EECE 7740 Neural Networks Assignment
Sept 23, 2021

Bereket Eshete Kebede
UID - U00827234

M.S. Student
University of Memphis, Fall 2021
EECE 7740 Neural Networks

Designing a Convolutional Neural Network (CNN)
Developing a Deep Convolutional Neural Network (DCNN)

Instructor: Prof. Zahangir Alom
Assignment #2 Report

EECE 7740 Neural Networks Assignment
Sept 23, 2021

Designing a Convolutional Neural Network (CNN)

Introduction

In this report we design and implement a convolutional neural network (CNN) to train an algorithm for Image Classification on CIFAR-10 dataset. CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images used commonly to train computer vision algorithms. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes with 6,000 images of each class. Students were paid to label the data manually. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. [1] Codes for this project can be found at, https://github.com/bereketeshete/Image-Classifier.

Methodology

In this CNN, we use Rectified Linear Unit (ReLU) activation function because it is the current most popular activation for machine learning networks. We also use batch normalization, because it makes the machine learning network faster and stable through normalization of inputs by re-scaling and re-centering, even though the reasons behind this method improvement aren't understood yet. [2] We use the softmax layer for classification. We use 100 numbers of epochs for training our network, the batch size is 32 and we select Scholastic Gradient Descent (SGD) as our optimizer.

Deep Learning Architecture

The CNN we use for our training is depicted below in the schematic diagram below. Descriptively, we use an input layer, 2 convolution layers, two max pooling layers, a flattening layer and a softmax layer.

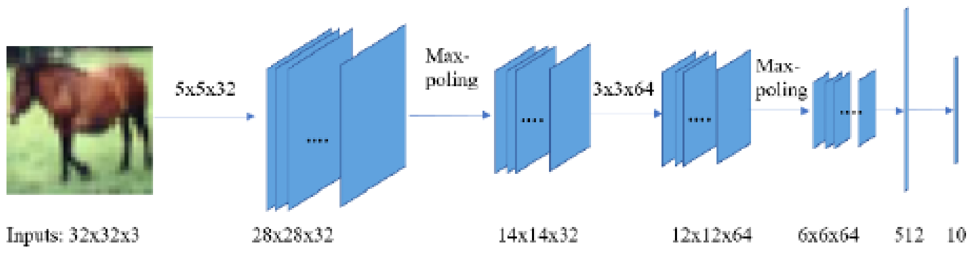
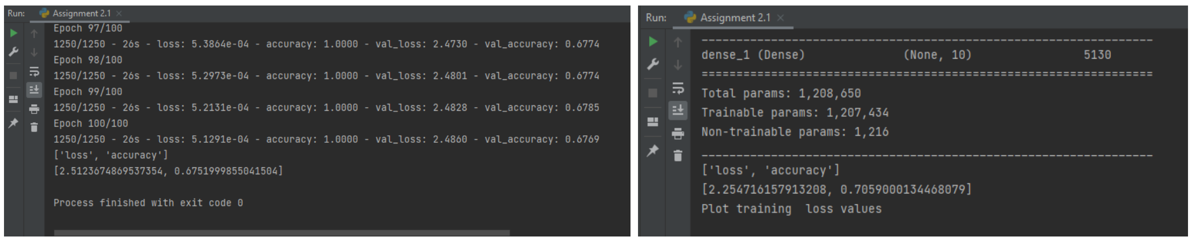


Fig 1. CNN Architecture used to train our training algorithm.

Experiment and Results

This experiment was conducted in Tensorflow v2.6 with Keras as backend and python 3.9 on a NVIDIA Quadro K420, core i7 processor and 32 RAM CPU. Tensor v2.6 isn't compatible with the GPU of K420, hence it reverted to CPU only. After noticing the slow training time on a CPU, we switched to a GPU environment with the intention of speeding up the training process. Finally, we used pycharm IDE for writing the python code and pip for installing the necessary packages.

