Convolutional Neural Network

Olivia Schultheis

In this project, I developed a Convolutional Neural Network (CNN) to predict whether pictures of hands are showing “rock”, “paper”, or “scissors” in a game of rock, paper, scissors. The dataset comes from ‘tensorflow\_datasets’, which I downloaded in Anaconda and imported into my code. The data was already split into a training and test set. I loaded in the data with the “tfds.load” function and specified the split to include the ‘train’ and ‘test’ data. The link to the Tensorflow documentation for the dataset is provided below:

<https://www.tensorflow.org/datasets/catalog/rock_paper_scissors>

I have provided some screenshots that give more insight into the structure of the data. Each image has a shape of 300 by 300, and the third argument of 3 in the shape corresponds to the fact that color is retained in the images. The images are of type ‘uint8’, which means the minimum value is 0, and the maximum value is 255. There are 372 images in the testing data set, and 2,520 images in the training data set. It will be confirmed below that there is an equal amount of ‘rock’, ‘paper’, and ‘scissors’ images in each set (372/3=124 of each in the testing data and 2520/3 = 840 of each in the training data).

A screenshot of a cell phone

Description automatically generated

A white rectangular object with a black stripe

Description automatically generated with medium confidence

First, the necessary packages are imported. We need to import tensorflow\_datasets as tfds to be able to load in the ‘rock\_paper\_scissors’ data.

import numpy as np   
import matplotlib.pyplot as plt  
import tensorflow as tf  
from tensorflow.keras import datasets, layers, models  
  
import tensorflow\_datasets as tfds

Since the data were already split into training and testing data, the data just needs to be loaded into two sets, with the ‘split’ in the ‘tfds.load’ specifying we want to bring in the training and testing data sets under the names ‘train\_rps’ and ‘test\_rps.’

train\_rps, test\_rps = tfds.load('rock\_paper\_scissors', split = ['train', 'test']) # split into training and test sets

Now that we have the training and testing data loaded, we need to further split into ‘X\_train, y\_train, X\_test, y\_test.’ The X values for each set will correspond to the image information, while the y values will correspond to the label, or classification, of each image. We need to do this by using the ‘map’ function with lambda parameters to be able to extract the ‘image’ and ‘label’ features from the data. Without breaking this down into a function for each subset, we are not able to select the ‘images’ and ‘labels’ from the tensorflow dataset. For X, we assign the images for both the training and testing data. For y, we assign the labels (rock, paper, or scissors) for both the training and testing data. Then, we need to make a list from the results of the mapping function so that information from each mapping output can be retained. There will be an output for the lambda function for each image and its label in the training and testing data for X and y. Finally, we can transform the results of the mapping within a list to a numpy array so that we can further investigate the shape of the image data.

X\_train = np.array(list(map(lambda x: x['image'], tfds.as\_numpy(train\_rps))))  
y\_train = np.array(list(map(lambda y: y['label'], tfds.as\_numpy(train\_rps))))  
X\_test = np.array(list(map(lambda i: i['image'], tfds.as\_numpy(test\_rps))))  
y\_test = np.array(list(map(lambda l: l['label'], tfds.as\_numpy(test\_rps))))

Evidently, the training data has 2,520 images that are of shape (300,300,3), which is exactly what is conveyed in the Tensorflow documentation.

X\_train.shape

# 2,520 images that have shape 300\*300 pixels. Final input is 3 because the images include color.

OUTPUT:

(2520, 300, 300, 3)

y\_train.shape

OUTPUT:

(2520,)

# Accordingly, y has 2,520 rows (1 corresponding to each image).

We already know that the testing data consists of fewer images than the training data, and looking at the shape of the testing data confirms that there are 372 images of shape (300, 300, 3).

X\_test.shape

#372 images in the test set. 300 by 300 pixels with color argument included.

