Deep Learning Project 1: Image Classification

Objective of the document: Details about the process of implementing algorithms for image classification.

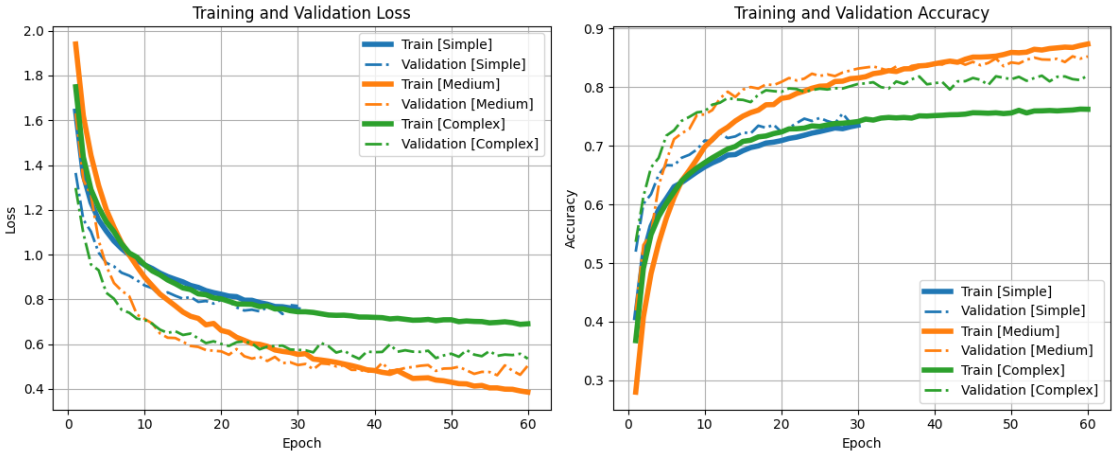
Part I: Models from scratch

The idea behind iterating over the model's complexity and hyperparameters is to gain insight into the complexity of the problem. If a simple model can solve the problem, the iterations should focus on tuning hyperparameters; otherwise, the focus should be on finding the proper model structure.

After various iterations I will comment three.

1. **The simple one**: No resizing was applied (although I initially resized the images to 32x32 in the code, which was unnecessary). This model consists of only two chains of operations, such as `self.pool1(self.relu1(self.conv1(x)))`, followed by a final affine linear transformation like `nn.Linear(128, 10)`. This approach provided the insight that the model learns quite easily, reaching an accuracy of up to 80%. Based on this, I decided to increase the model's complexity and reduce the learning rate.
2. **The complex one**: In this iteration, I maintained the original image size but increased the complexity of each layer and added an extra layer, making it three in total. Each layer follows the structure `self.pool1(self.bn1(self.relu1(self.conv1(x))))`, with a final affine linear transformation like `nn.Linear(512, 10)`. However, this approach resulted in a model that was too complex to effectively learn from the data.
3. **The medium one**: For this iteration, I used a larger image size of 64x64, added an extra layer, and modified the order of operations within each layer. This new model performed better than the previous ones, but its learning stopped after approximately 30 epochs.

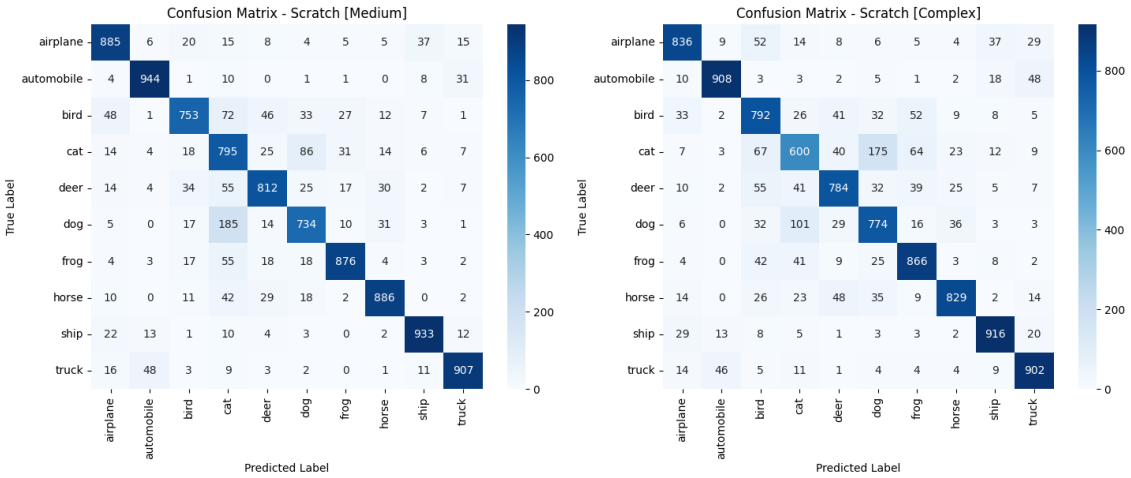
Figure 1: Train and Validation / Loss and Accuracy



