EEE 598: Deep Learning - Foundations and Applications

Assignment 1[1]

[1]

1. Problem 1: Cats vs Dogs

1.1. Part 1

The link to teachable machine model can be found **here**.

Dataset Segregation:

- Dogs: 1500 images sourced from the Stanford Dogs Dataset. These images feature a variety of dog breeds and are of high quality.
- Cats: 1500 images obtained from a Kaggle dataset. The collection includes images of different cat breeds, also of high quality.
- Others: 500 images captured using a webcam. This category includes: Personal images (e.g., pictures of myself and my living space), various wild animals (e.g., lions, tigers, wolves) and other miscellaneous items (e.g., furniture and objects resembling dogs or cats). The statistics can be found in 1

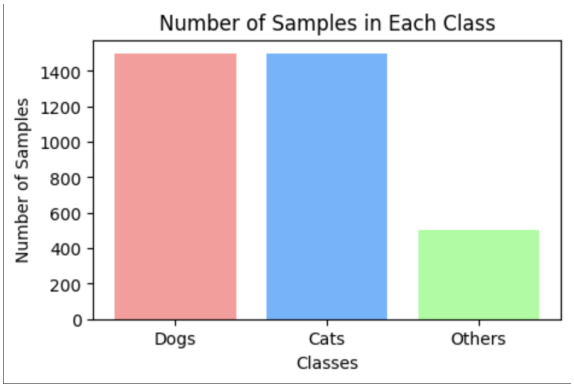

Figure 1.

1.2. Part 2

Initial Training and Data Imbalance:

The model was initially trained with 50 epochs achieving **a validation score of 80%**, but satisfactory performance was not achieved. Human images were notably misclassified as dogs, which indicated a significant class imbalance in the dataset. This misclassification highlighted the need for adjustments in both the data and training approach.

1.3. Part 3

Data Augmentation: To address the class imbalance, the amount of data in the underrepresented class was increased. More human images were added, including pictures of the researcher and several other images that shared visual similarities with both dogs and

cats. This step was crucial in providing the model with a more balanced and diverse dataset from which to learn.

Model Re-training: With the augmented dataset, the model was re-trained. The number of epochs was increased from 50 to 200, allowing more iterations for the model to learn from the new, more balanced dataset.

Learning Rate Adjustment: As the training progressed, the need to fine-tune the model's ability to learn the new features without overfitting was noticed. To achieve this, the learning rate was reduced to 0.0001. This lower learning rate allowed smaller, more precise updates to be made to the model's parameters, helping it to capture the nuances of the new data without overgeneralizing.

Results: These adjustments led to a significant improvement in the model's performance, resulting in a validation **score of 90%.** Figures 2, 3, 4 in the report illustrate examples of correct and incorrect classifications, demonstrating the model's enhanced ability to differentiate between humans, dogs, and cats.

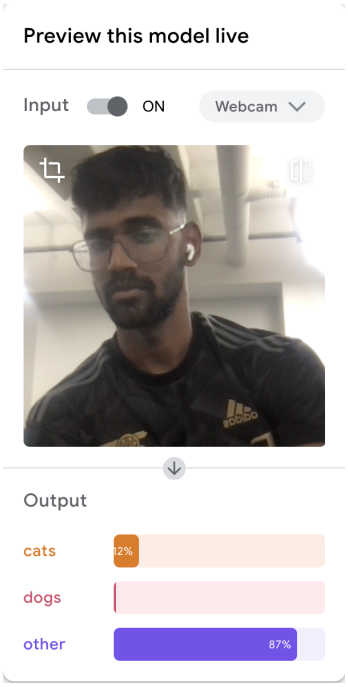

Figure 2. Incorrect Prediction

2. Introduction to SOL

2.1. Part 1

The figure 5 shows a Jupyter notebook interface running Python 3.11.6, packaged by conda-forge, on a Linux system. The required packages have been installed and verified: PyTorch (version 2.2.3+cu121) with GPU support via CUDA 12.1, and tqdm (version 4.65.0). The successful import statements for torch and tqdm, along with their version outputs, are visible in the Terminal 1 tab. While not explicitly shown, glob is part of Python's standard library and doesn't require separate installation. The file browser on the left displays various Python scripts and Jupyter notebooks, indicating an active development environment.

