Designing and Training Neural Networks: Image Classification

Assessment 3 Report of ZZEN9444 - Neural Networks, Deep Learning

Rushmila Islam

2024-03-02

Contents

1 Introduction

In this assignement we design and model a neural network that takes images of cats as input and learns to classify eight different breeds - Bombay, Calico, Persian, Russian Blue, Siamese, Tiger, Tortoiseshell and Tuxedo. The training set contains 1000 images of each categories of cat images in 80x80 resolution. We develop a variant of VGG11 for this exercise and found the results to be satisfactory to be within the preset constraint of model file size limit. We discuss our findings below.

2 Model Architecture, Algorithms and Enhancements

The major constraints of designing the model is the size limitation of the model. The model size must be under 50.00MB. To establish that we could not choose to implement a bigger network that performs better. We have considered VGG11 and ResNet34 to start with. Our derived model is inspired by VGG11 [1] but because of size constraint, we have adjusted the network model.

Below is a comparison between ResNet34 and VGG11_bn (batch normalization) on the given dataset:

Table 1: Model Comparison

Model	Highest Training Accuracy (%)	Highest Validation Accuracy (%)	Model Size
ResNet34	85.50%	71.75%	87.3MB
VGG11_bn	80.33%	72.75%	531MB

We found that model performance is very close between these two models. We have started with a variant of ResNet34 model but was unable to fit the model under 50MB. Considering our network size limitation and based on PyTorch performance evaluation on ImageNet dataset [2], we consider building model that follows the VGG architecture. We have started with batch normalization version of modified VGG 11 model but the test performance on validation data was not very good compared to the model without batch normalization. So we proceed with the model without batch normalization.

Model with Batch Normalization	Model without Batch Normalization
ep 1, loss: 86.08, 6400 train 14.88%, 1600 test 14.69%	ep 1, loss: 234.29, 6400 train 11.25%, 1600 test 11.00%
ep 2, loss: 56.55, 6400 train 35.02%, 1600 test 12.44%	ep 2, loss: 84.79, 6400 train 18.94%, 1600 test 25.75%
ep 3, loss: 49.01, 6400 train 43.58%, 1600 test 14.12%	ep 3, loss: 58.29, 6400 train 27.89%, 1600 test 32.81%
ep 4, loss: 46.73, 6400 train 45.78%, 1600 test 19.38%	ep 4, loss: 54.77, 6400 train 33.55%, 1600 test 34.44%
ep 5, loss: 46.13, 6400 train 46.08%, 1600 test 17.69%	ep 5, loss: 52.87, 6400 train 37.39%, 1600 test 38.81%
ep 6, loss: 44.57, 6400 train 48.22%, 1600 test 17.81%	ep 6, loss: 51.98, 6400 train 38.81%, 1600 test 42.00%
ep 7, loss: 43.36, 6400 train 50.67%, 1600 test 25.37%	ep 7, loss: 49.42, 6400 train 41.67%, 1600 test 40.38%
ep 8, loss: 42.36, 6400 train 51.16%, 1600 test 22.56%	ep 8, loss: 48.84, 6400 train 43.25%, 1600 test 46.00%
ep 9, loss: 41.61, 6400 train 52.12%, 1600 test 28.19%	ep 9, loss: 46.53, 6400 train 45.80%, 1600 test 46.31%
ep 10, loss: 40.29, 6400 train 53.44%, 1600 test 21.00%	ep 10, loss: 46.17, 6400 train 45.94%, 1600 test 43.12%

The architecture of the network that we are going to implement in this study is shown in Fig xxx. The inputs to the convolution layer are 224x224 RGB images. As these are RGB images, we have 3 channels as the input, we have done few pre-processing such as horizontal flip, auto contrast, adjust sharpness, vertical flip along with resizing the image. The input layer goes through a convolution layer that has kernel size of 3x3 with stride 1 and padding 1. The first convolution layer has 3 in_channels and 64 out_channels. Then we have added batch normalization followed by a pooling layer with max pooling operation and has a kernel size of 2x2 (stride=2). Altogether we have eight convolution layers followed by pooling layers. Then the final convolution layer has 256 in_channels and 256 out_channels with a kernel size 3x3. All the convolution layers are using ReLU activation. We have added an average pooling layer 7x7 an then a fully connected linear layer with ReLU activation which is fully connected to output layer with log_softmax activation. We have only used two fully connected linear layers as adding more linear layers make the model size much bigger than 50MB.

