notebookbc6846cd6c

December 16, 2024

[1]:
```
# imports
from typing import Callable

import numpy as np

import torch
from torch import nn
from torch.utils.data import DataLoader
from torch.optim import Optimizer

from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor

from tqdm.notebook import trange

#from solution import *
```

[2]:
```
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

print(DEVICE)
```

```
cuda
```

1 Exercise 8 - Convolutional Neural Networks

In this exercise we will create and train convolutional neural networks on the PyTorch dataset.

We will first write functions for both creation of MLPs (already implemented) and CNNs. Then we will train multiple MLPs and CNNs on the CIFAR10 dataset and compare their performance and number of parameters, as well as visualize the feature maps of the CNN. Finally we will investigate the robustness of CNNs against small translations of images.

2 Hyperparameters

[3]:
```
TRAIN_SPLIT = 0.5
VAL_SPLIT = 0.05

BATCH_SIZE = 128
EPOCHS = 5
LEARNING_RATE = 0.001
```

3 Datasets

3.1 Task 1 (10 P)

Use the `torch.utils.data.random_split()` function to further split the training set into train and validation set. Use the constants `TRAIN_SPLIT` and `VAL_SPLIT` as the percentages to take from the original training set.

[4]:
```
dataset = CIFAR10("sample_data", train=True, transform=ToTensor(),␣
 ↪download=True)

# ------------------------------------------------
# TODO: Replace by your code
# ------------------------------------------------c
#train_set, val_set = solution_1_train_and_val_split(dataset, TRAIN_SPLIT,␣
 ↪VAL_SPLIT)
#print(len(dataset)) #50000

dataset_len = len(dataset)
train_len = int(dataset_len * TRAIN_SPLIT)
val_len = int(dataset_len * VAL_SPLIT)
len_set = [train_len,val_len,dataset_len - train_len - val_len]
train_set, val_set,_ = torch.utils.data.random_split(dataset,len_set)


# ------------------------------------------------

test_set = CIFAR10("sample_data", train=False, transform=ToTensor(),␣
 ↪download=True)

train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_set, batch_size=BATCH_SIZE)
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
sample_data/cifar-10-python.tar.gz
```

```
100%|      | 170498071/170498071 [00:02<00:00, 57952907.65it/s]
```

```
Extracting sample_data/cifar-10-python.tar.gz to sample_data
Files already downloaded and verified
```

4 MLP

Implementation of MLP is given.

[5]:
```python
def create_mlp(input_shape: tuple[int, int, int], hidden_dims: list[int],␣
↪n_classes: int):
    dims = [np.prod(input_shape), *hidden_dims, n_classes]

    layers = []
    for i in range(len(dims) - 1):
        if i < len(dims) - 2:
            layers.extend([nn.Linear(dims[i], dims[i+1]), nn.ReLU()])
        else:
            layers.extend([nn.Linear(dims[i], dims[i+1])])

    model = nn.Sequential(
        nn.Flatten(),
        *layers,
    )

    return model.to(DEVICE)
```

5 CNN

5.1 Task 2 (20 P)

Implement a CNN. Differently than in the previous tasks, the function should work for arbitrary input sizes, and an arbitrary number of classes.

The CNN consists of the convolutional part, and a final projection head, which will be a simple `nn.Linear()` layer.

The convolutional part consists of a list of convolutional blocks, which are of the form `nn.Conv2d(...)` → `nn.MaxPool(...)` →`nn.ReLU()`. For the convolutional block we will use a kernel size of 5, with a padding of 2, and the max pool layrs will use a a kernel size of 4 with a stride of 2 and a padding of 1. This ensures that the convolutions will keep the image height and width and the pooling layers will halve the image dimensions.

The Projection head maps the output of the convolutional part onto a vector of size `n_classes`, so that we can use the cross-entropy loss to optimize our classifier. Since the nn.Linear layer needs a fixed number of input features, but this function should work for arbitrary input sizes, you will have to figure out the output shape of the convolutional part programatically. This is, because the output of a convolutional layer depends not only on how we specify our layer, but also on the input size. Do not hardcode the input shape of the CIFAR dataset into the function.

