Lunak

A Feed-forward Artificial Neural Network Implemented in Python

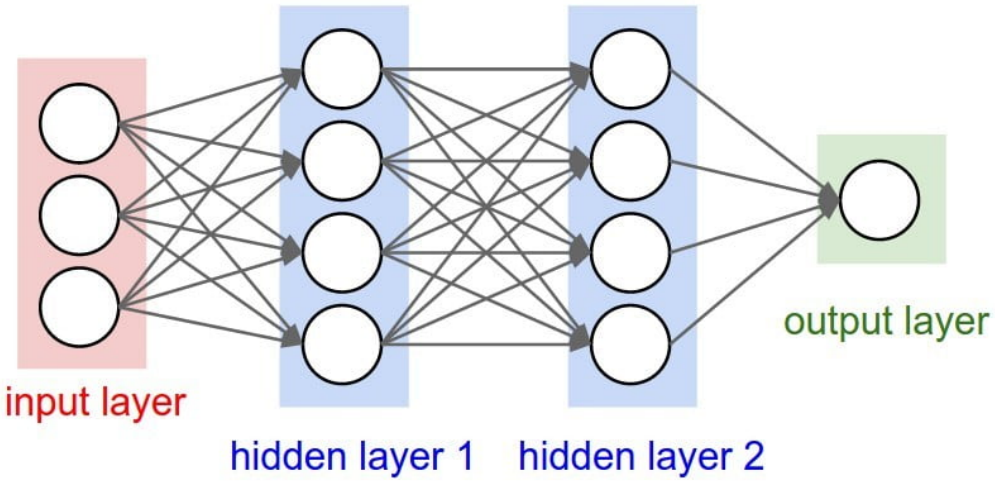
Felix Limanta—13515065
Holy Lovenia—13515113
Agus Gunawan—13515143

November 15, 2018

1 Introduction

An artificial neural network (ANN) is made of interconnected layers of nodes, emulating the structures and functions of neurons in human brain.

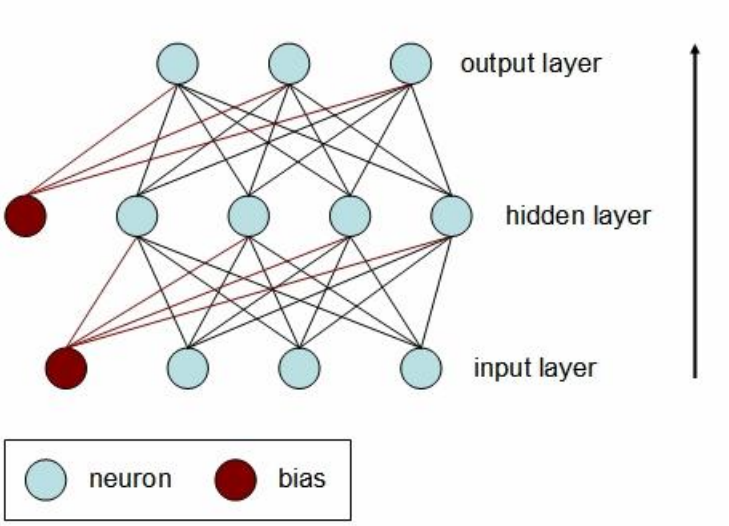
Normally, connections between the nodes (neurons) are called edges. These edges are associated with weights, which adjusts themselves during the learning process.



Artificial neural network

Nodes are typically grouped into specific layers. Different layers perform different transformations on their inputs. The first layer is regarded as input layer; the last is output layer. The output layer represents the final outputs of the network their corresponding predicted classes. Between the two layers, hidden layers may be present to process and transform the inputs by applying an activation function, then produce results according to the needs of output layer.

The disconnected nodes in the network are called as bias nodes, which are useful to shift the activation function into a desired direction. Below is an example of network with the presence of bias nodes.



Network with bias

2 Implementing an Artificial Neural Network

We introduce Lunak, a simple feed-forward neural network implementation in Python. The entire source code for Lunak is contained within this report and shall be shown in the following subsections.

```
In [1]: from random import random, randint
        from sklearn.metrics import confusion_matrix, mean_squared_error
        from sklearn.model_selection import train_test_split
        from tqdm import tqdm, tnrange

        import math
        import numpy as np
        import pandas as pd
```

2.1 Implementing the layers

We implement the `LunakDense` class to represent our hidden layer and output layer. The parameters for this class are as follows.

