# DeepLearningProject

July 19, 2023

## 1 Deep Learning Home Assignment

### 1.0.1 Ashley Honeycutt

In this assignment I build a convolutional neural network (CNN) model to identify three classes of images in the Real Sense depth image dataset.

First, I import all the necessary libraries and packages.

[31]:
```
import numpy as np
import os
import cv2
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.callbacks import ModelCheckpoint
```

### 1.0.2 Preparing the Dataset

From the terminal I downloaded the dataset and unzipped the folder. Now, this notebook and the three image directories are saved in the same folder.

The Real Sense dataset was downloaded from this link: https://drive.google.com/uc?id=11UOCSc1ynUX1Mqj9M36s9_d0t-SFgjFG&export=download&authuser=0

The code below is copied from the home assignment instructions. It serves to convert the image dataset to a numpy array so we can use it to train our model. In the code, we use the OS module to create three lists with the contents of each directory/folder. Then, we create labels for our data. We use the numpy ones function to create three separate arrays with 0's, 1's, and 2's in place of the images, and then we concatenate the arrays vertically. By viewing the shape we can see that our final label array is 1-dimensional with 900 items.

[32]:
```
list_of_class2 = os.listdir("real_sense_woman100x100")
list_of_class1 = os.listdir("real_sense_fadam100x100")
list_of_class0 = os.listdir("real_sense_fake100x100")
```

```
labels2 = np.ones(len(list_of_class2)) * 2 #creates a label of 2
labels1 = np.ones(len(list_of_class1))
labels0 = np.zeros(len(list_of_class0))
labels = np.concatenate((labels2, labels1, labels0), axis=0)
print(labels.shape)
```

```
(900,)
```

In this code we create a dataset with resized images.

First, we define a variable dataset as an empty list and dimensions as 100x100.

Then, we have three for loops, one for each class. We use the cv2 module's imread function to read in each image file, and then the resize function to resize each image to the 100x100 dimensions specified above, and use inter_area interpolation to reduce distortion. We append the adjusted images to the dataset list.

Then we convert the dataset list to a numpy array and assign it to X. We define y as the labels array.

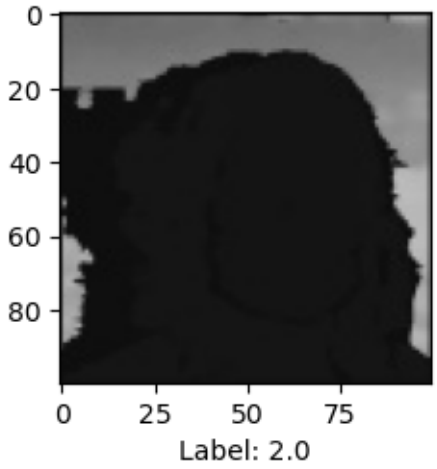
[33]:
```
dataset = []
dim = (100, 100)


for img in list_of_class2:
  image = cv2.imread("real_sense_woman100x100/" + str(img))
  resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
  dataset.append(resized)
for img in list_of_class1:
  image = cv2.imread("real_sense_fadam100x100/" + str(img))
  resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
  dataset.append(resized)
for img in list_of_class0:
  image = cv2.imread("real_sense_fake100x100/" + str(img))
  resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
  dataset.append(resized)

X = np.array(dataset)
y = labels
```