The dimensions of the tensors of each layer:

Input: 224x224x3
Conv1: 224x224x32
Pool1: 112x112x32
Conv2: 112x112x64
Pool2: 56x56x64
Conv3: 56x56x64
Conv4: 56x56x64
Pool3: 28x28x64
Conv5: 28x28x128
Conv6: 28x28x128
Pool4: 14x14x128
Conv7: 14x14x128
Conv8: 14x14x128
Pool5: 7x7x128
Pool6: 7x7x128
fc_1: 1024
output and softmax: 8

```python
class Network(nn.Module):

    def __init__(self):
        super().__init__()

        # Convolution layers
        #1
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)

        #2
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)

        #3
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1)

        #4
        self.conv5 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.conv6 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1)

        #5
        self.conv7 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.conv8 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=1)


        # Fully connected layers
        #6
        self.fc1 = nn.Linear(7*7*128, 1024)

        #7
        self.out = nn.Linear(1024, 8)

        # Adding dropouts to avoid overfitting
        self.dropout = nn.Dropout(0.5)
    def forward(self, x):
        #1
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        #2
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        #3
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        #4
        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        #5
        x = F.relu(self.conv7(x))
        x = F.relu(self.conv8(x))
        x = F.max_pool2d(x, kernel_size=2, stride=2)

        x = F.adaptive_avg_pool2d(x, 7)

        x = torch.flatten(x, 1) # flatten all dimensions

        #6
        x = F.relu(self.fc1(x))

        x = self.dropout(x)

        #7
        x = self.out(x)

        x = F.log_softmax(x, dim=-1)

        return x
```

Initially we have trained the network with the whole dataset. Then we performed validation testing. We have splitted train and validation data 80:20 ratio, so the training datatset has 6400 images and validation test dataset has 1600 images. We have started with a smaller batch size (e.g. 64) but it makes the loss much higher. And setting larger batch size takes more time to train and validate. Also we have 8000 data considering that any batch size that is multiplied by 10 will be ideal in this case.

```
Network parameters:  Network(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv6): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv8): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=6272, out_features=1024, bias=True)
  (out): Linear(in_features=1024, out_features=8, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
batch size:  200 epochs 150
Using device: mps
```

We have run the experiment in MacOS and used MPS enabled for faster processing:

```
import torch

# Use a GPU if available, as it should be faster.
# device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')
```

3 Loss Function and Optimizer

Here we are using CrossEntropyLoss function for the multi-label classification and Adam optimizer with scheduler which has learning rate starting from 0.001 and ending at 1. The reason of choosing Adam optimizer is - it is better than other adaptive learning rate algorithms due to its faster convergence and robustness[4].

4 Image Transformations

Data should be formatted into appropriately and converted to tensors before being fed into the model. So we resize them to a size of use 224 $\times$ 224. We have performed transformation using torchvision transforms [3]. After exploring the dataset, we have observed there are many sorts of impurity in the image datasets, for example - brightness, contrast, cropped image, blurred image, multiple photos in one image etc. We have applied few transformation techniques to the the better perfomance in the training and test dataset-

1. Resizing image to 224x224
2. Flipping image horizontally
3. Flipping image vertically
4. Rotation
5. Normalization

We have performed same image transformation on both train and test dataset.

