Project 1: DNN Pruning via NNI

Oliver Fowler and Brian Park

North Carolina State University, Computer Science 591/791-025

October 2022

1 Pruning Methods

For this project, we applied pruning models to a simple CNN and a state of the art ResNet model [1]. To warm up, we used a simple `L1NormPruner` [2] from NNI [3] to quickly get familiar with the workflow on a simple CNN provided by PyTorch. We performed this dataset on the MNIST digit dataset, which is a relatively easy dataset to train on for the purposes of quick development and experimental results.

Then we stepped it up a notch and used ResNet-101 to train the weights using the CIFAR-10 dataset, a much more complex dataset than MNIST. We used torchvision's collection of premade models rather than writing the model from scratch [4]. This took longer to train (180 epochs), but we wanted to see the magnitude of effect that structured pruning can have on a very deep neural network. For this network, we tried evaluated three different pruners: `LevelPruner`, `L1NormPruner`, `L2NormPruner` [5, 2]. But we still found `L1NormPruner` to be the best performing in terms of accuracy and performance tradeoffs.

2 Pruning and Hardware Configurations

2.1 Hardware Configuration

For hardware configurations, we fortunately had many devices with different architectures available to us. Not only on the ARC Cluster [6] do we have multiple GPUs, but we were also able to experiment on Brian's M1 MacBook Pro. Brian's M1 MacBook Pro features a 10-Core CPU with 8 performance cores and 2 efficiency cores, and 16-Core GPU with 32GB of unified memory. It also features a 16-core Neural Engine [7]. PyTorch recently announced native M1 support via Apple's Metal Performance Shaders [8]. It's able to perform operations like convolutional kernels optimally on the M1 GPU. Oliver used an Nvidia RTX 3080 on his local desktop computer.

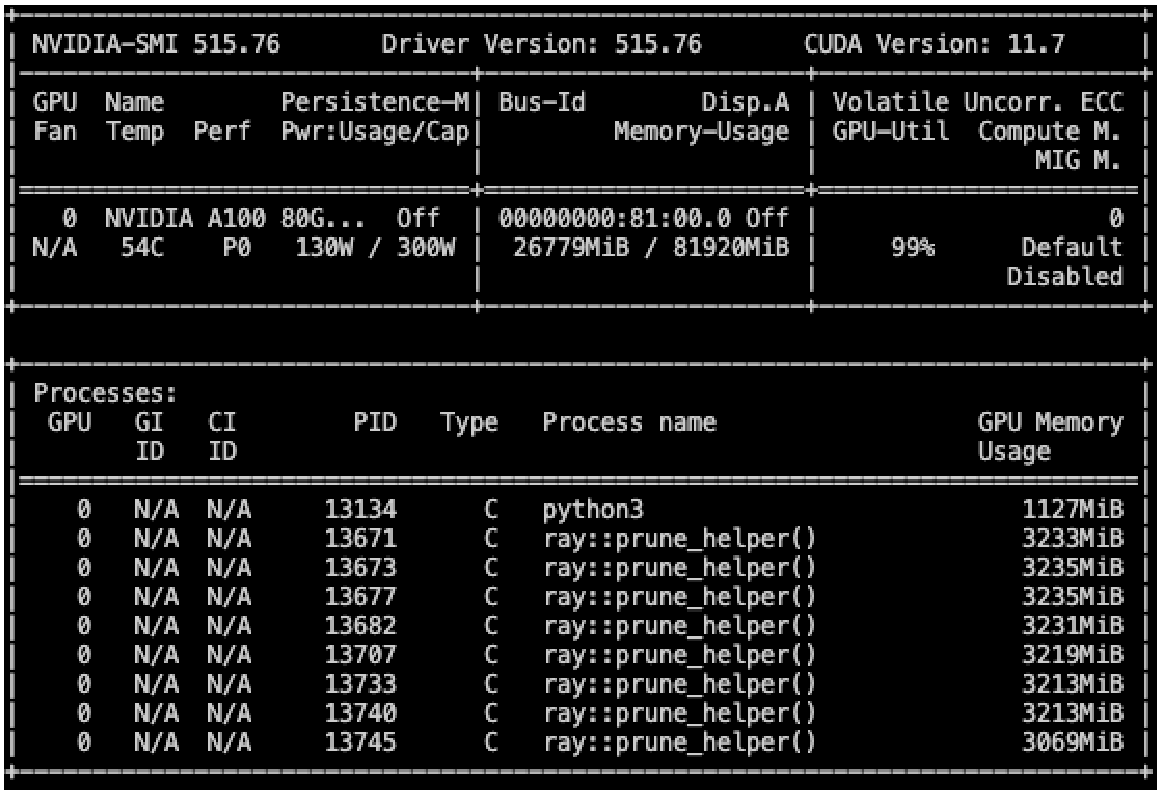
For ARC, we used the NVIDIA A100 GPU [9]. It has 80GB of HBM and it claims 19.5 TFLOPS of FP32 performance. Fortunately, PyTorch enables CUDA specific features some by default, and others by some flags. We were also able to use NVIDIA's custom floating point format, TensorFloat32, as well as use 16 bit floating point numbers which the PyTorch API was aware of.