1. `units`: `int`, the number of nodes in the layer
2. `activation`: `'sigmoid'`, activation function
3. `input_dim`: `int`, dimension of the input (e.g. 2D, 3D, ...)
4. `init`: `'uniform'`, `'random'`, type of distribution for the initial weight matrix
5. `use_bias`: `boolean`, whether there will be bias node present or not (default=`False`)
6. `seed`: `int`, the number of random seed (default=`None`)

The layer transforms input $p_j(t)$ to neuron j from the outputs $o_i(t)$ of predecessor neurons and bias w_{0j} according to the following propagation function.

$$p_j(t) = \sum_i o_i(t) w_{ij} + w_{0j}$$

```
In [2]: class LunakDense:
            def __init__(self, units, activation,
                         input_dim, init, use_bias=False, seed=None):
                self.units = units
                self.input_dim = input_dim

                if activation == 'sigmoid':
                    self.activation_function = self.sigmoid
                else:
                    print('Activation function not supported')

                np.random.seed(seed)

                if init == 'uniform':
                    self.weight_matrix = np.random.uniform(-0.05, 0.05,
                                                           size=(self.units, input_dim))
                elif init == 'random':
                    self.weight_matrix = np.random.random(size=(self.units, input_dim))
                else:
                    print('Init function not supported')

                self.delta_weight_matrix_before = np.zeros((self.units, input_dim))
                self.delta_weight_matrix = np.zeros((self.units, input_dim))

                self.use_bias = use_bias
                if self.use_bias:
                    bias = np.zeros((units, 1))
                    self.weight_matrix = np.hstack((self.weight_matrix, bias))
                    self.delta_weight_matrix_before = np.hstack(
                        (self.delta_weight_matrix_before, np.zeros((units, 1))))
                    self.delta_weight_matrix = np.hstack(
                        (self.delta_weight_matrix, np.zeros((units, 1))))

            def init_delta_weight_zero(self):
                for idx_unit, unit in enumerate(self.delta_weight_matrix):
                    for idx_source, source in enumerate(unit):
                        self.delta_weight_matrix[idx_unit] = 0
                        self.delta_weight_matrix_before[idx_unit] = 0

            def calculate_sigma(self, input_list):
                if self.use_bias:
                    input_list = np.append(input_list, 1)

                result_list = np.array([])
                for weight_neuron in self.weight_matrix:
                    result_list = np.append(result_list, np.dot(weight_neuron, input_list))
                return np.array(result_list)
```

```
def calculate_output(self, input_list):
    output_list = np.array([])
    for sigma_neuron in self.calculate_sigma(input_list):
        output_list = np.append(output_list,
                                self.activation_function(sigma_neuron))
    self.output_list = output_list
    return output_list.copy()

def calculate_local_gradient_output_layer(self, target_list):
    """
    Use this if the layer is output layer
    """
    result_list = np.array([])
    for index, output in enumerate(self.output_list):
        local_gradient = output * (1 - output) * (target_list[index] - output)
        result_list = np.append(result_list, local_gradient)
    self.local_gradient = result_list
    return result_list.copy()

def calculate_local_gradient_hidden_layer(self,
                                          local_gradient_output_list,
                                          output_layer_weight_matrix):
    """
    Use this if the layer is hidden layer
    """
    result_list = np.array([])
    for index, output in enumerate(self.output_list):
        sigma_local_gradient_output = 0
        for unit_number, local_gradient in enumerate(local_gradient_output_list):
            sigma_local_gradient_output += \
                output_layer_weight_matrix[unit_number][index] * local_gradient
        error_hidden = output * (1 - output) * sigma_local_gradient_output
        result_list = np.append(result_list, error_hidden)
    self.local_gradient = result_list
    return result_list.copy()

def update_delta_weight(self, lr, input_list, momentum=None):
    """
    Function to update delta weight
    """
    if self.use_bias:
        input_list = np.append(input_list, 1)
    if momentum == None:
        for j, unit in enumerate(self.weight_matrix): #j
            for i, source in enumerate(unit): #i
                delta_weight = self.delta_weight_matrix[j][i] \
                    + lr * self.local_gradient[j] * input_list[i]
                self.delta_weight_matrix[j][i] = delta_weight.copy()
```

```
            else:
                for j, unit in enumerate(self.weight_matrix): #j
                    for i, source in enumerate(unit): #i
                        delta_weight = self.delta_weight_matrix[j][i] \
                            + lr * self.local_gradient[j] * input_list[i] \
                            + momentum * self.delta_weight_matrix_before[j][i]

                        # Update Delta Weight
                        self.delta_weight_matrix_before[j][i] = delta_weight.copy()

                # Copy Last Update of Weight Matrix Before (Equal to Last Weight Matrix)
                for j, unit in enumerate(self.delta_weight_matrix_before):
                    for i, source in enumerate(unit):
                        self.delta_weight_matrix[j][i] = \
                            self.delta_weight_matrix_before[j][i].copy()

        def update_weight(self):
            """
            Function to update weight
            """
            for j, unit in enumerate(self.delta_weight_matrix_before):
                for i, source in enumerate(unit):
                    self.weight_matrix[j][i] += self.delta_weight_matrix[j][i]

        def sigmoid(self, x):
            return 1 / (1 + math.exp(-x))
```