OUTPUT:

(372, 300, 300, 3)

y\_test.shape

OUTPUT:

(372,) # 372 labels for the testing data

Next, we must standardize the images. First, we convert them to 32-bit floats and then we divide by 255, since the maximum pixel intensity is a value of 255. We do this for images both in the training and testing data.

X\_train = X\_train.astype('float32') / 255

#Convert to 32 bit floats and divide by 255 to standardize, since pixel intensity ranges from 0 to 255

X\_test = X\_test.astype('float32') / 255

The next part of the code will show an output of 36 of the images in the training dataset through a for loop. I assigned the rock, paper, and scissors strings to an object called ‘class\_labels’ so that I could print the hand motion underneath each image, along with the label assigned to that class of hand shape from the dataset.

class\_labels = ['rock', 'paper', 'scissors']

# In the data, 0 label is for rock, 1 is for paper, and 2 is for scissors  
  
"""" A for loop runs 36 times to print out a 6 by 6 plot of images in the training set, along with the class label  
that is assigned above and the respective index """  
  
plt.figure(figsize=(10,10))  
for i in range(36):  
 plt.subplot(6,6,i+1)  
 plt.xticks([])  
 plt.yticks([])  
 plt.grid(False)  
 plt.imshow(X\_train[i])  
 plt.xlabel((class\_labels[y\_train[i]],(y\_train[i])))  
plt.show()

![](data:image/png;base64;base64,)

We can confirm the data labels are correct by looking at an output of ‘y\_train.’

y\_train #Currently, y\_train simply shows the label corresponding to the hand-shape

Out[87]:

array([2, 2, 0, ..., 1, 1, 1])

# Confirmed that 2 corresponds to scissors, 0 to rock, and 1 to paper.

However, for our classification that will eventually use the ‘categorical\_crossentropy’ loss function, we will want the y-variable to be one-hot encoded, so that we have a numpy array that contains a 1 in the column corresponding to the class of hand-motion that is present in the image and 0 in all other columns of the array.

We can do this with the ‘to\_cateogorical’ function from tensorflow.keras.utils

from tensorflow.keras.utils import to\_categorical

#Use this to convert to one-hot encoded form for the response variable

y\_train = to\_categorical(y\_train)

y\_train.shape #2,520 images have 3 labels

OUTPUT:

(2520, 3)

y\_train[0]

# Now, y\_train shows the labels for each category, with a "1" in the place of the index that corresponds to the picture shape

OUTPUT:

array([0., 0., 1.], dtype=float32)

This suggests that the first training image has class 2, which is scissors.

y\_test = to\_categorical(y\_test) #Convert the testing data to one-hot encoded

y\_test.shape

(372, 3)

y\_test[0]

OUTPUT:

array([0., 0., 1.], dtype=float32)

We can see the testing data is also in the proper format now and that the first testing image is scissors as well.

The next chunk of code shows sum of each category of image-types present in the testing data.

print(y\_test.sum(axis=0)) # Even number of image-types in the test set.

[124. 124. 124.]

We can see that there are 124 of each hand motion in the testing data.

Doing this for the training data, we can see that there are 840 of each hand motion in the training data.

print(y\_train.sum(axis=0)) # Even number of image-types in the train set.

[840. 840. 840.]

Now, we can start to build the CNN.

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, Dense, Flatten, MaxPooling2D

CNN = Sequential() #Start building convolutional neural network

"""Start with 32 filters that are 3 by 3 matrices  
using relu activation. Take the maximum value from a 2 by 2 matrix.  
Then, add a layer with 64 filters that are 3 by 3 matrices using relu activation.  
Use another pooled layer that takes the maximum value from a 2 by 2 matrix.  
End with a final layer of 64 filters (3 by 3 matrices again).   
"""  
CNN.add(Conv2D(filters = 32, kernel\_size = (3,3), activation = 'relu', input\_shape = (300, 300, 3)))  
CNN.add(MaxPooling2D(pool\_size = (2,2)))  
CNN.add(Conv2D(filters = 64, kernel\_size = (3,3), activation = 'relu'))  
CNN.add(MaxPooling2D(pool\_size=(2, 2)))  
CNN.add(Conv2D(filters = 64, kernel\_size = (3,3), activation = 'relu'))