2.2 Pruning Configuration

For the pruning configurations, we used `L1NormPruner` on the simple MNIST CNN as outlined by the NNI QuickStart Guide. We retrained for 3 more epochs after the pruning step.

For the second model, we chose ResNet-101, which is a much deeper DNN. We wanted to see the effects that residual connections have on deep neural networks. We used the CIFAR-10 dataset as that's a relatively more challenging dataset than MNIST. As for which pruner we used, we brute forced the search for the best pruning method. Training multiple pruning methods on the same dataset is an embarrassingly parallel problem, so we used Ray to exploit this parallelism on the A100 GPU [10]. Ray is a distributed framework for scaling AI and Python applications. Since there is no communication between the training processes, we really could've used any other distributed programming framework. But we chose Ray as it has GPU support, and it's able to support multi process Python execution on a single GPU. We see that once we train in parallel, we can fully utilize the memory usage as well as the GPU utilization as shown in the output of `nvidia-smi` in Figure 1. We retrained for 20 more epochs after the pruning step, since this is a much deeper network.

For both networks, we pruned from 10% to 90% with intervals of 10%. We experimented with pruners such as `LevelPruner`, `L1NormPruner`, `L2NormPruner` [5, 2].



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.76       Driver Version: 515.76       CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100 80G...  Off  | 00000000:81:00.0 Off |                    0 |
| N/A   54C    P0   130W / 300W |  26779MiB / 81920MiB |     99%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     13134      C   python3                          1127MiB |
|    0   N/A  N/A     13671      C   ray::prune_helper()              3233MiB |
|    0   N/A  N/A     13673      C   ray::prune_helper()              3235MiB |
|    0   N/A  N/A     13677      C   ray::prune_helper()              3235MiB |
|    0   N/A  N/A     13682      C   ray::prune_helper()              3231MiB |
