

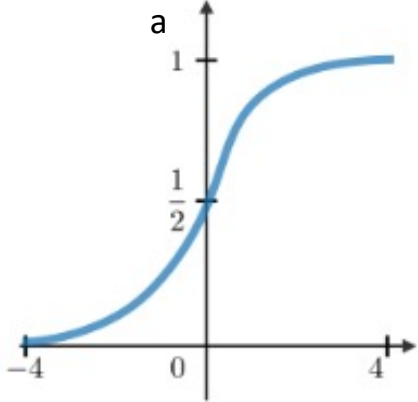
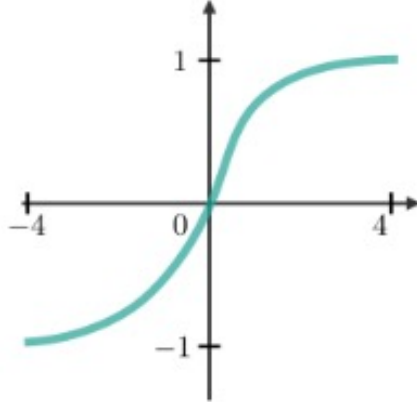
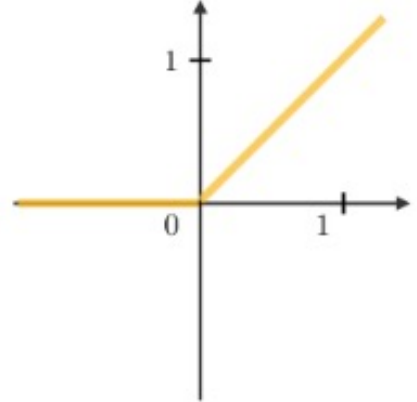
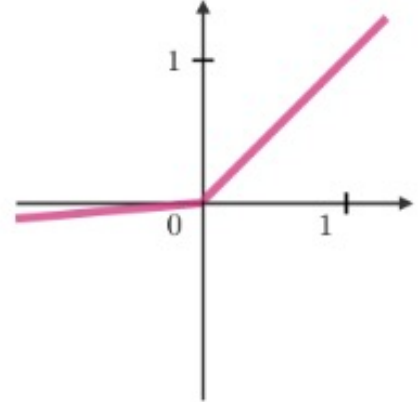
Build a Neural Network

Qingrun Zhang

Table of content

- Activation function
- Loss functions
- Parameters and Hyperparameters

Common Activation Functions

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

$$\begin{aligned}
 g'(z) &= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right) \\
 &= g(z) - (1 - g(z)) \\
 &= a(1 - a)
 \end{aligned}$$

When z is big or small, $g'(z) = 0$
 $z=0$. $g(z)=0.5$, $g'(z)=0.25$

$$\begin{aligned}
 g'(z) &= 1 - (\tanh(z))^2 \\
 &= 1 - a^2
 \end{aligned}$$

When z is big or small, $g'(z) = 0$
 $z=10$, $\tanh(10) \approx 1$
 $z=-10$, $\tanh(10) \approx -1$
 $z=0$, $\tanh(0) = 0$

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \\ \text{undefined} & \text{if } z = 0 \end{cases} \quad g'(z) = \begin{cases} \epsilon & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$$

