Benton Luu 10175103

CPSC 501 – Assignment 4: TensorFlow

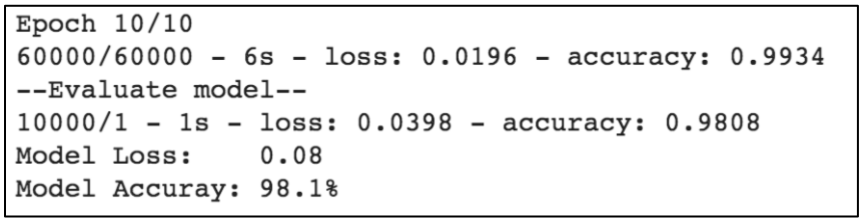
Link to GitHub: https://github.com/bentonluu/CPSC501-Assignment4-TensorFlow

Part 1: MNIST Logistic Regression

- Modifications:
    - Added a dense layer with 256 units (tensor nodes) and an activation function of 'relu'. This layer is the 1st hidden layer of the model. Found that with more units, the model was slightly more accurate on the training dataset but the cost was it took a longer time for each epoch iteration to complete the training. Since the result was approximately the same after 10 epochs, decided to keep the units at 256 as it maintained a balance between accuracy and the time it took to train the model. The reason for using the 'relu' activation function was that it trains faster compared to other activation functions, since any input value that is negative is equal to 0 thus there is less computation involved. Additionally, since 'relu' is a linear function, it has a faster learning rate as with increasing positive values the output will increase as well.
    - Added a dropout layer between the 1st hidden layer and the output layer to randomly drop a portion of inputs that feed into the output layer from the 1st hidden layer. The rate that the inputs are dropped at is set to 0.1 or 10%. The reason for the inclusion for the dropout layer is that it helps mitigate the issue with overfitting the model.
    - Changed the 'sigmoid' activation function to a 'softmax' activation function for the output layer. The reason for this is because 'sigmoid' is best used for two-class classification whereas 'softmax' is best used for multi-class classification. Since the MNIST dataset has more than two classes, the 'softmax' activation function would be more beneficial in improving the model's accuracy with classifying multiple classes.
    - Changed the epoch from 1 to 10. Having the model train on more iterations of the training data improved the overall accuracy of the model. However, found that training the model with more than 10 iterations had minimal improvement on the model's accuracy thus kept the epoch value at 10.
    - Changed the 'sgd' optimizer to an 'adam' optimizer. Using all the possible optimizers with TensorFlow, the best result I found was when using the 'adam' optimizer which outperformed the other optimizers in the model's accuracy.

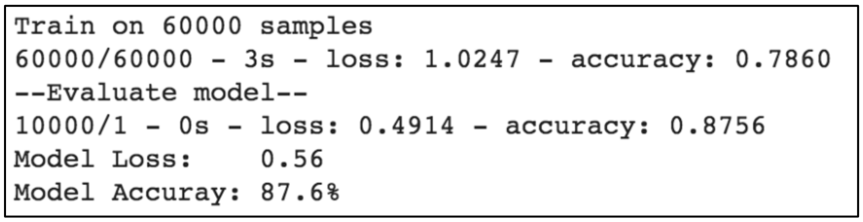
- Instructions:
    - Upload file 'MNISTStarter.ipynb' into Google Colab
    - Run the 1st section to load TensorFlow
    - Run the 2nd section to training and evaluate the model with the MNIST dataset
    - Run the 3rd section to save the model
    - Run the 4th section to download the model

- Results:
  - After the changes were implemented, the model's accuracy was improved to ~99% on the training data and ~98% on the test data.



```
Epoch 10/10
60000/60000 - 6s - loss: 0.0196 - accuracy: 0.9934
--Evaluate model--
10000/1 - 1s - loss: 0.0398 - accuracy: 0.9808
Model Loss:    0.08
Model Accuray: 98.1%
```

  - This is significant improvement from the original model that was provided where it was only ~79% accurate on the training data and ~88% accurate on the test data.