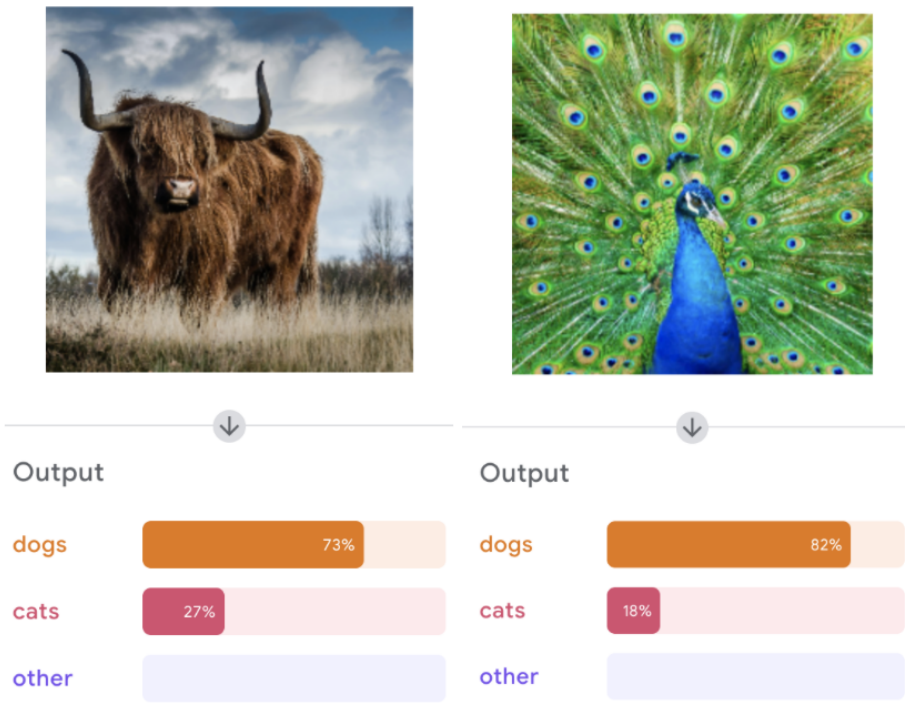

Figure 3. GPU Allocation

2.2. Part 2

The NVIDIA driver version is 555.58.02 as shown in figure 6, and the CUDA version is 12.5. The GPU has 80GB of memory, with only 1MiB currently in use. The GPU utilization is at 0%, indicating it's not actively processing any tasks at the moment. This confirms the presence of a high-performance GPU suitable for machine learning tasks, and verifies that the CUDA toolkit is properly installed, aligning with the PyTorch CUDA support observed in the previous environment setup.

The figure 7 shows the access to the Conda environment as a kernel when running code in .ipynb files inside SOL.

3. Pytorch on SOL

3.1. Part 2

Tensor Operations: As shown in figure 9, two tensors (t1 and t2) of shape (5, 4) were created with random float32 values. The element-wise multiplication of these tensors was performed, resulting in a new tensor t3. This operation is demonstrated by the line: t3 = t1.mul(t2).view(10,2)

Original input tensors (t1 and t2) are displayed, each of shape (5, 4). Data type of tensors: torch.float32 (as shown in the output). The reshaped output tensor t3 is shown with shape (10, 2).

3.2. Part 3

Autograd Example: Figure **??** illustrates the autograd example:

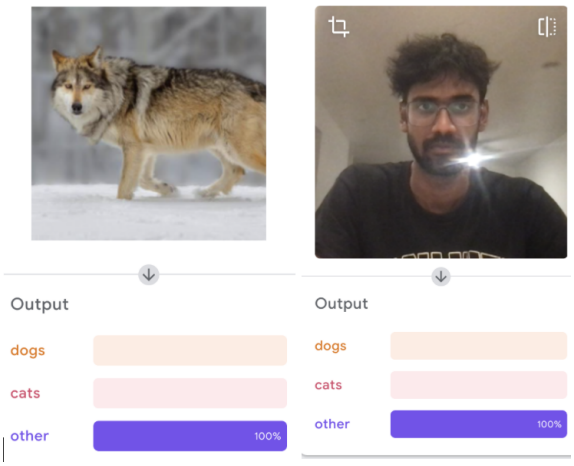

Figure 4. Correct Predictions

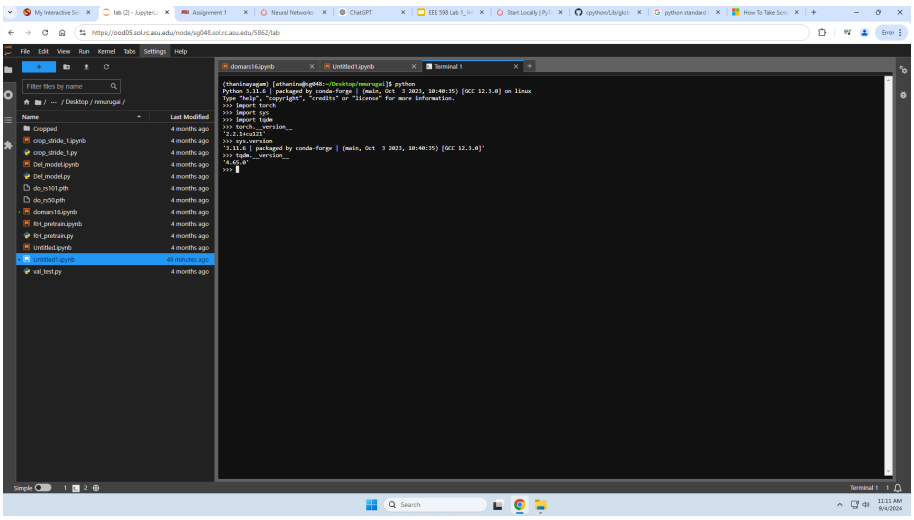

Figure 5. Virtual Enviornment

- **Original tensor with requires_grad=True:** tensor([[2., -1., 3.], [4., 0., -2.], [5., 7., -4.]], requires_grad=True)
- **Result after adding scalar value 5:** tensor([[7., 4., 8.], [9., 5., 3.], [10., 12., 1.]], grad_fn=<AddBackward0>)
- **Result of element-wise multiplication with multiplier_tensor:** tensor([[-7., 8., 8.], [27., -5., 6.], [10., 24., -1.]], grad_fn=<MulBackward0>)
- **Scalar output (sum of elements):** 70.0
- **Gradient of scalar output with respect to original tensor:** tensor([[-1., 2., 1.], [3., -1., 2.], [1., 2., -1.]])

4. Neural Network Parameters

4.1. Neural Network Architecture and GPU Utilization Report

Initial Network Architecture: The network (Net) is defined with the following structure shown in figure 10:

- conv1: Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))

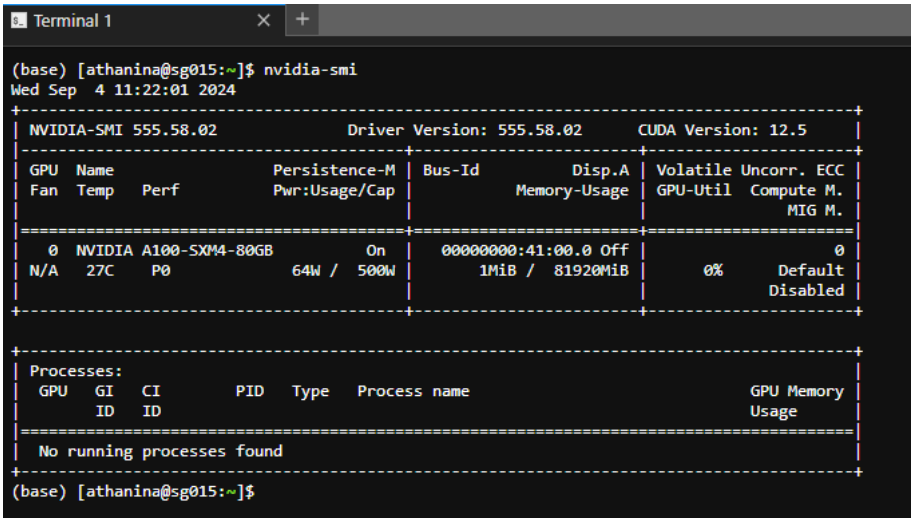

Figure 6. GPU Allocation

Figure 7. Screenshot illustrating the .ipynb file thaninayagam

- conv2: Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
- fc1: Linear(in_features=400, out_features=120, bias=True)
- fc2: Linear(in_features=120, out_features=84, bias=True)
- fc3: Linear(in_features=84, out_features=10, bias=True)