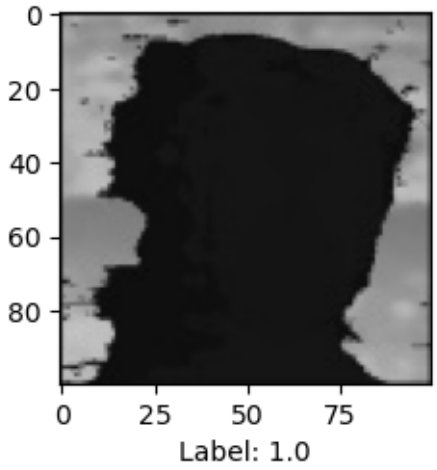
### 1.0.3 Displaying Images

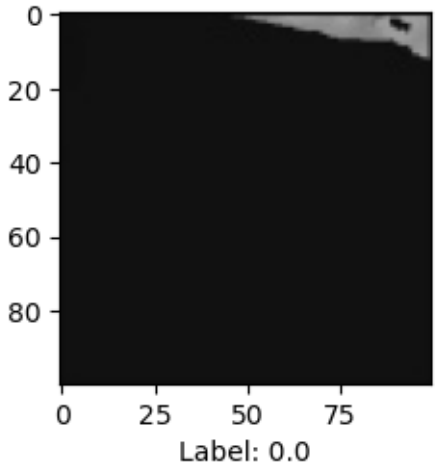
To display some of the images in our dataset, we defined a function called 'plot_sample' that uses matplotlib's pyplot to display images from our numpy array (X) and label them from the corresponding labels array (y).

[34]:
```
import matplotlib.pyplot as plt

def plot_sample(x,y,index):
  plt.figure(figsize=(5,2.5))
```

```
  plt.imshow(x[index])
  plt.xlabel(f'Label: {y[index]}')

plot_sample(X,y,5) #displays an image from class 2 (woman)
plot_sample(X,y,450) #displays an image from class 1 (Adam)
plot_sample(X,y,899) #displays an image from class 0 (fake)
```






### 1.0.4 Train, Test, Validation Split

We use the sklearn function train_test_split to divide our data into separate arrays. The training data tunes the parameters of our model (e.g. weights and biases), validation data fine-tunes the hyperparameters (e.g. learning rate) and test data is used to evaluate the model's performance by comparing its results with the training results.

To avoid overfitting we don't want to allocate too much to the training data. We've set 20% of the data for testing, and 75%/25% of the remaining data for training/validation.

[35]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=1)

X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
↪25, random_state=1)
```

### 1.0.5 Building a Convolutional Neural Network

First we instantiate an object of a Sequential model. This model is appropriate since we have a single input and single output, and it is a streamlined structure that is easy to build.

Next, we add layers to our model.

#### Convolutional Layer

The first layer is a 2D convolutional layer, which creates a filter (or kernel) to apply to the image inputs to detect features. The first parameter is the number of filters we will apply to each input (32), and the next is the kernel size, which we have specified as 3x3 pixels. We have chosen our activation function as ReLu (rectified linear unit). For each input, relu returns the maximum: (0,input value), which removes negative values from the equation and introduces non-linearity to the model, which allows for more complexity. Lastly, we specify the dimensions of the input images.

The second layer is a max pooling layer. This layer is responsible for reducing the image size, which

lessens the computational power needed to train the model. Additionally, the max pooling layer helps our model filter out 'noise' since it captures only the main features, and this helps the model handle distortions and variations in input images. We have chosen a 2x2 filter which is applied to the feature map generated by the conv2D layer, and the maximum value is chosen from that section and used to generate a new feature map.

We repeated these two layers to try to maximize the image detection power of the model. In the second conv2D layer, the input shape is smaller thanks to the max pooling layer above it.

**Dense Layer**

The second 'section' of our model is the dense layers, which are fully connected layers that serve to refine and eventually produce an output, which in this case is image classification.

The first four layers of our model (above) have produced a 2D array, which we need to flatten to 1D in order to apply the dense layers. Hence, the first layer is a flatten function.

Next, we have chosen two activation functions. First is relu, which is an efficient and useful function that also mitigates the vanishing gradient problem. This problem can lead to overfitting which results in poor performance on newly introduced data, so it is important to avoid. We set the number of layers to 64 after some trial and error. Fewer layers result in lower accuracy and more layers are less efficient.

Finally, we have 10 layers of the softmax activation function. This function is often used for multiclass classification problems with no overlap, like this one. It creates a probability distribution of classes from the raw outputs.

[36]:
```
model = keras.models.Sequential()
#Convolutional layer
model.add(keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu",␣
 ↪input_shape=(100, 100, 3))) #detects features
model.add(keras.layers.MaxPooling2D((2, 2)))
model.add(keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu",␣
 ↪input_shape=(32, 32, 3))) #detects features
model.add(keras.layers.MaxPooling2D((2, 2)))
#Dense layer
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(64, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