2.2 Implementing the model

The ANN model is implemented using stochastic gradient descent (SGD) as the learning rule. SGD is known as a strategy for searching through a large or infinite hypothesis space.

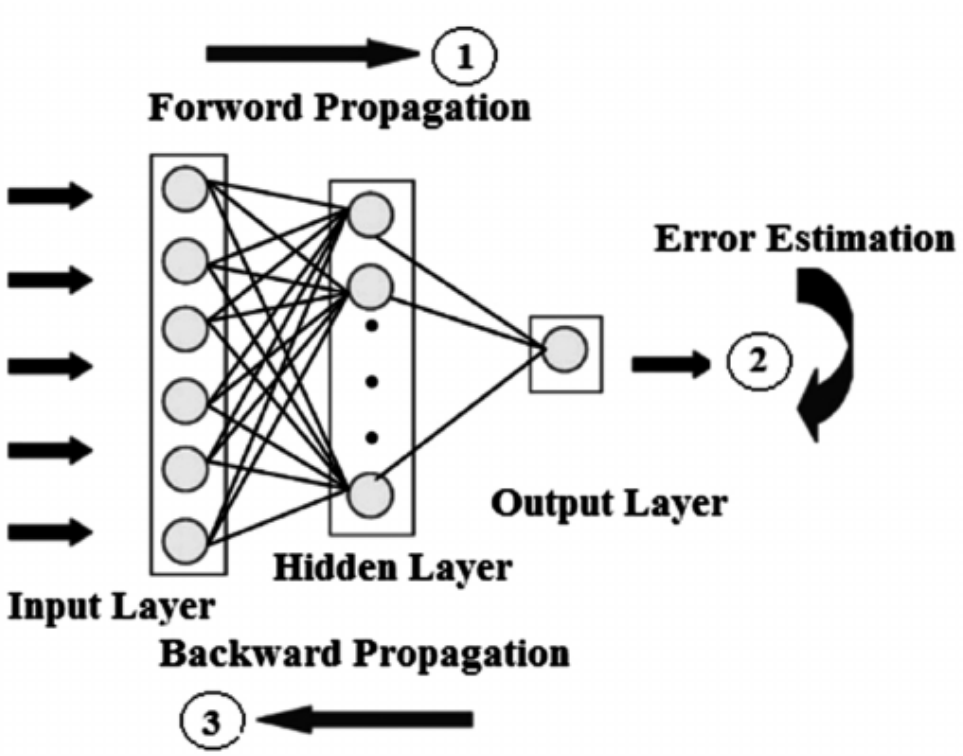
Training an ANN typically entails two steps: feed-forward and backpropagation.

- **Feed-forward** predicts output classes from the given inputs using the weights the network currently posseses.
- **Backpropagation** calculates the error in each neuron (based on the predicted class and its ground truth) and updates the weights accordingly.

The algorithm used for the feed-forward and backpropagation is based on the algorithm in the book Machine Learning (Mitchell, 1997, p. 98).

We implement the `LunakArtificialNeuralNetwork` class to represent our neural network model. The class possesses two notable methods: - `fit(·)`: Trains a model with a given dataset, and - `predict(·)`: Receives input data and produces predictions according to the trained model.

