CVPR 2023 Project Report - CNN Classifier

Michele Scomina

January 21, 2024

Contents

1 Problem Statement **2**

2 Tools, libraries and structure **3**

3 Base reference model **4**

3.1 Implementation of the model 4

3.2 Training of the model 5

3.2.1 Dataset loading 5

3.2.2 Training 5

3.2.3 Results 6

4 Optimized model **7**

4.1 Data augmentation 8

4.2 Regularization 9

4.3 Architectural changes 9

5 Transfer learning **11**

5.1 Implementation of the model 11

5.2 Fine-tuned classifier 12

5.3 SVM classifier 13

References **15**

1 Problem Statement

The project requires the implementation of an image classifier based on convolutional neural networks. The provided dataset (from [Lazebnik et al., 2006]) contains 15 categories (office, kitchen, living room, bedroom, store, industrial, tall building, inside city, street, highway, coast, open country, mountain, forest, suburb), and is already divided in training set and test set.[1] An example of each category is shown in Fig. 1.

Figure 1: Examples of images from each of the 15 categories of the provided dataset (the same as [Lazebnik et al., 2006]).

The tasks of the given project can be divided in three parts:

1. **Base reference model**: Initialization and training of a CNN model with a fixed architecture, without applying any data augmentation.
2. **Optimized model**: Optimization of the previous result by making use of data augmentation techniques and different architectures.
3. **Transfer learning**: Improvement on the previous results through transfer learning, by fine-tuning a pre-trained model to the given dataset and by using the pre-trained model as a feature extractor for a SVM classifier.

[1] Of 1500 and 3000 images respectively.

2 Tools, libraries and structure

The project has been developed in Python, using the following libraries:

- **PyTorch** for the implementation of the CNN models.
- **Scikit-learn** for the implementation of the SVM classifier.
- **Matplotlib** for the visualization of the results.
- **Seaborn** for the visualization of the confusion matrix.
- **Scipy** for the estimates of the confidence intervals.

The project has also been designed to be as modular as possible, in order to allow easy testing of different architectures and hyperparameters. There might then be some differences with the displayed code.[2]
The main folders of the project are:

- **dataset/**: contains the dataset, respectively in /train/ and /test/.
- **models/**: contains the code for the implementation of the CNN models, as well as the saved models.
- **plots/**: contains the visualization of the results through graphs and confusion matrices.
- **tasks/**: contains the code for the implementation of the tasks.
- **utils/**: contains various helper libraries, like the training and plot code.

The project can be run through the **main.py** file, which will train the models and save the various losses and accuracies graphs during training in the **plots/** folder, as well as save the trained models in the **models/** folder and the test accuracies in **results.txt**.
All of the code is available on GitHub at `https://github.com/MScomina/computer-vision-project`.

Label	Description	Label	Description	Label	Description
0	Bedroom	5	Inside city	10	Open country
1	Coast	6	Kitchen	11	Store
2	Forest	7	Living room	12	Street
3	Highway	8	Mountain	13	Suburb
4	Industrial	9	Office	14	Tall building

Table 1: Dataset labels.

[2]More specifically, some of the variables in this report have been replaced with constants for the specific task, for ease of readability.

3 Base reference model

The first task of the project consists in training a CNN model with the following architecture:

#	type	size
1	Image Input	64x64x1
2	Convolution	8 3x3 convolutions with stride 1
3	ReLU	
4	Max Pooling	2x2 max pooling with stride 2
5	Convolution	16 3x3 convolutions with stride 1
6	ReLU	
7	Max Pooling	2x2 max pooling with stride 2
8	Convolution	32 3x3 convolutions with stride 1
9	ReLU	
10	Fully Connected	15
11	Softmax	softmax
12	Classification Output	crossentropyex

Table 2: Layout of the base reference model.

The model has to be initialized with starting weight values sampled from a normal distribution with mean 0 and standard deviation 0.01 and no bias. The model has to then be trained on minibatches of size 32 on a 85/15 train/validation split.