It is possible to use the `create_mlp()` function above in this module, it is not necessary however.

[6]:
```
def create_cnn(input_shape: tuple[int, int, int], channels: list[int],␣
↪n_classes: int):
    # ------------------------------------------------
    # TODO: Replace by your code
    # ------------------------------------------------
    #model = solution_2_create_cnn(input_shape, channels, n_classes, create_mlp)

    layers = []
    in_channels = input_shape[0]
    for out_channels in channels:
        layers.append(nn.
↪Conv2d(in_channels,out_channels,kernel_size=5,padding=2))
        layers.append(nn.MaxPool2d(kernel_size=4,stride=2,padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels

    conv_part = nn.Sequential(*layers)


    ex_input = torch.zeros(1, *input_shape)  # wir nehmen batch size = 1 an
    conv_output = conv_part(ex_input)
    conv_output_shape = conv_output.shape
    flattened_size = conv_output_shape[1] * conv_output_shape[2] *␣
↪conv_output_shape[3]

    projection_head = nn.Sequential(
        nn.Flatten(),
        nn.Linear(flattened_size, n_classes)
    )

    model = nn.Sequential(
        conv_part,
        projection_head
    )


    # ------------------------------------------------

    return model.to(DEVICE)
```

6 Train and validation functions

The functions for training and evaluating the models are given.

[7]:
```
def train_one_epoch(model: nn.Module, train_loader: DataLoader,
                    optimizer: Optimizer, loss_fn: Callable):
```

```
    losses = []

    for batch, labels in train_loader:
        batch, labels = batch.to(DEVICE), labels.to(DEVICE)

        optimizer.zero_grad()
        preds = model(batch)
        loss = loss_fn(preds, labels)
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

    return np.mean(losses)


def validate(model: nn.Module, val_loader: DataLoader, loss_fn: Callable):
    losses = []

    model.eval()

    with torch.no_grad():
        for batch, labels in val_loader:
            batch, labels = batch.to(DEVICE), labels.to(DEVICE)
            preds = model(batch)
            loss = loss_fn(preds, labels)

            losses.append(loss.item())

    model.train()

    return np.mean(losses)


def train_model(model: nn.Module, train_loader, val_loader,
                optimizer: Optimizer, loss_fn: Callable, n_epochs: int,
                validate_every: int = 1):
    train_losses = []
    val_losses = []

    # stores the iterations in which we validated our model for plotting
    val_epochs = []

    progress_bar = trange(n_epochs)
    for epoch in progress_bar:
        train_loss = train_one_epoch(model, train_loader, optimizer, loss_fn)
        train_losses.append(train_loss)
```

```
        if epoch % validate_every == 0 or epoch in [1, n_epochs - 1]:
            val_loss = validate(model, val_loader, loss_fn)
            val_losses.append(val_loss)
            val_epochs.append(epoch)

        progress_bar.set_description(f"Train loss: {train_loss:.4f}\t Val loss:␣
↪{val_loss:.4f}")

    # The first epoch often results in very high losses due to random guesses at
    # the beginning of training. To avoid the effect on the plot you may
    # uncomment the following line.

    # return train_losses[1:], val_losses[1:], [e-1 for e in val_epochs][1:]

    return train_losses, val_losses, val_epochs
```