The following parameters are used for `fit(·)`. 1. `X`: `data`, training data 2. `y`: `data`, labels for training data 3. `epochs`: `int`, the number of epochs that will be run 4. `lr`: `float`, learning rate 5. `momentum`: `float`, momentum (used to prevent local minima) 6. `batch_size`: `int`, incremental when 1 (default=`len(X)`) 7. `val_data`: `(X_val, y_val)`, for validation purposes (default=`None`, will use `val_size`=0.1) 8. `val_size`: `float`, used to split X and y to get validation data (default=0)



ANN steps

```
In [3]: class LunakArtificialNeuralNetwork:
            def __init__(self, loss='root_mean_squared', optimizer='sgd'):
                assert loss == 'root_mean_squared', 'loss function not supported'
                assert optimizer == 'sgd', 'optimizer not supported'
                self.layers = []

            def add(self, layer):
                self.layers.append(layer)

            def feed_forward(self, X_instance):
                # Calculate output with the first hidden layer
                output_list = self.layers[0].calculate_output(X_instance)
                # Calculate output with the second until the last layer
                for layer in self.layers[1:]:
                    next_output_list = layer.calculate_output(output_list)
                    output_list = next_output_list
                return output_list.copy()

            def backpropagation(self, y_instance):
                # Calculate local gradient for output layer
                next_local_gradient_list = \
                    self.layers[-1].calculate_local_gradient_output_layer([y_instance])
                next_layer_weight_matrix = self.layers[-1].weight_matrix
```

```
        # Calculate local gradient for hidden layer(s)
        for layer_idx, layer in enumerate(reversed(self.layers[0:-1])):
            next_local_gradient_list = \
                layer.calculate_local_gradient_hidden_layer(next_local_gradient_list,
                                                            next_layer_weight_matrix)
            next_layer_weight_matrix = layer.weight_matrix

    def calculate_delta_weight(self, X_instance, lr, momentum):
        # Update delta weight for first hidden layer
        self.layers[0].update_delta_weight(lr, X_instance, momentum)

        # Update delta weight for other layers
        for layer_idx, layer in enumerate(self.layers[1:]):
            layer.update_delta_weight(lr, self.layers[layer_idx].output_list,
                                      momentum)

    def fit(self, X, y, epochs, lr, momentum=None,
            batch_size=None, val_data=None, val_size=0):
        assert X.shape[1] == self.layers[0].input_dim, \
            'Input dimension must be same with the column'
        self.classes_ = np.unique(y)

        if batch_size == None:
            batch_size = len(X)

        if val_data is None:
            val_size = 0.1
            X, X_val, y, y_val = train_test_split(X, y, test_size=val_size)
        else:
            X_val = val_data[0]
            y_val = val_data[1]

        print('Train on {} samples, validate on {} samples'.format(len(X),
                                                                   len(X_val)))

        if val_data is not None and val_size != 0:
            print('Validation data will be used instead of val_size.')

        for epoch in range(epochs):
            delta = batch_size

            with tnrange(0, len(X), delta,
                         desc='Epoch {}'.format(epoch + 1)) as pbar:
                for start in pbar:
                    X_batch = X[start:start+delta]
                    y_batch = y[start:start+delta]
```

```
                    for idx, X_instance in enumerate(X_batch):
                        self.feed_forward(X_instance)
                        self.backpropagation(y_batch[idx][0])
                        self.calculate_delta_weight(X_instance, lr, momentum)

                    for layer in self.layers:
                        layer.update_weight()
                        layer.init_delta_weight_zero()

                    pred = self.predict(X)
                    pred_val = self.predict(X_val)

                    pred_proba = self.predict_proba(X)
                    pred_proba_val = self.predict_proba(X_val)

                    acc = self.calculate_accuracy(y, pred)
                    val_acc = self.calculate_accuracy(y_val, pred_val)
                    loss = mean_squared_error(y, pred_proba)
                    val_loss = mean_squared_error(y_val, pred_proba_val)

                    postfix = {
                        'loss': loss,
                        'acc': acc,
                        'val_loss': val_loss,
                        'val_acc': val_acc
                    }
                    pbar.set_postfix(postfix, refresh=True)

    def predict_proba(self, X):
        predictions = []
        for idx, X_instance in enumerate(X):
            X_pred = self.feed_forward(X_instance)
            predictions.append([np.mean(X_pred.copy())])
        return predictions

    def predict(self, X):
        predictions = []
        for idx, X_instance in enumerate(X):
            X_pred_proba = self.feed_forward(X_instance)
            X_pred = min(self.classes_,
                         key=lambda pred_class:abs(pred_class - np.mean(X_pred_proba)))
            predictions.append([X_pred])
        return predictions

    def calculate_accuracy(self, y_true, y_pred):
        if len(confusion_matrix(y_true, y_pred).ravel()) > 1:
            tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
        else:
```

```
            tp = confusion_matrix(y_true, y_pred).ravel()[0]
            fp = 0
            fn = 0
            tn = 0
        return (tp + tn) / (tp + tn + fp + fn)
```