(a) Training and testing logs



```
Epoch 97/100
1250/1250 - 26s - loss: 5.3864e-04 - accuracy: 1.0000 - val_loss: 2.4730 - val_accuracy: 0.6774
Epoch 98/100
1250/1250 - 26s - loss: 5.2973e-04 - accuracy: 1.0000 - val_loss: 2.4801 - val_accuracy: 0.6774
Epoch 99/100
1250/1250 - 26s - loss: 5.2131e-04 - accuracy: 1.0000 - val_loss: 2.4828 - val_accuracy: 0.6785
Epoch 100/100
1250/1250 - 26s - loss: 5.1291e-04 - accuracy: 1.0000 - val_loss: 2.4860 - val_accuracy: 0.6769
['loss', 'accuracy']
[2.5123674869537354, 0.6751999855041504]

Process finished with exit code 0
```

```
_________________________________________________________________
dense_1 (Dense)              (None, 10)                5130
=================================================================
Total params: 1,208,650
Trainable params: 1,207,434
Non-trainable params: 1,216
_________________________________________________________________
['loss', 'accuracy']
[2.254716157913208, 0.7059000134468079]
Plot training  loss values
```

Fig 2. (a) Left image, Training CNN model with a CPU, a 40 min total computation time, a **100%** training accuracy and a test accuracy of **67.5%** was observed. (b) Right image, Training CNN model with a GPU, a total computation time of 8.3 min was recorded. For this training, the training accuracy was **99.92%** and the test accuracy was **70.59%**.

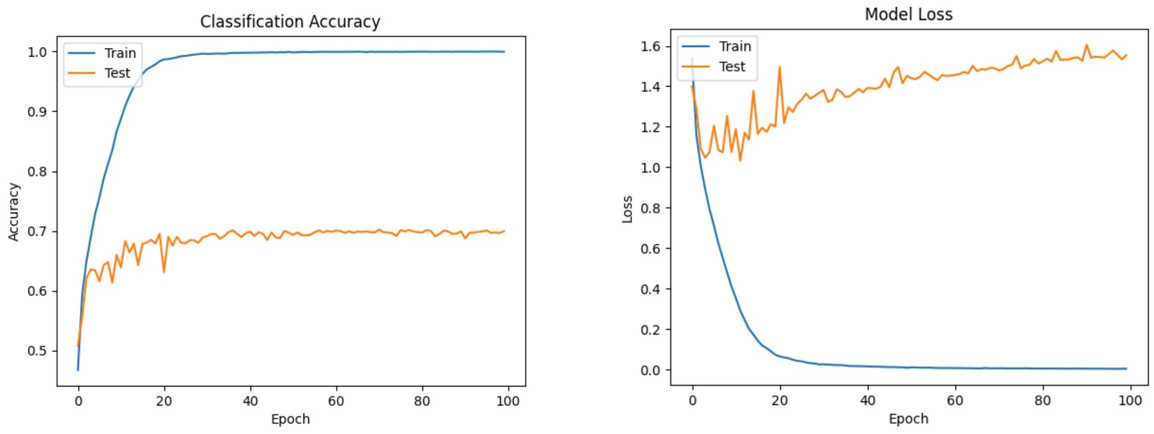


Fig 3. (a) Left image, classification accuracy graph of our CNN mode during training (b) Right image, loss graph of our CNN model during training. Graph is based on GPU training.

(b) Discussion and comparison

Looking at Fig (3) we can observe that classification accuration saturates at about 20 epochs, hence we can observe that 100 epochs isn't necessary for training our model. Using 20 epochs is sufficient to get a nearly similar accuracy of 60-70% and it will save us 6 min of extra training time. Finally, comparing the CPU and GPU results, the GPU performs better as intended yielding a 70.59% testing accuracy on classifying the 10 categories of our image dataset and also achieving 5x the speed of CPU training.