### 1.0.6 Compiling the Model

**Optimizer**

We selected the optimizer known as 'adam', or Adaptive Moment Estimation for our model. Adam is the most popular optimizer for its effectiveness and efficiency. In trial and error we also tried stochastic gradient descent (SGD) but achieved higher accuracy with adam.

**Loss Function**

We use the sparse categorical crossentropy loss function since our data contains more than two classes and the labels are presented as integers.

**Metrics**

We will measure the accuracy of our model, meaning the percentage of correct predictions.

[37]:
```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])
```

### 1.0.7 Training the Model

We use the ModelCheckpoint function to save our best performing model in a separate file.

Then we use the fit function to train our model. As parameters we've added our training datasets, doubled the default batch size to 64, changed epochs to 15 (the number of times we put our data through the model), added the validation datasets, and added a callback to refer to the ModelCheckpoint function above.

Finally, we set a variable 'score' to evaluate our model and passed in the test datasets.

[38]:
```
check = ModelCheckpoint('best.h5', save_best_only=True, monitor='val_accuracy',␣
↪mode='max', verbose=1)
history = model.fit(X_train, y_train, batch_size=64, epochs=15,␣
↪validation_data=(X_val, y_val), callbacks=[check])
score = model.evaluate(X_test, y_test)
```

```
Epoch 1/15
9/9 [==============================] - ETA: 0s - loss: 71.4286 - accuracy:
0.3537
Epoch 1: val_accuracy improved from -inf to 0.76667, saving model to best.h5
9/9 [==============================] - 5s 430ms/step - loss: 71.4286 - accuracy:
0.3537 - val_loss: 1.6560 - val_accuracy: 0.7667
Epoch 2/15
9/9 [==============================] - ETA: 0s - loss: 1.1542 - accuracy: 0.7963
Epoch 2: val_accuracy improved from 0.76667 to 0.85556, saving model to best.h5
9/9 [==============================] - 3s 348ms/step - loss: 1.1542 - accuracy:
0.7963 - val_loss: 0.4750 - val_accuracy: 0.8556
Epoch 3/15
9/9 [==============================] - ETA: 0s - loss: 0.2916 - accuracy: 0.9093
Epoch 3: val_accuracy improved from 0.85556 to 0.93333, saving model to best.h5
9/9 [==============================] - 3s 329ms/step - loss: 0.2916 - accuracy:
0.9093 - val_loss: 0.2164 - val_accuracy: 0.9333
Epoch 4/15
9/9 [==============================] - ETA: 0s - loss: 0.1683 - accuracy: 0.9685
Epoch 4: val_accuracy improved from 0.93333 to 0.96667, saving model to best.h5
9/9 [==============================] - 3s 366ms/step - loss: 0.1683 - accuracy:
0.9685 - val_loss: 0.1371 - val_accuracy: 0.9667
Epoch 5/15
9/9 [==============================] - ETA: 0s - loss: 0.0716 - accuracy: 0.9926
Epoch 5: val_accuracy improved from 0.96667 to 0.97778, saving model to best.h5
9/9 [==============================] - 3s 350ms/step - loss: 0.0716 - accuracy:
0.9926 - val_loss: 0.1349 - val_accuracy: 0.9778
```

```
Epoch 6/15
9/9 [==============================] - ETA: 0s - loss: 0.0426 - accuracy: 1.0000
Epoch 6: val_accuracy did not improve from 0.97778
9/9 [==============================] - 3s 330ms/step - loss: 0.0426 - accuracy:
1.0000 - val_loss: 0.1189 - val_accuracy: 0.9722
Epoch 7/15