3 Testing our implementation

To test whether Lunak works correctly, we train both our model and a Keras ANN model as baseline, then compare their results and performance.

```
In [4]: from scipy.io.arff import loadarff
        import pandas as pd
        from IPython.utils import io
```

Our dataset is the Play Tennis dataset, obtained from the Weka Data Sets.

```
In [5]: raw_data = loadarff('dataset/weather.arff')
        data = pd.DataFrame(raw_data[0])
        data.head()
```

Out[5]:

	outlook	temperature	humidity	windy	play
0	b'sunny'	85.0	85.0	b'FALSE'	b'no'
1	b'sunny'	80.0	90.0	b'TRUE'	b'no'
2	b'overcast'	83.0	86.0	b'FALSE'	b'yes'
3	b'rainy'	70.0	96.0	b'FALSE'	b'yes'
4	b'rainy'	68.0	80.0	b'FALSE'	b'yes'

3.1 Preprocessing

Before we train our model, we must first preprocess our data, as the data types of our chosen dataset cannot be fed to our ANN out-of-the-box.

```
In [6]: def convert_to_binary_vector(data):
            return pd.get_dummies(data)
```

First, we decode strings and boolean data in the dataset as UTF-8.

```
In [7]: for idx, column in enumerate(['outlook', 'windy', 'play']):
            data[column] = data[column].str.decode('utf-8')
```

Then, string categoricals in the data (i.e., `outlook` and `windy`) are converted to one-hot vectors.

```
In [8]: bv_outlook = convert_to_binary_vector(data['outlook'])
        bv_outlook.head()
```

Out[8]:

	overcast	rainy	sunny
0	0	0	1
1	0	0	1
2	1	0	0
3	0	1	0
4	0	1	0

```
In [9]: bv_windy = convert_to_binary_vector(data['windy'])
        bv_windy.head()
```

Out[9]:

	FALSE	TRUE
0	1	0
1	0	1
2	1	0
3	1	0
4	1	0

We drop the categorical `outlook` and `windy` data and concatenate our one-hot vectors to the dataset.

```
In [10]: preproc_data = data.drop('outlook', 1).drop('windy', 1)
         preprocessed_data = pd.concat([bv_outlook, bv_windy, preproc_data], axis=1)
```

Our target column, `play`, is processed as categorical data.

```
In [11]: preproc_data['play'] = preproc_data['play'].astype('category')
         preprocessed_data.head()
```

Out[11]:

	overcast	rainy	sunny	FALSE	TRUE	temperature	humidity	play
0	0	0	1	1	0	85.0	85.0	no
1	0	0	1	0	1	80.0	90.0	no
2	1	0	0	1	0	83.0	86.0	yes
3	0	1	0	1	0	70.0	96.0	yes
4	0	1	0	1	0	68.0	80.0	yes

The, the `play` column is converted to binary $0/1$ integers.

```
In [12]: y = pd.DataFrame({'play': preproc_data['play'].cat.codes})
         y.head()
```

Out[12]:

	play
0	0
1	0
2	1
3	1
4	1

For training, we delete our target column from the dataset.

```
In [13]: X = preprocessed_data.drop('play', 1)
```

We then cast all data remaining in the dataset as floats.

```
In [14]: for column in X.columns:
             X[column] = X[column].astype('float')
```

We normalize real number data (`temperature` and `humidity`) to $[0, 1]$.

```
In [15]: for column in ['temperature', 'humidity']:
             X[column] = (X[column] - min(X[column])) / (max(X[column]) - min(X[column]))

In [16]: X.head()
```

