IL2230 Lab 3

Linnéa Ridderström, Sofie Franzén,
Daniel Saber Tehrani, Silvia Barrett

December 2021

3.1 Learning From Scratch

We built our own CNN using LeNet as a base. We tried changing a few filter sizes and adding one convolutional layer, producing the code that can be seen in listing 1, to see if we could increase the accuracy. In the end we always got an accuracy of around 62.75%. The network probably performs this poorly because the bees & ants subset of the imagenet dataset does not contain very many images, so the network has a hard time learning. The two classes are also quite similar since they are both insects which might make it harder for the network do distinguish them.

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 12, 5)     # Changed filter sizes
        self.pool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(12, 30, 5)
        self.conv3 = nn.Conv2d(30, 120, 5)
        self.conv4 = nn.Conv2d(120, 300, 5,1,2) # Added layer
        self.fc1 = nn.Linear(300, 84)
        self.fc2 = nn.Linear(84, 2)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = torch.relu(self.conv3(x))
        x = torch.relu(self.conv4(x))
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Listing 1: Our CNN

3.2 Transfer Learning

We chose to import a pretrained ResNet18 to use for transfer learning with:

```python
model = models.resnet18(pretrained=True)
```

Listing 2: Importing ResNet18

In the first step, the model was to be adapted to have an output layer with only 2 outputs instead of 1000. Then All layers except the output layer were to be frozen. The code used to achieve this was:

```python
1 for param in model.parameters():    # Freeze all parameters
2     param.requires_grad = False
3 
4 # Parameters of newly constructed modules have requires_grad=True
      by default
5 num_ftrs = model.fc.in_features
6 model.fc = nn.Linear(num_ftrs, 2)   # Add fc-layer with 2 outputs
```

Listing 3: Freezing and modifying

The model trained for 40 epochs and this resulted in an accuracy of 76.74%.

In the second step, the model was to be fully retrained to obtain an optimized model, so we commented out line 1 & 2 of listing 3 to unfreeze the layers and trained for 40 epochs. The accuracy achieved in this step was 82.35%.

Transfer learning can be useful if the problem one is trying to solve is a subset of a problem already solved. It will then save time as one will not have to train the network from scratch.

3.3 Network Pruning

CNNOptimized was pruned with two different techniques. The first was random pruning of the first convolutional layer as shown in listing 4. We pruned 30% of weights and before training, it gave an accuracy of 54.25%. After training for 20 epochs, the accuracy had been restored to 84.31%.

```python
module = model.conv1    # Choose convolutional layer 1 for pruning
prune.random_unstructured(module, name="weight", amount=0.3)
prune.remove(module, "weight")
```

Listing 4: Random pruning of convolutional layer 1

The second technique was random global pruning. As can be seen in listing 5, we pruned 30% and got an accuracy of 46.41% before training and 81.05% after 20 epochs of training.

```python
parameters_to_prune = ((model.conv1, 'weight'), # Choose all layers
                       (model.fc, 'weight'))
prune.global_unstructured(parameters_to_prune,
                          pruning_method=prune.L1Unstructured,
                          amount=0.3)
```

```python

for module in parameters_to_prune:
    prune.remove(module[0], "weight")
```

Listing 5: Random global pruning

When comparing the inference performance we noted that the pruned models did not take up less space or perform computations faster than the unpruned models. We reckon this is because pruning only replaces weights with zeroes which doesn't affect the size or the number of computations performed.

Pruning has the benefit of making the network smaller and faster but with unstructured pruning there is a risk of removing important connections and damaging the accuracy.

3.4 Data Quantization

Quantization improves the performance gain by reducing the model size, lowering the bandwidth and enables a faster inference by focusing on the computations and memory accesses. Quantization archives this reduction by converting to and using lower precision data, usually 8bit integer (int8) instructions instead of floating point, but at the cost of slightly reduced accuracy. This technique is important because this could be one of the only ways of fitting all resources on a limited device, such as a mobile device, while still getting the needed quality to achieve service goals.

3.4.2 Run Demos and Tutorials

1 Observe the Data-type Changes

When running the `demo0` code provided we first observe that the results are different, `tensor([[-0.5329]], grad_fn=<AddmmBackward0>)` for the 32 bit floating point and `tensor([[-0.5332]])` for the quantized 8 bit integer.

