Saaif Ahmed

661925946

ahmeds7

4/1/2022

Introduction to Deep Learning: Programming Assignment #3 Report

1: Problem Statement

To develop a machine learning model that can identify between various objects, we could build a NN that is able to support multi-classification, but that many layers of connectivity would result in an immensely slow system that learns slowly because computation itself is large in size. Thus we develop a Convolutional Neural Network or CNN that is able to downsize a classification problem into a size order of magnitudes smaller than if it was fully connected. We build a CNN that is able to distinguish between 10 categories. These categories are as follows: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. For this we use data from the CIFAR-10 Dataset, a set of 60,000 32x32 colored images of these 10 classes. Thus each data point is 32x32x3 as the images are RGB. We analyze the model on its ability to correctly predict the class based on the requisite data.

2: Model Description

The model is built as follows. Outside of the input layer there are 3 convolutional layers, 2 pooling layers, a fully connected layer, and an output layer.

The first convolutional layer has 16 5x5 filters with no zero padding and stride of 1.

The first max pooling layer has an area of 2x2 with a stride of 2.

The second convolutional layer has 32 5x5 filters with no zero padding and stride of 1.

The second max pooling layer has an area of 2x2 with a stride of 2.

The third convolutional layer has 64 3x3 filters with no zero padding and stride of 1.

Each of the convolution layers has the ReLU activation functions which is defined as such.

$$ReLu(x) = \max\{0, x\}$$

The fully connected layer that follows from this convolution map has 500 nodes and is also activated with the ReLU function. And the output layer has 10 nodes to match the one-hot encoding of the output labels Y. This output layer is activated with the softmax function.

Softmax Function is as follows: $\sigma_M(\vec{x}^T[m]\Theta_k) = \frac{e^{\vec{x}^T[m]\Theta_k}}{\sum_{k=1}^{K} e^{\vec{x}^T[m]\Theta_k}}$

For each class k the function determines the value of the node through the functions computation for that class k. The output is then determined by the node with the largest value in that output layer.

To optimize this model we minimize the cross-entropy loss function. We define the negative log likelihood as

$$L(\vec{y}[m], \hat{y}[m]) = -\log\left(p(\vec{y}[m]|\vec{x}[m])\right) = -\sum_{\{k=1\}}^{K} \vec{y}[m][k] * \log(\hat{y}[m][k])$$

We construct the model in Tensorflow 2.X and as such the code definition to define the model is as follows.

```
class cnn_model(tf.keras.Model):
  def __init__(self):
    super(cnn_model, self).__init__()
    self.conv_layer_1 = layers.Conv2D(16, 5, strides=1, padding='valid', activation='relu')
    self.pooling_layer_1 = layers.MaxPool2D(2, 2)
    self.conv_layer_2 = layers.Conv2D(32, 5, strides=1, padding='valid', activation='relu')
    self.pooling_layer_2 = layers.MaxPool2D(2, 2)
    self.conv_layer_3 = layers.Conv2D(64, 3, strides=1, padding='valid', activation='relu')

    self.flatten = layers.Flatten()

    self.fully_connected = layers.Dense(500, activation='relu')
    self.output_layer = layers.Dense(10, activation=tf.keras.activations.softmax)
  def call(self, training_data):
    conv_1 = self.conv_layer_1(training_data)
    pool_1 = self.pooling_layer_1(conv_1)
    conv_2 = self.conv_layer_2(pool_1)
    pool_2 = self.pooling_layer_2(conv_2)
    conv_3 = self.conv_layer_3(pool_2)

    flat = self.flatten(conv_3)

    full_conn = self.fully_connected(flat)
    return self.output_layer(full_conn)
```

The model is built with tf.keras.Model and uses the Conv2D and MaxPool2D functions to build the model layer by layer.

To build and train the model we use the following code.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=7.65e-4, beta_1 = .75, beta_2 = .95,
                                                 epsilon = 1e-7),loss='categorical_crossentropy', metrics=['accuracy'])