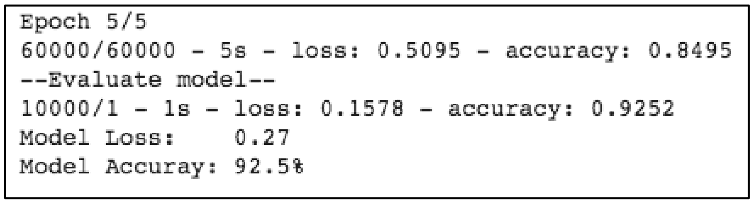


```
Train on 60000 samples
60000/60000 - 3s - loss: 1.0247 - accuracy: 0.7860
--Evaluate model--
10000/1 - 0s - loss: 0.4914 - accuracy: 0.8756
Model Loss:    0.56
Model Accuray: 87.6%
```

Part 2: Logistic Regression on a replacement for MNIST dataset

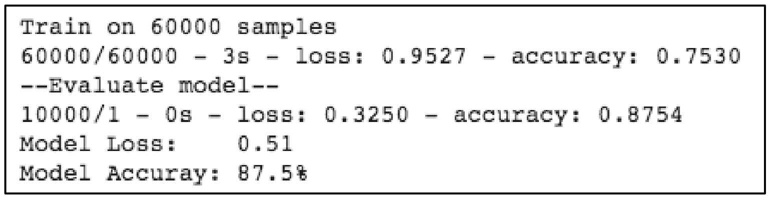
- Modifications (before incorrect image classification changes):
  - Added a dense layer with 256 units (tensor nodes) and an activation function of 'relu'. This layer is the 1st hidden layer of the model. Similar to part 1) where with more units, the model was slightly more accurate on the training dataset but the cost was it took a longer time for each epoch iteration to complete the training, the same happened with this model. Thus, decided to keep the units at 256 as it maintained a balance between accuracy and the time it took to train the model. The reason for using the 'relu' activation function was that it trains faster as well as has a faster learning rate compared to other activation functions.
  - Added a dropout layer between the 1st hidden layer and the output layer to randomly drop a portion of inputs that feed into the output layer from the 1st hidden layer. The rate that the inputs are dropped at is set to 0.5 or 50%. The reason for the inclusion for the dropout layer was that it helps mitigate the issue with overfitting the model.
  - Changed the 'sigmoid' activation function to a 'softmax' activation function for the output layer. Similar to part 1) where it was a classification problem with multiple classes, part 2) is nearly the same problem expect with a different dataset, thus using the 'softmax' activation function is best suited for this type of classification problem.
  - Changed the epoch from 1 to 5. Having the model train on more iterations of the training data improved the overall accuracy of the model but with more iterations there was minimal improvement.
  - Changed the 'sgd' optimizer to an 'adam' optimizer. Using all the possible optimizers with TensorFlow, the best result I found was when using the 'adam' optimizer which outperformed the other optimizers in the model's accuracy.

- Results:
    - After the changes were implemented, the model's accuracy was improved to ~85% on the training data and ~93% on the test data.



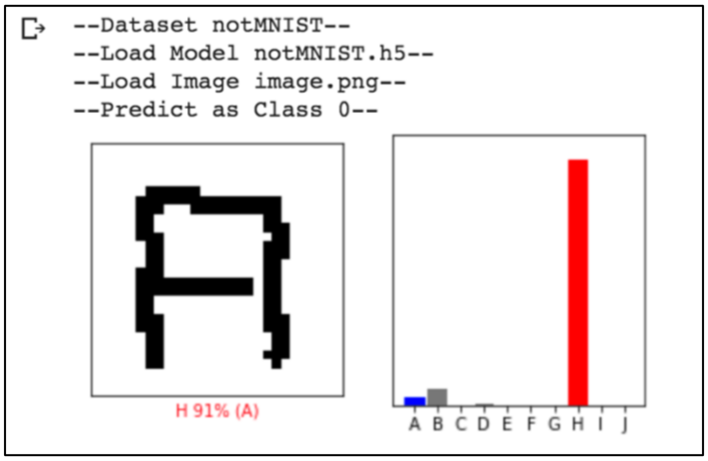
```
Epoch 5/5
60000/60000 - 5s - loss: 0.5095 - accuracy: 0.8495
--Evaluate model--
10000/1 - 1s - loss: 0.1578 - accuracy: 0.9252
Model Loss:    0.27
Model Accuray: 92.5%
```