3.1 Implementation of the model

This architecture can be implemented in PyTorch by creating a torch.nn.Module class with the following layers:

```
import torch.nn as nn
self.layers = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=8, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(in_features=8192, out_features=15)
)
```

The number of final features can be calculated from the following formula, considering that every 2x2 max pooling layer with stride 2 halves the channels' sizes:

$$\text{in\_features} = \left(\frac{\text{image\_size}}{4}\right)^2 \cdot \text{n\_of\_final\_kernels} = \left(\frac{64}{4}\right)^2 \cdot 32 = 8192$$

The Convolutional and Linear layers are then initialized with the following code:

```
if type(m) == nn.Conv2d or type(m) == nn.Linear:
    nn.init.normal_(m.weight, mean=0.0, std=0.01)
    if m.bias is not None:
        nn.init.constant_(m.bias, 0.0)
```

3.2 Training of the model

3.2.1 Dataset loading

First, the transformations to be applied to the images are defined:

```
from torchvision import transforms, datasets
transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x * 255)
])
```

This transformation rescales the images anisotropically to a size of 64x64, converts them to grayscale[3], converts them to a tensor and then scales the values back to [0,255][4].

The dataset is then loaded into three DataLoaders (train, validation and test) by employing ImageFolder, using the previously defined transformations:

```
from torch.utils.data import DataLoader
from datasets import ImageFolder

full_training_data = ImageFolder(root="dataset/train", transform=transform)
test_dataset = ImageFolder(root="dataset/test", transform=transform)

train_size = int(0.85 * len(full_training_data))
val_size = len(full_training_data) - train_size
train_dataset, val_dataset = torch.utils.data.random_split(full_training_data, [
    train_size, val_size])

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)
```

3.2.2 Training

For the choice of hyperparameters, the following values have been used:

- **Learning rate**: 0.001. This value has been chosen because it is commonly used in CNNs (because it's not too big to overshoot nor too small to slow down the training too much).
- **Momentum**: 0.9. This value helps the model converge faster and avoid getting stuck in local minima.
- **Patience**: 10. This value describes the number of epochs without improvement after which the training is stopped. 10 epochs were chosen as a reasonable period for the model to converge.

After initializing the model, the optimizer (stochastic gradient descent) and the loss function (cross entropy) are defined:

```
model = CNN_task_1()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
loss_function = nn.CrossEntropyLoss()
```

[3]since ImageFolder loads them as RGB images, therefore with 3 channels
[4]since ToTensor converts the image to a tensor with values in the range [0,1]

The model is then trained using the following code:

```
model.train()
for x, y in iter(train_loader):
    optimizer.zero_grad()
    y_pred = model(x)
    l = loss_function(y_pred, y)
    l.backward()
    optimizer.step()
```

The training and validation losses and accuracies are calculated at each epoch, and the validation loss is used as a stopping criterion (if it doesn't improve for 10 epochs, the training is stopped):

```
model.eval()
loss = 0
correct = 0
total = 0
with torch.no_grad():
    for x, y in iter(loader):
        y_pred = model(x)
        l = loss_function(y_pred, y)
        loss += l.item()
        _, predicted = torch.max(y_pred.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()
loss /= len(loader)
accuracy = correct / total
```

3.2.3 Results

The model was trained for a maximum of 300 epochs. The training and validation losses and accuracies are as follows:

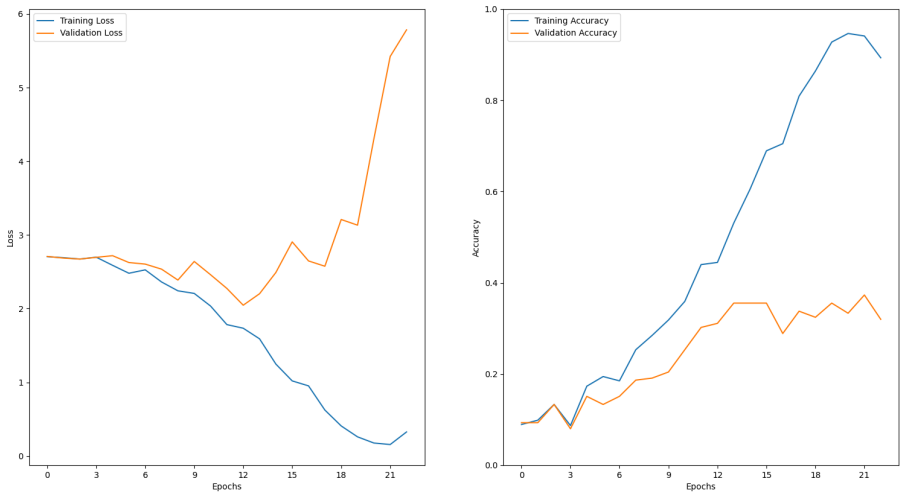

Figure 2: Training and validation losses and accuracies of the base reference model.