5 Overfitting

We are using weight initializer for both linear and convolution layers. Here we are using Kaiming normal initializer for weight initialization as we are using ReLU activation. The standard approach for initialization of the weights of neural network layers that use the ReLU activation function is Kaiming initialization [5]. We have also introduced dropout 0.5 at linear layer and 0.0001 weight decay at the optimizer to avoid overfitting.

6 Result Analysis

We have trained our model for the whole dataset initially. We have used the savedModel.pth file and used the pretrained model's weight parametes to run validation testing.

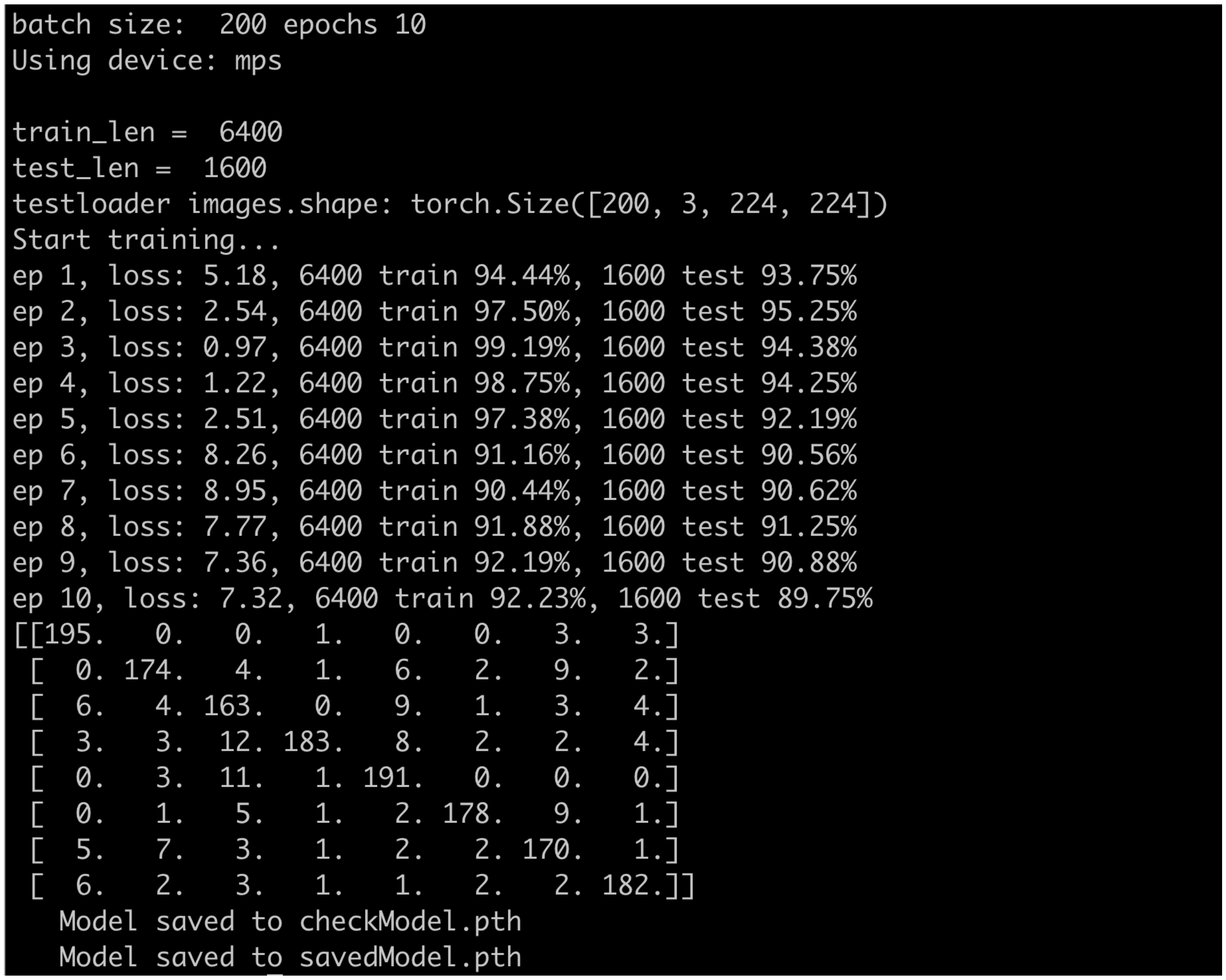
Highest training accuracy recorded 93.35% while training the model for 150 epochs.