GPU Utilization:

- The code uses CUDA if available: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
- Confirmation of GPU usage: All parameters (conv1.weight, conv1.bias, conv2.weight, etc.) are on device: cuda:0
- Total Parameter Count: The initial network has a total of 61,706 parameters.
- GPU Confirmation: The network is indeed running on the GPU, as evidenced by the 'cuda:0' device assignment for all parameters.

4.2. Modified Neural Network Architecture Report

Original Architecture: The initial network had 61,706 parameters, which was significantly higher than the target of 10,000 as shown in figure 11.

Modification Process:

- I calculated the average number of parameters per layer to understand the distribution.
- I identified that the last three layers (fully connected layers) were easy to manipulate to get the required parameter count.
- I focused on adjusting the output units of the last two fully connected layers to reduce the total parameter count.

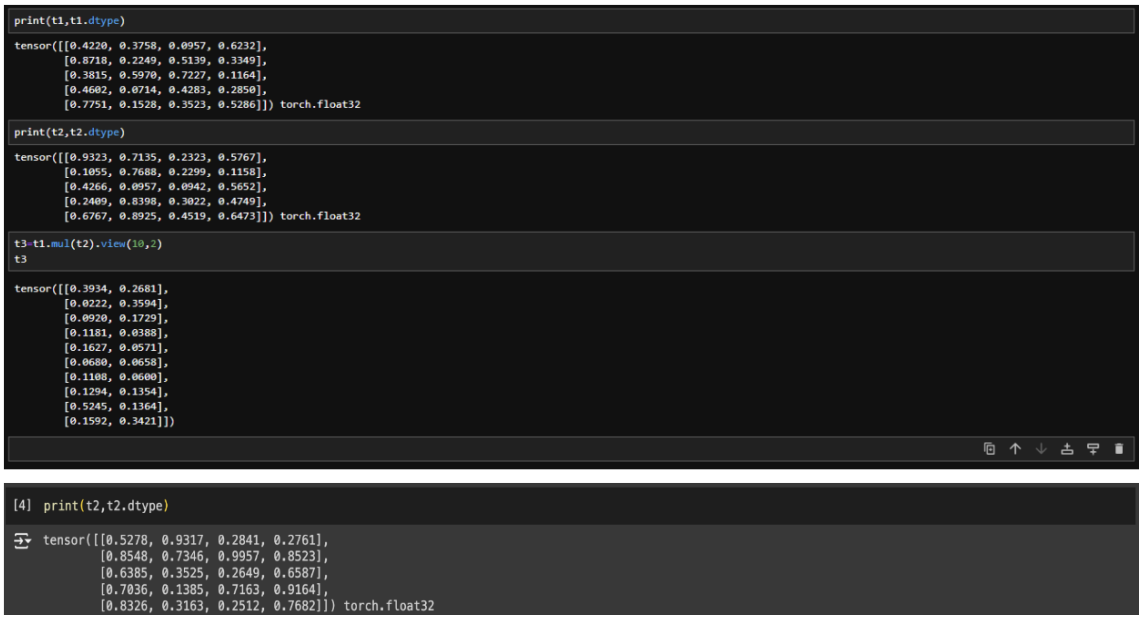

Figure 8. Tensor Operations

Modified Network Architecture: The network (Net) is now defined with the following structure:

- conv1: Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
- conv2: Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
- fc1: Linear(in_features=400, out_features=15, bias=True)
- fc2: Linear(in_features=15, out_features=67, bias=True)
- fc3: Linear(in_features=67, out_features=10, bias=True)

Updated Parameter Count:

The modified network now has a total of 10,004 parameters, which is very close to the target of 10,000.

