Kaggle Challenge Report
Abhishank Gaba
20481729
Kaggle Username: abhishankgaba2
Google Collab Link

https://colab.research.google.com/drive/1tDSTbBsaV3aypK7N1_VseubrJMB0sSkO

Attempt 1: Retraining Prebuilt VGG Model

In the first attempt, I tried to retrain the vgg model given on the keras applications website, however I kept getting an accuracy of 20% to 30% when I applied it to my training dataset regardless of the number of epochs I used. During this approach I froze all the layers of the vgg model, and then added several dense functions, to fully connect the network. I then performed model.fit to fit the vgg model to my datasets (training & training label). Despite this, the accuracy was far to low to continue.

Attempt 2: Built My Own Network

I then decided to build my own network, taking the basic framework from the CNN tutorial of this course. I first loaded the training, test, and train label datasets using numpy load.

I then determined how many unique classes there were by performing np.unique on the train label dataset. The labels were then stored as tensor flow readable labels in a numpy array. The Train dataset was also sliced into a numpy array. Both these datasets (training and training label) were augmented into 1 dataset of batch size = 32.

Batch size determines how many samples are processed at a time. Therefore, a batch size of 32 will take the first 32 samples and then will update the neural network based on those 32 samples. It will then take the next 32 samples and then update the neural network on that dataset, and so on.

The dataset was repeated 1000 times because the original dataset was way too small, and thus with higher epochs, the model would run out of data to train with unless repeated.

The convolutional 2D model was used because it extracts features on each layer. The convolutional neural network is made up of 3 parts: convolution, polling, and flattening. Convolution extracts features from input and preserves spatial relationships between the pixels. It does this by learning features by splitting the input data into small squares. The output matrix is a feature map. The Relu function is used after every convolution operation.

Pooling is performed to down sample the data and reduce the dimensionality of each feature map while keeping all the most important features. Max pooling is used to take the largest element within a neighborhood window.

Flattening is then used to convert the matrix into a linear array that will be input into the nodes of the neural network.

The add and dense operations are then performed to fully connect the convolutional network to a neural network and then compiling the results.

Actual Implementation

Kaggle Challenge Report
Abhishank Gaba
20481729
Kaggle Username: abhishankgaba2
Looking at the actual implementation, a Keras sequential model was created. 3 layers of 2 convolutions directly followed by a max pool were then applied in a sequential order. The convolution 2D model was used.

The number of filters that the convolutional layer will learn were increased from 1 layer to another. Layers closer to the input learned fewer convolutional filters while layers deeper into the network/closer to the output learned more filters. Layer 1 had a total of 32 filters, directly followed by max pooling to reduce spatial dimensions of the output volume. Layer 2 had 64 filters directly followed by another max pooling. The final 3rd layer had 128 filters.

A padding parameter of 'same' was chosen in order to keep the spatial dimensions of the input volume.

An activation function of 'relu' was chosen because it is robust, and it gets rid of the vanishing gradient problem.

Overfitting was noticed in my data because accuracy to training data was 100%, but when submitted to Laggle, the accuracy was 30%. Therefore, a kernel regularizer was used to make the model be able to generalize. A L2 regularizer of 0.0005 was used.

Epoch is how many times you need to go over the data, and each time a different accuracy is found. A higher epoch increases the accuracy. However when an epoch = 100 was chosen, the resultant accuracy was 60% while an epoch of 10 resulted in an accuracy of 80%. Thus an epoch of 10 was chosen.

The **keras optimizer** is a predefined function that performs the following tasks

For epoch = 10

    Split data according to batch size

    For each batch size

        Feed batches(chunks) into the model

        According to the output, find errors and perform backpropagation

The **Adam optimizer** with learning rate = **1e-4** was used.

Model.fit fits the data using the loss model equal to softmax_cross_entropy, with the metric being 'accuracy'. Several other loss functions, such as min_squared_error, were tried but they either resulted in errors or they gave a lower accuracy.

Dropout were also tried to decrease overtraining, but they ended up decreasing accuracy when submitted to Kaggle.

Since the training data set is relatively small, data **augmentation** was tried. The idea was to make minor alterations to the existing dataset. Minor changes such as flips, translations, and rotations causes the neural network to perceive the altered data as completely new images. Data augmentation also allows for the reduction of model overfitting. However when this was implemented, the accuracy when submitting on Kaggle decreased for some unknown reason.