Advanced ML Assignment 2

Erik Dahlberg, er8251da-s

May 2024

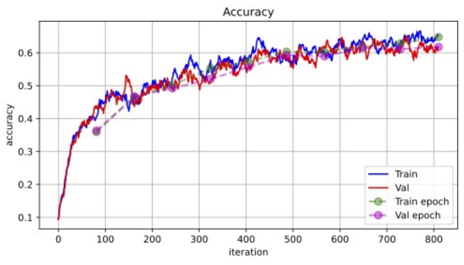
1 Method and Results

I did two models from scratch and two based on the pytorch pretrained models for efficientnet and resnet respectively.

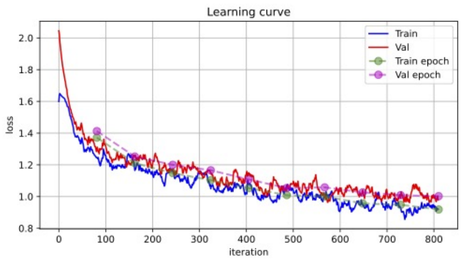
1.1 Scratch-models

My first model was based on a baseline a CNN-architecture created for classification on MNIST data set, see here. I adapted this baseline to the take in more features in it's linear layer. This because I anticipated that flower classification would be more complex than MNISt. However, I only changed the linear input_features because I wanted to maintain use a simple architecture to start with. See Appendix for model.

The first model produced accuracy and loss as in below figures:

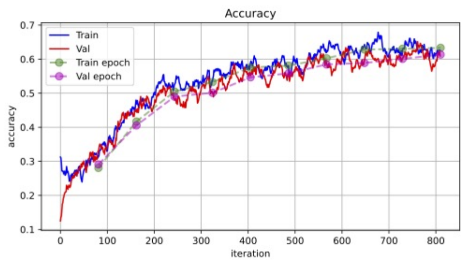


(a) 1s CNN accuracy

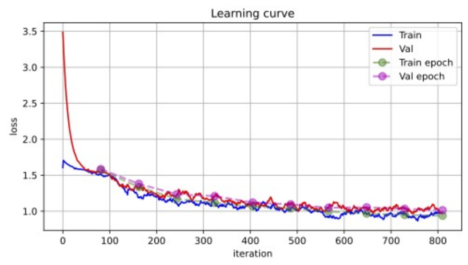


(b) 1s CNN loss

Since the accuracy start to plateau at a low value of 60% accuracy I wanted to increase the complexity. I thus added another convolutional block and increase the output features of each block. See appendix for the second model.



(a) 2nd CNN accuracy

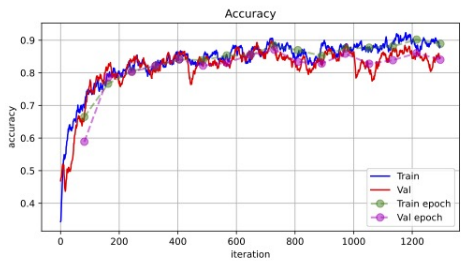


(b) 2nd CNN loss

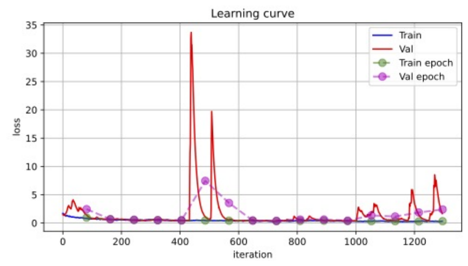
These results are more or less the same as the first. My conclusion was that I needed to increase training time and complexity. Since I don't have a particularly strong computer I decided to continue with transfer-learning and LUNARC.

1.2 Pretrained Models

I started with the efficientnet. In the lecture notes for image classification ResNet was mentioned as a behemoth of image classification but I wanted to try efficientnet since the pytorch documentation states it has similar performance on the ImageNet1K dataset but less resources required. Specifically I chose the V2 L 1K version of EfficientNet.



(a) EfficientNet accuracy

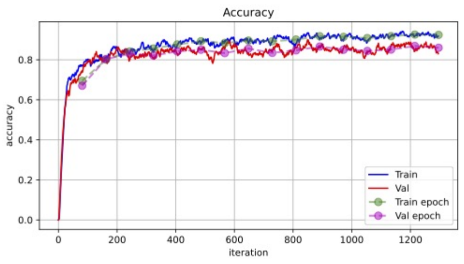


(b) EfficientNet loss

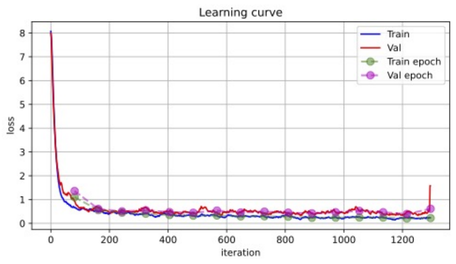
Note that this model adapts faster to the new data and achieves higher accuracy and lower loss.

EfficientNet achieved a macro F1-score of 85.75% with transforms applied.