    - This is an improvement from the original model that was provided where it was only ~75% accurate on the training data and ~88% accurate on the test data.



```
Train on 60000 samples
60000/60000 - 3s - loss: 0.9527 - accuracy: 0.7530
--Evaluate model--
10000/1 - 0s - loss: 0.3250 - accuracy: 0.8754
Model Loss:    0.51
Model Accuray: 87.5%
```

    - Testing the improved model with a created image, the model incorrectly identified the image to be in class 'H' with a 91% confidence rate however the correct class was 'A'.

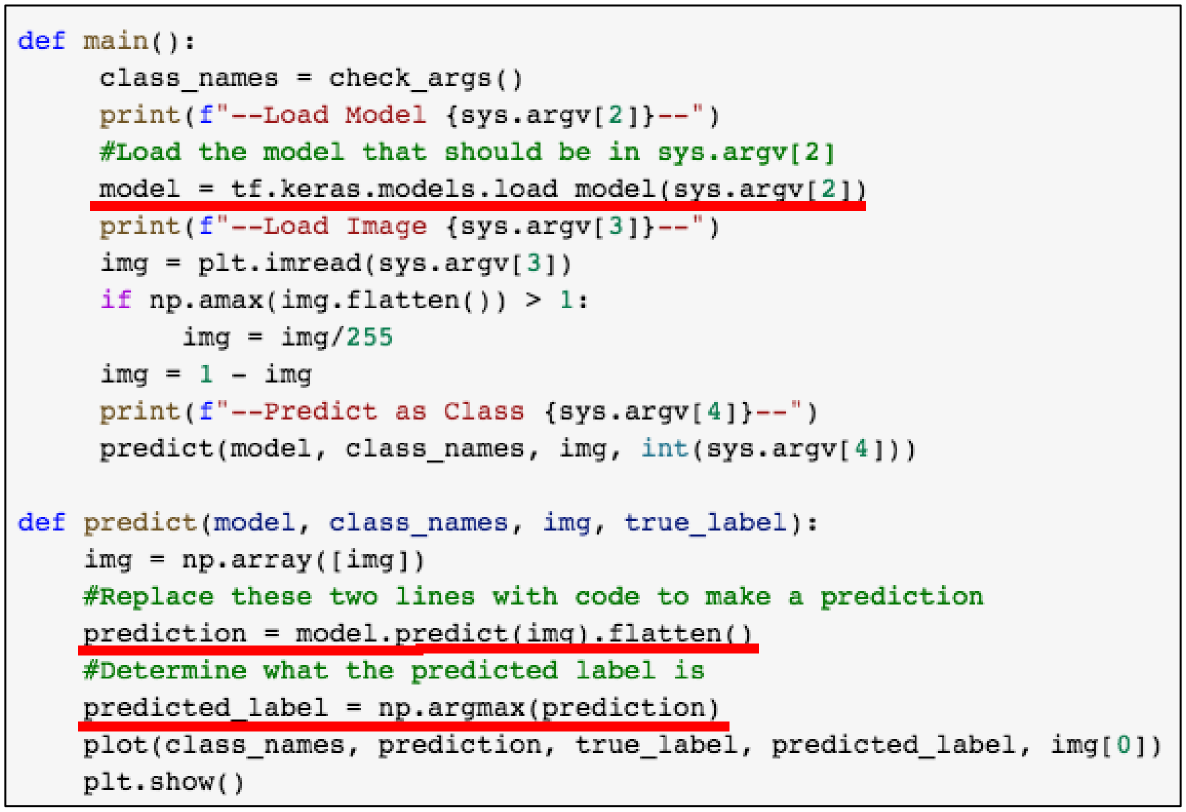


```
--Dataset notMNIST--
--Load Model notMNIST.h5--
--Load Image image.png--
--Predict as Class 0--
```