As expected, without using any data augmentation techniques or batch normalization, the model quickly starts overfitting just after a few epochs.

The confusion matrix of the model on the test set is as follows:

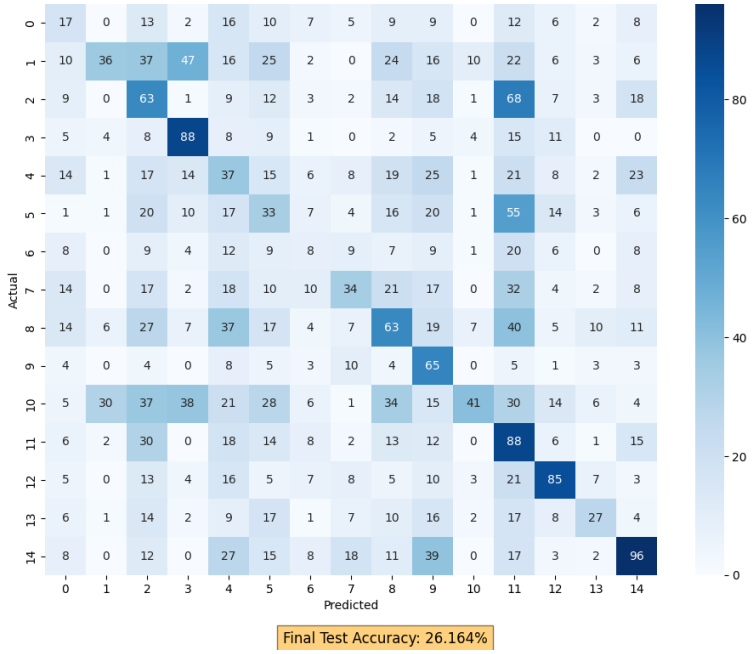

Figure 3: Confusion matrix of the base reference model on the test set.

The average test accuracy of the model is 27.9±1.9%[5]
From the confusion matrix, it's clear that the model primarily specialized in the easiest categories, like *tall buildings* (label 14)[1]. It's likely that the model wasn't able to learn the more complex categories because of the lack of data augmentation techniques and batch normalization, and therefore the model just fitted the training set without generalizing well to the test set.
Nonetheless, the model performed better than a random classifier, which would have an expected accuracy of approximately 7%.

4 Optimized model

The second task of the project consists in improving the previous result by applying data augmentation techniques and using different architectures. The following techniques have been used:

- **Data augmentation**: Data augmentation can be applied to the training set in order to improve the generalization of the model by effectively providing more data for training.
- **Regularization**: Regularization techniques can be applied to the model in order to prevent overfitting and further improve the generalization of the model.
- **Architectural changes**: The architecture of the model can be changed in order to improve the performance.

[5]95% confidence interval over 10 runs using the t-distribution.

4.1 Data augmentation

One of the main issues of the previous model is that it overfits the training set very quickly. This is due to the fact that the dataset is relatively small (only 1500 images for 15 categories). In order to improve the generalization of the model, data augmentation techniques can be applied to the training set. The following data augmentation techniques have been applied:

- **Random horizontal flip**: The image is flipped horizontally with a probability of 0.5.
- **Random rotation**: The image is rotated by a random angle between -20 and 20 degrees.
- **Random crop**: The image is randomly cropped first into a 160x160 image, and then resized to 64x64 with a probability of 0.33.
- **Random noise**: The image is perturbed with Gaussian noise with mean 0 and a standard deviation 0.02.
- **Data normalization**: The image is normalized with mean 0.5 and standard deviation 0.5.

