Convolutional Neural Net Assignment

Artificial Intelligence – Spring 2021

Thomas Trankle

April 1, 2021

Summary

This report describes the development of a convolutional artificial neural network to be used as a classifier for predicting one hundred different classes within the CIFAR-100 data set. The background and category list of the images can be found here, but were omitted from this document for simplification. Overall, the best model that was trained had an accuracy of 0.63, which was primarily due to an additional Dropout layer added before the fully connected layer. The introduction describes the modifiable parameters of this convolutional neural network that is built using a Sequential model. Additionally, it describes some basic characteristics about the image input and the features/columns transformations used to predict the given classes. The body of the report steps through necessary preprocessing procedures for the model and data, as well as the main adjustments made for developing a base model, and then using additional layers to fine-tune the model.

Introduction

The purpose of this report is to provide a detailed analysis of a convolutional neural network, and the layer architecture within, to predict labels for images representing captured objects – which we refer to as classes. The data set we are using is hosted by *keras* and is called CIFAR-100. This data comes pre-split into training and testing sets. Therefore, it is unnecessary for us to partition the data after importing, which eliminates a common preprocessing step. The data consists of 5000 and 1000 images for the train and test sets respectively. Each image has the shape 32x32x3. This means that each image is 32 wide, 32 high and has a 3 color channels (typically RGB). Ideally, we hope to accurately predict every test label based on the test features. So, the overall goal is to maximize the prediction accuracy of our model, that is the number of correctly predicted labels over the total number of attempted predictions.

$$Accuracy = \frac{TruePredictions}{TruePredictions + FalsePredictions}$$

To do this, we will be using Categorical Cross Entropy as the loss metric, since it is a very intuitive and common metric when evaluating the performance of a multi-class classification model. The formula is defined as follows,

$$CE = -\,log\left(\frac{e^{s_p}}{\sum_j^C e^{s_j}}\right)$$

Additionally, we will be using Softmax as our activation function for my output layer since this is considered best practice for models with our criteria – multi-class. As we know, there is no procedural method to define the specific parameters of our model to optimize our objective function. Instead, we utilize experimentation to identify different parameters that will, at least on average, produce sufficient results. However, we do utilize a Randomized Grid Search approach to try and best select the hyper-parameters to optimize parts of the model. For the purpose of this model and due to memory constraints, we only used grid search on the fully connected part of our network while freezing the convolutional base layer. The parameters considered were:

- Optimizer
- One or Two Fully Connected Layers
- Number of Nodes per Layer
- Activation Function of Fully Connected Layer

There are certainly more details and/or parameters that could be added to the above list, however, for the purpose of this exercise they were omitted due to hardware constraints. Additionally, a couple of common best practices and preprocessing steps were held constant when designing, tuning, and evaluating the model.

Model Development and Parameter Tuning

The reason we are implementing a convolutional neural network is because this type of network is extremely good at extracting features from an image. This, in turn, helps identify and characterize features (image) to predict a label. Convents with image classification consist of two parts:

1. Convolutional Base Layer
2. Densely or Fully Connected Layer

In essence, feature extraction occurs in the *convolutional base* and then the *fully connected layers* can use these features to train, much like a regular classifier would. It is noted that, in the convolutional base layer, the level of generality of extracted feature representations is dependent on the depth of the layer in the model. Generally, layers at the beginning of the model are used to extract highly generic features whereas layers at the bottom are used to extract more abstract representations (Chollet, 2018). This is an important quality of convolutional layers which becomes apparent later. Additionally, since neural networks make use of this kind of approach, one must make sure the inputted data is normalized.
However, since convolutional networks use backwards propagation, that is the model gets updated after each layer starting at the output and moving backwards to the input, to assume the weights of the previous layers – it is essentially always chasing a moving target. To counter this phenomenon, a common technique suggested by literature is to use batch normalization. This helps to adaptively normalize the data even as the mean and variance change/update as the model trains. Essentially, it operates by internally maintaining an exponential moving average of the bath-wise mean and variance for the data seen (Chollet, 2018). This helps coordinate the gradient propagation by stabilizing the outputs of each layer. For this reason, we generally place a BatchNormilzation layer after each convolutional layer since convents are usually deep networks to maximize feature extraction.