- Modifications (after incorrect image classification changes):
    - Added a dense layer with 128 units and an activation function of 'relu'. This layer is the 2nd hidden layer of the model. To keep consistent with the 1st hidden layer used the same activation function but reduced the number of units in half to 128 units.
    - Added a dropout layer between the 2nd hidden layer and the 3rd hidden layer to randomly drop a portion of inputs that feed into the 3rd hidden layer from the 2nd hidden layer. The rate that the inputs are dropped at is set to 0.5 or 50%. The reason for the inclusion for the dropout layer was that it helps mitigate the issue with overfitting the model.
    - Added a dense layer with 64 units and an activation function of 'relu'. This layer is the 3rd hidden layer of the model. To keep consistent with the 2nd hidden layer used the same activation function but reduced the number of units in half to 64 units.

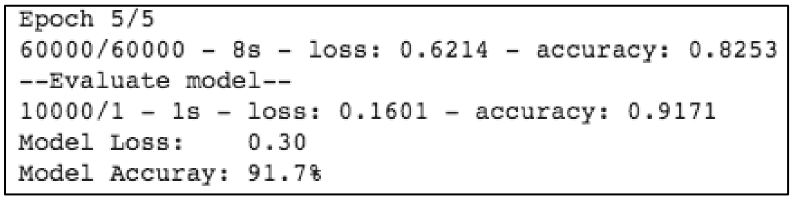
- Added a dropout layer between the 3rd hidden layer and the output layer to randomly drop a portion of inputs that feed into the output layer from the 3rd hidden layer. The rate that the inputs are dropped at is set to 0.5 or 50%. The reason for the inclusion for the dropout layer was that it helps mitigate the issue with overfitting the model.

- Code added 'predict.ipynb' file:



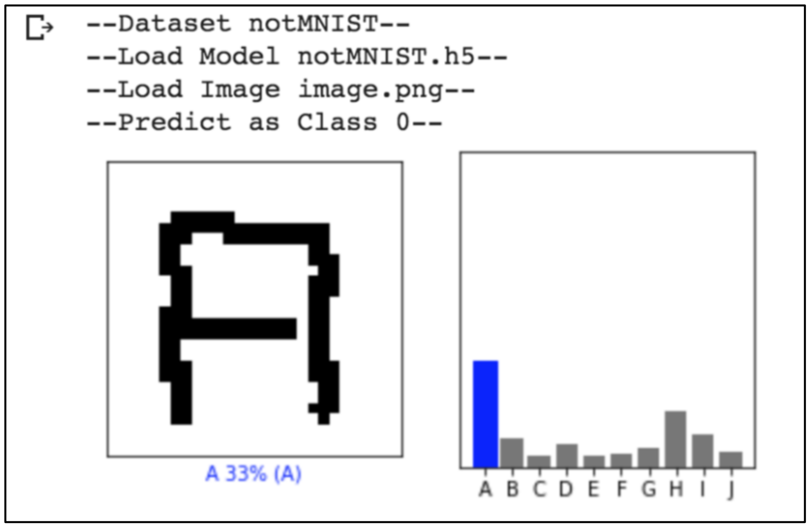
```
def main():
     class_names = check_args()
     print(f"--Load Model {sys.argv[2]}--")
     #Load the model that should be in sys.argv[2]
     model = tf.keras.models.load_model(sys.argv[2])
     print(f"--Load Image {sys.argv[3]}--")
     img = plt.imread(sys.argv[3])
     if np.amax(img.flatten()) > 1:
          img = img/255
     img = 1 - img
     print(f"--Predict as Class {sys.argv[4]}--")
     predict(model, class_names, img, int(sys.argv[4]))

def predict(model, class_names, img, true_label):
    img = np.array([img])
    #Replace these two lines with code to make a prediction
    prediction = model.predict(img).flatten()
    #Determine what the predicted label is
    predicted_label = np.argmax(prediction)
    plot(class_names, prediction, true_label, predicted_label, img[0])
    plt.show()
```