These data augmentation techniques can be easily added to the dataset by modifying the transformations of the training set specifically:

```
data_augmentation_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5)
    transforms.RandomRotation(degrees=20),
    transforms.RandomChoice([
        transforms.Resize((64, 64)),
        transforms.Resize((64, 64)),
        transforms.Compose([
            transforms.RandomCrop(160),
            transforms.Resize(64)
        ])
    ]),
    transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Lambda(lambda x: torch.clamp(x + 0.02*torch.randn_like(x),min=0.,max
    =1.)),
    transforms.Normalize(mean=[0.5], std=[0.5])
])
```

These data augmentation techniques enhance the performance of the model by improving its generalization. Data augmentation does present some drawbacks, though:

- The model requires more training time and a higher patience value of 30, since it takes longer to train and converge on the augmented data.
- The learning rate has to be lowered to 0.0003, since the model starts to experience the "dying ReLU" problem.[6]

The average test accuracy of the model with data augmentation is 43.7±2.5%, which is a significant improvement over the previous model. The model also stops overfitting on the training set.

[6]The "dying ReLU" problem is a phenomenon that occurs when a lot of the neurons of the model always output 0, due to the ReLU function being 0 for negative values, slowing down the training of the model.

4.2 Regularization

Another way to improve the generalization of the model is to apply regularization techniques. The following regularization techniques have been applied:

- **Batch normalization**: Batch normalization is a technique that normalizes the output of the previous layer by subtracting the batch mean and dividing by the batch standard deviation. This technique helps the model to converge faster and to not get stuck in local minima. [Ioffe and Szegedy, 2015]
- **Dropout**: Dropout is a technique that randomly sets a fraction of the input units of a layer to 0 during training. The dropout rate has been set to 0.2 due to the model not overfitting with data augmentation, therefore not requiring a high dropout rate.

These regularization techniques can be easily added to the model by changing the layers of the model:

```
nn.Conv2d(in_channels=n, out_channels=m, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(m),
nn.ReLU(),
nn.Dropout(p=0.2),
nn.MaxPool2d(kernel_size=2, stride=2),
```

The final average test accuracy of the model with data augmentation and regularization is 54.2±1.9%. The model also converges much faster than using only data augmentation.

4.3 Architectural changes

The final way to improve the performance of the model is to change its architecture. The following architectural changes have been applied:

- **Swish activation**: Implementation of the Swish function, which is defined as follows:

  $$\text{Swish}(x) = x \cdot \sigma(\beta x)$$

  where $\sigma(x)$ is the sigmoid function and β is a hyperparameter. This function effectively solves the "dying ReLU" problem due to its non-zero gradient for negative values. [Ramachandran et al., 2017] For this specific model, $\beta = 0.8$ has been chosen.
- **Bigger kernel sizes**: Use of a 7x7 kernel size in the first convolutional layer and 5x5 on the second one.
- **More layers**: The addition of another 3x3 convolutional layer with 64 output channels and another max pooling layer allows more complex patterns to be learned.
- **Kaiming initialization**: Use of the Kaiming initialization [He et al., 2015] instead of the normal initialization. Although the Kaiming initialization has been designed for ReLU, it works well with Swish too due to its similar shape.[7]
- **Ensemble of models**: Use of an ensemble of 10 models, each trained on the dataset indipendently of each other. The final prediction is the average of the predictions of the 10 models. [Szegedy et al., 2015]

[7] In fact, $\beta \to \infty \implies \text{Swish}(x) \to \text{ReLU}(x)$.

The Swish function has to be manually implemented in PyTorch as follows, and then used as an activation function in place of ReLU:

```
class Swish(nn.Module):
    def __init__(self, beta=1.0):
        super().__init__()
        self.beta = beta
    def forward(self, x):
        return x * torch.sigmoid(self.beta * x)
```

The Kaiming initialization can be applied to the model by changing the initialization of the layers as follows:

```
def _init_weights(self, m):
    if type(m) == nn.Conv2d or type(m) == nn.Linear:
        if type(m) == nn.Conv2d:
            nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
        else:
            nn.init.kaiming_normal_(m.weight, mode="fan_in", nonlinearity="relu")
```

The model's training and validation losses and accuracies are as follows:

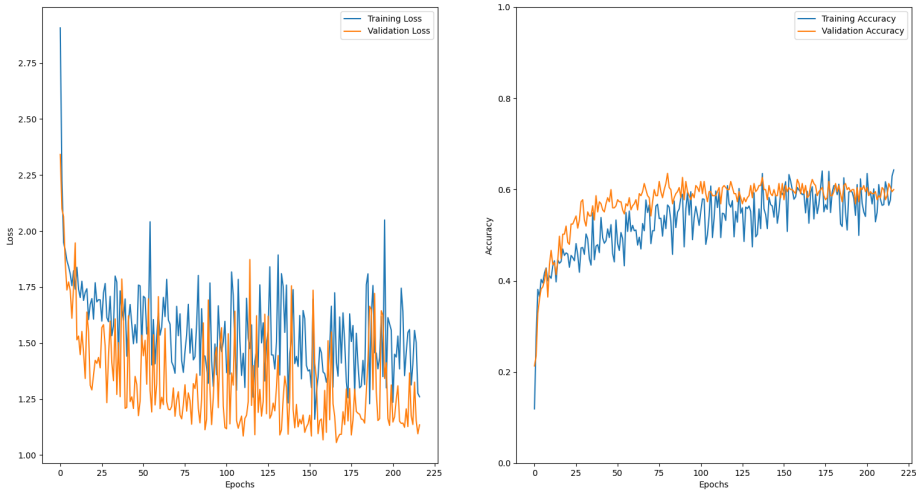

Figure 4: Training and validation losses and accuracies of the optimized model.

As shown, the model performs much better and doesn't overfit the training data.

The confusion matrix of the model on the test set is as follows:

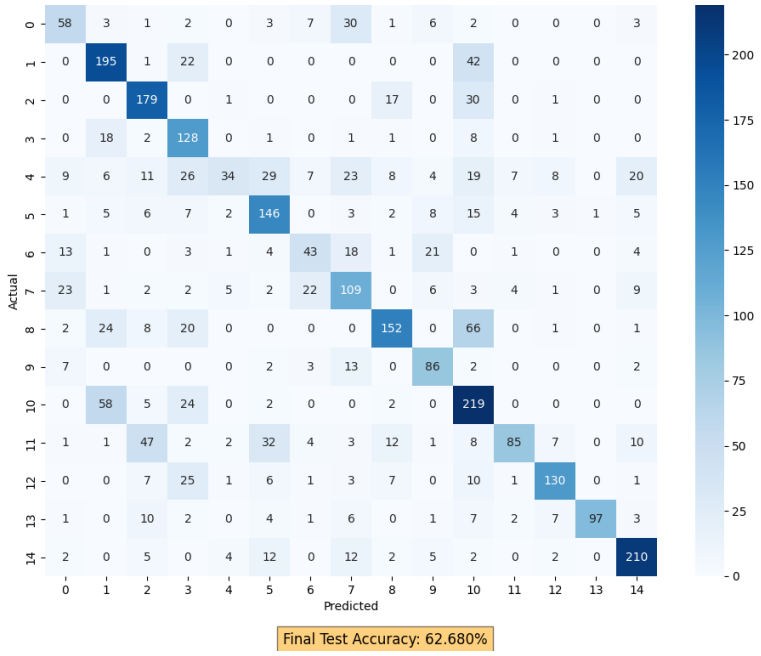

Figure 5: Confusion matrix of the optimized model on the test set.

By increasing the patience to 50, the final test accuracy of the ensemble of models ends up being 62.7%, which is a significant improvement over the previous model.[8]

5 Transfer learning

The third task consists in using a previously trained model and fine-tuning it to the given dataset. The chosen model is the AlexNet model [Krizhevsky et al., 2012].

5.1 Implementation of the model

The AlexNet model can be downloaded and implemented with torchvision as follows:

```
from torchvision import models
from torchvision.models.alexnet import AlexNet_Weights

alexnet = models.alexnet(weights=AlexNet_Weights.DEFAULT)
```

The model has three distinct parts: the features extractor, the avgpool and the classifier.
The dataset has to be adjusted to the model:

- The images have to be resized to 224x224, which is the input size of the model.
- The images have to be duplicated 3 times, since the model expects 3 channels.
- The images have to be normalized with the mean and standard deviation of the ImageNet dataset, on which the model was trained on.

[8] Due to the model's complexity and the associated computational limitations, the model was only trained once.

The dataset can be adjusted to the model by changing the transformations as follows:

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.Grayscale(num_output_channels=3),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

All of the data augmentation techniques used in the previous task were applied to the dataset in this task too.