Furthermore, we make sure to add pooling after each convolutional layer. The purpose of this is to is reduce the size of the filter layer above. This helps decrease the run-time of the model and can help prevent overfitting by removing unwanted values from the filter tensor created by the convolutional filter. The simplest implementation of this is to use MaxPooling (Chollet, 2018). This layer allows us to use an extracting window and take the maximum value from the input feature maps of each channel. This is extremely important because if we did not down sample then the size of our model would rapidly increase which is not feasible. So, by using this method we can reduce the number of feature-map coefficients to process, as well as introduce special hierarchies that allow successive convolutional layers to look at increasingly large windows (Chollet, 2018).

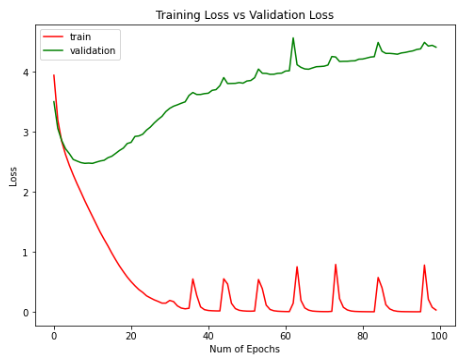
Preprocessing

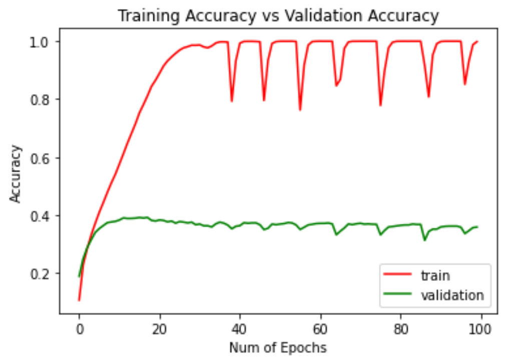
The imported feature data (what we use to train our model) is represented by images so the first step is to reshape and scale the feature sets for the train and test sets. This is easily achieved by using *numpy* to transform the images into matrix representation and then divide each value by 255, which is the maximum number possible in each of the 3 channels. Additionally, we make sure to change the data type of every value held in the matrix to a floating-point representation.

Next, we one-hot encode our Y sets so that the model can predict probabilities of class allocation that line up with the respective class index allocation.
Lastly, during certain experiments and model evaluations, data augmentation was utilized. However, this will be appropriately explained at a later stage.

Course of Action and Model Development

After all the preprocessing steps had been implemented, I created an initial model with three convolutional layers and only one fully connected layer as suggested in the prompt. This was done to make sure all the data passed through in the correct format and to determine a performance baseline. The number of filters for the three convolutional layers were 32, 64 and 128 respectively while the single fully connected had 512 nodes. Each of the above nodes used ReLu as the activation function and the batch size used was 128. This model had an accuracy of 0.366 and a run-time of 7 minutes and 9 seconds.