6.1 Task 3 (10 P)

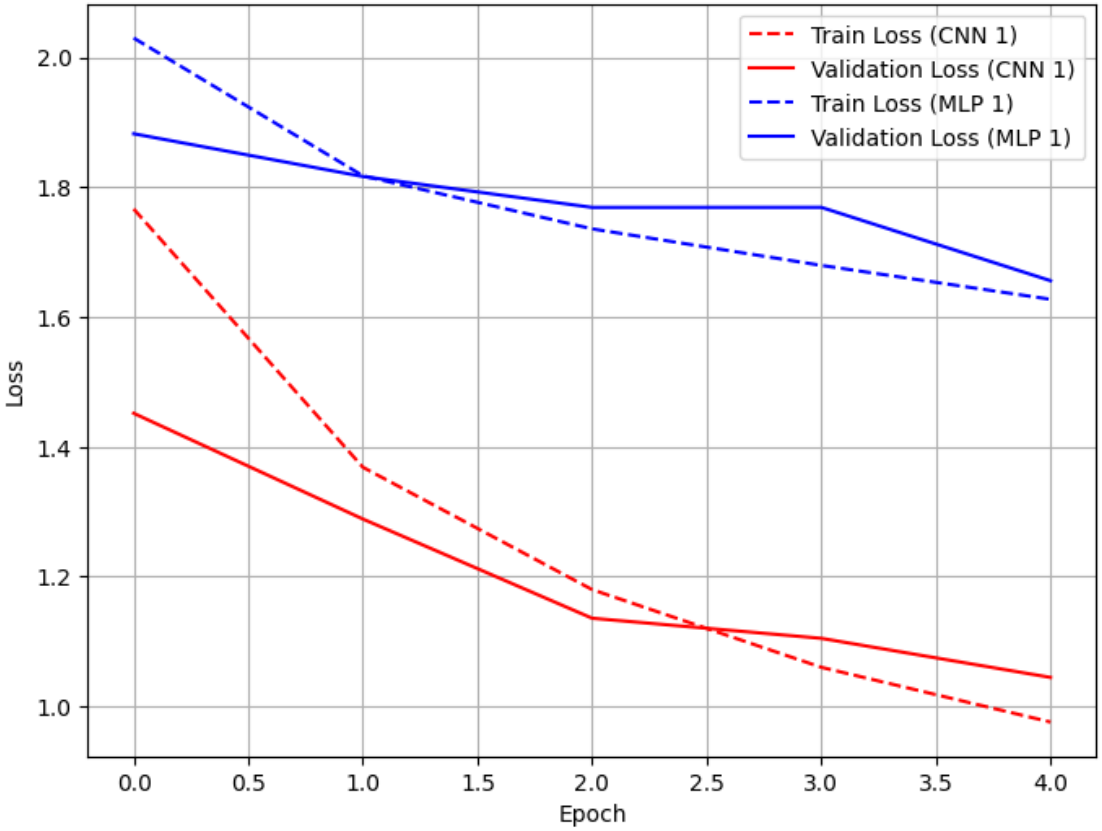
Use the functions above to train three CNNs and three MLPs using the Adam optimizer with the provided learning rate. After having trained one MLP and one CNN, plot their validation and training losses as shown below. The progress bars are made using the `tqdm` package, in this case the `trange` function, but this is not a requirement for this task. The models should be appended to the lists `cnns` and `mlps`. The parameters for the functions for creating the models are given.