Out[16]:

	overcast	rainy	sunny	FALSE	TRUE	temperature	humidity
0	0.0	0.0	1.0	1.0	0.0	1.000000	0.645161
1	0.0	0.0	1.0	0.0	1.0	0.761905	0.806452
2	1.0	0.0	0.0	1.0	0.0	0.904762	0.677419
3	0.0	1.0	0.0	1.0	0.0	0.285714	1.000000
4	0.0	1.0	0.0	1.0	0.0	0.190476	0.483871

We also cast our target labels to float.

```
In [17]: y = y.astype('float')
```

Both our dataset and our target labels are converted to `numpy` arrays for faster processing.

```
In [18]: X = np.array(X)
         y = np.array(y)
```

The following is our preprocessed data and their labels. We shall feed this preprocessed data into our ANN implementations.

```
In [19]: X

Out[19]: array([[0.        , 0.        , 1.        , 1.        , 0.        ,
                 1.        , 0.64516129],
                [0.        , 0.        , 1.        , 0.        , 1.        ,
                 0.76190476, 0.80645161],
                [1.        , 0.        , 0.        , 1.        , 0.        ,
                 0.9047619 , 0.67741935],
                [0.        , 1.        , 0.        , 1.        , 0.        ,
                 0.28571429, 1.        ],
                [0.        , 1.        , 0.        , 1.        , 0.        ,
                 0.19047619, 0.48387097],
                [0.        , 1.        , 0.        , 0.        , 1.        ,
                 0.04761905, 0.16129032],
                [1.        , 0.        , 0.        , 0.        , 1.        ,
                 0.        , 0.        ],
                [0.        , 0.        , 1.        , 1.        , 0.        ,
                 0.38095238, 0.96774194],
                [0.        , 0.        , 1.        , 1.        , 0.        ,
                 0.23809524, 0.16129032],
                [0.        , 1.        , 0.        , 1.        , 0.        ,
                 0.52380952, 0.48387097],
                [0.        , 0.        , 1.        , 0.        , 1.        ,
                 0.52380952, 0.16129032],
                [1.        , 0.        , 0.        , 0.        , 1.        ,
```

```
                0.38095238, 0.80645161],
               [1.        , 0.        , 0.        , 1.        , 0.        ,
                0.80952381, 0.32258065],
               [0.        , 1.        , 0.        , 0.        , 1.        ,
                0.33333333, 0.83870968]])
```

```
In [20]: y
```

```
Out[20]: array([[0.],
                [0.],
                [1.],
                [1.],
                [1.],
                [0.],
                [1.],
                [0.],
                [1.],
                [1.],
                [1.],
                [1.],
                [1.],
                [0.]])
```

3.1.1 Hold-out split

For training and validation, we employ a 90-10 holdout validation scheme, i.e., 90% data is used for training, while the remaining 10% is used for validation.

```
In [21]: from sklearn.model_selection import train_test_split
```

```
In [22]: X, X_val, y, y_val = train_test_split(X, y, test_size=0.1)
```

3.2 Building and training our models

We shall now build, train, and test Lunak and Keras.

3.2.1 Lunak

We shall first initialize our Lunak ANN model.

```
In [23]: lunak_ann = LunakArtificialNeuralNetwork()
```

On our model, we add a hidden layer with two units and an output layer with one unit.

```
In [24]: lunak_ann.add(LunakDense(2, 'sigmoid', 7, 'uniform', use_bias=True, seed=5))
         lunak_ann.add(LunakDense(1, 'sigmoid', 2, 'uniform', use_bias=True, seed=5))
```

We then train our Lunak model using the preprocessed data and labels from the previous section across 10 epochs.

As the dataset is quite small, training should not take too much time. We use `jupyter`'s `%timeit` function to measure the average training time.

```
In [25]: %%timeit
         with io.capture_output() as captured:
             lunak_ann.fit(X, y, epochs=10, momentum=0.001,
                           lr=0.01, batch_size=4, val_data=(X_val, y_val))
```

```
533 ms ± 62.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Predicting the original dataset using the trained models yield the following predicted probabilities.