Conclusion

We were able to achieve a stageering **70.59%** classification accuracy on the CIFAR-10 dataset with our trained model utilizing a relatively shallow Convolutional Network Network (CNN). We also conclude 20 epochs is sufficient for training our model to yield similar classification accuracy. Finally, we observed the utilization of GPU has increased our CNN model training speed by five fold.

References

[1] https://en.wikipedia.org/wiki/CIFAR-10
[2] http://home.mit.bme.hu/~hadhazi/Oktatas/NN18/dem3/html_demo/CIFAR-10Demo.html
[3] https://en.wikipedia.org/wiki/Batch_normalization

EECE 7740 Neural Networks Assignment
Sept 23, 2021

Developing a Deep Convolutional Neural Network (DCNN)

Introduction

In this report, our goal is to have a better accuracy for scene understanding for our CNN. Our data set has 15 natural scenes (categories) for supervised learning. To achieve this goal we will use tensorflow and keras deep learning tools in a python environment. The 15 scene classes contain a variety of outdoor and indoor environments. The 15 classes are office, kitchen, living room, bedroom, store, industrial, tall building, inside city, street, highway, coast, open country, mountain, forest, and suburb. The images in the dataset have a 250*300 dimension, with 210 to 410 images per class. [1] Codes for this project can be found at, https://github.com/bereketeshete/Image-Classifier.

Methodology

For training our classification, we create a simple and intuitive function in python that trains our model by providing our input dataset and custom parameters to modify our algorithm.

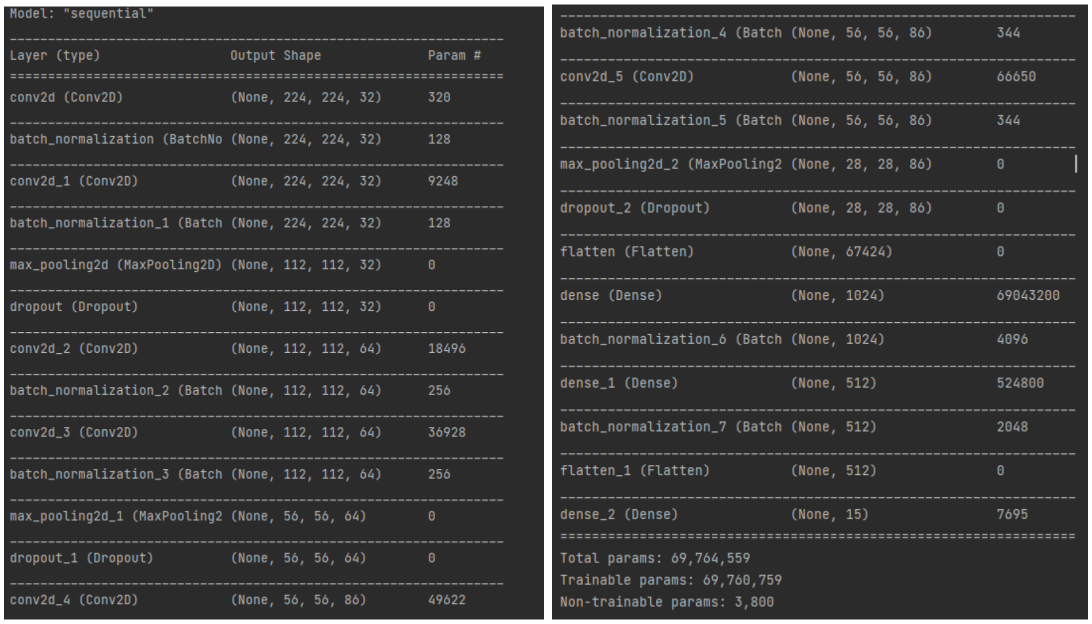
```
Memphis_DNN_Custom(X_train, Y_train, X_test, Y_test, epoch, optimizer)
```