ReLU Pros and Cons

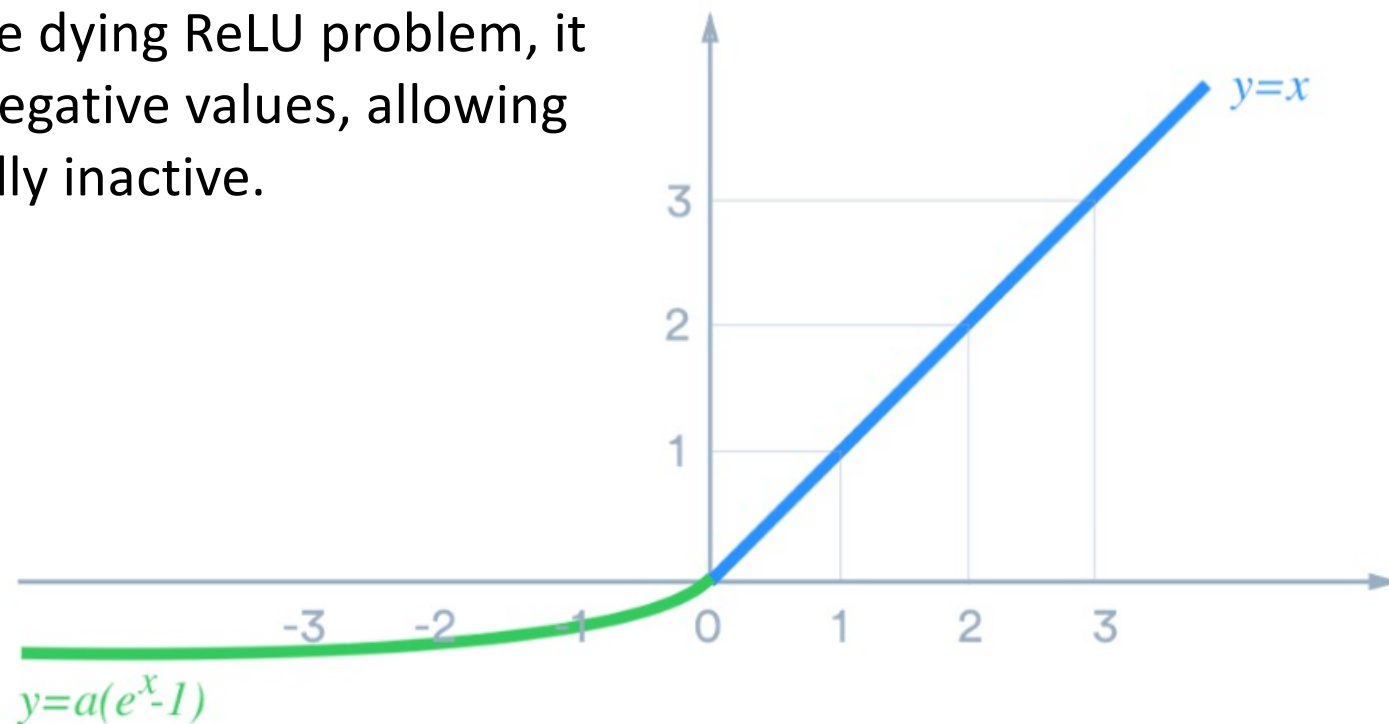
- It's cheap to compute. The model can take less time to train or run.
- It converges faster. Linearity means that the slope doesn't "saturate," It doesn't have the vanishing gradient problem suffered by activation functions like sigmoid or tanh.
- It's sparsely activated. Since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all. A sparse network is faster than a dense network
- Dying ReLU problem: A ReLU neuron is "dead" if it's stuck in the negative side and always outputs 0.
- The dying problem is likely to occur when learning rate is too high or there is a large negative bias. Lower learning rate often mitigates the problem

Leaky ReLU & Parametric ReLU (PReLU)

- Leaky ReLU may have $y = \overset{\varepsilon}{\downarrow} 0.01x$ when $x < 0$.
- Parametric ReLU (PReLU), a type of Leaky ReLU, it makes 0.01 a parameter for the neural network to figure out itself: $y = ax$ when $x < 0$.
- Leaky ReLU fixes the “dying ReLU” Problem
- It speeds up training.
- Leaky ReLU isn’t always superior to plain ReLU and should be considered only as an alternative.