```
In [26]: pred_proba = lunak_ann.predict_proba(X)
         pred_proba
```

```
Out[26]: [[0.5986759362416141],
          [0.5977620134260081],
          [0.5978382872859325],
          [0.5976941219147527],
          [0.5979907498942453],
          [0.5983622001595633],
          [0.5985349949605846],
          [0.5986546575535244],
          [0.5984597055343299],
          [0.5983042184694954],
          [0.5983673455929349],
          [0.5984547249383486]]
```

The probabilities are then converted to predicted classes as following.

```
In [27]: predictions = lunak_ann.predict(X)
         predictions
```

```
Out[27]: [[1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0],
          [1.0]]
```

3.2.2 Keras

```
In [28]: from keras.models import Sequential
         from keras.layers import Dense
```

```
from keras.initializers import RandomUniform
import keras
import pandas as pd
import tensorflow as tf
```

```
Using TensorFlow backend.
```

We shall first initialize our Lunak ANN model.

```
In [29]: keras_ann = Sequential()
```

We initialize the weight matrix with a random uniform distribution.

```
In [30]: initializer = RandomUniform(minval=-0.05, maxval=0.05, seed=5)
```

Similiarly with our Lunak model, we add a hidden layer with 2 units and an output layer with 1 unit to our Keras model.

```
In [31]: keras_ann.add(Dense(2, activation='sigmoid',
                               input_dim=7, use_bias=True, kernel_initializer=initializer))
         keras_ann.add(Dense(1, activation='sigmoid',
                               use_bias=True, kernel_initializer=initializer))
```

We use Stochastic Gradient Descent as our Keras model optimizer.

```
In [32]: optimizer_ = keras.optimizers.SGD(momentum=0.001, lr=0.01)
```

```
In [33]: keras_ann.compile(loss='mean_squared_error',
                             optimizer=optimizer_, metrics=['accuracy'])
```

We then train our Keras model using the preprocessed data and labels from the previous section. Similiarly, we use `%timeit` to measure average training time.

```
In [34]: %%timeit
         with io.capture_output() as captured:
             keras_ann.fit(X, y, epochs=10, batch_size=4, validation_data=(X_val, y_val))
```

```
39.4 ms ± 10.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Predicting the original dataset using the trained models yield the following predicted probabilities.

```
In [35]: pred_prob = keras_ann.predict(X)
         pred_prob
```

```
Out[35]: array([[0.5604244 ],
                [0.56077284],
                [0.5607391 ],
                [0.5608534 ],
                [0.56050277],
                [0.5607991 ],
                [0.5605264 ],
                [0.5604657 ],
                [0.56070906],
                [0.5607866 ],
                [0.560738  ],
                [0.5607262 ]], dtype=float32)
```

We then convert the predicted probabilities to binary classes, with a threshold of 0.5.

```
In [36]: threshold = 0.5
         pred = [1 if x > threshold else 0 for x in pred_prob]
         pred

Out[36]: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

4 Analysis

4.1 Accuracy

Both our Lunak model and our Keras model produced the same predictions, i.e., all data are predicted as 1. This proves that our implementation works correctly for small datasets.

For results predicted both by Lunak and by Keras, even predicting the original dataset yields low accuracy. This is due to the small dataset used for training and testing. For larger datasets, the accuracy should improve.

4.2 Performance

Even though both our Lunak model and the baseline Keras model produces the same results, training the Keras model is an order of magnitude faster than training our Lunak model (~500ms vs. ~20ms). This is due to the following factors.

- Lunak is a textbook implementation of machine learning algorithms; no performance tricks are used. Tensorflow (and, by extension, Keras) is very much optimized to eke out the smallest edge in performance.
- Lunak is build purely on Python, even though libraries used here may be implemented in other languages. The core of Tensorflow is written in C++/CUDA.
- Lunak uses `numpy` for numerical operations, while Tensorflow, uses Eigen (a high-performance C++/CUDA numerical library), in addition to Nvidia's cuDNN.
- Lunak is single-threaded and runs purely in the CPU. Our Keras model uses Tensorflow-GPU, which utilizes both the CPU and the GPU for processing.

Random weight initialization may impact performance, but running the training process several times produce roughly the same 6:1 execution time ratio.

5 Roles and Responsibilities

Name	SID	Role
Felix Limanta	13515065	Layer and model implementation, debugging, Keras exploration, writing report
Holy Lovenia	13515113	Layer and model implementation, debugging, Keras exploration, writing report
Felix Limanta	13515143	Layer and model implementation, debugging, Keras exploration, writing report