Dietrich Kruse

Intro to AI

Neural Network Final Project

I began by inspecting the data for nan values, checking for correlation, and checking for imbalance in the dataset. I found that the dataset was very imbalanced, so I used different techniques such as random oversampling and random undersampling to balance the data [1]. It was also found that several columns had an uncomfortable amount of nan values. I played with omitting and including these columns, although the correct solution is probably to fill nan values with the mean or the most frequent value in the column.

Table 1) Percentage of NaN values

Column	Percentage of NaN values
Gender	23.5%
Major Discipline	14.7%
Company Size	31.0%
Company Type	32.0%

I then one-hot encoded the categorical data using pandas get_dummies() function and calculated the top 9 most correlated columns (really just my own curiosity). I found that the top 2 most correlated attributes with the target column were Full Time enrollment (15%) and No Relevant Experience (12.8%). The rest of the top 9 was rounded out by low experience levels (<1 year to 4 years) and STEM majors. It was also found that City Development Index had a correlation of -35%, the lowest of any attribute.

My model started with just two Dense layers which used rectified linear activation functions and an output layer using a softmax activation function. My loss function started as Binary Crossentropy. After training and making sure everything was working properly, I began to add layers of Batch Normalization and Dropout between the Dense Layers. I also implemented Batch Normalization as the first layer in order to do some preprocessing before getting to the dense layers, as discussed in [2]. I explored many different values of neurons in each layer, different values of dropout, different activation functions in my Dense layers, and different loss functions. I also used GridSearchCV to explore the optimal batch size and number of epochs.

I found that including the columns with the NaN values increased the score. I found the optimal batch size to be around 50 and optimal number of epochs to be 10. Sparse Categorical Crossentropy was determined to be the best loss function, as MSE and Binary Crossentropy both yielded under 50% accuracy. 128 Neurons, 0.4 dropout in the first layer and 64 neurons/0.2 dropout in the second layer was found to be the optimal layer design. The final model structure is shown in Figure 1. No matter what I tried, my results seemed to fall in the 75% to 76% range. However, once I applied oversampling, my results with the testing data improved to 80.5% (77.5% with undersampling). The highest accuracy I achieved was 81.1%. Unfortunately, I did not have time to implement any k-fold cross-validation. Another concern I have is that I oversampled without doing anything with NaN values, which may cause some errors in the model.

```python
def create_model():
    model = Sequential()
    model.add(BatchNormalization())
    model.add(Dense(128, input_dim=num_inputs, activation='relu'))
    model.add(Dropout(0.4))
    model.add(BatchNormalization())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())
    model.add(Dense(2, activation='softmax'))
    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics='accuracy')
    return model

model = create_model()
history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10, batch_size=batch_size)
```

Figure 1) Final Model Structure

```
202/202 [==============================] - 1s 4ms/step - loss: 0.6321 - accuracy: 0.6859 - val_loss: 0.5031 - val_accuracy: 0.7609
Epoch 2/10
202/202 [==============================] - 1s 3ms/step - loss: 0.5153 - accuracy: 0.7544 - val_loss: 0.4751 - val_accuracy: 0.7762
Epoch 3/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4933 - accuracy: 0.7611 - val_loss: 0.4617 - val_accuracy: 0.7868
Epoch 4/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4709 - accuracy: 0.7735 - val_loss: 0.4578 - val_accuracy: 0.7836
Epoch 5/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4662 - accuracy: 0.7767 - val_loss: 0.4537 - val_accuracy: 0.7943
Epoch 6/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4549 - accuracy: 0.7804 - val_loss: 0.4512 - val_accuracy: 0.7956
Epoch 7/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4485 - accuracy: 0.7834 - val_loss: 0.4488 - val_accuracy: 0.7947
Epoch 8/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4442 - accuracy: 0.7843 - val_loss: 0.4500 - val_accuracy: 0.7905
Epoch 9/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4463 - accuracy: 0.7845 - val_loss: 0.4502 - val_accuracy: 0.7915
Epoch 10/10
202/202 [==============================] - 1s 3ms/step - loss: 0.4369 - accuracy: 0.7924 - val_loss: 0.4496 - val_accuracy: 0.7938
68/68 [==============================] - 0s 1ms/step - loss: 0.4327 - accuracy: 0.8113
Scores on testing data: [0.4326697587966919, 0.8113120198249817]
```