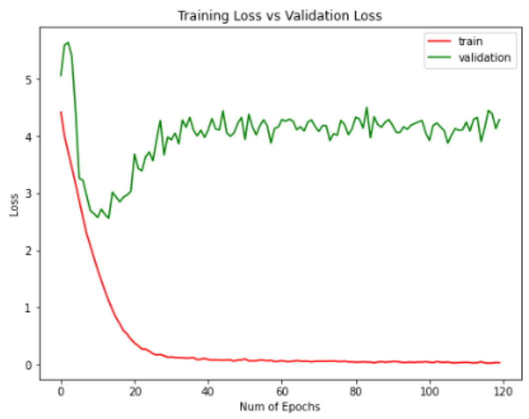


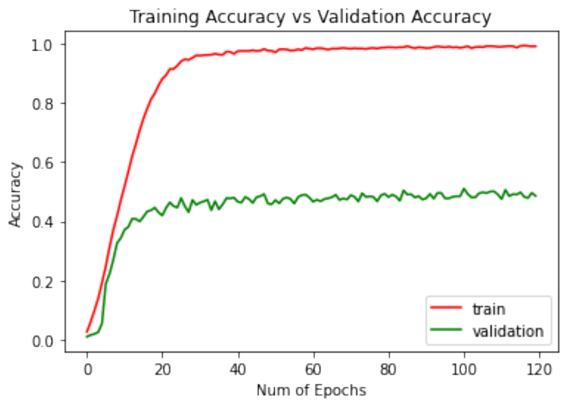
I then decided to increase the number of filters in my convolutional layers because we are dealing with a reasonably small data set with which we hope it classify 100 classes. This meant that I needed to let the model extract more features. This decision, resulted in my base convolutional model, which differs from the above mentioned by having more inputs and layers as seen in the model summary below.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 64)        1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 64)        36928
_________________________________________________________________
batch_normalization (BatchNo (None, 32, 32, 64)        256
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 128)       147584
_________________________________________________________________
batch_normalization_1 (Batch (None, 16, 16, 128)       512
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 256)         590080
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 256)         590080
_________________________________________________________________
batch_normalization_2 (Batch (None, 8, 8, 256)         1024
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 256)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 4, 4, 512)         1180160
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 4, 4, 512)         2359808
```

```
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 4, 4, 512)         2359808
_________________________________________________________________
batch_normalization_3 (Batch (None, 4, 4, 512)         2048
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 2, 2, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dropout (Dropout)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1049088
_________________________________________________________________
dense_1 (Dense)              (None, 100)               51300
_________________________________________________________________
predictions (Activation)     (None, 100)               0
=================================================================
Total params: 8,739,492
Trainable params: 8,737,572
Non-trainable params: 1,920
```

This deeper network had an accuracy of 0.4856, however the time required for training was 36 minutes and 14 seconds. So, the model takes longer to train however it is now able to generalize better. As you can see, this model still allows the training accuracy to achieve approximately 100% but we the overfitting in the model is less drastic but still apparent.





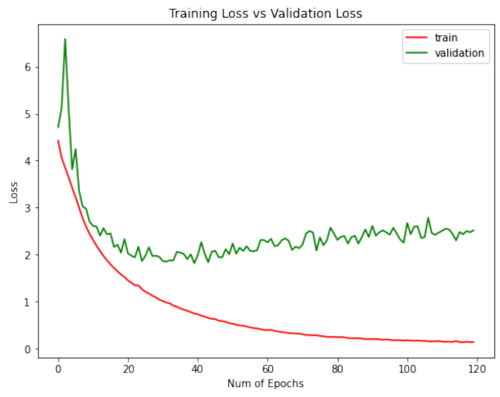
At this stage I decided to use Randomized Grid Search to determine if some of the parameters in my fully connected layer needed to be modified. The results were as follows,

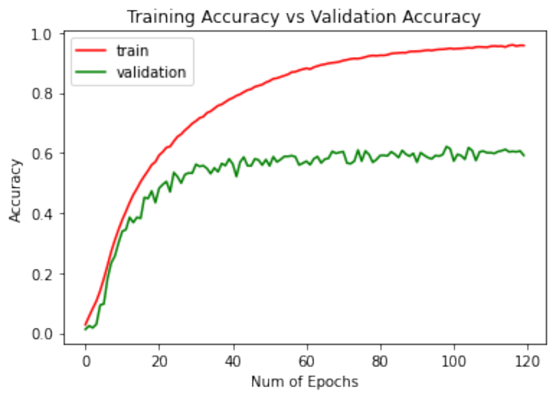
Best: 0.250360 using {'optimizer': 'Adam', 'neurons_second': 512, 'activation': 'sigmoid'}
0.016640 (0.002269) with: {'optimizer': 'SGD', 'neurons_second': 512, 'activation': 'relu'}
0.007440 (0.000450) with: {'optimizer': 'Adamax', 'neurons_ second ': 0, 'activation': 'hard_sigmoid'}
0.184600 (0.030524) with: {'optimizer': 'Adamax', 'neurons_ second ': 256, 'activation': 'linear'}
0.100480 (0.018786) with: {'optimizer': 'Adagrad', 'neurons_ second ': 128, 'activation': 'linear'}
0.007360 (0.000372) with: {'optimizer': 'SGD', 'neurons_ second ': 0, 'activation': 'linear'}