9/9 [==============================] - ETA: 0s - loss: 0.0358 - accuracy: 1.0000
Epoch 7: val_accuracy improved from 0.97778 to 0.98889, saving model to best.h5
9/9 [==============================] - 3s 328ms/step - loss: 0.0358 - accuracy:
1.0000 - val_loss: 0.0994 - val_accuracy: 0.9889
Epoch 8/15
9/9 [==============================] - ETA: 0s - loss: 0.0331 - accuracy: 1.0000
Epoch 8: val_accuracy did not improve from 0.98889
9/9 [==============================] - 3s 320ms/step - loss: 0.0331 - accuracy:
1.0000 - val_loss: 0.1087 - val_accuracy: 0.9889
Epoch 9/15
9/9 [==============================] - ETA: 0s - loss: 0.0317 - accuracy: 1.0000
Epoch 9: val_accuracy did not improve from 0.98889
9/9 [==============================] - 3s 311ms/step - loss: 0.0317 - accuracy:
1.0000 - val_loss: 0.1180 - val_accuracy: 0.9722
Epoch 10/15
9/9 [==============================] - ETA: 0s - loss: 0.0311 - accuracy: 0.9981
Epoch 10: val_accuracy did not improve from 0.98889
9/9 [==============================] - 3s 325ms/step - loss: 0.0311 - accuracy:
0.9981 - val_loss: 0.2207 - val_accuracy: 0.9722
Epoch 11/15
9/9 [==============================] - ETA: 0s - loss: 0.0325 - accuracy: 1.0000
Epoch 11: val_accuracy did not improve from 0.98889
9/9 [==============================] - 3s 332ms/step - loss: 0.0325 - accuracy:
1.0000 - val_loss: 0.2195 - val_accuracy: 0.9667
Epoch 12/15
9/9 [==============================] - ETA: 0s - loss: 0.0320 - accuracy: 1.0000
Epoch 12: val_accuracy did not improve from 0.98889
9/9 [==============================] - 3s 334ms/step - loss: 0.0320 - accuracy:
1.0000 - val_loss: 0.2344 - val_accuracy: 0.9667
Epoch 13/15
9/9 [==============================] - ETA: 0s - loss: 0.0303 - accuracy: 1.0000
Epoch 13: val_accuracy did not improve from 0.98889
9/9 [==============================] - 3s 315ms/step - loss: 0.0303 - accuracy:
1.0000 - val_loss: 0.3628 - val_accuracy: 0.9611
Epoch 14/15
9/9 [==============================] - ETA: 0s - loss: 0.0295 - accuracy: 1.0000
Epoch 14: val_accuracy did not improve from 0.98889
9/9 [==============================] - 3s 310ms/step - loss: 0.0295 - accuracy:
1.0000 - val_loss: 0.8196 - val_accuracy: 0.9444
Epoch 15/15
9/9 [==============================] - ETA: 0s - loss: 0.0296 - accuracy: 1.0000
Epoch 15: val_accuracy did not improve from 0.98889
```

```
9/9 [==============================] - 3s 297ms/step - loss: 0.0296 - accuracy:
1.0000 - val_loss: 0.1187 - val_accuracy: 0.9833
6/6 [==============================] - 0s 41ms/step - loss: 0.0723 - accuracy:
0.9944
```

### 1.0.8 Evaluating the Model

Finally, we can see how our model is performing. In the output above we can see that the accuracy is increasing and loss is decreasing with each epoch, especially from the first to the second epoch. This shows that our model is improving with each iteration, just as we want.

Below, we have printed the accuracy and loss scores, which are based on comparing the test data results with the training/validation data results. Our accuracy score is very high, nearly 100%, which is excellent. This shows that our model is able to accurately predict the label of a new input image. Our loss score is fairly low but could be improved upon even further.