We use the relu activation function for all Conv2d and MaxPooling2D layers. This function prevents linearity since the parameters of the images are not linear. Each filter is a 3 by 3 matrix, and we start with 32 filters before doubling to 64 to try and discover more patterns. This section of code develops the convolutional base. The max pooling layers decrease the dimensionality of the output of the later and gets rid of certain features. In this code, since the pool\_size is (2,2) in the MaxPooling2D layer, 2 by 2 squares of features are analyzed one at a time and only the maximum feature is retained.

In [101]:

CNN.summary()

#Check the structure of the neural network. Width and height are decreasing over time.

Model: "sequential\_1"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 conv2d\_3 (Conv2D) (None, 298, 298, 32) 896   
   
 max\_pooling2d\_2 (MaxPooling (None, 149, 149, 32) 0   
 2D)   
   
 conv2d\_4 (Conv2D) (None, 147, 147, 64) 18496   
   
 max\_pooling2d\_3 (MaxPooling (None, 73, 73, 64) 0   
 2D)   
   
 conv2d\_5 (Conv2D) (None, 71, 71, 64) 36928   
   
=================================================================  
Total params: 56,320  
Trainable params: 56,320  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

We can see that the final convolutional layer has shape (71, 71, 64). This will now be passed into dense layers after the pooled feature map is flattened. After flattening, we have vectors of shape 71\*71\*64 = 322,624. In the dense layers, classification occurs. We will need the final dense layer to consist of 3 units, since we have 3 possible classifications of hand gestures. We will also need to use the ‘softmax’ activation, since we have multi-class classification with a one-hot encoded response. The activation function will assign probabilities to each of the 3 neurons, and the neuron with the highest probability will be the prediction for a hand-gesture image.

CNN.add(Flatten()) #Flatten the pooled feature map that can be passed to next (input) layer

CNN.add(Dense(units = 64, activation = 'relu')) # Dense input layer with relu activation

CNN.add(Dense(units = 3, activation = 'softmax')) #Output layer with softmax activation because this is multiclass classification with 3 classes

CNN.summary() # (71, 71, 64) outputs were flattened to shapes (322624) to become vectors passed through dense layers

Model: "sequential\_1"  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Layer (type) Output Shape Param #   
=================================================================  
 conv2d\_3 (Conv2D) (None, 298, 298, 32) 896   
   
 max\_pooling2d\_2 (MaxPooling (None, 149, 149, 32) 0   
 2D)   
   
 conv2d\_4 (Conv2D) (None, 147, 147, 64) 18496   
   
 max\_pooling2d\_3 (MaxPooling (None, 73, 73, 64) 0   
 2D)   
   
 conv2d\_5 (Conv2D) (None, 71, 71, 64) 36928   
   
 flatten\_1 (Flatten) (None, 322624) 0   
   
 dense\_2 (Dense) (None, 64) 20648000   
   
 dense\_3 (Dense) (None, 3) 195   
   
=================================================================  
Total params: 20,704,515  
Trainable params: 20,704,515  
Non-trainable params: 0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

We now need to compile the model . We will use the ‘adam’ optimizer with categorical\_cross entropy loss for our multi-class classification, returning accuracy as our metric of interest.

