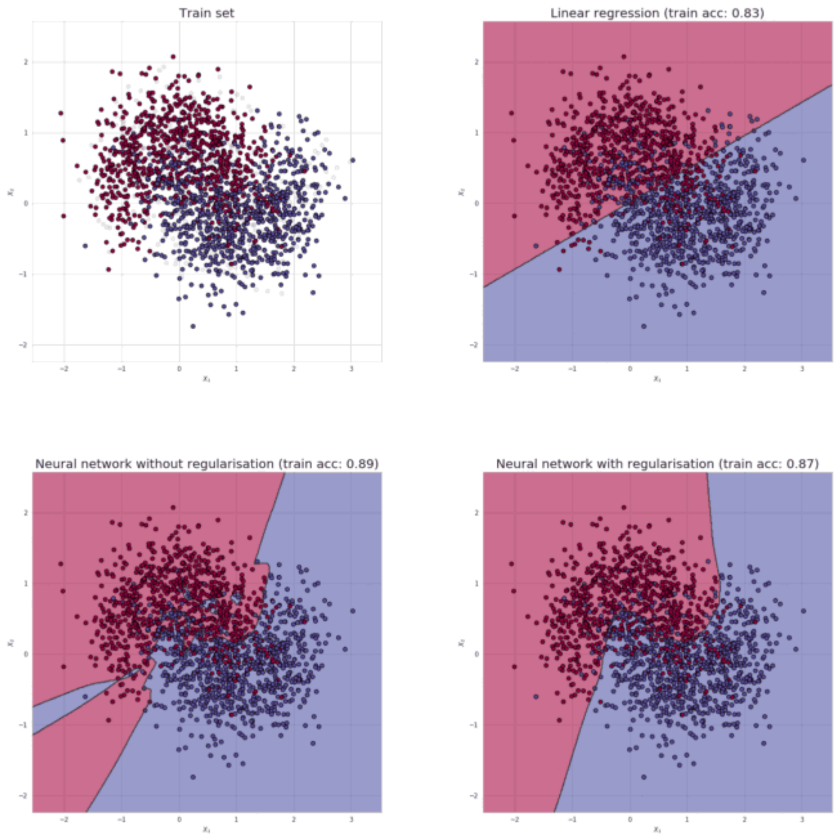


Bias and Variance

To give us a better understanding of this some how complex issue, we will use a simple example, that hopefully allow us to develop a valuable intuition. Our dataset consisting of two classes of points, located in a two-dimensional space.



The first model in the top right corner is very simple and therefore has a high bias, i.e. it is not able to find all significant links between features and result. This is understandable - our dataset has a lot of noise in it, and therefore simple linear regression, is not able to deal with it effectively. Neural networks performed much better, but the first one (shown in the lower left corner) fitted into the data too closely, which made it work significantly worse on the hold-out set. This means that it has a high variance - it fits into the noise and not into the intended output. This undesirable effect was mitigated, in the last model, by the use of regularisation.



Ways to prevent overfitting

L1 and L2 Regularizations

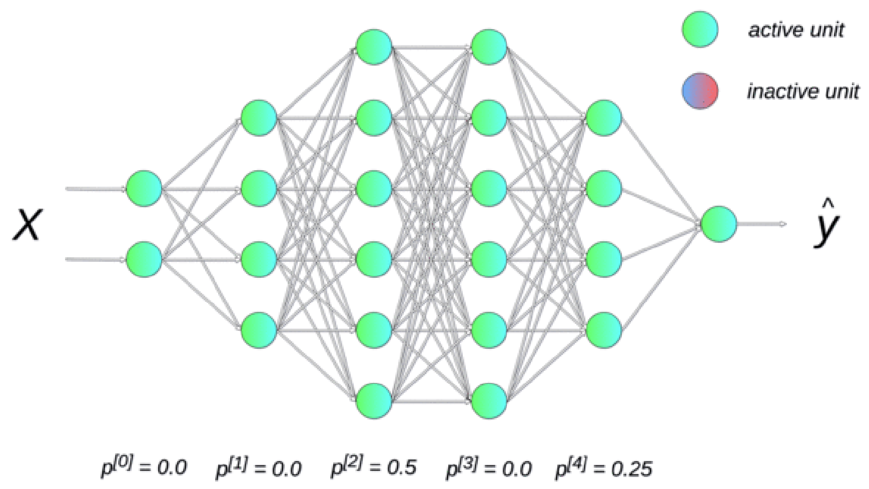
One of the first methods we should try when we need to reduce overfitting is regularisation. It involves adding an extra element to the loss function, which punishes our model for being too complex or, in simple words, for using too high values in the weight matrix. This way we try to limit its flexibility, but also encourage it to build solutions based on multiple features. Two popular versions of this method are L1 - Least Absolute Deviations (LAD) and L2 - Least Square Errors (LS). Equations describing these regularisations are given below. In most cases the use of L1 is preferable, because it reduces the weight values of less important features to zero, very often eliminating them completely from the calculations. In a way, it is a built-in mechanism for automatic featur selection. Moreover, L2 does not perform very well on datasets with a large number of outliers. The use of value squares results in the model minimizing the impact of outliers at the expense of more popular examples.