|    0   N/A  N/A     13707      C   ray::prune_helper()              3219MiB |
|    0   N/A  N/A     13733      C   ray::prune_helper()              3213MiB |
|    0   N/A  N/A     13740      C   ray::prune_helper()              3213MiB |
|    0   N/A  N/A     13745      C   ray::prune_helper()              3069MiB |
+-----------------------------------------------------------------------------+
```

Figure 1: Peak GPU utilization

3 Experimental Results

The outputs of `print(model)` is shown for Simple CNN and ResNet-18 in Appendix A. For visuals, we outputted the graph of the DNNs as well in Figure 2. Note that although we used ResNet-101 for the report, the print output and computation graph are showing ResNet-18 for readability.

After training each model, we benchmarked both models for their inference time as a baseline. Then we used NNI to prune as aggressively as we could to showcase the performance vs. accuracy degradation trade-off. Unfortunately, we could not run the pruning process on the M1 MacBook Pro as the top k function is not supported with MPS backend for Apple Silicon. Thus we could not run NNI unless we changed the source code to avoid this issue.

The results are shown in Figure 3 for NVIDIA A100 GPU. We report the inference time, training, and validation accuracy. We ran inference models for 25 trials and also plot the variance. We did this to see if there are any effects on cache misses on the GPU, as a higher variance across multiple trials may indicate it.

For M1 Mac, we're able to transfer the ResNet-101 weights from the node with the A100 we trained on. Then we ran just the inference models and got the results as reported in Figure 4. There seems to be less variance for the inference times on the M1 Mac. We don't know the root cause of this difference as the PyTorch backend for M1 is still in early stages of development. Another thing to keep in mind is that we don't see any synchronization primitives for the MPS kernels, and we're not exactly sure if the MPS GPU kernels are synchronous, or asynchronous like CUDA. The A100 inference time is also much more flat and variance is much higher. We suspect that this might be due to overhead of memory transfer of the dataset from main memory to GPU HBM, making this memory bounded for such a small model. We don't see this effect in M1, as the memory between GPU and CPU is *unified*, or shared.