```
0.164700 (0.019298) with: {'optimizer': 'Adam', 'neurons_ second ': 128, 'activation': 'hard_sigmoid'}
0.250360 (0.036097) with: {'optimizer': 'Adam', 'neurons_ second ': 512, 'activation': 'sigmoid'}
0.007360 (0.000372) with: {'optimizer': 'Adagrad', 'neurons_ second ': 0, 'activation': 'relu'}
0.041180 (0.004696) with: {'optimizer': 'Adadelta', 'neurons_ second ': 256, 'activation': 'linear'}
0.091820 (0.014715) with: {'optimizer': 'RMSprop', 'neurons_ second ': 256, 'activation': 'linear'}
Time required for training: 0:47:51.886454
```

The reason for such small accuracy is because of small epochs. This is due to my kernel crashing and memory constraints. So, I decided to perform the search with only 20 epochs as this would still provide insight on how the model performs. The results of the search indicate that the only thing I needed to change was my activation functions and add a second fully connected layer. Rerunning the above model with the altered parameters only led to a small increase in training accuracy to .51 percent with a similar run time. Therefore, I decided to omit the added activation layer since it creates a more complicated model with no added value.

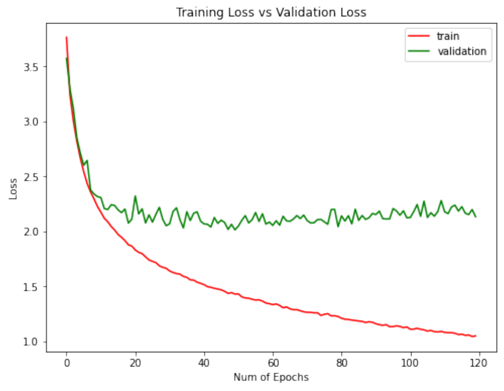
Since our model still suffers severely from overfitting, we need to investigate ways to create a model with less variance between training and testing. One of the ways overfitting occurs is when we do not have enough samples to train our model on. The results in our inability to train a model that can generalize well. Since we have 100 classes but only 5000 samples, it is not surprising that our model is overfitting. So, to try remedy this predicament, we will use data augmentation to generate more training data from the existing training data we have. In simple terms, this can be thought of as feature expansion in Machine Learning. This is achieved by augmenting the images we have via several random transformations. Keras has an Image Generator that can help us with this implementation (Chollet, 2018). Adding data augmentation and rerunning the model we see the following results.

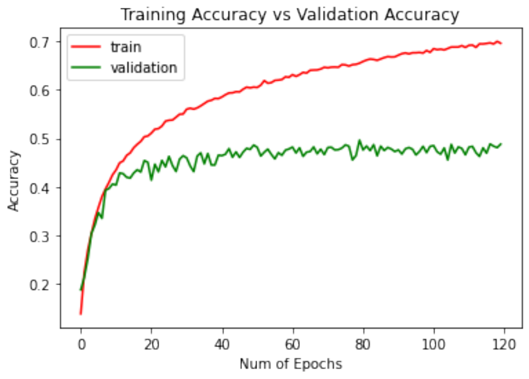