[8]:
```
import matplotlib.pyplot as plt

mlps: list[nn.Module] = []
cnns: list[nn.Module] = []

input_shape = (3, 32, 32)
cnn_channels = [64, 96, 128]
mlp_dims = [1024, 512, 64]
n_classes = 10

# ------------------------------------------------
# TODO: Replace by your code
# ------------------------------------------------
'''
mlps, cnns = solution_3_train_n_models(n_models=3,
                                       input_shape=input_shape,
                                       create_cnn=create_cnn,
                                       cnn_channels=cnn_channels,
                                       create_mlp=create_mlp,
                                       mlp_dims=mlp_dims,
                                       n_classes=n_classes,
                                       lr=LEARNING_RATE,
```

```
                                        n_epochs=EPOCHS,
                                        train_fn=train_model,
                                        train_loader=train_loader,
                                        val_loader=val_loader)
'''


for i in range(3):
    plt.figure(figsize=(8, 6))

    cnn = create_cnn(input_shape, cnn_channels, n_classes)
    optimizer = torch.optim.Adam(cnn.parameters(),lr= LEARNING_RATE)
    loss_fn = nn.CrossEntropyLoss()
    train_losses, val_losses, val_epochs =␣
 ↪train_model(cnn,train_loader,val_loader,optimizer,loss_fn,EPOCHS)
    cnns.append(cnn)
    plt.plot(range(len(train_losses)), train_losses, 'r--', label=f'Train Loss␣
 ↪(CNN {i+1})')
    plt.plot(val_epochs, val_losses, 'r-', label=f'Validation Loss (CNN {i+1})')


    mlp = create_mlp(input_shape, mlp_dims,n_classes)
    optimizer = torch.optim.Adam(mlp.parameters(),lr= LEARNING_RATE)
    loss_fn = nn.CrossEntropyLoss()
    train_losses, val_losses, val_epochs =␣
 ↪train_model(mlp,train_loader,val_loader,optimizer,loss_fn,EPOCHS)
    mlps.append(mlp)
    plt.plot(range(len(train_losses)), train_losses, 'b--', label=f'Train Loss␣
 ↪(MLP {i+1})')
    plt.plot(val_epochs, val_losses, 'b-', label=f'Validation Loss (MLP {i+1})')



    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()




# ------------------------------------------------

print(f"Trained {len(mlps)} MLPs and {len(cnns)} CNNs.")
```

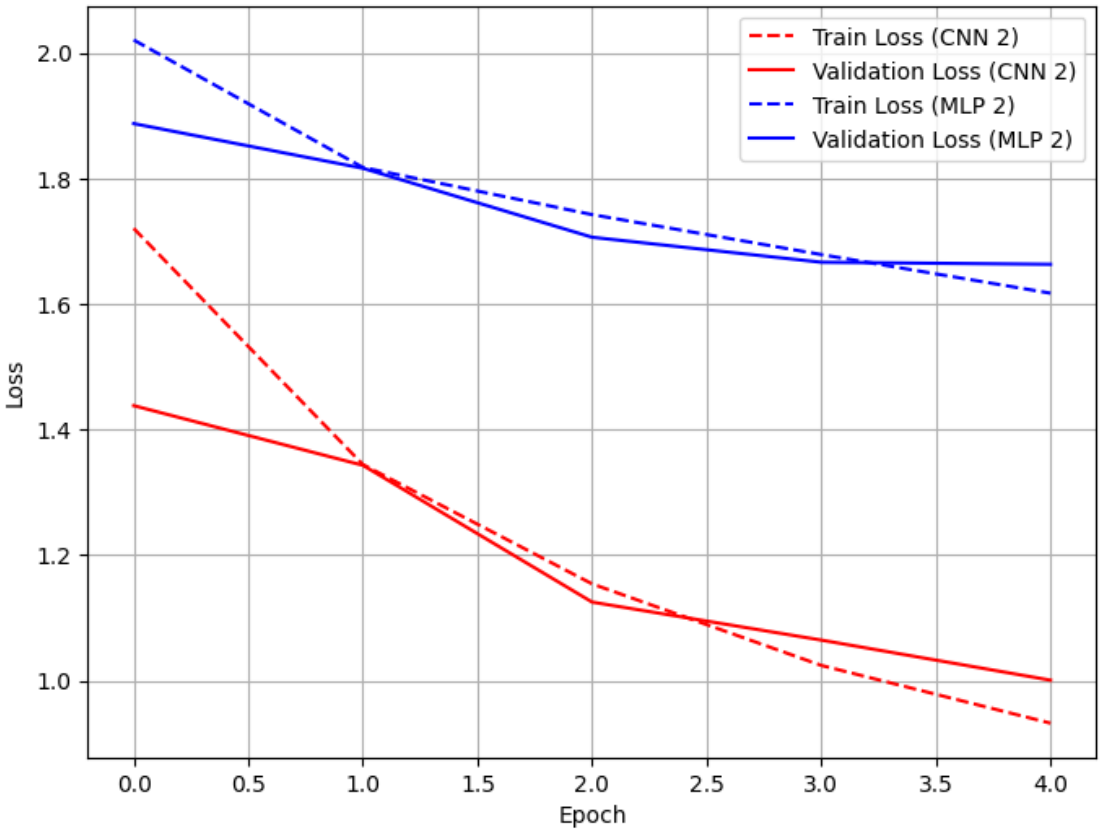
```
  0%|          | 0/5 [00:00<?, ?it/s]
```