We then observe further differences in the weights. The 32 bit floating point weights are given as `tensor([[-0.0053, 0.3793]]))`, while the quantized 8 bit integer weights are given as `tensor([[-0.0060, 0.3778]], size=(1, 2), dtype=torch.qint8, quantization_scheme=torch.per_tensor_affine, scale=0.0029750820249319077, zero_point=0)`.

2 Dynamic Quantization

Dynamic quantization, just like all quantization techniques, converts the weights to a lower data precision. Most often the conversion is to int8. The difference in dynamic quantization is that the technique that does this also converts the activations to a lower data precision, and the scales and zero points are collected during inference, right before the computation.
For the `ResNet-18` model we recorded the data displayed in table 1.

Table 1: ResNet Dynamic Quantization

	Mean of Output Tensor	Size [KB]	Latency [μs]
Before Quantization	0.12887	3.743	816 ± 9.3
After Quantization	0.12912	2.719	968 ± 155

3 Static Quantization

Static quantization is a technique that quantizes a model's weights and activation functions statically. The weights are converted from float to int, and batches of data are fed through the network to determine the distribution of the activations. The distributions are then used to compute how to quantize the activations prior to inference time.

Table 2: ResNet Static Quantization

	Accuracy [%]	Size [KB]	Latency [s]	Memory Usage [MB]
Before Quantization	94.77	44787.341	60	346.852
After Quantization	47.71	11308.193	52	286.711

For the `ResNet-18` model we recorded the data displayed in table 2.

4 Quantization Aware Training (QAT)

Compared to other quantization techniques, Quantization Aware Training (QAT) gives a higher accuracy effect as it does all of its calculations and training still in floating point; to mimic quantization the technique instead rounds and clamp. Afterwards, the weights and activations are quantized.

Table 3: ResNet QAT

	Accuracy [%]	Size [KB]	Latency [ms]
Before Quantization	39.87	46837.645	726.840
After Quantization	96.73	11222.521	377.900

For the `ResNet-18` model we recorded the data displayed in table 3. Comparing back to Lab 2's runtime for LeNet and the MPL's we see that they were alot faster then our codes today. This maybe because of how much more complex the design has gotten. But we were not expecting some of these adaptations to be as fast as they are.

3.4.3 LeNet Quantization

1 Dynamic Quantization

From the table 4 it can be seen that the model after being dynamically quantized is 1.14 times smaller than before the quantization. This was an expected result. Accuracy was not effected, but memory usage was lowered. The mean values of the output tensors was not effected much. In conclusion the overall performance was not effected noteworthy, probably because the size reduction was not much.

Table 4: LeNet Dynamic Quantization

	Accuracy [%]	Size [KB]	Latency [s]	Memory Usage [KB]
Before Quantization	96.31	249.735	35	368.699
After Quantization	96.31	218.363	40	368.184

Table 5: Mean values of output tensor values for LeNet dynamic quantization

	Mean of Output Tensor
Before Quantization	2.55747
After Quantization	2.55733
Mean Difference	0.01101 (0.48%)

2 Static Quantization

From the table 6, it can be seen that the model after being statically quantized is 3.33 times smaller than before the quantization and that memory usage goes down slightly, but this is at the cost of lower accuracy. But in comparison with Static Quantization, the size was drastically lowered and still keeping a decent performance.

Table 6: LeNet Static Quantization

	Accuracy [%]	Size [KB]	Latency [s]	Memory Usage [KB]
Before Quantization	97.31	249.735	32	346.852
After Quantization	76.73	75.077	51	346.086

3 LeNet Quantization Aware Training (QAT)

Table 7: LeNet QAT

	Accuracy [%]	Size [KB]	Latency [ms]
Before Quantization	86.03	249.799	2.783
After Quantization	97.52	100.591	1.925

In the table 7 we can see the result from running the code before and after fine tuning and quantization. Here we see that the accuracy has increased with

11.49 percentage points, while at the same time the size decreased down to 100 KB. Lastly we also see a speed up where the latency is decreased.