Exponential Linear (ELU, SELU)

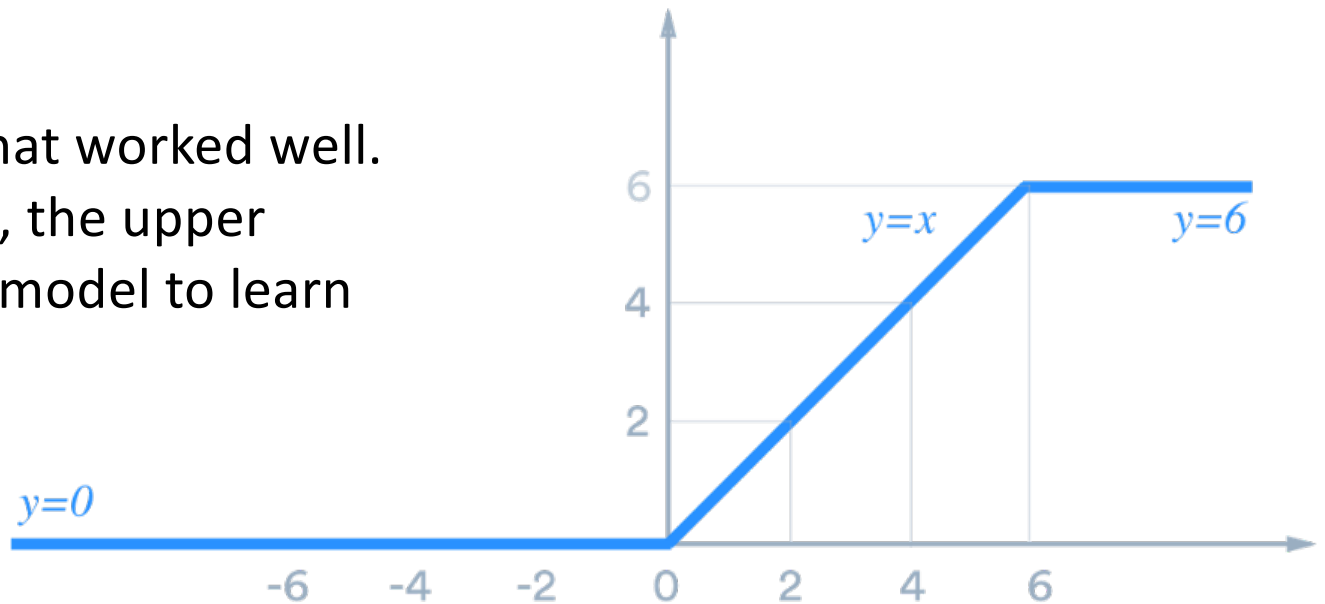
ELU doesn't have the dying ReLU problem, it saturates for large negative values, allowing them to be essentially inactive.



Clevert et al, 2015 Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)

ReLU-6

6 is an arbitrary choice that worked well. According to the authors, the upper bound encouraged their model to learn sparse features earlier.



https://www.tensorflow.org/api_docs/python/tf/nn/relu6

https://keras.io/api/layers/activation_layers/

Alex Krizhevsky, 2010 Convolutional Deep Belief Networks on CIFAR-10

Softmax activation function

- Softmax is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.
- $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}$ for $i = 1, \dots, k$ and $z = (z_1, \dots, z_k) \in R^k$
- $\sum_{i=1}^k \sigma(z)_i = 1$

Why non-linear activation function?

- When the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator.
- When multiple layers use the identity (linear) activation function, the combination of linear activation function is still linear.
- $a^{[1]} = z^{[1]} = w^{[1]}x + b^{[1]}$
- $a^{[1]} = z^{[2]} = w^{[2]}a^{[1]} + b^{[2]} = w^{[2]T}(w^{[1]}x + b^{[1]}) + b^{[2]}$
- $= w^{[2]}w^{[1]}x + (w^{[2]}b^{[1]} + b^{[2]})$

Loss Functions

- Regression Loss Functions
 - Mean Squared Error Loss
 - Mean Squared Logarithmic Error Loss
 - Mean Absolute Error Loss
- Binary Classification Loss Functions
 - Binary Cross-Entropy
 - Hinge Loss
 - Squared Hinge Loss
- Multi-Class Classification Loss Functions
 - Multi-Class Cross-Entropy Loss
 - Sparse Multiclass Cross-Entropy Loss
 - Kullback Leibler Divergence Loss