completed_model = model.fit(training_data, training_label, validation_data=(testing_data, testing_label),
                            batch_size=batch, epochs=epochs)
```

The model is compiled use the Adam optimizer with the “categorical_crossentropy” loss as discussed above. We then use the model.fit() method to train and evaluate our model.

The model is saved using the model.save_weights() method and can be loaded with the model.load_weights() function in Tensorflow.

Hyper Parameters:

There are a few hyper parameters relevant to the model.

Batch Size: This determined the size of the mini batch runs per epoch of the model's training period. A large size took more time but was able to achieve convergence faster. For our model we used a batch size of 90.

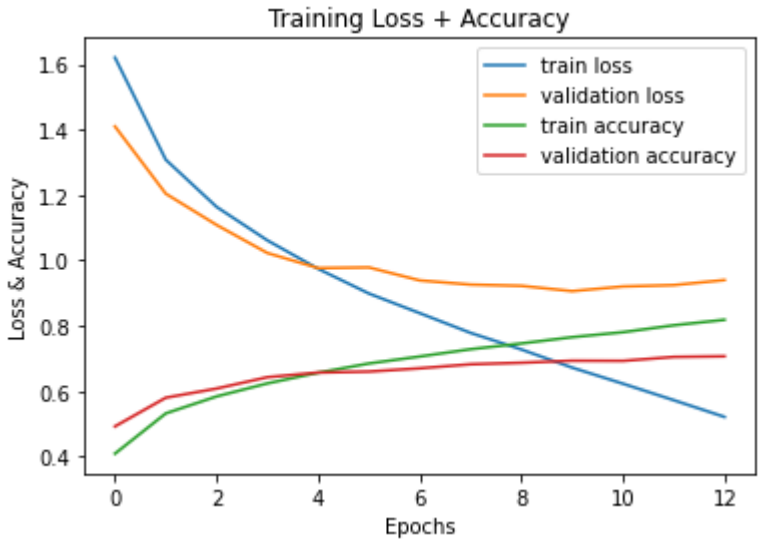
Learning Rate: This was a constant that was multiplied by the gradients to update the weights. We used a learning rate of 0.000765

Epochs: This is synonymous with the iterations of the model's learning period. Because our network had many layers to it, 13 epochs were chosen to reach desired accuracy and convergence.

Beta 1 & Beta 2: The Adam optimizer in Tensorflow have two beta constants that determine exponential decay and growth rate of the model's learning. We diverge from the default setting by choosing Beta 1 as 0.75 and Beta 2 as 0.95.

Input Data: The dataset used had 60,000 data points. We used 50,000 for training and 5,000 for our testing. As $D = \{\vec{x}[m], \vec{y}[m]\}, m = 1,2,..,M$ and input data $X^{50000\times32\times32\times3}$ with output labels $\vec{y}^{10\times1}$.

3: Results



Above we show the Training Loss vs. Epochs, the Validation Loss vs. Epochs, the Training Accuracy vs. Epochs, and the Validation Accuracy vs. Epochs.

	precision	recall	f1-score	support
0	0.69	0.77	0.73	488
1	0.81	0.85	0.83	505
2	0.59	0.62	0.60	512
3	0.54	0.55	0.54	497
4	0.70	0.63	0.66	507
5	0.64	0.48	0.55	488
6	0.65	0.85	0.74	491
7	0.83	0.73	0.78	495
8	0.83	0.83	0.83	504
9	0.80	0.75	0.78	513
accuracy			0.71	5000
macro avg	0.71	0.71	0.70	5000
weighted avg	0.71	0.71	0.70	5000

Above we show the classification accuracy for each specific class labeled 0-9. The precision column shows the accuracy and to obtain the error, one can do $1 - k_{precision}$ in order to validate it for each class k. It is clear the error is minimized.

```
157/157 - 1s - loss: 2.3060 - accuracy: 0.0916 - 611ms/epoch - 4ms/step
Untrained model, accuracy:  9.16%
157/157 - 0s - loss: 0.9397 - accuracy: 0.7064 - 405ms/epoch - 3ms/step
Restored model, accuracy: 70.64%
```

Above we show the overall accuracy of an untrained model versus a trained model. This was performed by saving the weights and loading them back up in a fresh instance. We see that the validation accuracy has reached the >70% margin.

Above we have the visualization of the filters in the first convolution layer. These are the 16 5x5 filters used in the first convolution layer in the CNN model we have made.

4: Discussion

To summarize the empirical results the model reached convergence after 13 epochs. Using Tensorflow's model.fit() function to train the model we reached >70% accuracy on the validation. The per-class accuracy was decently high at around the 65-80% margin which led our average validation accuracy to be high as well.

To discuss the runtime performance, training the model was very quick. If we were to do it with a fully connected multi-layered neural network the model would take extremely long to work. This is because each data point had roughly 2500 features that were a part of a dataset 50,000 long. Not to mention the full gradients needed to be computed in the back propagation. However, with our CNN we were able to still generate a trained model in a reasonable amount of time that had respectable accuracy. This may not have been as accurate as achievable with a fully connected NN, but the computation required to do it would simply not be possible. Thus the CNN was a very good way to build a model that was able to identify the classes in the dataset.

The usage of hyper parameters was crucial in achieving desired accuracy levels. If using any learning rate on the 10^{-3} scale or higher, the model would hit a convergent point far below the desired accuracy, like 50%. Thus numbers of the 10^{-4} scale were used. The batch size was important as well. The accuracy of the model was dependent on how much data it could utilize to learn the weights in each epoch. Thus choosing a batch size that allowed for more learning per epoch led to better results. Adjusting the beta values in the Adam optimizer allowed the model to achieve that marginal difference in accuracy that made it reach the accuracy threshold. However pushing them below 0.80 prevented the model from learning at any desirable pace.

The Adam optimizer was the best optimizer possible within Tensorflow for this model. Other optimizers were tested but were unable to achieve reliable results with hyper parameter tuning. This gives rise to the reason of Adam being the most popular in industry. However the method used to teach the model were very close to hitting the "overfitting" category. The train loss continued to decline, but the accuracy gains were diminishing over epochs. As such epochs had to be limited since node dropping was not implemented in the model.fit() method.