- Results:
  - After the revised model changes were implemented, the model's accuracy was ~83% on the training data and ~92% on the test data. Compared to the improved model, the accuracy on the training data decreased by ~2% on the training data and ~1% on the test data.



```
Epoch 5/5
60000/60000 - 8s - loss: 0.6214 - accuracy: 0.8253
--Evaluate model--
10000/1 - 1s - loss: 0.1601 - accuracy: 0.9171
Model Loss:    0.30
Model Accuray: 91.7%
```

- Testing the revised model on the created image again, the model correctly identifies the image to be in class 'A' with a 33% confidence rate.



```
--Dataset notMNIST--
--Load Model notMNIST.h5--
--Load Image image.png--
--Predict as Class 0--
```

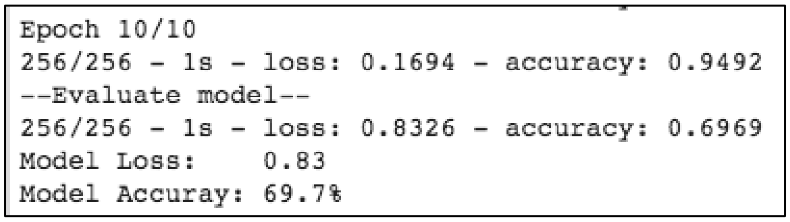
- Note that depending on the way the model is trained with the training data and the weights that are set in the model after training, the prediction of the created image will be correct some of the time and incorrect the other times. The model that was submitted correctly identifies the created image.

- Instructions:
  - Upload files 'notMNISTStarter.ipynb' and 'predict.ipynb' into Google Colab
  - In the 'notMNISTStarter' notebook – run all sections:
    - Run section 2 to upload the notMNIST.npz data
    - Run section 3 to train and evaluate the model with the data
    - Run section 4 to save the model
    - Run section 5 to download the saved model 'notMNIST.h5'
  - In the 'predict.ipynb' notebook – run all sections:
    - Run section 3 to upload files 'notMNIST.h5' and 'image.png'
    - Run section 5 and enter the following parameters:
      - Dataset:notMNIST
      - Model:notMNIST.h5
      - Image:image.png
      - Class index:0
    - Run section 8 to load the model and evaluate it against the uploaded image

Part 3: Build a logistic regression model to predict if someone has coronary heart disease

- Initial Model:
  - A dense layer with 128 units (tensor nodes) and an activation function of 'relu'. This layer is the 1st hidden layer of the model. The reason for using the 'relu' activation function is that it trains faster as well as has a faster learning rate compared to other activation functions. Found that with more units, the model was slightly more accurate on the training dataset but the cost was it took a longer time for each epoch iteration to complete the training thus decided to keep the layer at 128 units.
  - A dense layer with 128 units (tensor nodes) and an activation function of 'relu'. This layer is the 2nd hidden layer of the model. To keep consistent with the 1st hidden layer used the same activation function and the same number of units.
  - A dense layer with 1 unit (tensor nodes) and a 'sigmoid' activation function. This layer is the output layer of the model. The reason for this is because the label is binary and 'sigmoid' is best used suited for two-class classifications. Since the 'chd' label is either a 0 or 1, using 'sigmoid' would be most beneficial in improving the model's accuracy with classifying between these two classes.
  - Set the epoch steps to 128. This allowed the model to train with the training dataset 128 times with a batch size of 5 for each epoch iteration. The higher the epoch step value was the higher accuracy the model was with the training data but less accurate with the test data.