CNN.compile(optimizer = 'adam', loss = 'categorical\_crossentropy', metrics = ['accuracy'])

In [107]:

history = CNN.fit(X\_train, y\_train, epochs = 5, batch\_size = 64, validation\_split = 0.10)

Epoch 1/5  
36/36 [==============================] - 77s 2s/step - loss: 1.5019 - accuracy: 0.6327 - val\_loss: 0.2750 - val\_accuracy: 0.9167  
Epoch 2/5  
36/36 [==============================] - 82s 2s/step - loss: 0.1267 - accuracy: 0.9612 - val\_loss: 0.0116 - val\_accuracy: 1.0000  
Epoch 3/5  
36/36 [==============================] - 84s 2s/step - loss: 0.0123 - accuracy: 0.9969 - val\_loss: 0.0073 - val\_accuracy: 1.0000  
Epoch 4/5  
36/36 [==============================] - 84s 2s/step - loss: 0.0028 - accuracy: 1.0000 - val\_loss: 0.0089 - val\_accuracy: 0.9960  
Epoch 5/5  
36/36 [==============================] - 85s 2s/step - loss: 7.3994e-04 - accuracy: 1.0000 - val\_loss: 0.0084 - val\_accuracy: 0.9960

# Plot training and validation accuracy

plt.plot(history.history['accuracy'], label='accuracy')  
plt.plot(history.history['val\_accuracy'], label = 'val\_accuracy')  
plt.title('Model Accuracy')  
plt.xlabel('Epoch')  
plt.ylabel('Accuracy')  
plt.legend(loc='lower right')  
plt.xlim([0,6])  
plt.ylim([0.5, 1])  
plt.show()  
  
test\_loss, test\_acc = CNN.evaluate(X\_test, y\_test, verbose=2)  
  
# We see increasing accuracy and validation accuracy that approaches 1.0.   
# However, this is likely overfitting, as overall accuracy is only 0.7016.

![](data:image/png;base64;base64,)

12/12 - 3s - loss: 0.8933 - accuracy: 0.7688 - 3s/epoch - 287ms/step

The testing accuracy of the model is about 0.77, but the training and validation accuracy are higher. We can see that the training and validation accuracy increase to 1 quickly, only within a few epochs. Even though the training accuracy begins fairly low, the model quickly picks up on the training data, which could lead to overfitting. While the training accuracy stays at 1.00 after 4 epochs, the validation accuracy decreases and the loss increases very minimally, still hovering at 1.00. The model seems to be memorizing the training data.

We can also look at a plot for the loss.

plt.plot(history.history['loss'], label = "loss")  
plt.plot(history.history['val\_loss'],label = 'val\_loss')  
plt.title('Loss')  
plt.ylabel('Loss')  
plt.xlabel('Epoch')  
plt.legend(loc = "lower right")  
plt.xlim([0,6])  
plt.show()  
  
Both validation loss and training loss decrease over time, but training loss starts higher than validation loss.

![](data:image/png;base64;base64,)

Like what was observed in the above plot, the validation set seems to enjoy a lower loss than the training data, just as the validation set started with a higher accuracy than the training accuracy. Both losses are quickly minimized by the 2nd epoch, and the model seems to pick up on trends in both sets quickly.

print(test\_loss) #Testing loss is about 0.80

0.8933207392692566

print(test\_acc) #Testing accuracy is about 0.77.

0.7688171863555908

We can allow the model to make predictions for image classification based on the testing data with the “predict” method.

predictions = CNN.predict(X\_test) #Make predictions by passing X test values through the CNN.

12/12 [==============================] - 3s 231ms/step

By checking the value in the index corresponding to 0 (first image present) in the predictions array, we can see that the model predicts the first testing image to be scissors, which is correct. The model makes this prediction because the 2nd class has the highest probability resulting from the soft max activation.

predictions[0] # The model predicts the highest probability that the first testing image is 2 (scissors).

array([0.01306122, 0.14219832, 0.84474045], dtype=float32)

y\_test[0] # It does indeed have a 1 in the categorical encoding for class 2, so the model predicts the first image correctly.

array([0., 0., 1.], dtype=float32)