[39]:
```
print('Accuracy:', score[1])
print('Loss:',score[0])
```

```
Accuracy: 0.9944444298744202
Loss: 0.0723387822508812
```

**Plotting Accuracy and Loss across Epochs**

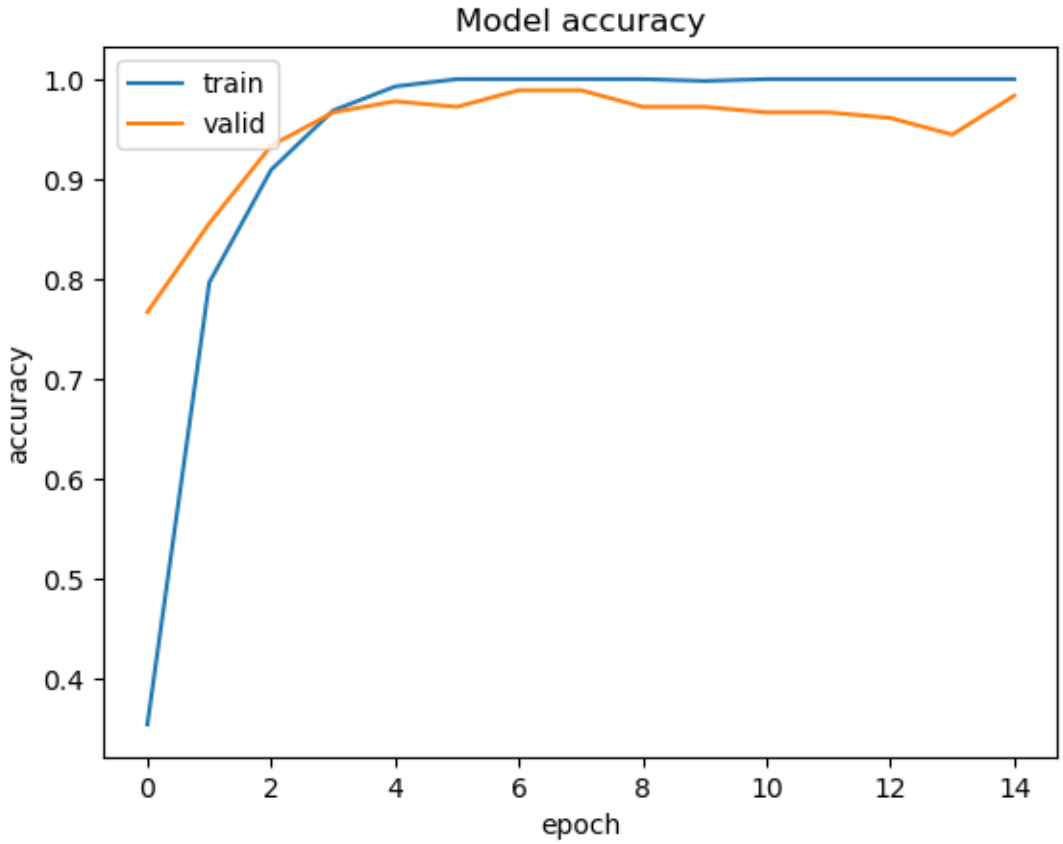
Below is a function to plot the change in accuracy and loss of training and validation data across epochs.