Since ResNet was mentioned as a behemoth of image classification I created a ResNet model as well. Specifically, I utilised ResNet101 pre-trained on ImageNet1K. When applied to the flowers dataset it produced measurements:



(a) EfficientNet accuracy



(b) EfficientNet loss

The ResNet model achieved a macro F1-score of 86.88%. Compared to the efficientnet the loss-function seems more stable for the ResNet. Moreover, ResNet achieves higher accuracy and lower loss than EfficientNet.

1.3 Data Augmentation and Comparison

In all models, a data transformation

```
T.Compose([
    T.Resize(size=(res,res)),
    T.RandomCrop(size=0.8*res),
    T.RandomHorizontalFlip(),
    T.RandomAffine(degrees=90),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])
```

was applied to the dataset. The transforms were chosen based on prior experience with transforms. These are typical transforms which have produced good results for a lot of previous ML-Engineers. The specific values of mean and std are taken from the documentation for the ResNet101 architecture.

In the test without augmentation ResNet achieved a macro F1-score of 83.76% (lower than Efficient-net with augmentation!).

1.4 Macro F1-score table

I performed F1-score for the pretrained models as they were the only contestants for the top performers.

Model	**Macro F1-Score**
ENet V2 L	0.8575
ResNet101	0.8688
ReSNet101 (No augmentation)	0.8376

2 Discussion

The implementation of the skeleton went well. The difficult part of any ML-project is to interpret results. For me, the scratch implementations give understanding of underlying models but are difficult to interpret. When they produce a result of 60%, what does it mean? With more time allocated, I would have developed more tests to evaluate bias and variance in the model. This way, I would iteratively increase complexity while evaluating for bottlenecks.

3 Summary

My main take-aways were:

- how to structure a flexible ML protocol. I really enjoyed adapting the skeleton and seeing how a series of scripts can be organised to optimise easy testing of multiple machine learning implementation and their configuration.
- how PyTorch tests can differ when utilising transfer learning. EfficientNet and ResNet had similar performance in the PyTorch documentation, but ResNet outperformed EfficientNet with a substantial margine in all my tests.
- how hands on cloud computing enable a different way of testing ML-implementation. The most amazing feature is by far the email notification (which I have now added to my own thesis project)
- how to use LUNARC and navigate the linux-based system.

I've had previous experience with augmentations, CNNs and pretrained models so that was good repetition but not part of my main take-aways.

4 Feedback

All in all, this was a great lab. Maybe a clearer description of how to use LUNARC would be good. I had no experience with cloud computing outside of google colab and it took me a long time to understand how to run the different commands. Also the information in the documentation is quite spread out. I found myself searching for information I knew I had read somewhere on the website multiple times, which was difficult to relocate. That was a bit annoying.

5 Appendix

5.1 First Scratch CNN

```
class YourFirstNet(torch.nn.Module):
def __init__(self, n_labels):
    super(YourFirstNet, self).__init__()

    # Conv-block
    self.conv1 = nn.Conv2d(in_channels=3, out_channels=20, kernel_size=(5, 5))
```

```
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

        # Conv-block
        self.conv2 = nn.Conv2d(in_channels=20, out_channels=50, kernel_size=(5, 5))
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

       # Linear-Block
        self.fc1 = nn.Linear(in_features=54450, out_features=33)
        self.relu3 = nn.ReLU()

        # Linear-Block
        self.fc2 = nn.Linear(in_features=33, out_features=n_labels)
        self.logSoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.maxpool1(self.relu1(self.conv1(x)))

        x = self.maxpool2(self.relu2(self.conv2(x)))

        x = torch.flatten(x, start_dim=1)
        x = self.relu3(self.fc1(x))

        x = self.fc2(x)

        output = self.logSoftmax(x)
        return output
```

5.2 Second Scratch CNN

```
class YourSecondNet(torch.nn.Module):
    def __init__(self, n_labels):
        super(YourSecondNet, self).__init__()

        # Conv-block
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=50, kernel_size=(5, 5))
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

        # Conv-block
        self.conv2 = nn.Conv2d(in_channels=50, out_channels=100, kernel_size=(5, 5))
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

        self.conv3 = nn.Conv2d(in_channels=100, out_channels=75, kernel_size=(5, 5))
        self.relu3 = nn.ReLU()
        self.maxpool3 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
```

```
   # Linear-Block
    self.fc1 = nn.Linear(in_features=14700, out_features=500)
    self.relu = nn.ReLU()

    # Linear-Block
    self.fc2 = nn.Linear(in_features=500, out_features=n_labels)
    self.logSoftmax = nn.LogSoftmax(dim=1)

def forward(self, x):
    x = self.maxpool1(self.relu1(self.conv1(x)))

    x = self.maxpool2(self.relu2(self.conv2(x)))

    x = self.maxpool3(self.relu3(self.conv3(x)))

    # print(x.shape)
    x = torch.flatten(x, start_dim=1)
    x = self.relu(self.fc1(x))

    x = self.fc2(x)

    output = self.logSoftmax(x)
    return output
```