The conclusions of this part are:

1. Approximately 60 epochs should be sufficient to reach 90% accuracy.

Figure 2: Confussion Matrix for from Scratch



2. With just images of 32x32 it can be possible to reach 80% accuracy.
3. As the image size increases, the model's complexity should also increase.
4. Since the algorithm learns quickly during the initial epochs, implementing a learning rate schedule or decay would be beneficial.

Part II: Transfer learning for image classification

1. **MobileNet**: As suggested, I started with this architecture, which gave me a glimpse of how easy it is to apply transfer learning.
   (a) **With a size of 224x224**: The first iteration was run using only the CPU, and each epoch took 25 minutes. Surprisingly, it easily matched the performance of the model I had trained from scratch. The key aspect of this iteration was that I resized the images to 224x224 and used the following optimizer: `optim.SGD(model.parameters(), lr=0.001, momentum=0.9)`. To my surprise, the model got stuck after a few epochs, so I decided to tweak the hyperparameters.
   (b) **With a size of 64x64**: The next iteration used a GPU, reducing the training time per epoch to 3 minutes. I noticed that the GPU was not fully utilized, so I increased the batch size to 256. Additionally, increasing the learning rate allowed for a more complex classification layer.
      i. **First 30 epochs**: Only the last classification layer was unfrozen, meaning all pre-trained layers remained frozen.
      ii. **Last 30 epochs**: All layers were unfrozen. The transition was performed manually in a Jupyter notebook cell, so the code used for this transition is not included here.
   (c) This transition improved performance and accelerated training over the epochs.
2. **ResNet**: This architecture comes in different levels of complexity. In this case, I used ResNet18 from the tutorial and ResNet50 from the official documentation.

(a) **ResNet18 - Fine-tuning (Tutorial Part 1)**: The tutorial suggests that this model is intended for small datasets. This might explain why the model got stuck after the initial epochs.

(b) **ResNet18 - Feature Extractor (Tutorial Part 2)**: Freezing all layers except the last one worsened the learning process. This was not surprising, given the behavior I had observed in the previous experiment. In both tutorial parts, images of size 224x224 were used.

(c) **ResNet50**: After experimenting with simpler models, I switched to ResNet50, applying the same strategy—freezing all layers except the last ones and later unfreezing them.

i. **First 30 epochs**: Only three residual blocks were unfrozen:

```
for param in model.parameters():
    param.requires_grad = False

for param in model.layer4.parameters():
    param.requires_grad = True

for param in model.fc.parameters():
    param.requires_grad = True
```

ii. **Last 30 epochs**: All layers were unfrozen. The transition was performed manually in a Jupyter notebook cell, so the code used for this transition might not be consistent in the final script.

iii. Again, this transition improved the model's performance.

3. **EfficientNet**: According to the documentation, this model is designed for small images, so I did not resize the original images. Unfortunately, with the hyperparameters I used, it performed worse than the other models. Additionally, my GPU quota was exhausted, so I could not run further iterations.