0%| | 0/5 [00:00<?, ?it/s]

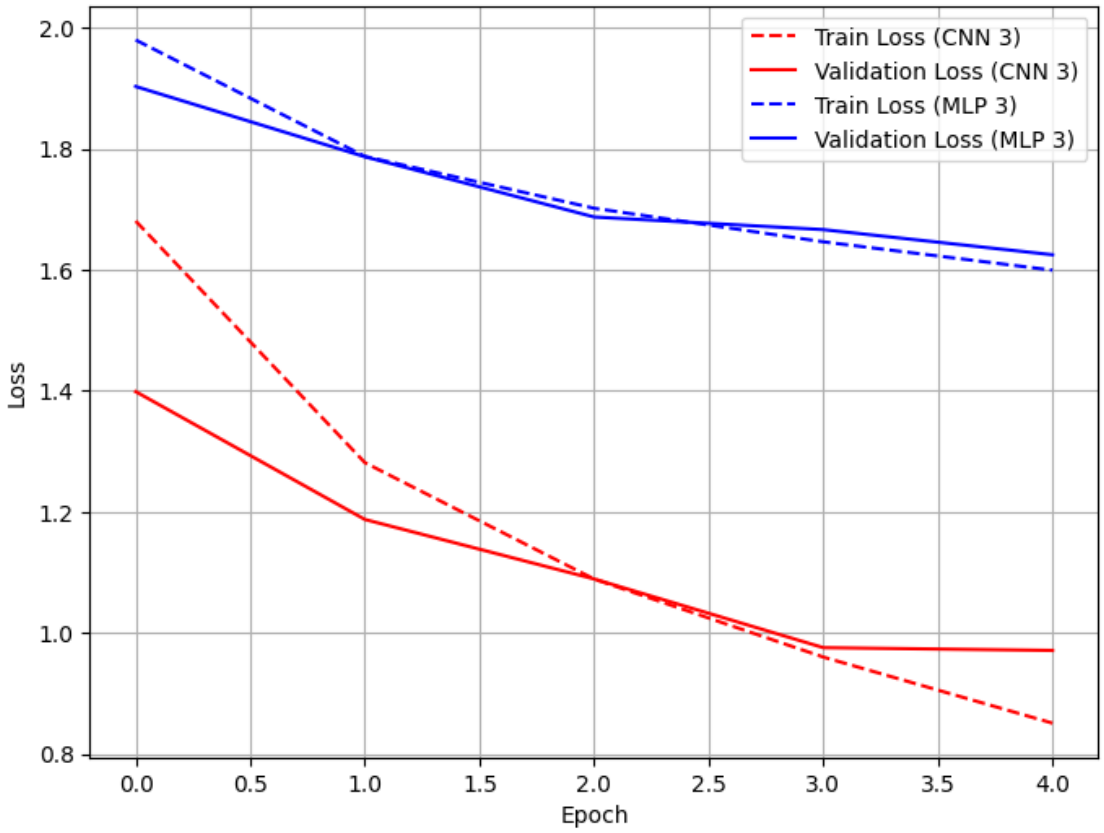


0%| | 0/5 [00:00<?, ?it/s]

0%| | 0/5 [00:00<?, ?it/s]




```
0%|          | 0/5 [00:00<?, ?it/s]
0%|          | 0/5 [00:00<?, ?it/s]
```




```
Trained 3 MLPs and 3 CNNs.
```