Modification Strategy: I iteratively adjusted the output units of fc1 and fc2 layers while maintaining the input and output dimensions of the network. This process involved trial and error to find the right balance that would bring the total parameter count close to 10,000 without changing the number of layers in the architecture.

5. Acknowledgements

Chat-GPT was used to help reformat the report.

```
[8]: tensor = torch.tensor([[2,-1,3],[4,0,-2],[5,7,-4]],dtype=torch.float32,requires_grad=True)
     tensor

[8]: tensor([[ 2., -1.,  3.],
             [ 4.,  0., -2.],
             [ 5.,  7., -4.]], requires_grad=True)

[9]: tensor_new= tensor.add(5)
     tensor_new

[9]: tensor([[ 7.,  4.,  8.],
             [ 9.,  5.,  3.],
             [10., 12.,  1.]], grad_fn=<AddBackward0>)

[12]: multiplier_tensor = torch.tensor([[-1,2,1],[3,-1,2],[1,2,-1]],dtype=torch.float32)

[13]: mult = tensor_new.mul(multiplier_tensor)
      print(mult)

      tensor([[-7.,  8.,  8.],
              [27., -5.,  6.],
              [10., 24., -1.]], grad_fn=<MulBackward0>)

[14]: scalar_value = mult.sum()
      scalar_value.item()

[14]: 70.0

[15]: scalar_value.backward()
      print(tensor.grad)

      tensor([[-1.,  2.,  1.],
              [ 3., -1.,  2.],
              [ 1.,  2., -1.]])
```

Figure 9. Autograd Operation

```
[4]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     net.to(device)

[4]: Net(
       (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
       (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
       (fc1): Linear(in_features=400, out_features=120, bias=True)
       (fc2): Linear(in_features=120, out_features=84, bias=True)
       (fc3): Linear(in_features=84, out_features=10, bias=True)
     )

[6]: len(list(net.parameters()))

[6]: 10

[7]: for name, param in net.named_parameters():
         print(f"Parameter '{name}' is on device:", param.device)

     Parameter 'conv1.weight' is on device: cuda:0
     Parameter 'conv1.bias' is on device: cuda:0
     Parameter 'conv2.weight' is on device: cuda:0
     Parameter 'conv2.bias' is on device: cuda:0
     Parameter 'fc1.weight' is on device: cuda:0
     Parameter 'fc1.bias' is on device: cuda:0
     Parameter 'fc2.weight' is on device: cuda:0
     Parameter 'fc2.bias' is on device: cuda:0
     Parameter 'fc3.weight' is on device: cuda:0
     Parameter 'fc3.bias' is on device: cuda:0

[8]: def count_parameters(model):
         return sum(p.numel() for p in model.parameters() if p.requires_grad)

     # Get the number of parameters
     total_params = count_parameters(net)
     print(f"Total number of parameters: {total_params}")

     Total number of parameters: 61706

[ ]:
```

Figure 10. Neural Network Architecture with Parameter Count

```
        s4 = F.max_pool2d(c3, 2)
        # Flatten operation: purely functional, outputs a (N, 400) Tensor
        s4 = torch.flatten(s4, 1)
        # Fully connected layer F5: (N, 400) Tensor input,
        # and outputs a (N, 120) Tensor, it uses RELU activation function
        f5 = F.relu(self.fc1(s4))
        # Fully connected layer F6: (N, 120) Tensor input,
        # and outputs a (N, 84) Tensor, it uses RELU activation function
        f6 = F.relu(self.fc2(f5))
        # Gaussian layer OUTPUT: (N, 84) Tensor input, and
        # outputs a (N, 10) Tensor
        output = self.fc3(f6)
        return output


net = Net()
print(net)

Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=15, bias=True)
  (fc2): Linear(in_features=15, out_features=67, bias=True)
  (fc3): Linear(in_features=67, out_features=10, bias=True)
)

[41]: def count_parameters(model):
          return sum(p.numel() for p in model.parameters() if p.requires_grad)

      # Get the number of parameters
      total_params = count_parameters(net)
      print(f"Total number of parameters: {total_params}")

      Total number of parameters: 10004
```

Figure 11. Modified Neural Network Architecture with 10,000 Parameters