Other reading: <https://yuan-du.com/post/2020-12-13-loss-functions/decision-theory/#:~:text=Exponential%20loss,of%20additive%20modeling%20is%20computational.>

Cross-entropy

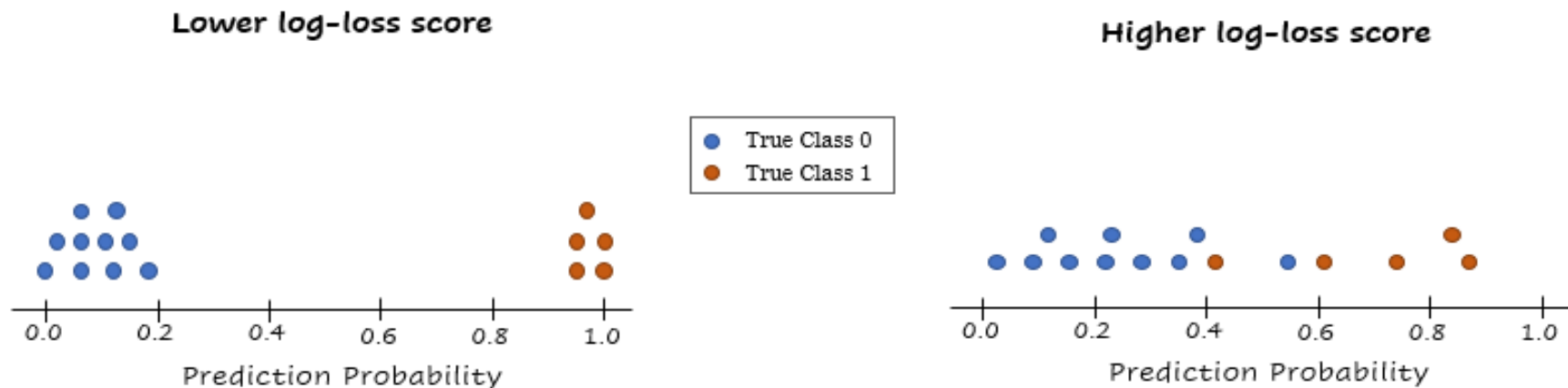
- Binary cross-entropy: for binary classification problem
- Categorical cross-entropy: binary and multiclass problem, the label needs to be encoded as categorical, one-hot encoding representation (for 3 classes: $[0, 1, 0]$, $[1, 0, 0]$...)
- Sparse cross-entropy: binary and multiclass problem (the label is an integer — 0 or 1 or ... n , depends on the number of labels)

Log-Loss

$$\text{Logloss} = -\frac{1}{N} \sum_{i=1}^N [y_i \ln p_i + (1 - y_i) \ln(1 - p_i)]$$

log loss = negative log-likelihood, under a Bernoulli probability distribution.

The Log-Loss is the Binary cross-entropy up to a factor $1 / \log(2)$. This loss function is convex and grows linearly for negative values (less sensitive to outliers). The common algorithm which uses the Log-loss is the ***logistic regression***.



Hinge Loss= $\max(0, 1 - y\hat{y})$

- Use with binary classification where the target values are in the set $\{-1, 1\}$.
- The Hinge loss is used for “maximum-margin” classification, most notably for support vector machine (SVM). It’s equivalent to minimize the loss function.
- The goal is to make different penalties at the point that are not correctly predicted or too closed of the hyperplane.
- The hinge loss function encourages examples to have the correct sign, assigning more error when there is a difference in the sign between the actual and predicted class values.
- The output layer of the network should use hyperbolic tangent activation function capable of outputting a single value in the range $[-1, 1]$.
- With the optimization problem is loss + penalty
- $\min_{w, b} \sum_{i=1}^m [1 - y_i \hat{y}_i]_+ + \frac{\lambda}{2} \|w\|^2$

