

Introduction

Deep Learning is a growing field in Artificial Intelligence in which multiple layers learn patterns through deep computations. The architecture of deep learning is a **neural network** which is composed of layers of neurons. A **neuron** is essentially a function that takes in bias values and input values (originally from samples) to give an output. The weights, which are randomly initialized, are applied to the input values to determine the output value. The output of a neuron is passed to an activation function which is responsible for interpreting the output. The **activation function** essentially determines how much to adjust the weights of features based on the output so the model's predictions better fit the shape of the target output. To summarize, a neuron takes in data and transforms it into a more meaningful form, and then passes the output (transformed data) to the next neuron and so on. The neural network can have multiple layers of these neurons with each layer utilizing a specific activation function such as sigmoid or relu. An implementation of a simple neural network using TensorFlow and the Keras API was performed. This report details the results and findings of the experiments on the model.

Background

Keras is an open source library that allows programmers to interface the **TensorFlow** libraries with Python. The most basic neural network in the Keras API is the Sequential model which is essentially a linear stack of neural networks. Although simple, this model can be very powerful and has several parameters such as number of layers, number of nodes, activation functions, etc that can be modified based on a model's data and goals. The Sequential model was implemented with a dataset from Kaggle pertaining to heart disease. A total of 11 features are provided from the dataset such as age, sex, chest pain type, etc and the output variable is heart disease. A value of 0 indicates no heart disease and a value of 1 indicates heart disease. After creating, fitting, and evaluating the model, parameter tuning experiments were performed to better understand neural networks.


Methods

Data Preprocessing

Preprocessing of the data was performed before applying the data to the model. Initial preprocessing consists of "cleaning" the data:

- **Check (and remove) unnecessary columns:** there are none
- **Check (and remove) for null values:** there are none
- **Check (and remove) for duplicate values:** there are none

After the cleaning of the data was completed, the categorical column data was converted into numerical data. This was performed with the `get_dummies` method provided by the pandas library. This method employs the one hot encoding technique which converts categorical values into new categorical columns consisting of binary values (0s or 1s). An example demonstrating hot encoding is show below:



Color		Red	Yellow	Green
Red		1	0	0
Red		1	0	0
Yellow		0	1	0
Green		0	0	1
Yellow		0	0	1

Source: Kaggle

Next the data was converted into numpy arrays to be of an acceptable format for the Sequential model. The data was then split into training sets (samples, labels) and testing sets (samples, labels). The data was then normalized via the `MinMaxScaler` class from the scikit learn API. The training samples were fitted and transformed to values between 0 and 1. The testing samples were transformed to values between 0 and 1, but not fitted. The reason the testing data is not fitted with the `MinMaxScaler` is because knowledge of the testing set would leak into the model. The datasets are now ready to be used by a machine learning model.

Creating the Sequential Model

The initialization of the **Keras Sequential** model involves constructing and passing the layers that the model will have. The first model with supplied layers that are recommended by François Chollet in his book, *Deep Learning with Python*. The input layer is essentially the training set that is passed to the model. The first deep layer defines the shape that expects from the input layer. The first and second layers are assigned 16 neurons each and they employ the `relu` activation function. The `relu` activation function is the most commonly used activation function and was used because of its strength of learning nonlinear patterns. The last deep layer has 1 neuron and employs the `sigmoid` activation function. The `sigmoid` activation function is commonly used in binary classification problems because it outputs a 0 or 1.

Next the model is compiled via the `compile` method which involves parameters: optimizer function, loss function, and metrics. The loss function tells the model what to minimize, the optimizer function determines how the model will minimize, and the metrics tell the model what criteria to use to assess its progression.

After compiling the Keras model, the model is ready to be fitted with the training data via the `fit` method. The training data was fit to the model with parameters batch size of 20 and epochs of

30. The batch size defines the number of samples that will be passed through the network at a time. In other words, the neural network will process a batch number of samples before updating weights and processing more data. The epoch is the number of iterations the model will perform. Each training sample has been processed in the network by the end of each epoch. The Keras API outputs the loss percentage and accuracy percentage of the model at the end of each epoch. Generally speaking, the accuracy of the model increases as the loss decreases. The accuracy is the model's accuracy relative to its predictions on the training set. The loss function is what is being optimized during the training of the model. The model attempts to minimize the loss function. The model's results started to stabilize around epoch 7 when the model achieved 86.38% accuracy. Afterwards, the model's accuracy fluctuated around 86 - 87%.

Evaluating Model Performance

Once the data is fitted to the Sequential model, the model is evaluated by supplying the test samples and test labels to the model's evaluate method. This method returns the loss percentage and accuracy percentage from supplying the testing data. The model achieved about 83.15% accuracy with 37.65% loss.

Further Experiments

Further experiments with the Sequential model were performed involving tuning of the following parameters: number of hidden layers, number of units, loss functions, and activation functions.