For accuracy, the simple CNN degrades pretty quickly as the network becomes more sparse. This is what we expected to see. But interestingly the accuracy of ResNet-101 actually increases significantly. We're seeing training accuracy increase from 84% to 96% and validation accuracy increase from 79% to 86%. Both of the accuracies then start to degrade as the model becomes more sparse. The only suspicion is that retraining for 20 more epochs helped the model become more accurate. We wish to do more thorough analysis of why this happens, if time permits.

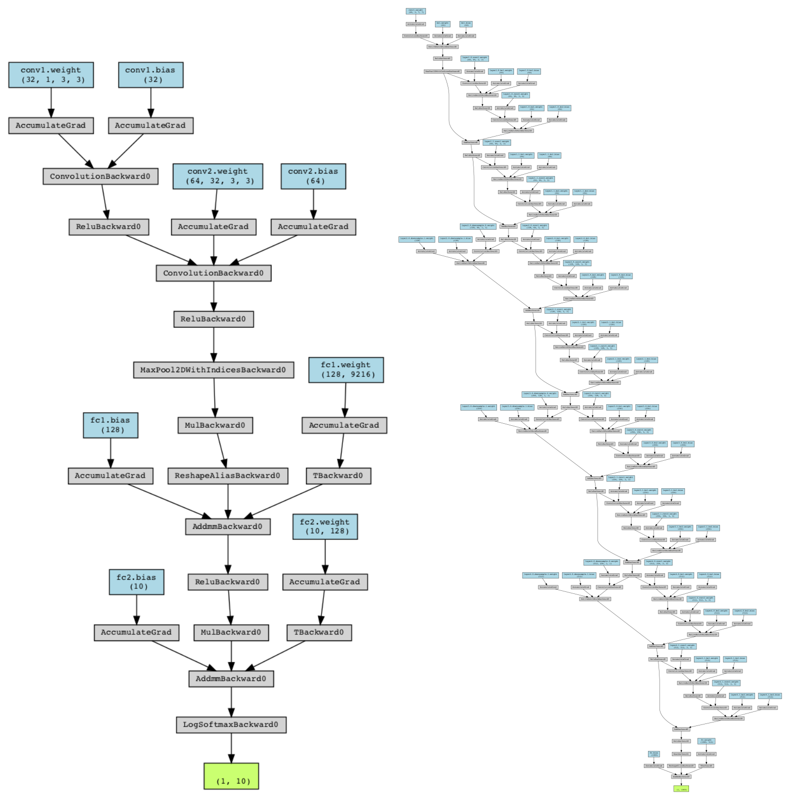


Figure 2: DNN Computation Graphs

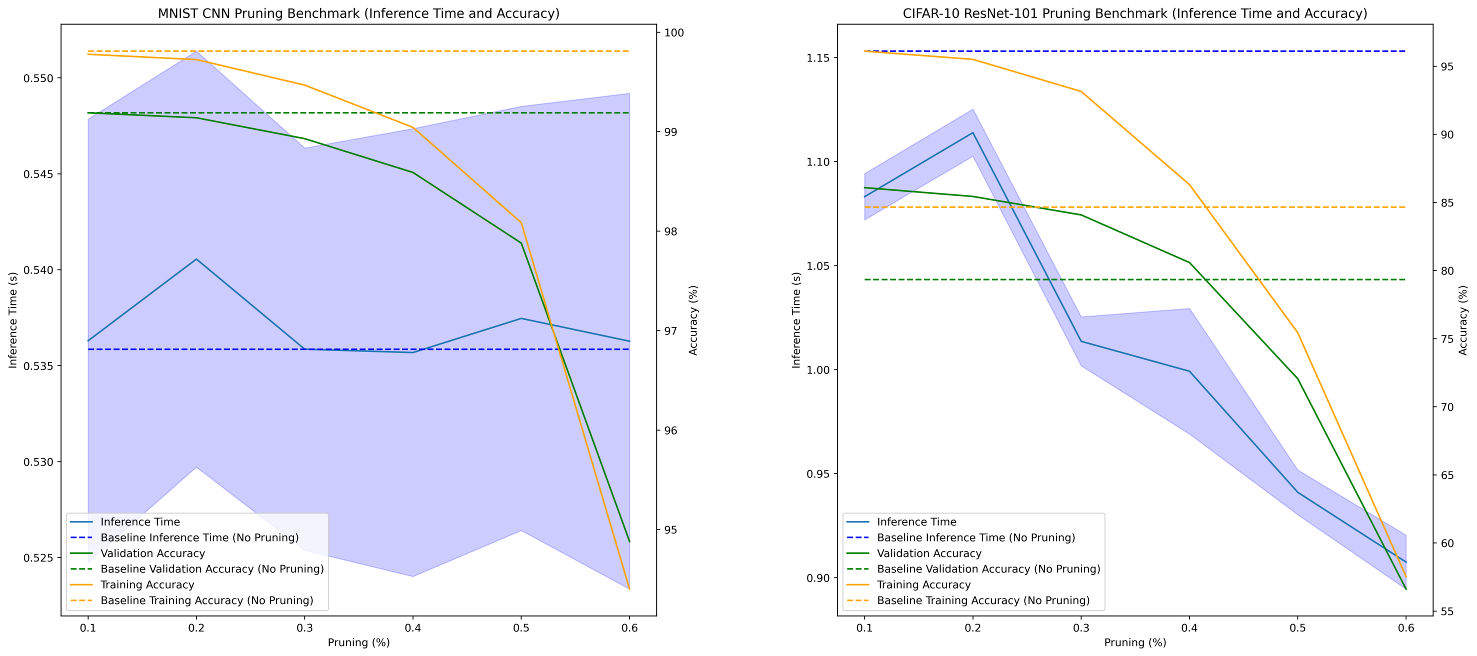


Figure 3: MNIST CNN and CIFAR 10 ResNet-101 Benchmark on NVIDIA A100

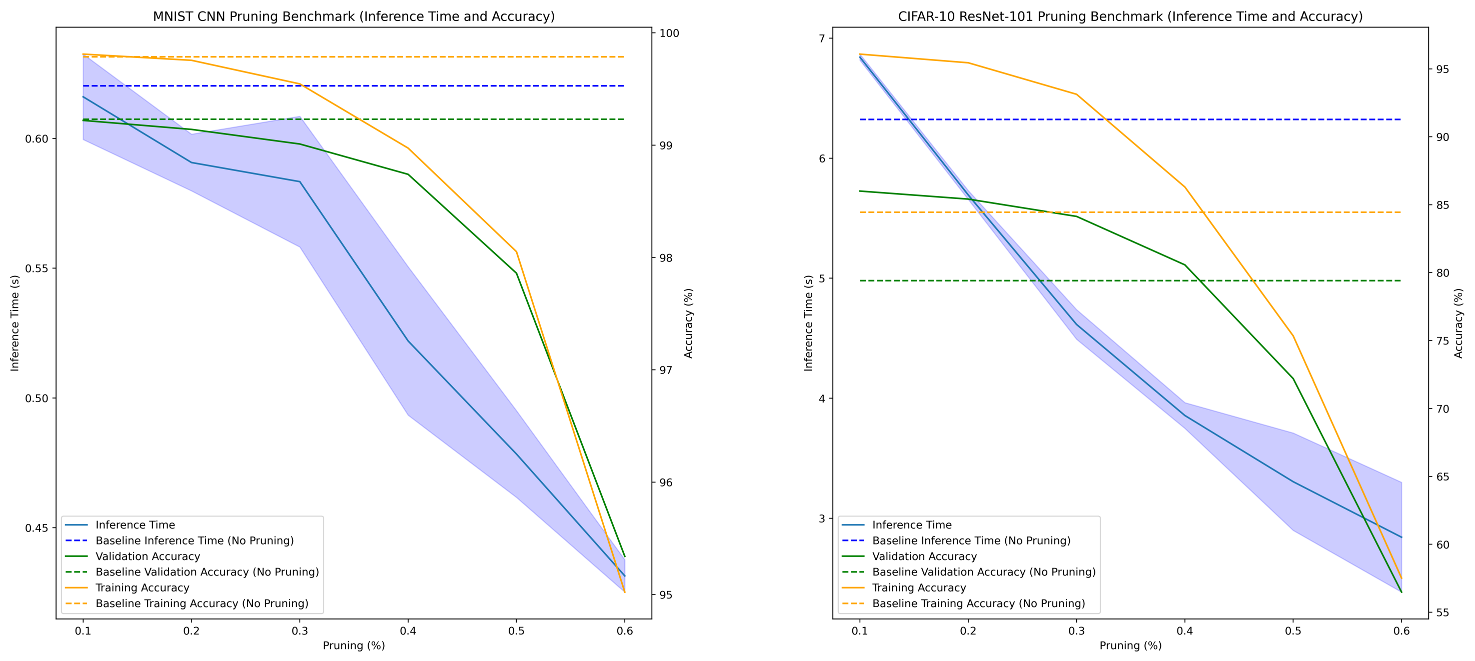


Figure 4: MNIST CNN and CIFAR-10 ResNet-101 Benchmark on M1 MacBook Pro

4 Lessons Learned

We learned how to use the PyTorch API at a much advanced level. Brian was a light PyTorch user, but he has learned how to utilize hardware acceleration via GPU on M1 Mac and CUDA to greatly speedup the training process for SOTA models, as well as other low level PyTorch functionalities.

We also learned that PyTorch also supports pruning, but only *unstructured pruning*. With unstructured pruning, we would not achieve any speedups because it cannot exploit any parallelism. It turns out NNI is able to support structured pruning, thus why we are able to see noticeable speedups.

Although training was the main bottleneck to get things running, we also learned a lot from the effects of pruning. Pruning can have a great effect on inference speed. There's a certain sweetspot where we can achieve speedups until we degrade greatly in validation accuracy. It's also very mysterious how we can initially achieve higher accuracy after pruning. We cannot understand this phenomenon and wonder if our methodology is flawed somewhere, or whether it's actually true that pruning *and* retraining can boost accuracy during the training process.