Kullback Leibler Divergence Loss

- The KL divergence is the score of two different probability distribution functions. The KL difference between a PDF of $q(x)$ and a PDF of $p(x)$ is noted $KL(Q || P)$ where $||$ means *divergence* (it is not symmetric $KL(P || Q) \neq KL(Q || P)$). $kl_{loss} = y(\log \hat{y})$
- KL can be used to measure how one probability distribution differs from a baseline distribution. A KL divergence loss of 0 suggests the distributions are identical
- The KL divergence loss is more commonly used to approximate a more complex function, e.g. an autoencoder used for learning a dense feature representation that must reconstruct the original input.

```
model.compile(optimizer='sgd', loss=tf.keras.losses.KLDivergence())
```

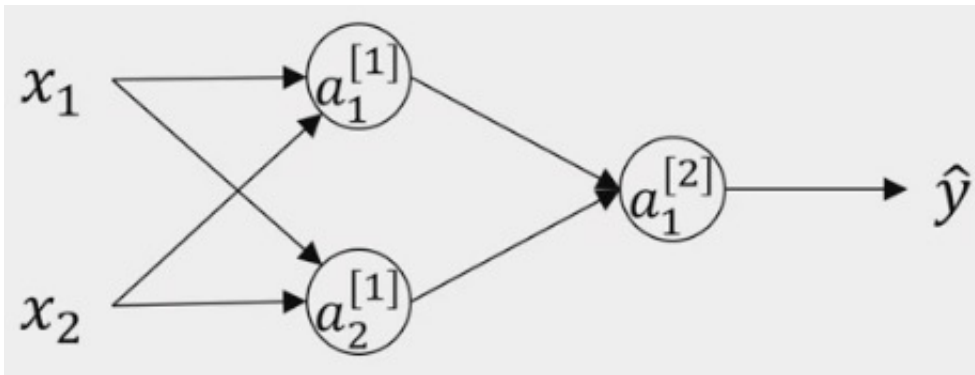
Parameters and Hyperparameters

- Parameters: $W^{[l]}, b^{[l]}$ Learning rate α
- Hyperparameters:
 - # of iterations
 - # hidden layers l
 - # hidden units
 - Choice of Activation Function
 - Momentum (Exponentially weighted average)
 - Minibatch size
 - Regularization terms

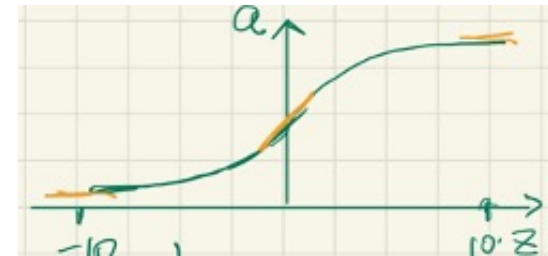
Weight initialization

- It is ok to initialize b to zero, but w should be randomly initialized to prevent the symmetric issue.

$$w = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \rightarrow a_1^{[1]} = a_2^{[1]} \quad dz_1^{[1]} = dz_2^{[1]}$$



$$dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix} \quad w^{[1]} = w^{[1]} - \alpha dw$$