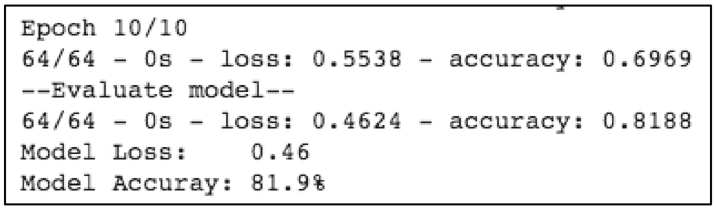
- Results:
  - Before the overfitting changes were implemented, the model's accuracy was ~95% on the training data and ~70% on the test data. This difference between the accuracy between the training and test data is attributed to the fact that the model was over trained with the training data also known as overfitting. Since the model trained specifically with the training data and no new data was provided, it formed patterns in the neutral network rather than continue to learn. When the model was given the test data that it had never seen before, the patterns that it developed did not help with correctly predicting the if the person had coronary heart disease or not thus resulted in a significant decrease in accuracy with the test data.



```
Epoch 10/10
256/256 - 1s - loss: 0.1694 - accuracy: 0.9492
--Evaluate model--
256/256 - 1s - loss: 0.8326 - accuracy: 0.6969
Model Loss:    0.83
Model Accuray: 69.7%
```

- Modification to Model for Overfitting:
  - Added a L2 kernel regularizer (sum of the squared weights) and set the penalty value to 0.0001 for the 1st hidden layer. Used to help mitigate the issue with overfitting.
  - Added a dropout layer between the 1st hidden layer and the 2nd hidden layer to randomly drop a portion of inputs that feed into the 2nd hidden layer from the 1st hidden

layer. The rate that the inputs are dropped at is set to 0.5 or 50%. Set the dropout rate high in order for the model to not be over trained with the training dataset.
  - Added a L2 kernel regularizer (sum of the squared weights) and set the penalty value to 0.0001 for the 2nd hidden layer. Used to help mitigate the issue with overfitting.
  - Added a dropout layer between the 2nd hidden layer and the output layer to randomly drop a portion of inputs that feed into the output layer from the 2nd hidden layer. The rate that the inputs are dropped at is set to 0.5 or 50%. Set the dropout rate high in order for the model to not be over trained with the training dataset.
  - Changed the epoch steps from 128 to 64. Lowered the epoch steps because with a batch size of 5 and there only being ~370 training samples, the model would be over trained since it would see the same samples more than once. By lowering the steps it reduces this issue and allows for the model to be more generalized.
- Results:
  - After the overfitting changes were implemented, the model's accuracy was ~70% on the training data and ~82% on the test data. The model was able to more accurately predict the test data since it was not over trained with the training data. It was able to learn what parameters likely are connected to coronary heart disease but since some of the inputs would be dropped it helped reduce overfitting of the model and thus was better able to accurately predict the test data which it had never seen before.



```
Epoch 10/10
64/64 - 0s - loss: 0.5538 - accuracy: 0.6969
--Evaluate model--
64/64 - 0s - loss: 0.4624 - accuracy: 0.8188
Model Loss:    0.46
Model Accuray: 81.9%
```

- Instructions:
  - Upload file 'heartPredict.ipynb' into Google Colab
  - Run the 1st section to load TensorFlow
  - Run the 2nd section to upload the 'heart.csv' data
  - Run the rest of the sections (the last section will train and evaluate the model)

- References:
  - Code for loading and pre-processing the CSV data for the 'heartPredict' file is based on code from:
    - https://www.tensorflow.org/tutorials/load_data/csv
  - Code for dealing with overfitting of the model is based on code from:
    - https://www.tensorflow.org/tutorials/keras/overfit_and_underfit