""""To investigate incorrect predictions, we can set "test\_images" to   
equal the arrays for the image testing data. Then, we can use a "for"  
loop to iterate over the predicted values and the actual class for the testing  
set. We keep track of the class that had the highest probability in the   
"predict" output, as well as the highest input in the y\_test array, which would  
be a one because the array for y\_test would contain a 1 for the class that the image belongs to. A correct prediction would have the same maximum  
value in the prediction output and the y\_test output. If these are not equal,  
then we can add them to an array called "incorrect\_predictions", entering the index,  
image, predicted class, and expected class into this array, which can later  
be plotted."""  
  
  
test\_images = X\_test  
incorrect\_predictions = []  
  
for i, (predicted, expected) in enumerate(zip(predictions, y\_test)):  
 predicted, expected = np.argmax(predicted), np.argmax(expected)  
   
 if predicted != expected:  
 incorrect\_predictions.append((i, test\_images[i], predicted, expected))

len(incorrect\_predictions) # 86 of the test images are incorrectly classifed

86

"""Plot incorrect classifications with their index,  
predicted class by the model, and actual class from the raw data.  
0 = rock, 1 = paper, 2 = scissors  
"""  
figure, axes = plt.subplots(nrows=5, ncols=5, figsize=(20,20))  
  
for axes, expected in zip(axes.ravel(), incorrect\_predictions):  
 index, image, predicted, expected = expected  
 axes.imshow(image)  
 axes.set\_xticks([])   
 axes.set\_yticks([])   
 axes.set\_title(f'index: {index}\np: {predicted}; e: {expected}')  
A collage of hands with different gestures

Description automatically generatedplt.tight\_layout()]

The above code printed out 25 images with their index in the testing data, the predicted class, and the expected (or actual class). At a quick glance, it seems the model struggles at accurately predicting scissors.

The following code will determine the number of each type of image that were incorrectly classified. The list(zip(\*incorrect\_predictions)) code makes the list of incorrect\_predictions (which consists of index, image, predicted class, and expected class) into an unpackable object. It allows an index to be specified in the new list to pull out the group of values corresponding to that item. To explain further, we assign expectedcount to list(zip(\*incorrect\_predictions)) to unpack the list and assign an index to each group of values in the list. Then, expectedcount[3] is used to pull out all the ‘expected’ or actual values in the testing data corresponding to images that were incorrectly classified.

If we took expectedcount[0], we would get the index of each mis-classified image (first element of incorrect\_predictions), and expectedcount[1] would yield the arrays conveying information about the images, which are 0-1 values resulting after the original unit8 format with values from 0-255 were standardized. Expectedcount[2] would give the incorrect predictions made by the model.

import pandas as pd  
  
expectedcount = list(zip(\*incorrect\_predictions)) #"Transpose" of list  
  
exp = expectedcount[3] # Gets the expected values out of the list  
  
print(exp.count(0)) # Counts number of images that were supposed to be predicted as rock  
print(exp.count(1)) # Counts number of images that were supposed to be predicted as paper  
print(exp.count(2)) # Counts number of images that were supposed to be predicted as scissors

25  
4  
57

These values mean that 25 rock images were incorrectly classified, 4 paper images were incorrectly classified, and 57 scissors images were incorrectly classified. The model seems to be doing better at classifying paper and may be getting rock and scissors confused.

We saw above there are an even number of rock, paper, and scissors images in the training and test sets.   
Some of the "rock" images seem to have patterns where the knuckles are not completely closed, causing incorrect predictions of scissors to be made.   
The model could likely be improved if more pictures distinguished better between rock and scissors, such as if the hands were turned around so that the palm shape may not be as misleading.

To conclude, a Convolutional Neural Network resulted in a testing accuracy of about 77%, with higher training and validation accuracy being observed. The model seems to struggle with classifying between scissors and rock. The model incorrectly classified 86 / 372 = 23.12% of the testing images. A future analysis could look into the issue of overfitting and different techniques, such as hyperparameter tuning, to address this issue of the model memorizing the data.