In this report, we explore the different optimizers using the following Application Program Interface (API).

```
Memphis_DNN_Custom(X_train, Y_train, X_test, Y_test, 100, 'SGD')
Memphis_DNN_Custom(X_train, Y_train, X_test, Y_test, 100, 'Adam')
Memphis_DNN_Custom(X_train, Y_train, X_test, Y_test, 100, 'RMSProp')
```

Deep Learning Architecture

In this report, we use an advanced and a deeper CNN compared to question 1, in hopes of achieving better test accuracy for image classification. In this architecture, we use similar building blocks and concepts of question 1 to build a deeper CNN (DCNN). In addition we introduce a new activation function ELU to replace ReLU, dropout (regularization method) and explore different varying optimizers.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 224, 224, 32)      320
_________________________________________________________________
batch_normalization (BatchNo (None, 224, 224, 32)      128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 224, 224, 32)      9248
_________________________________________________________________
batch_normalization_1 (Batch (None, 224, 224, 32)      128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 32)      0
_________________________________________________________________
dropout (Dropout)            (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 64)      18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 112, 112, 64)      256
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 112, 112, 64)      36928
_________________________________________________________________
batch_normalization_3 (Batch (None, 112, 112, 64)      256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 56, 56, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 56, 56, 86)        49622
```

```
_________________________________________________________________
batch_normalization_4 (Batch (None, 56, 56, 86)        344
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 56, 56, 86)        66650
_________________________________________________________________
batch_normalization_5 (Batch (None, 56, 56, 86)        344
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 86)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 28, 28, 86)        0
_________________________________________________________________
flatten (Flatten)            (None, 67424)             0
_________________________________________________________________
dense (Dense)                (None, 1024)              69043200
_________________________________________________________________
batch_normalization_6 (Batch (None, 1024)              4096
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
batch_normalization_7 (Batch (None, 512)               2048
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 15)                7695
=================================================================
Total params: 69,764,559
Trainable params: 69,760,759
Non-trainable params: 3,800
```

Fig 1. Deep Learning Architecture for a 15-class image classification.

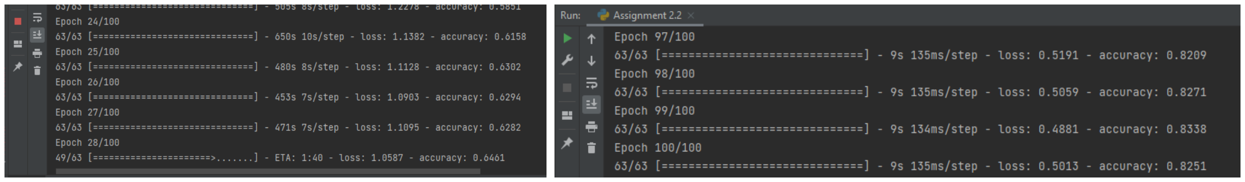
Experiment and Results

(a) Training and testing logs

The results will be divided into two for comparison with two initializers. First is the default tensor flow initializer, glorot initializer and the second one is the heuniform initializer.

1. Using Tensorflow Default/glorot initializer

- **Using a SGD Optimizer**

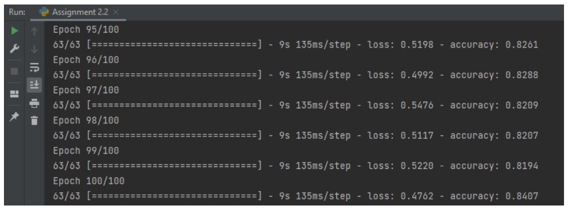