Lastly, it was our first time using PyTorch with a high end GPU and compare it against a mobile laptop GPU. We found the comparison of today's state of the hardware interesting from what is available in high performance clusters and consumer-end hardware.

5 GitHub Repository

The GitHub repository for this report is publicly accessible here [11]. To reproduce our findings, please read the `README.md` under the `proj1` directory. If there are any setup issues on ARC cluster, please contact bcpark@ncsu.edu.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[2] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2016.

[3] "Neural network intelligence github repository." `https://github.com/microsoft/nni/`.

[4] "Torchvision: Models and pre-trained weights." `https://pytorch.org/vision/stable/models.html`.

[5] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient DNN inference on GPU," *CoRR*, vol. abs/1811.00206, 2018.

[6] "Arc: A root cluster for research into scalable computer systems." `https://arcb.csc.ncsu.edu/~mueller/cluster/arc/`.

[7] "Apple macbook pro specifications." `https://www.apple.com/macbook-pro-14-and-16/specs/`.

[8] "Introducing accelerated pytorch training on mac." `https://pytorch.org/blog/introducing-accelerated-pytorch-training-on-mac/`.

[9] "Nvidia a100 datasheet." `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf`.

[10] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," 2017.

[11] "Csc 791-025 github repository." `https://github.com/briancpark/csc791-025/tree/main/proj1`.

6 Appendix

6.0.1 A1: Layers of Simple CNN for MNIST

```
Net(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (dropout1): Dropout(p=0.25, inplace=False)
  (dropout2): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=4608, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

6.0.2 A2: Layers of ResNet18 for CIFAR-10

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```