After a training time of 50 minutes and 16 seconds we now have test accuracy of 0.592, which is a dramatic improvement of the model. By adding more data, we have been able to reduce the variance between the training and test set so that our model generalizes better. Unlike before, we see an increase in training time, however this is more than acceptable since we have improved the performance of the model.

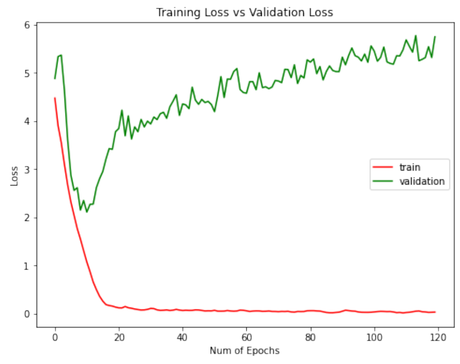
As I mentioned above, it is important that convolutional layers can extract more abstract features as the model gets deeper. If we simply let it extract general upper level features, then our model will not be able to learn well. For example, I will now take the same model from above, however I am going to remove all convolutional layers except for one. The filters of the convolutional layer in this block is 256. Running the model resulted in the following,

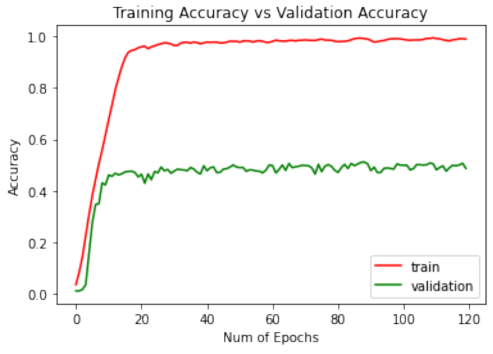




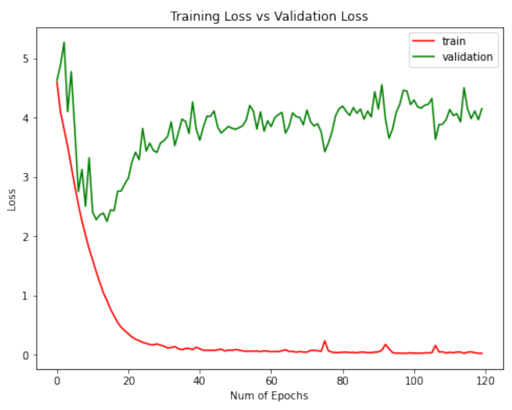
Unsurprisingly, we see that neither our training nor our validation accuracy climbed as high as the above example. The model took 1 hour and 7 minutes to run but only yielded a training accuracy of 0.48. In fact, we look more closely as the performance of our model, we notice that the validation accuracy stays around that percentage for most of the training cycle. This is because our model is unable to distinguish further between simples since we have limited its extraction capabilities.

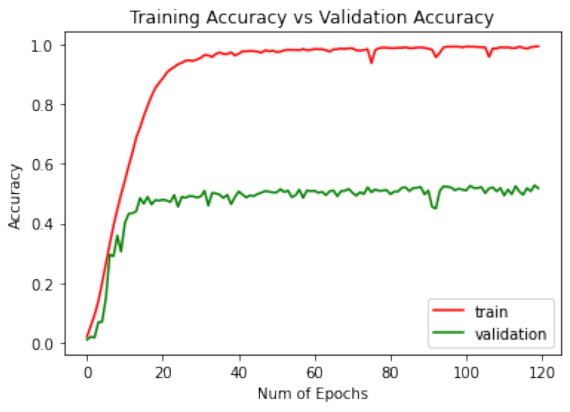
However, what if we decide to add more convolutional layers so we can maybe extract and more abstract features and further aid our model's ability to differentiate training features and generalize better. To see this, I am going to add another convolutional block with each layer having 512 filters respectively. The results from this test gave us a 0.4875 accuracy and took 38 minutes to 17 seconds to run. This result is very similar to our base model and it is probably because our model can no longer extract abstract features dramatically more than it could in the base case.