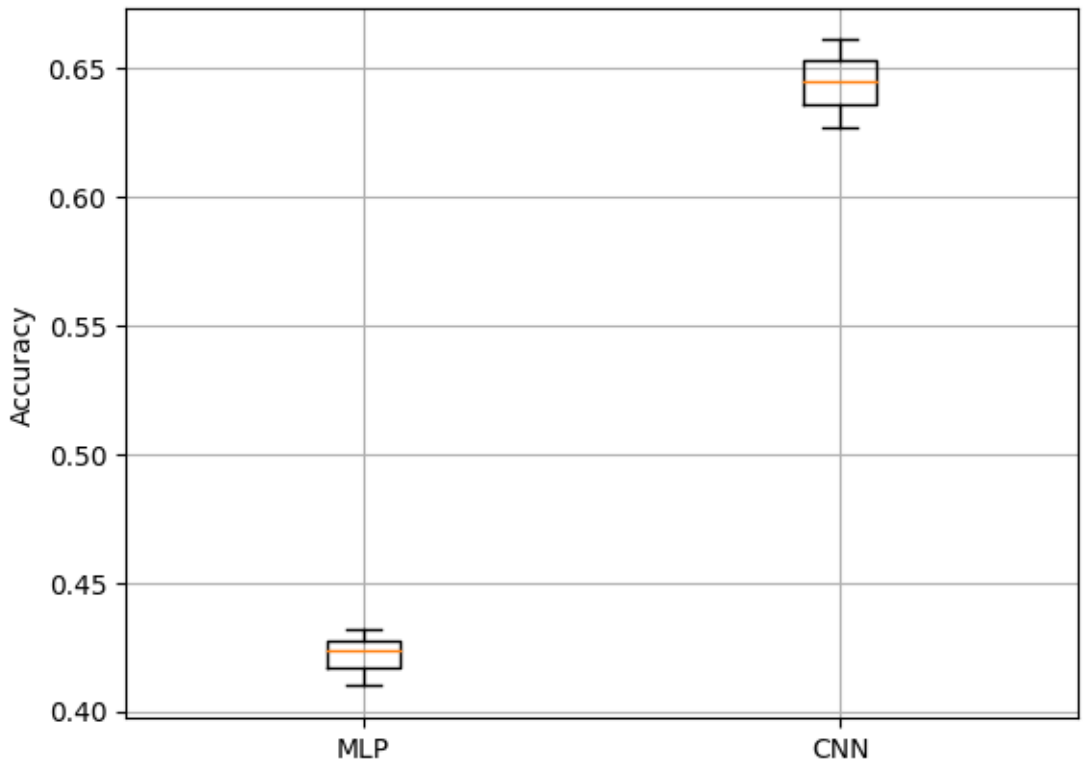
7 Evaluation

7.1 Task 4a - Accuracy (20 P)

You are given a function to compute the accuracy of a model on a given dataloader. Compute the accuracies of all the cnns and mlp and plot them as a boxplot as shown below.

[9]:
```python
def get_accuracy(model: nn.Module, dataloader: DataLoader):
    """Compute accuary of modle on test set"""
    accuracies = []

    model.eval()

    with torch.no_grad():
        for batch, labels in dataloader:
            batch, labels = batch.to(DEVICE), labels.to(DEVICE)
            preds = model(batch)
            accuracy = torch.mean((torch.argmax(preds, dim=-1) == labels).
↪type(torch.FloatTensor))
```

```
            accuracies.append(accuracy)

    model.train()

    return np.mean(accuracies)

# ------------------------------------------------
# TODO: Replace by your code
# ------------------------------------------------
'''
solution_4a_accuracy_boxplots(get_accuracy_fn=get_accuracy,
                              mlps=mlps,
                              cnns=cnns,
                              dataloader=test_loader)
'''
mlp_accuracies = []
cnn_accuracies = []

for i, mlp in enumerate(mlps):
    acc = get_accuracy(mlp, test_loader)
    mlp_accuracies.append(acc)

for i, cnn in enumerate(cnns):
    acc = get_accuracy(cnn, test_loader)
    cnn_accuracies.append(acc)

plt.figure()
plt.boxplot([mlp_accuracies, cnn_accuracies], labels=["MLP", "CNN"])
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()

print(f"MLP mean: {np.mean(mlp_accuracies)}")
print(f"CNN mean: {np.mean(cnn_accuracies)}")
# ------------------------------------------------
```




```
MLP mean: 0.42171016335487366
CNN mean: 0.6445147395133972
```