$$w^{[1]} = np.random.rand(n^{[2]}, n^{[1]}) * \underline{0.01}$$

$$b^{[1]} = np.zeros(n^{[1]}, 1) * 0.01$$

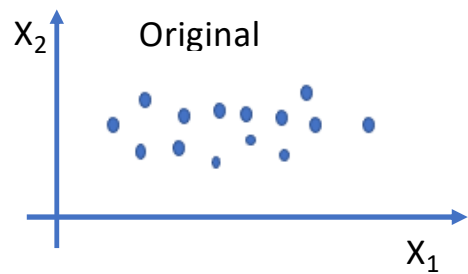
$$z^{[1]} = w^{[1]}x + b^{[1]}$$

Big z makes learning rate small

$$a^{[1]} = g^{[1]}(z^{[1]})$$

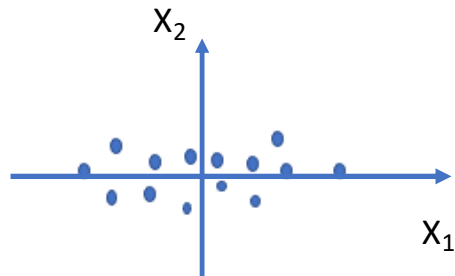
Layer weight initializers with Keras

- <https://keras.io/api/layers/initializers/>
- W2.2_Weight Initializers.ipynb

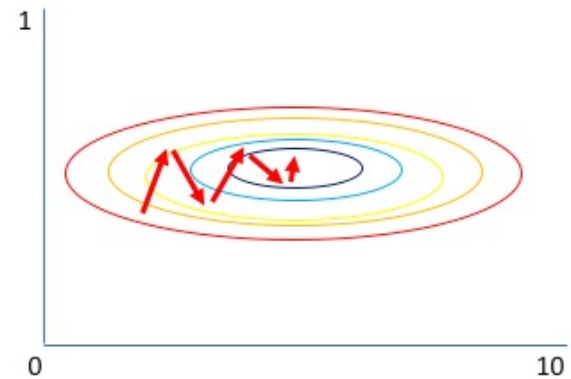
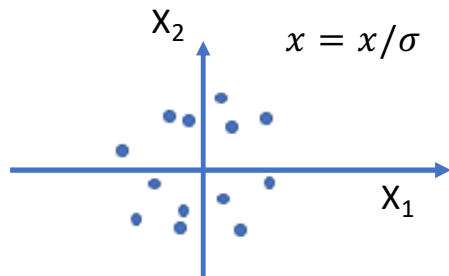


normalize your inputs

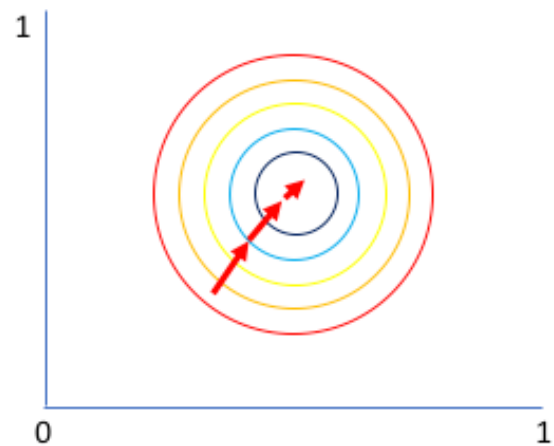
Subtract mean $\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}, x = x - \mu$



Normalize variances $\sigma^2 = \frac{1}{m} \sum_{i=1}^m x^{(i)} ** 2$



Gradient of larger parameter dominates the update



Both parameters can be updated in equal proportions

Tensorflow Keras API

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(50, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam',
              loss='BinaryCrossentropy',
              metrics=['accuracy'])

print('Training model...\n')
model.fit(x_train.T, y_train.T, epochs=20, batch_size=32)
```

Build a neural network:

- This tutorial, we will use tensorflow provided notebook to learn how to Training a neural network on MNIST with Keras
- https://colab.research.google.com/github/tensorflow/datasets/blob/master/docs/keras_example.ipynb
- Hyperparameter tuner
- https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/keras_tuner.ipynb

Overview of the Project 1-part a

- **Problem Statement:** You are given a dataset ("Iris.csv") containing:
 - Four features and one target column
 - Convert the target column into 1=Setosa and 0!=Setosa
- You will build a simple algorithm that can correctly classify species Setosa or not-Setosa.
- Please remember the following steps:
 - Inspect data first, such as histogram of each feature etc.
 - Split data into training and testing datasets
 - Standardize the dataset
 - Build a Feed Forward Neural Network, using ReLU and other activation functions. Let's see whether your neural network accuracy can beat logistic regression

Project 1- part b

- **Problem Statement:** using TensorFlow dataset “beans”
- <https://www.tensorflow.org/datasets/catalog/beans>
- Build a Feed Forward Neural Network, to classify the three classes of beans. 2 disease classes and the healthy class.
- Use Hyperparameter tuner to tune the hyperparameters.