Another option is for us to add a dropout layer after flattening extracted features from our convolutional base layers before passing it to the fully connected layers. This step randomly ignores some of the nodes and reduces the size of the network. In essence, this form of regularization helps prevent spoon feeding amongst certain neurons and rather creates a more robust model as nodes are forced to learn from different subsets of each other. This technique is supposed to help with overfitting. Adding this to our model we see,

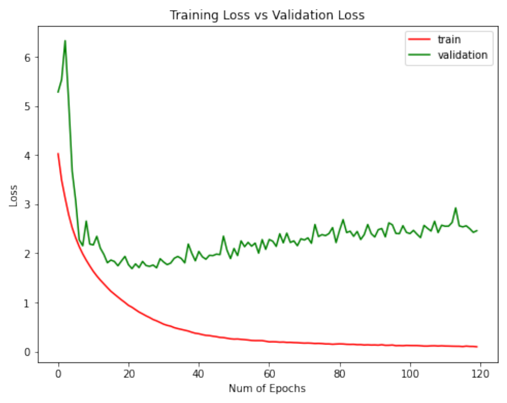


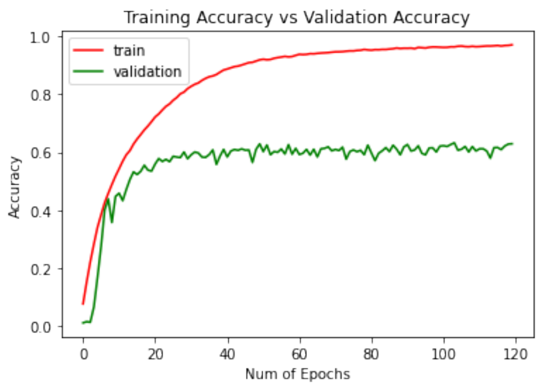


This model took 15 minutes and 40 seconds to run and resulted in a test accuracy of 0.53. Therefore, dropout not only speeds up the model but can help create a more generalized model.

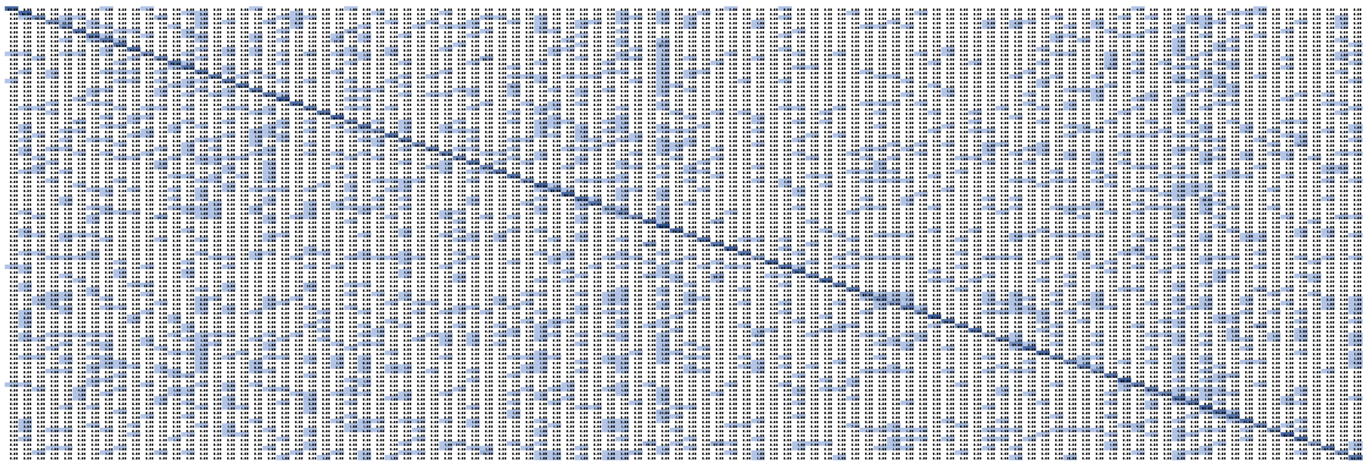
Conclusion

Convolutional neural networks are excellent ways to make prediction (classification) based on images. This is because the convolutional base layer can extract various levels of features to help distinguish classes when making predictions. Throughout this experiment, we looked at many types of layers and preprocessing (data augmentation) that could be added to a convolutional neural network and the impacts of either including or excluding these layers. For this experiment, we saw that adding data augmentation was the most beneficial followed by dropout and batch normalization. We also notice the impact of not creating a deep enough model so that the convolutional layer can extract enough abstract features to help prediction accuracy and generalization. We looked at how these alterations impacted our base model, but, the best approach is to use a combination of all these various layers and methods. For example, taking the base model and using both data augmentation and dropout results in a training accuracy of 0.63 with a running time of 48 minutes. It is interesting that this combination decreased the training time (benefit of dropout) and increased our accuracy.





Confusion Matrix of 100 categories. My best attempt to at least use color as an indicator.



Base Model with Data Augmentation and Dropout