Hidden Layers

Different models were created and evaluated with different numbers of layers. The results are below:

```
6/6 [=====] - 0s 1ms/step - loss: 0.3875 - accuracy: 0.8641
6/6 [=====] - 0s 940us/step - loss: 0.3684 - accuracy: 0.8696
6/6 [=====] - 0s 968us/step - loss: 0.3511 - accuracy: 0.8587
```

```
Achieved 0.3875 loss 0.8641 accuracy with 1 layer(s)
Achieved 0.3684 loss 0.8696 accuracy with 2 layer(s)
Achieved 0.3511 loss 0.8587 accuracy with 3 layer(s)
```

The model achieved very similar accuracy results when being passed 1 layer versus being passed 2 layers. The model with 2 layers was able to achieve a slightly higher accuracy with more loss making it superior to the model with 1 layer. The model with 3 layers achieved a lower accuracy score despite having a lower loss percentage than the model with 2 layers. And so the model with 3 layers must be overfitting more than the other models. It is learning the training

data better than the other (hence lower loss) yet results in poorer performance on unseen data as demonstrated by its accuracy score.

Hidden Units

Different models were created and evaluated with different numbers of units in the layers. The results are below:

```
6/6 [=====] - 0s 1ms/step - loss: 0.3605 - accuracy: 0.8587
6/6 [=====] - 0s 816us/step - loss: 0.3748 - accuracy: 0.8370
6/6 [=====] - 0s 846us/step - loss: 0.3888 - accuracy: 0.8370
```

```
Achieved 0.3605 loss 0.8587 accuracy with 16 units/nodes per deep layer
Achieved 0.3748 loss 0.8370 accuracy with 32 units/nodes per deep layer
Achieved 0.3888 loss 0.8370 accuracy with 64 units/nodes per deep layer
```

Generally speaking, the more the nodes that a layer has, the more that layer is going to learn about the small details and patterns in the data. In above experiments, adding more neurons to the layers significantly degraded the model's performance on the testing data. This is occurring because the presence of more nodes in the layers is causing the model to learn undesired patterns or noise. The loss is increasing because the model is making more errors on the testing dataset and accuracy is decreasing because the model is performing more poorly on the testing set.

Loss Functions

Different models were created and evaluated with different loss functions. The results are below:

```
6/6 [=====] - 0s 854us/step - loss: 0.3638 - accuracy: 0.8696
6/6 [=====] - 0s 916us/step - loss: 0.6380 - accuracy: 0.8478
6/6 [=====] - 0s 739us/step - loss: 0.7288 - accuracy: 0.8370
```

```
Achieved 0.3638 loss 0.8696 accuracy with the binary_crossentropy loss function
Achieved 0.6380 loss 0.8478 accuracy with the hinge loss function
Achieved 0.7288 loss 0.8370 accuracy with the squared_hinge loss function
```

For models predicting a binary outcome, the binary cross entropy loss function is usually applied to the Sequential model as it is intended to predict binary outcomes. Binary cross works by computing the average difference of output values to their respective target values for each feature. The hinge loss and squared hinge loss functions are alternative loss functions for binary classification problems. They are intended to be used when the target values are -1 or 1. Given the nature of their uses, it is no surprise that they performed significantly worse than the binary cross entropy loss function. Despite the hinge functions having very high loss percentages, the model is still able to predict test results with 84% and 83% accuracy. This is because the model making "big" errors in some of its predictions resulting in the model's loss increases significantly.

Activation Functions

Different models were created and evaluated with different activation functions of the output later. The results are below:

```
6/6 [=====] - 0s 792us/step - loss: 0.3668 - accuracy: 0.8533
6/6 [=====] - 0s 721us/step - loss: 0.4986 - accuracy: 0.8207
6/6 [=====] - 0s 1ms/step - loss: 0.7425 - accuracy: 0.8370
```

Achieved 0.3668 loss 0.8533 accuracy with the sigmoid activation function.

Achieved 0.4986 loss 0.8207 accuracy with the tanh activation function.

Achieved 0.7425 loss 0.8370 accuracy with the relu activation function.

Generally speaking, a sigmoid activation function is the preferred activation function for the last layer of a binary classifying model. For the sake of experiments, the tanh and relu functions were applied as the activations of the last layers. The model achieved lower accuracy scores and high loss percentages when using tanh and relu. Tanh compresses the input to be in the range of (-1, 1). To ensure, the results were not skewed from the feature range, the same tests were performed but with features being transformed into the range of -1 and 1. The results are below:

```
6/6 [=====] - 0s 843us/step - loss: 0.4098 - accuracy: 0.8533
{'tanh': [0.40980401635169983, 0.85326087474823]}
```

The tanh accuracy and loss significantly improved with the correct feature range assigned to the model, however, the model still does not perform as well as sigmoid. Sigmoid has less loss and a slightly higher accuracy making it superior. The sigmoid loss function most likely performed better because the data of the dataset tends to center around 0. The relu function performed the worst when it was the last layer for binary classification. This is because relu is non-differentiable at 0 which results the neurons tending to become inactive for all inputs. This highly reduces the model's learning capacity and hence explains the significant amount of loss and poorer results with testing data.

Conclusion

Neural network models can be very effective for a multitude of problems. This report demonstrated the effectiveness of a neural network in the form of a Keras Sequential model in predicting a binary outcome, whether a person has heart disease or a person does not have heart disease. The Sequential model has several parameters that can be adjusted such as number of layers, number of nodes, activation functions, and loss functions. Determining the optimal values for these parameters are dependent on the data, the goal of the model, and usually determined through experiments. The report details the effect of tuning the parameters individually for the sake of learning, however, a method such as GridSearchCV could be implemented to more efficiently determine optimal parameters.