Figure 3: Train and Validation / Loss and Accuracy

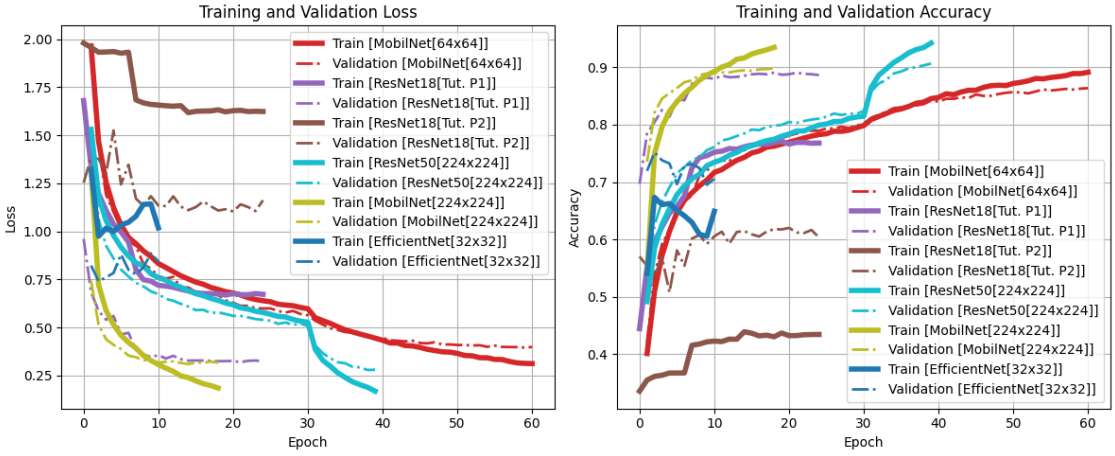
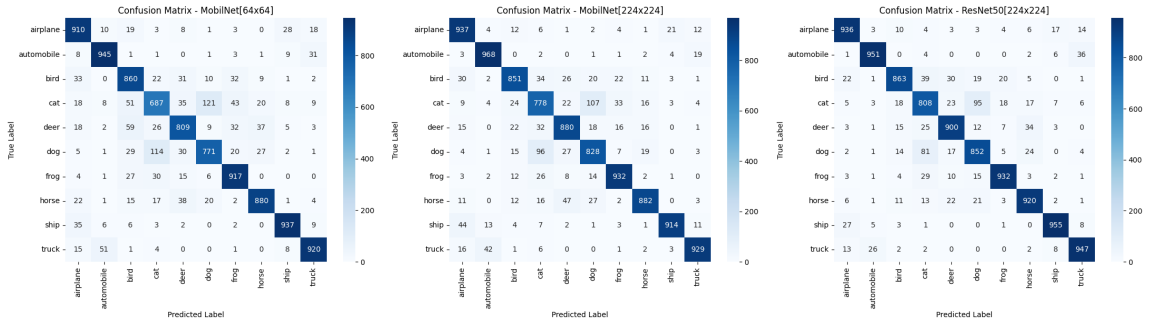


Figure 4: Confussion Matrix for pre-trained models



The conclusions of this part are:

1. MobilNet performed well even for small and bigger images.
2. Resnet50 performed better, that was expected since it is a complex model.
3. The strategy of freezing almost all layers then unfreezing them after some epochs worked pretty well.

Part III: Prediction Among all models three of them perform quite similar, MobilNet64x64, MobilNet224x224 and Resnet50. For practice purposes I've decided to use them at once to make the prediction or final model. The strategy to follow is by vote, if there is a tie (all of them indicates different classifications) the Resnet50 is preferred.