```
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 24 19:24:57 2021

@author: ttran
"""

# import appropriate packages for input, preprocessing, model creation and
# training, and output
import datetime
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Dropout,Flatten,Conv2D,MaxPooling2D,\
                           BatchNormalization, Activation
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.datasets import cifar100

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
GPU Configuration for OOM
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
from tensorflow.compat.v1 import ConfigProto,Session
config = ConfigProto()
config.gpu_options.allow_growth = True
sess = Session(config=config)
sess.as_default()

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
                    Code Flow
The first step in define some of the hyper parameters used in the model:
   batch_size
   num_classes
   epochs
   optimizer
   activation

Next we import the relevant data and assign it to our train and test variables.
   x_train, y_train, x_test and y_test

Next, we display some of the images in the data set as examples we are trying
to predict.

The next section we begin our preprocessing procedure. This is where we
standardize our feature input and one hot encode our categorical outputs.
Additionally, we place the construction of the ImageGenerator, which is used
for data augmentation, in this section since we will be fitting it on our data
and then creating batches thereafter. Since this is done on the data, it is
considered a preprocess step. The train_generator is an object that constitues
our training samples and additional augmented samples of our training set.

The next section is where define the model we want to use.
This model consists of two parts. The first part is the convolutional
base model and the second part is the fully connected layers.
   - Convnet portion is used for feature extraction
   - Full Connected is used to train classifier portion of the network

After this, we fit our model which basically means we allow it to train
on the training data using the training parameters we have either constructed
or passed thus far.

Lastly, a very important step is to visualize the performance of the model
so that we can evaluate it and look for issues like overfitting, underfitting
or perhaps even skewness of inputs. This section is how we determine how
well our model is performing and how well it can generalize.
```

```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""


""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Parameters Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# Define the batch size. This variable defines the size of the training batch
# per epoch (forward and backward pass)
batch_size = 128
# Define the number of classification labels. Used in final dense layer
num_classes = 100
# Defien the number of epochs you wish to use for training. Defines the number
# of training cycles through the full data set
epochs = 120        # Defien the number of epochs you wish to use for training
# Select the appropriate optimizer. Used to reduce losses
optimizer = 'adam'
# Select the activation function for fully connected layer. Activation
# functions determine the outpout to the next neuron
activation = 'sigmoid'

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Load Data Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
'''
Loading the CIFAR-100 datasets and assign them to the appropriate variables
    x_train = variable to hold 5000 training images with a 32x32x3 shape
        (50000, 32, 32, 3)
    y_train = variable to hold 5000 training images labels with 100 classes
        (50000, 100)
    x_test  = variable to hold 1000 training images with a 32x32x3 shape
        (10000, 32, 32, 3)
    Y_test  = variable to hold 1000 training images labels with 100 classes
        (10000, 100)
'''
# Import data by accessing the cifar100 dataset from keras.
# Used to train and test our model.
(x_train, y_train), (x_test, y_test) = cifar100.load_data()

# Print figure with 10 random images as examples of what we are trying to
# classify
fig = plt.figure(figsize=(8,3))
for i in range(10):
    ax = fig.add_subplot(2, 5, 1 + i, xticks=[], yticks=[])
    idx = np.where(y_train[:]==i)[0]
    features_idx = x_train[idx,::]
    img_num = np.random.randint(features_idx.shape[0])
    im = np.transpose(features_idx[img_num,::],(0,1,2))
    #ax.set_title(class_names[i])
    plt.imshow(im)
plt.show()
```

```
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Pretreat Data Section
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

# Convert values to float and rescale pixel values by diving by 255
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# One-hot encode output lables using keras to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

'''
Construct a image generator to perform data augmentation
     - rotation_range specifies the degree range for random rotations
     - width_shift_range is the fraction of total width to shift the image by
     - height_shift_range is the fraction of total height to shift image by
     - horizontal_flip is a boolean to allow randomly flip inputs horizontally.
'''
train_datagen =  ImageDataGenerator(
    rotation_range=15, width_shift_range=0.1, height_shift_range=0.1,
    horizontal_flip=True)
'''
Now we fit the generator to the training data we are going to use it on
This computes the internal data stats related to the data-dependent
transformations, based on an array of training data.