5.2 Fine-tuned classifier

In order to fine-tune the model, the classifier has to be replaced with a new one, with 4096 input features and 15 output features. Then, the model's weights have to be frozen, except for the classifier's last layer:

```
alexnet.classifier[6] = nn.Linear(4096, 15)
for param in alexnet.parameters():
    param.requires_grad = False
alexnet.classifier[6].requires_grad = True
alexnet.classifier[6].weight.requires_grad = True
alexnet.classifier[6].bias.requires_grad = True
```

The model is then trained on the dataset with 50 epochs of patience and a learning rate of 0.001:

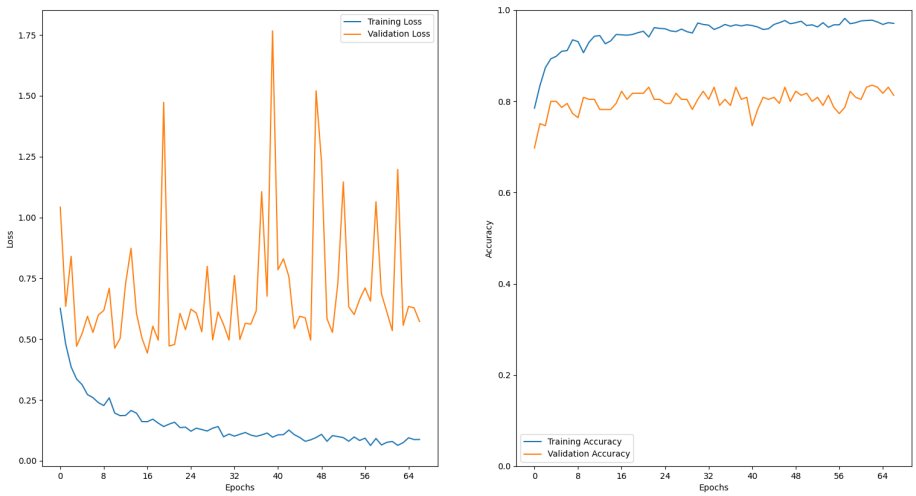

Figure 6: Training and validation losses and accuracies of the fine-tuned model.

As expected, the model converges very quickly, since AlexNet is already a very good model for image classification, and through the use of transfer learning it only has to adapt to the new training set.

The confusion matrix of the model on the test set is as follows:

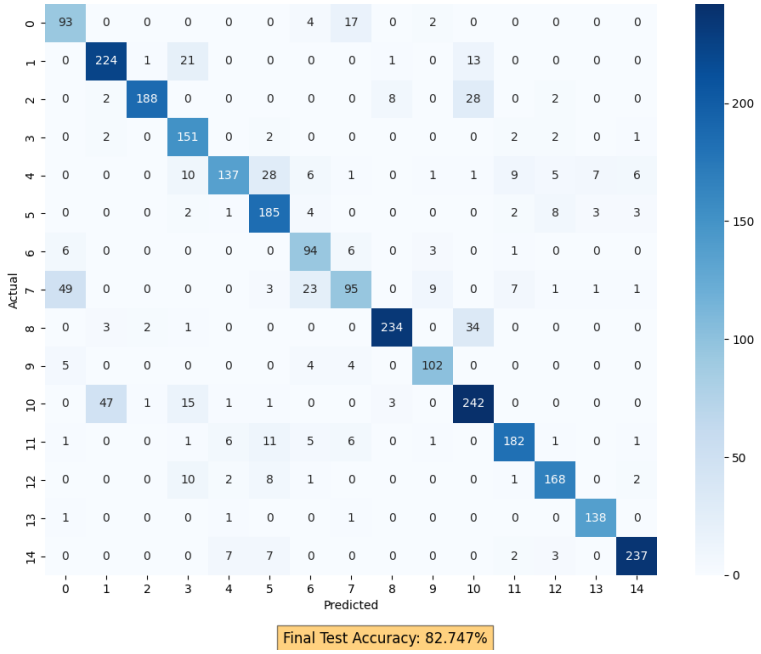

Figure 7: Confusion matrix of the fine-tuned model on the test set.

The final average test accuracy of the fine-tuned model is 83.3±0.6%.

