Note_3_TestTunedModel

April 19, 2022

# Test the tuned Model

In the last notebook, we have seen the steps to fine tune AlexNet model with MNIST dataset. In this notebook, we will load the tuned model shared by NICD and then will test it with MNIST data.

In the import section below, we have used two new import which are torch.nn and matplotlib.pyplot. In which torch.nn provide us many more classes and modules to implement and train the neural network and matplotlib.pyplot is a collection of functions that make matplotlib work like MATLAB through which we can create figures.

```
[ ]: import torch
     from torchvision import models, transforms, datasets
     import torch.nn as nn
     import matplotlib.pyplot as plt
```

The Transforms module used in the below snippet helped us in transforming the greyscaled MNIST data into a tensor.

```
[ ]: transform = transforms.Compose([
      transforms.Resize(224),
      transforms.ToTensor(),
      # MNIST is greyscale so we stack the image to create 3 channels
      transforms.Lambda(lambda x: x.repeat(3, 1, 1)),
      transforms.Normalize(
      mean=[0.485, 0.456, 0.406],
      std=[0.229, 0.224, 0.225]),
      ])
```

Torchvision provides many built-in datasets in the torchvision.datasets module. we have downloaded the MNIST datasets here and stored it in the directory named as Data. Data loader used here combines a dataset and a sampler, and provides an iterable over the given dataset.

```
[ ]: mnist_test_dataset = datasets.MNIST(root='./data', train=False, download=True,␣
      ↪transform=transform)
     MNIST_test_loader = torch.utils.data.DataLoader(mnist_test_dataset,␣
      ↪batch_size=1, shuffle=True)
```

In the below snippet, we have created a alexnet model without its pretrained feature. If we execute the model, we can see the model architecture with out_feature =1000 in its last layer.

```
[ ]: model = models.alexnet()
     model
```

```
[ ]: AlexNet(
       (features): Sequential(
         (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
         (1): ReLU(inplace=True)
         (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
     ceil_mode=False)
         (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
         (4): ReLU(inplace=True)
         (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
     ceil_mode=False)
         (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (7): ReLU(inplace=True)
         (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (9): ReLU(inplace=True)
         (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (11): ReLU(inplace=True)
         (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
     ceil_mode=False)
       )
       (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
       (classifier): Sequential(
         (0): Dropout(p=0.5, inplace=False)
         (1): Linear(in_features=9216, out_features=4096, bias=True)
         (2): ReLU(inplace=True)
         (3): Dropout(p=0.5, inplace=False)
         (4): Linear(in_features=4096, out_features=4096, bias=True)
         (5): ReLU(inplace=True)
         (6): Linear(in_features=4096, out_features=1000, bias=True)
       )
     )
```

# Redefining the layers The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively. So we need to replace the last layer with 10 neurons. nn. Linear(n,m) is a module that creates single layer feed forward network with n inputs and m output. Through which we can update the output of our last layer.

```
[ ]: num_ftrs = model.classifier[6].in_features
     model.classifier[6] = nn.Linear(num_ftrs,10)
```

If we check the model architecture, we can see the last layer out_features was updated to 10.

```
[ ]: model
[ ]: AlexNet(
       (features): Sequential(
         (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
         (1): ReLU(inplace=True)
         (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
     ceil_mode=False)
         (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
         (4): ReLU(inplace=True)
         (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
     ceil_mode=False)
         (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (7): ReLU(inplace=True)
         (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (9): ReLU(inplace=True)
         (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
         (11): ReLU(inplace=True)
         (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
     ceil_mode=False)
       )
       (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
       (classifier): Sequential(
         (0): Dropout(p=0.5, inplace=False)
         (1): Linear(in_features=9216, out_features=4096, bias=True)
         (2): ReLU(inplace=True)
         (3): Dropout(p=0.5, inplace=False)
         (4): Linear(in_features=4096, out_features=4096, bias=True)
         (5): ReLU(inplace=True)
         (6): Linear(in_features=4096, out_features=10, bias=True)
       )
     )
```

Now, we have our expected model architecture for MNIST data pretection. But it needs to be trained to predict MNIST data. Instead of training the model with MNIST dataset, we directly loaded the finetuned alexnet model head shared by NICD. Training a model in a CPU machine will take a long time. So with the help of NICD, we finttuned our pretrained alexnet model with MNIST dataset in NICD's GPU machine which took approximately 40 to 50 mins for 50 epochs.

To load this finetuned model, we need to place the model file (mnist_head_alexnet.pth) in the same directory of our source code. Then using torch.load() we can deserialize python's object files to memory. Finally, using load_state_dict, we can loads a model's parameter dictionary using a deserialized state_dict.

In PyTorch, the learnable parameters (i.e. weights and biases) of an torch.nn.Module model are contained in the model's parameters (accessed with model.parameters()). A state_dict is simply a Python dictionary object that maps each layer to its parameter tensor. Note that only layers with learnable parameters (convolutional layers, linear layers, etc.) and registered buffers have entries in the model's state_dict.