After that we call the flow method to take data & label arrays and
generates batches of augmented data.
'''
train_datagen.fit(x_train)
train_generator = train_datagen.flow(x_train,y_train, batch_size= batch_size)

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Define Model Section
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# begin stopwatch to record training time of model
start_time = datetime.datetime.now()
def base_model():
    # Construct a sequential model. Can add layers for appropriate depth
    model = Sequential()
    ############### Begin Ddefining Convolutional Base Layer(s) ######
    '''
    BatchNormilization is used to standardize the mean and variance of values
    as the model trains. Used per convolutional block.
    MaxPooling selects the maximum values from its defined 2x2 window in order
    to reduce the amount of features being generated so that we can still
    be able to train and predict model without reaching infeasibility
    '''
```

```
'''
#block1
# Add a convolutional layer with 64 filters, 3x3 window, same sided
# padding and ReLu activation function. Used for general feature extraction
'''
model.add(Conv2D(64, (3, 3), padding='same',  activation='relu',
                input_shape=(32, 32, 3)))
model.add(Conv2D(64, (3, 3), padding='same',  activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
'''
#block2
# Add a convolutional layer with 128 filters, 3x3 window, same sided
# padding and ReLu activation function. Used for less general feature
# extraction'''
model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
'''
#block3
# Add a convolutional layer with 256 filters, 3x3 window, same sided
# padding and ReLu activation function. Used for abstract
# feature selection '''
model.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(256, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

'''
# block4
# Add a convolutional layer with 512 filters, 3x3 window, same sided
# padding and ReLu activation function. Used for more abstract feature
# extraction'''
model.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(512, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))


################### Begin Defining Fully Connected Layer(s) ##########
# flattens multi-dimensional tensor into a single dimensional tensor
model.add(Flatten())
# Randomly select 50% of nodes to not fire
model.add(Dropout(0.5))
# Fully connected layer used to
model.add(Dense(512, activation=activation))
```

```
    model.add(Dense(100))
    model.add(Activation('softmax', dtype='float32', name='predictions'))

    # configures the model for training with defined loss (see paper) and opt
    model.compile(loss='categorical_crossentropy', optimizer=optimizer,
              metrics=['accuracy'])
    return model

# initialize convolutional base model
cnn_n = base_model()
# output summary of convolutional base model
cnn_n.summary()

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Train Model Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

# Trains the model for a fixed number of epochs (iterations on a dataset).
cnn = cnn_n.fit(train_generator, epochs=epochs,
            validation_data=(x_test,y_test),steps_per_epoch=len(x_train)/batch_size)

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Show output Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
'''
Plots for training and testing process: loss and accuracy
'''
# Training Loss and Validation Loss
plt.figure(0)
plt.plot(cnn.history['accuracy'],'r')
plt.plot(cnn.history['val_accuracy'],'g')
plt.xticks(np.arange(0, epochs+1, 20.0))
plt.rcParams['figure.figsize'] = (8, 6)
plt.xlabel("Num of Epochs")
plt.ylabel("Accuracy")
plt.title("Training Accuracy vs Validation Accuracy")
plt.legend(['train','validation'])

# Training Accuracy and Validation Accuracy
plt.figure(1)
plt.plot(cnn.history['loss'],'r')
plt.plot(cnn.history['val_loss'],'g')
plt.xticks(np.arange(0, epochs+1, 20.0))
plt.rcParams['figure.figsize'] = (8, 6)
plt.xlabel("Num of Epochs")
plt.ylabel("Loss")
plt.title("Training Loss vs Validation Loss")
plt.legend(['train','validation'])
plt.show()
```

```
print('Accuracy: ', cnn_n.evaluate(x_test,y_test,verbose=0)[1])
# stop stopwatch so we can see how long the model took to train
stop_time = datetime.datetime.now()
# Print the time taken to train the model
print ("Time required for training:",stop_time - start_time)
```

References

Chollet, F. (2018). *Deep Learning with Python.* Shelter Island, NY: Manning Publications Co.