Figure 2) Final Model Output

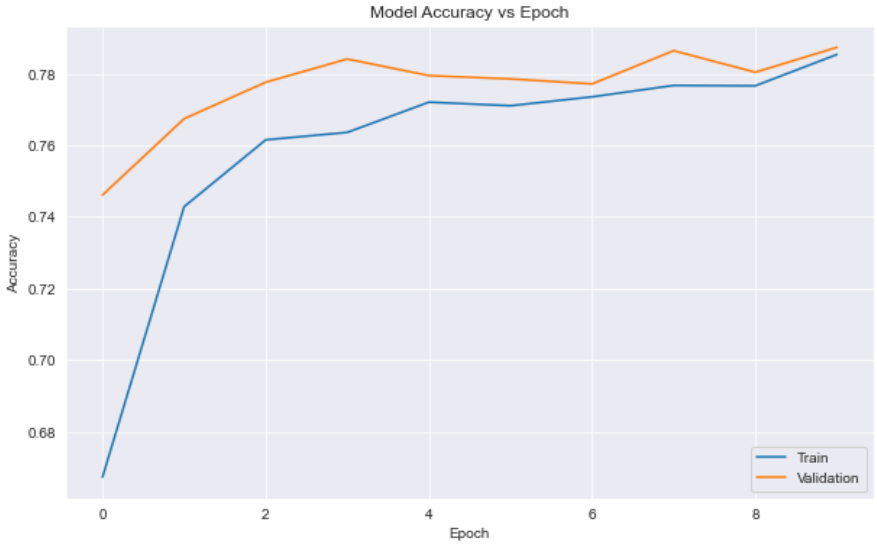

Figure 3) Model Accuracy vs Epoch

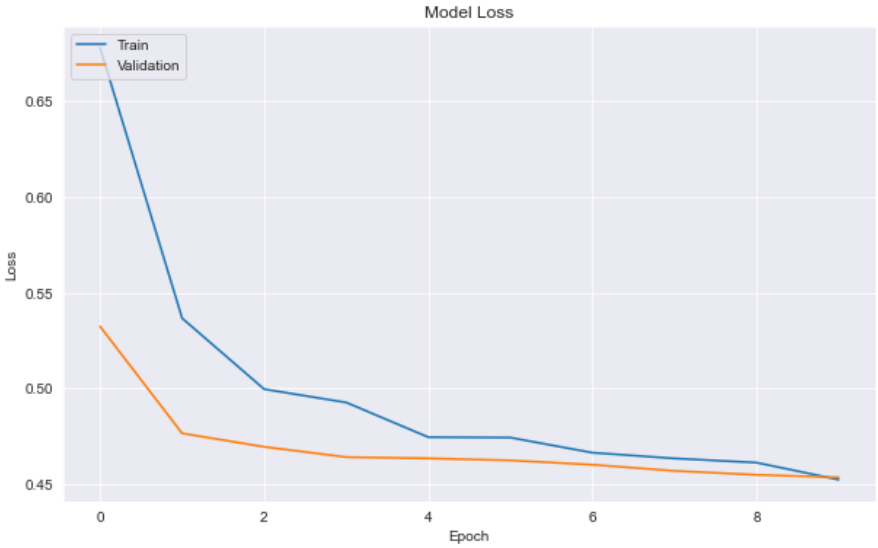

Figure 4) Model Loss vs Epoch

References

[1] https://www.machinecurve.com/index.php/2020/11/10/working-with-imbalanced-datasets-with-tensorflow-and-keras/

[2] https://www.kaggle.com/ryanholbrook/dropout-and-batch-normalization