$$J_{L1}(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \lambda\|w\|_1 \quad \|w\|_1 = \sum_{j=1}^{n_x} |w_j|$$
$$J_{L2}(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \lambda\|w\|_2 \quad \|w\|_2 = \sum_{j=1}^{n_x} w_j^2$$

Increasing the λ value also increases the regularisation effect. Models with a very low λ coefficient value are very “turbulent”.

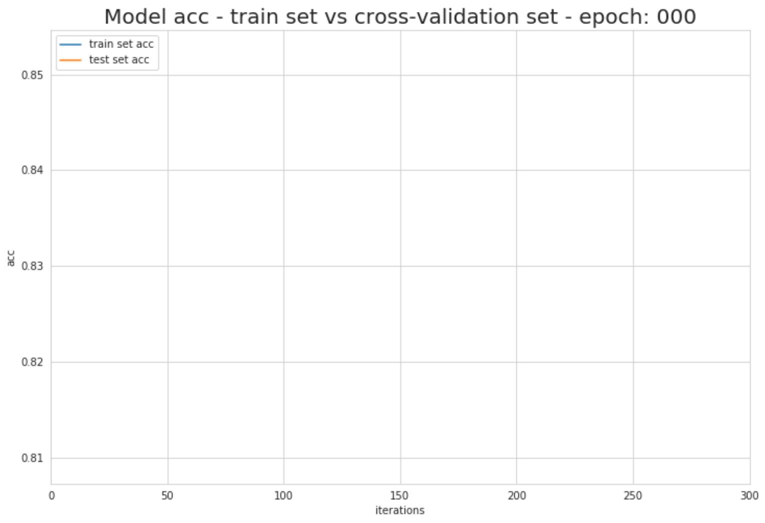
Dropout

Another very popular method of regularization of neural networks is dropout. This idea is actually very simple - every unit of our neural network (except those belonging to the output layer) is given the probability p of being temporarily ignored in calculations. Hyper parameter p is called dropout rate and very often its default value is set to 0.5. Then, in each iteration, we randomly select the neurons that we drop according to the assigned probability. As a result, each time we work with a smaller neural network. The visualization below shows an example of a neural network subjected to a dropout. We can see how in each iteration random neurons from second and fourth layer are deactivated.



Early Stopping

The graph below shows the change in accuracy values calculated on the test and cross-validation sets during subsequent iterations of learning process. We see right away that the model we get at the end is not the best we could have possibly create. To be honest, it is much worse than what we have had after 150 epochs. Why not interrupt the learning process before the model starts overfitting? This observation inspired one of the popular overfitting reduction method, namely early stopping.