```
63/63 [==============================] - 505s 8s/step - loss: 1.2278 - accuracy: 0.5851
Epoch 24/100
63/63 [==============================] - 650s 10s/step - loss: 1.1382 - accuracy: 0.6158
Epoch 25/100
63/63 [==============================] - 480s 8s/step - loss: 1.1128 - accuracy: 0.6302
Epoch 26/100
63/63 [==============================] - 453s 7s/step - loss: 1.0903 - accuracy: 0.6294
Epoch 27/100
63/63 [==============================] - 471s 7s/step - loss: 1.1095 - accuracy: 0.6282
Epoch 28/100
49/63 [======================>.......] - ETA: 1:40 - loss: 1.0587 - accuracy: 0.6461
```

```
Epoch 97/100
63/63 [==============================] - 9s 135ms/step - loss: 0.5191 - accuracy: 0.8209
Epoch 98/100
63/63 [==============================] - 9s 135ms/step - loss: 0.5059 - accuracy: 0.8271
Epoch 99/100
63/63 [==============================] - 9s 134ms/step - loss: 0.4881 - accuracy: 0.8338
Epoch 100/100
63/63 [==============================] - 9s 135ms/step - loss: 0.5013 - accuracy: 0.8251
```

Fig 2. Using CPU. **64%** training accuracy at 28th epoch. (b) The image on the right, we have to revert to GPU for faster speed and observed **82.51%** training accuracy.

❖ **Using an Adam optimizer**



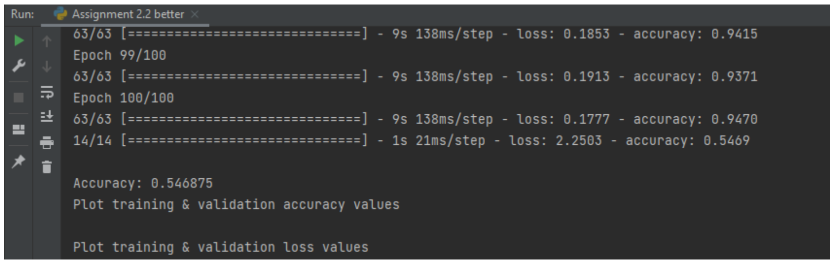
```
Epoch 95/100
63/63 [==============================] - 9s 135ms/step - loss: 0.5198 - accuracy: 0.8261
Epoch 96/100
63/63 [==============================] - 9s 135ms/step - loss: 0.4992 - accuracy: 0.8288
Epoch 97/100
63/63 [==============================] - 9s 135ms/step - loss: 0.5476 - accuracy: 0.8209
Epoch 98/100
63/63 [==============================] - 9s 135ms/step - loss: 0.5117 - accuracy: 0.8207
Epoch 99/100
63/63 [==============================] - 9s 135ms/step - loss: 0.5220 - accuracy: 0.8194
Epoch 100/100
63/63 [==============================] - 9s 135ms/step - loss: 0.4762 - accuracy: 0.8407
```

Fig 3. Since we know GPU performs better on the previous figure, we will start working with GPU. This time using Adam optimizer we observe an **84.87%** training accuracy.

From the above two figures, we can observe a different optimizer didn't change the training accuracy significantly so instead of trying RMSProp Optimizer let's switch initializer to He initialization to explore a better classification performance.

2. Using HeUniform initializer

❖ **Using** SGD Optimizer



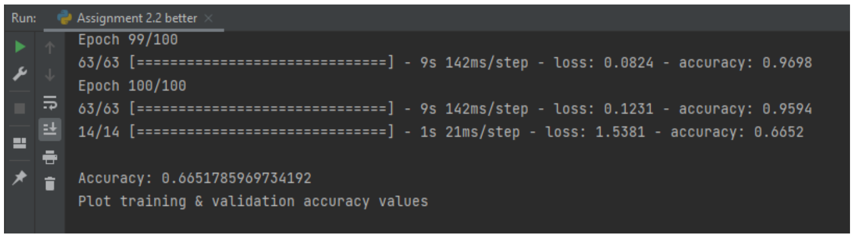
```
63/63 [==============================] - 9s 138ms/step - loss: 0.1853 - accuracy: 0.9415
Epoch 99/100
63/63 [==============================] - 9s 138ms/step - loss: 0.1913 - accuracy: 0.9371
Epoch 100/100
63/63 [==============================] - 9s 138ms/step - loss: 0.1777 - accuracy: 0.9470
14/14 [==============================] - 1s 21ms/step - loss: 2.2503 - accuracy: 0.5469