Table 4: Training accuracy

Epochs	Accuracy
10	50.08%
20	62.41%
30	69.51%
40	75.29%
50	77.65%
60	81.30%
70	84.03%
80	87.45%
90	87.62%
100	90.25%
110	90.75%
120	91.61%
130	90.88%
140	91.41%
150	92.86%

After pre-training we have loaded the saved model to perform validation testing.

95.25% highest test accuracy received on validation data and after running for 10 epochs.



```
batch size:  200 epochs 10
Using device: mps

train_len =  6400
test_len =  1600
testloader images.shape: torch.Size([200, 3, 224, 224])
Start training...
ep 1, loss: 5.18, 6400 train 94.44%, 1600 test 93.75%
ep 2, loss: 2.54, 6400 train 97.50%, 1600 test 95.25%
ep 3, loss: 0.97, 6400 train 99.19%, 1600 test 94.38%
ep 4, loss: 1.22, 6400 train 98.75%, 1600 test 94.25%
ep 5, loss: 2.51, 6400 train 97.38%, 1600 test 92.19%
ep 6, loss: 8.26, 6400 train 91.16%, 1600 test 90.56%
ep 7, loss: 8.95, 6400 train 90.44%, 1600 test 90.62%
ep 8, loss: 7.77, 6400 train 91.88%, 1600 test 91.25%
ep 9, loss: 7.36, 6400 train 92.19%, 1600 test 90.88%
ep 10, loss: 7.32, 6400 train 92.23%, 1600 test 89.75%
[[195.   0.   0.   1.   0.   0.   3.   3.]
 [  0. 174.   4.   1.   6.   2.   9.   2.]
 [  6.   4. 163.   0.   9.   1.   3.   4.]
 [  3.   3.  12. 183.   8.   2.   2.   4.]
 [  0.   3.  11.   1. 191.   0.   0.   0.]
 [  0.   1.   5.   1.   2. 178.   9.   1.]
 [  5.   7.   3.   1.   2.   2. 170.   1.]
 [  6.   2.   3.   1.   1.   2.   2. 182.]]
   Model saved to checkModel.pth
   Model saved to savedModel.pth
```

Confusion Matrix:

```
[[195.   0.   0.   1.   0.   0.   3.   3.]
 [  0. 174.   4.   1.   6.   2.   9.   2.]
 [  6.   4. 163.   0.   9.   1.   3.   4.]
 [  3.   3.  12. 183.   8.   2.   2.   4.]
 [  0.   3.  11.   1. 191.   0.   0.   0.]
 [  0.   1.   5.   1.   2. 178.   9.   1.]
 [  5.   7.   3.   1.   2.   2. 170.   1.]
 [  6.   2.   3.   1.   1.   2.   2. 182.]]
```

As reading from the confusion matrix above we can say that Persian (class-2) cats mostly classified for Russian Blue (class-3) and Siamese (class-4).

7 Conclusion

There is a lot more area to explore for fine tuning the model. Training the model took much time and longer epochs to reach above 90% accuracy. It would be great to fine tune more meta parameter to train the model faster with higher accuracy. Another area to explore would be to use different model like YOLO, so it can classify different categories of cats.

8 References

1. https://arxiv.org/pdf/1409.1556.pdf
2. https://pytorch.org/vision/0.12/models.html
3. https://pytorch.org/vision/stable/transforms.html
4. https://builtin.com/machine-learning/adam-optimization
5. https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/