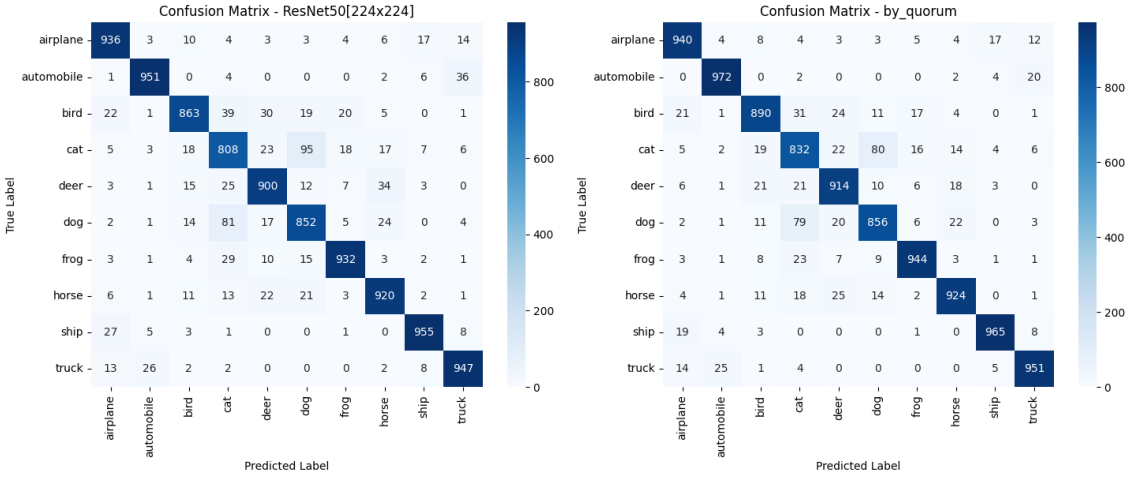
The idea is as follows: `predict_only_labels` returns the file names (for further ordering) and the predictions of the label. Then I apply the vote with the following code:

```
def predict_by_quorum_only_labels(models_with_loaders):
    predictions = [(predict_only_labels(model_iteration, model_loader), w
    final_predictions = []
    for ((a, b), c) in zip(zip(predictions[0][0][1], predictions[1][0][1])
        if a == b or b == c:
            final_predictions.append(b)
        elif a == c:
            final_predictions.append(a)
        else:
            final_predictions.append(b)
    final_predictions = np.array(final_predictions)
    return predictions[0][0][0], final_predictions
```

The result of this approach can be shown in the next table, when all three of them participant the accuracy reaches to 91.88%. The confusion matrix also is improved, I will show it side to side against ResNet50.

Model	Accuracy
MobilNet[64x64]	86.36%
MobilNet[224x224]	89.82%
ResNet50[224x224]	90.64%
All three of them by vote	91.88%

Figure 5: Confussion Matrix comparing ResNet50[224x224] vs (MobilNet 64x64, MobilNet 224x224, Resnet50 224x224)



With that final predictor I completed the predictions of unlabelled images.

Part IV: Code with comments All code used for experiments mentioned, and the scripts to generate this report are in this repository of GitHub https://github.com/vmchura/practice_pytorch_imag I will mention some relevant parts of the code.

1. From Scratch - Medium

Converting the input image from 64x64 through the steps 64, 32, 16, 8, 4. For convenience, I will use just padding=1.

- With pool1, relu1, bn1, conv1 the image change from 3x64x64 to 64x32x32.
- With pool2, relu2, bn2, conv2 the image change from 64x32x32 to 128x16x16.
- With pool3, relu3, bn3, conv3 the image change from 128x16x16 to 256x8x8.
- With pool4, relu4, bn4, conv4 the image change from 256x8x8 to 512x4x4.
- Finally is flattened to a linear dimension of 8192 (512x4x4).

```
class ImageClassifier(nn.Module):
    def __init__(self, num_classes=10):

        super(ImageClassifier, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(512)
        self.relu4 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(512 * 4 * 4, 1024)
        self.dropout1 = nn.Dropout(0.5)
        self.relu5 = nn.ReLU()
        self.fc2 = nn.Linear(1024, 512)
        self.dropout2 = nn.Dropout(0.5)
        self.relu6 = nn.ReLU()
        self.fc3 = nn.Linear(512, num_classes)
```

2. MobilNet 64x64

After some iterations:

- Adding complexity with a new fully connected layer.
- Adding generalization with Dropout(0.5)
- Dropout(0.5) it was too much, changed to Dropout(0.3)

```
model = torch.hub.load('pytorch/vision:v0.10.0', 'mobilenet_v2', weights=models.MobileNet_V2_Weights.DEFAULT)

model.classifier[1] = nn.Sequential(
    nn.Linear(model.classifier[1].in_features, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, len(train_dataset.classes))
)
model = model.to(device)

optimizer_ft = optim.SGD(model.parameters(), lr=0.002, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```