Accuracy: 0.546875
Plot training & validation accuracy values

Plot training & validation loss values
```

Fig 4. Train accuracy for SGD Optimizer is **94.7%** and Test accuracy of **54.69%.**

❖ **Using** Adam Optimizer

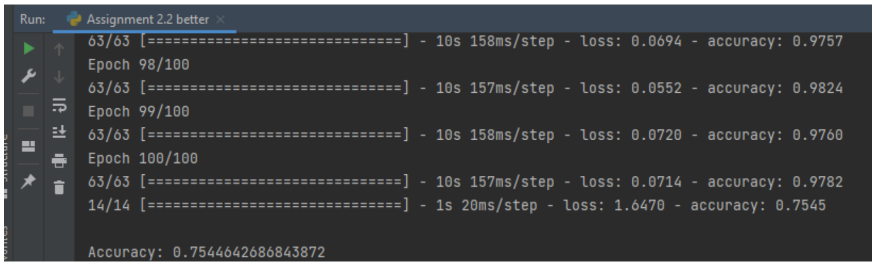


```
Epoch 99/100
63/63 [==============================] - 9s 142ms/step - loss: 0.0824 - accuracy: 0.9698
Epoch 100/100
63/63 [==============================] - 9s 142ms/step - loss: 0.1231 - accuracy: 0.9594
14/14 [==============================] - 1s 21ms/step - loss: 1.5381 - accuracy: 0.6652

Accuracy: 0.6651785969734192
Plot training & validation accuracy values
```

Fig 4. Training accuracy for Adam Optimizer is **95.94%** and Test accuracy is **66.52%.**

- **Using** RMSProp Optimizer



```
63/63 [==============================] - 10s 158ms/step - loss: 0.0694 - accuracy: 0.9757
Epoch 98/100
63/63 [==============================] - 10s 157ms/step - loss: 0.0552 - accuracy: 0.9824
Epoch 99/100
63/63 [==============================] - 10s 158ms/step - loss: 0.0720 - accuracy: 0.9760
Epoch 100/100
63/63 [==============================] - 10s 157ms/step - loss: 0.0714 - accuracy: 0.9782
14/14 [==============================] - 1s 20ms/step - loss: 1.6470 - accuracy: 0.7545

Accuracy: 0.7544642686843872
```

Fig 5. Training accuracy for RMSProp Optimizer is **97.82%** and Test accuracy is **75.45%.**

(b) Discussion and comparison

Comparing test accuracy and computational time respectively we observed the following results as summarized below achieved by He intializer. We can also observe He Intializer training accuracy trumps over the default/

Using Default Intializer

Using He Intializer
Using SGD - 54.69% Te.A
Using Adam - 66.52% Te.A
Using RMSProp - 75.45% Te.A

By comparing the testing accuracy of the above results, we observe the RMSProp optimization function performs better than the other two optimizers we used.

Conclusion

In summary, the main purpose of this report is to explore the different optimizers for our model and investigate their effect. We also implemented an We concluded the RMSProp optimization function is the best optimizer for getting a better classification algorithm on our dataset.

References

[1] https://qixianbiao.github.io/Scene.html
[2] https://figshare.com/articles/15-Scene_Image_Dataset/7007177
[3] https://www.kaggle.com/zaiyankhan/15scene-dataset
[4] https://www.kaggle.com/zaiyankhan/zaiyan
[5] https://www.kaggle.com/zaiyankhan/zaiyan-assignment-5

[6] https://www.kaggle.com/puneet6060/image-scene-classification