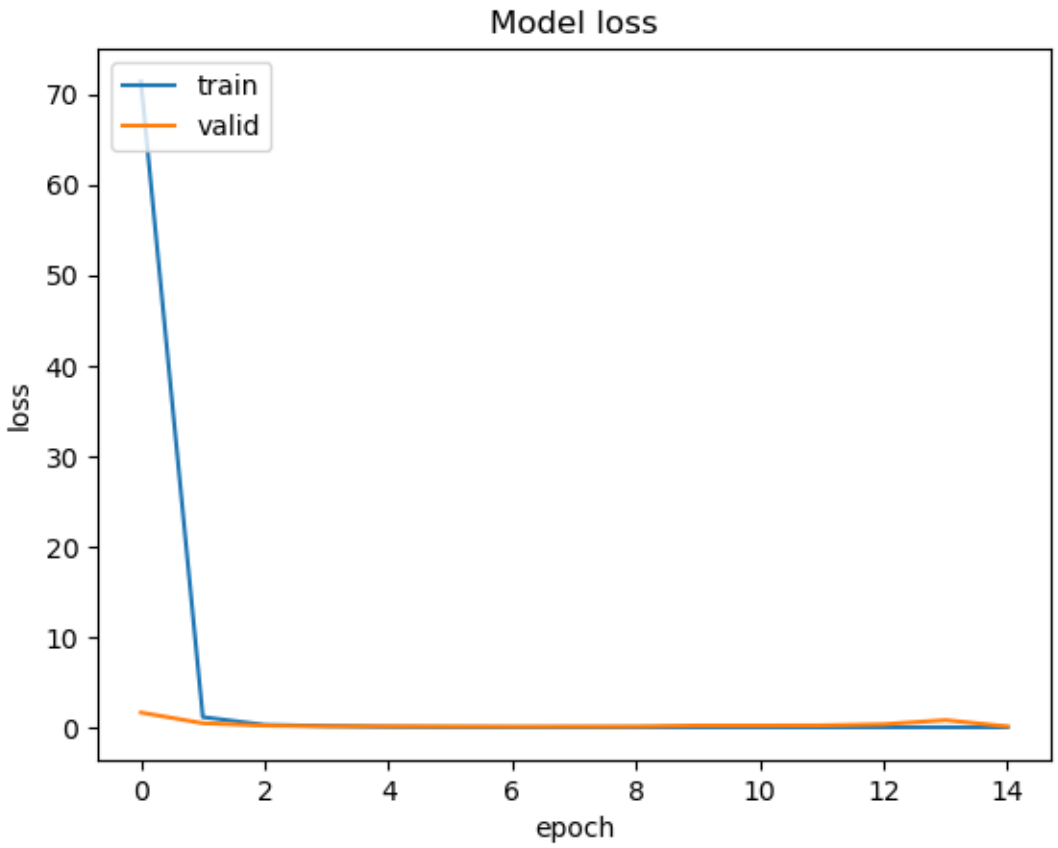
We can observe in these plots that the accuracy increases sharply after the first epoch, and has nearly reached 100% by the third. After that it stays steady around 100%.

We can observe the opposite for the loss, as it drops sharply after the first epoch and reaches nearly zero by the third, where it remains. As we would expect, the validation loss starts lower than the training loss (and validation accuracy starts higher than training accuracy), because the training loss has tuned the parameters of the model through backpropagation before the validation data is passed in.

[40]:
```
def display_history(history):
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'valid'], loc='upper left')
    plt.show()
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'valid'], loc='upper left')
    plt.show()
```

```
display_history(history)
```






### 1.0.9 Reflections

Overall I am satisfied with the performance of the model. The convergence behavior indicates that the model learns well, as the loss score drops steeply after the first epoch. There doesn't appear to be *severe* overfitting occuring, as we would observe a bigger difference between the performance of train and validation data, but ideally we would like to see no difference between them.

We achieved these results by tuning the hyperparameters of the model through trial and error and observing the results.

I tried allocating a larger proportion of data to the test set and the validation set, but observed the loss score increase and accuracy decrease, so I settled on the 20%/75%/25% split.

I tried increasing the number of filters applied in the conv2D layer, but found that the improvements were negligible combined with the increase in computational power needed to run the model, so I kept it at 32.

I also added a regularizer to the dense layers but did not achieve the desired results so I removed it.

I played around with the number of relu and softmax dense layers and finally settled on 64 and 10 as they produced the best results with the least amount of time/computational power.

I achieved higher accuracy and lower loss by increasing the number of epochs from 10 to 15. Any more than 15 took too much time to train.

The particular functions I chose, including relu, softmax, adam, and sparse categorical crossentropy loss were chosen for their applicability to a multi-class image classification problem, their efficiency, and their effectiveness.

Other actions that could improve the predictive power of the model include increasing the amount of input data, manipulating the images to introduce more variation in the input data, and continuing to tune the hyperparameters through trial and error to find the best results.