5.3 SVM classifier

Another possible approach for transfer learning is to use the feature extractor part of AlexNet as a feature extractor for a One-vs-One SVM classifier[9]:

```
from sklearn import svm

features_layer = models.alexnet(weights=AlexNet_Weights.DEFAULT).features
for param in self.features_layer.parameters():
    param.requires_grad = False
svm = svm.SVC(kernel="linear", C=1.0)
```

A function to fit the SVM classifier is then defined:

```
def fit_svm(dataloader, epochs):
    features = []
    labels = []
    for _ in range(epochs):
        for x, y in iter(dataloader):
            features.append(np_features(x))
            labels.append(y.detach().numpy())
    features = np.concatenate(features)
    labels = np.concatenate(labels)
    svm.fit(features, labels)
```

[9]The SVM classifier is linear and uses a C value of 1.0. Using a non-linear SVM classifier or a different C value didn't improve the performance of the model.

```
def np_features(x):
    features = features_layer(x)
    features = features.view(x.size(0), -1)
    return features.detach().numpy()
```

By one-hot encoding the prediction, the SVM classifier can be fit to the training data and be used as a classifier for the test data as follows:

```
def forward(x):
    features = features_layer(x)
    features = features.view(x.size(0), -1)
    features_np = features.detach().numpy()
    predictions = svm.predict(features_np)
    probabilities = np.zeros((predictions.shape[0], svm.classes_.shape[0]))
    probabilities[np.arange(predictions.shape[0]), predictions] = 1
    return torch.from_numpy(probabilities)
```

The confusion matrix of the model on the test set is as follows:

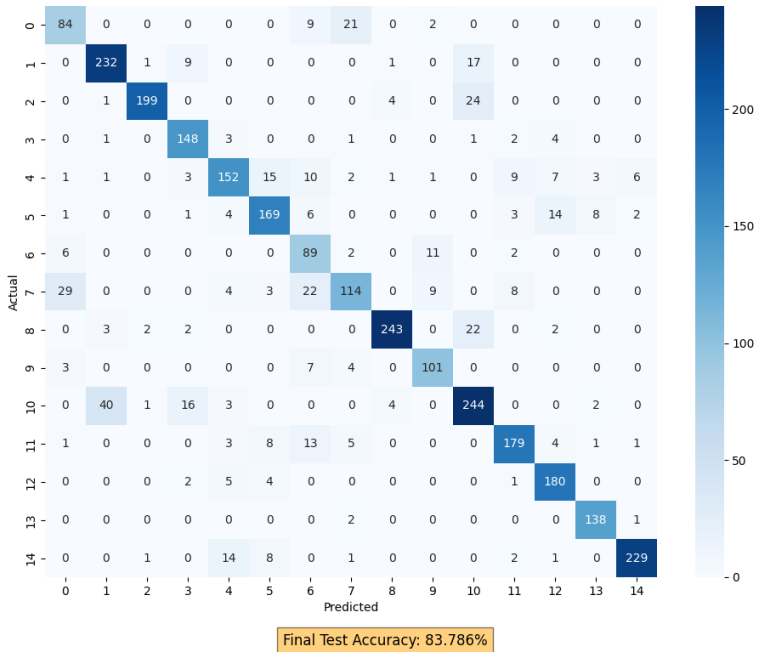

Figure 8: Confusion matrix of the SVM classifier on the test set.

The final average test accuracy of the SVM classifier is 84.4±0.5%.

Model	**Avg test accuracy**	**Confidence interval**[10]
Base reference model	27.9%	1.9%
Base model + data augmentation	43.7%	2.5%
Base model + data augmentation + regularization	54.2%	1.9%
Ensemble model	62.7%	N/A
Fine-tuned AlexNet	83.3%	0.6%
AlexNet feature extractor + OvO SVM	84.4%	0.5%

Table 3: Comparison of the results of the different models.

References

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

[Lazebnik et al., 2006] Lazebnik, S., Schmid, C., and Ponce, J. (2006). Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, volume 2, pages 2169–2178. IEEE.

[Ramachandran et al., 2017] Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Searching for activation functions. *arXiv preprint arXiv:1710.05941*.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[10] 95% confidence interval over 10 runs using the t-distribution.