[ ]:
```
model.load_state_dict(torch.load('mnist_head_alexnet.pth',map_location=torch.
  ↪device('cpu')))
model.eval()
```

[ ]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Since, we are going to test the model directly we are using eval mode in the above snippet. Next, we need to pass our MNIST data into our model and predict the output. So, using the below code snippet we iterated and picked a single data from our dataset.

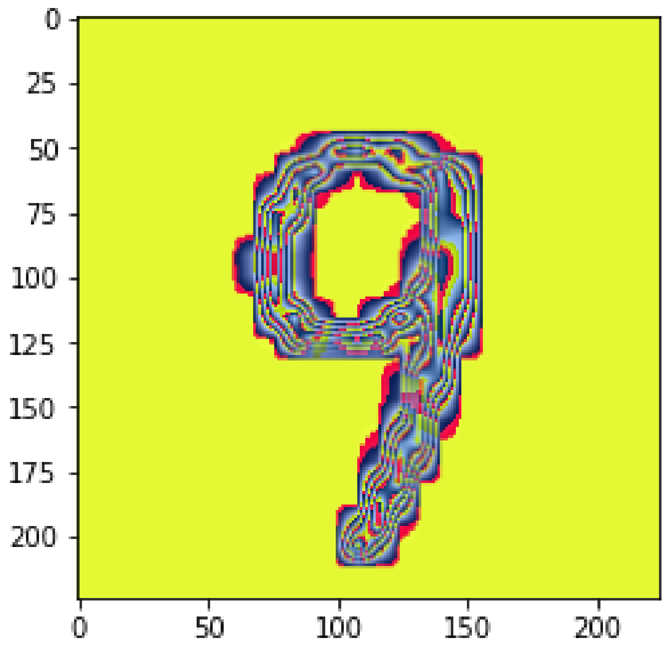
[ ]:
```
sample_image = iter(MNIST_test_loader).next()[0]
sample_image
```

[ ]:
```
tensor([[[[-2.1179, -2.1179, -2.1179,  …, -2.1179, -2.1179, -2.1179],
          [-2.1179, -2.1179, -2.1179,  …, -2.1179, -2.1179, -2.1179],
          [-2.1179, -2.1179, -2.1179,  …, -2.1179, -2.1179, -2.1179],
          …,
          [-2.1179, -2.1179, -2.1179,  …, -2.1179, -2.1179, -2.1179],
          [-2.1179, -2.1179, -2.1179,  …, -2.1179, -2.1179, -2.1179],
          [-2.1179, -2.1179, -2.1179,  …, -2.1179, -2.1179, -2.1179]],

         [[-2.0357, -2.0357, -2.0357,  …, -2.0357, -2.0357, -2.0357],
          [-2.0357, -2.0357, -2.0357,  …, -2.0357, -2.0357, -2.0357],
          [-2.0357, -2.0357, -2.0357,  …, -2.0357, -2.0357, -2.0357],
          …,
          [-2.0357, -2.0357, -2.0357,  …, -2.0357, -2.0357, -2.0357],
          [-2.0357, -2.0357, -2.0357,  …, -2.0357, -2.0357, -2.0357],
          [-2.0357, -2.0357, -2.0357,  …, -2.0357, -2.0357, -2.0357]],

         [[-1.8044, -1.8044, -1.8044,  …, -1.8044, -1.8044, -1.8044],
          [-1.8044, -1.8044, -1.8044,  …, -1.8044, -1.8044, -1.8044],
          [-1.8044, -1.8044, -1.8044,  …, -1.8044, -1.8044, -1.8044],
          …,
          [-1.8044, -1.8044, -1.8044,  …, -1.8044, -1.8044, -1.8044],
          [-1.8044, -1.8044, -1.8044,  …, -1.8044, -1.8044, -1.8044],
          [-1.8044, -1.8044, -1.8044,  …, -1.8044, -1.8044, -1.8044]]]])
```

Our sample image is in tensor format. To visualise the selected data, we used matplotlib.pyplot to create the figure of the image.

```
[ ]: pil_iamge = transforms.ToPILImage()(sample_image.squeeze(0))
     plt.imshow(pil_iamge, cmap='gray')
```

```
[ ]: <matplotlib.image.AxesImage at 0x22db8534040>
```



Finally, we just passed our sample test data into our model and model precits the probablities of each tensor[0-9].

```
[ ]: pred =model(sample_image)
     pred
```

```
[ ]: tensor([[-3.6049e+00, -1.0361e+01, -5.9384e-03, -2.9405e+00, -8.4072e-01,
              -5.0797e+00, -6.5814e+00,  4.7631e-01,  2.4504e-01,  6.4359e+00]],
            grad_fn=<AddmmBackward0>)
```

```
[ ]: torch.argmax(pred)
```

```
[ ]: tensor(9)
```

Finally, using torch.argmax tensor with highest probabilties is returned and that is our expected output.