Model acc - train set vs cross-validation set - epoch: 000
train set acc
test set acc
acc
0.85
0.84
0.83
0.82
0.81
0
50
100
150
200
250
300
iterations

In [0]:
```
#@title ## Generate random x, y

ov_num_samples = 40 #@param {type:"integer"}
ov_dimensions = 2 #@param {type:"integer"}
ov_num_classes = 3 #@param {type:"integer"}

ov_x = np.random.randn(ov_num_samples * ov_num_classes, ov_dimensions)
ov_y = np.array([[i] * ov_num_samples for i in range(ov_num_classes)])
ov_y = ov_y.flatten()


print()
print('x: %s' % str(ov_x.shape))
print('y: %s' % str(ov_y.shape))
```

```
x: (120, 2)
y: (120,)
```

In [0]:
```
#@title ## Preprocessing
```

In [0]:
```
#@title ## Create non-linear model
```

In [0]:
```
#@title ## Fit overfie model.
```

In [0]:
```
#@title ## Evaluating
```

In [0]:
```
#@title ## Plotting
```

In [0]:
```
#@title ## Visualize the decision boundary
```

In [0]:
```
#@title ## Confusion matrix
```

Regularization

In [0]:
```
#@title ## Build non-linear model [MLP Model](https://en.wikipedia.org/wiki/Multilayer_perceptron)

def build_non_linear_with_regularization(n_classes, n_units, n_features):
    model = None

    return model
```

In [0]:
```
#@title ## Create non-linear model
```

In [0]:
```
#@title ## Fit non-linear model with regularization.
```

In [0]:
```
#@title ## Evaluating
```

In [0]:
```
#@title ## Plotting
```

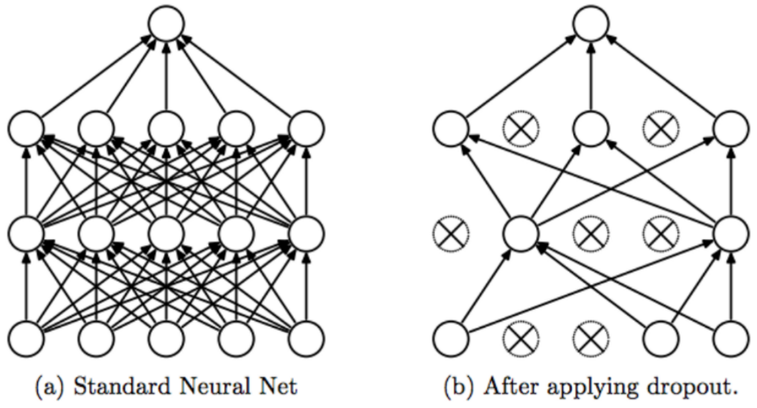
In [0]:
```
#@title ## Visualize the decision boundary
```

In [0]:
```
#@title ## Confusion matrix
```

Dropout

A great technique to overcome overfitting is to increase the size of your data but this isn't always an option. Fortuntely, there are methods like regularization and dropout that can help create a more robust model. We've already seen regularization and we can easily add it in our optimizer to use it in PyTorch.

Dropout is a technique (used only during training) that allows us to zero the outputs of neurons. We do this for p% of the total neurons in each layer and it changes every batch. Dropout prevents units from co-adapting too much to the data and acts as a sampling strategy since we drop a different set of neurons each time.



```
In [0]: #@title ## Build non-linear model [MLP Model](https://en.wikipedia.org/wiki/Multilayer_perceptron)

        def build_non_linear_with_dropout(n_classes, n_units, n_features):
            model = None

            return model
```

```
In [0]: #@title ## Create non-linear model
```

```
In [0]: #@title ## Fit non-linear model with dropout.
```

In [0]:
```
#@title ## Evaluating
```

In [0]:
```
#@title ## Plotting
```

In [0]:
```
#@title ## Visualize the decision boundary
```

In [0]:
```
#@title ## Confusion matrix
```

Visualization

In [0]:
```
#@title ## Build non-linear model [MLP Model](https://en.wikipedia.org/wiki/Multilayer_perceptron)

def build_non_linear_v2(n_classes, n_units, n_features):
    model = None

    return model
```

In [0]:
```
#@title ## Create non-linear model
```

In [0]:
```
#@title # Summary of the model
```

In [0]:
```
#@title ## Model Arch
```

In [0]:
```
#@title ## Tensorboard callbacks
```

In [0]:
```
#@title ## Fit non-linear model.
```

In [0]:
```
#@title ## Plotting
```

In [0]:
```
#@title ## Visualize the decision boundary
```

In [0]:
```
#@title ## Confusion matrix
```

Tensorboard

In [0]:
```
# %load_ext tensorboard
```

In [0]:
```
# %tensorboard --logdir logs/scalars
```

In [0]:
```
```