7.2 Task 4b - Number of parameters (10 P)

Write a function that returns the number of trainable weights of a model.

[10]:
```python
def count_params(model: nn.Module):
    # ----------------------------------------------------
    # TODO: Replace by your code
    # ----------------------------------------------------
    #return solution_4b_count_params(model)
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
    # ----------------------------------------------------


mlp_params = count_params(mlps[0])
print(f"MLP:\t{mlp_params} trainable paramerters")
cnn_params = count_params(cnns[0])
print(f"CNN:\t{cnn_params} trainable parameters")

print(f"\nCNN is ~{(mlp_params / cnn_params):.2f} times smaller than the MLP.")
```

```
MLP:    3705034 trainable paramerters
CNN:    486378 trainable parameters

CNN is ~7.62 times smaller than the MLP.
```

7.3 Task 4c - Plot feature maps (20 P)

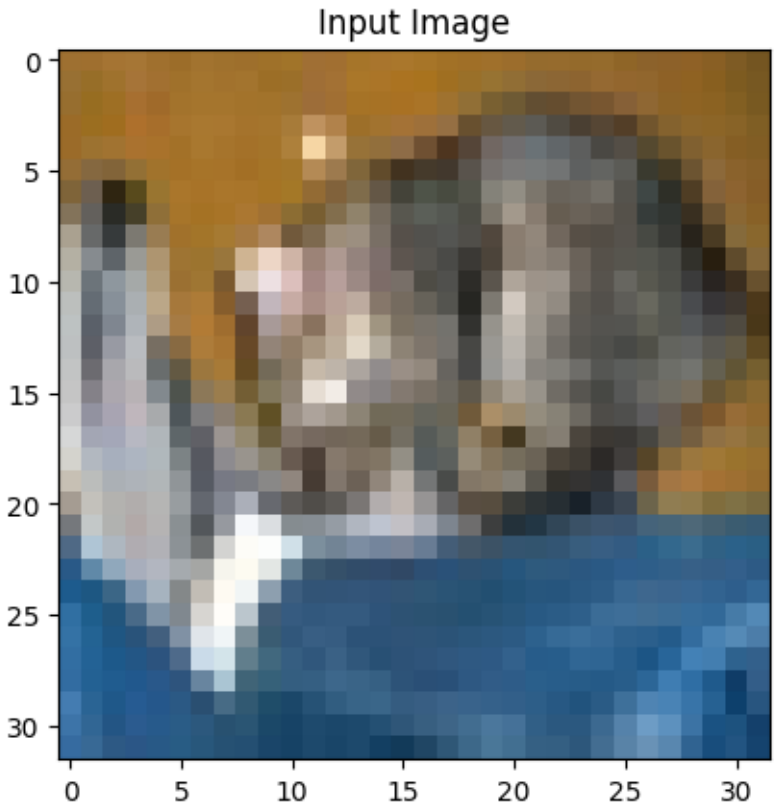
Visualize the feature maps of the convolutional neural networks.

A feature map is simply a single channel of some intermediate conv layer. E.g. in the above trained CNNs, the first conv layer outputs 32 channels. For each of these channels we get a 16 x 16 feature map. We will consider the feature map after applying the max pooling and relu activation. The feature map then can be visualized using matplotlib as a grayscale image. Make sure to normalize your images to the range [0,1].

Hint: You might want to use the `torchvision.utils.make_grid`.

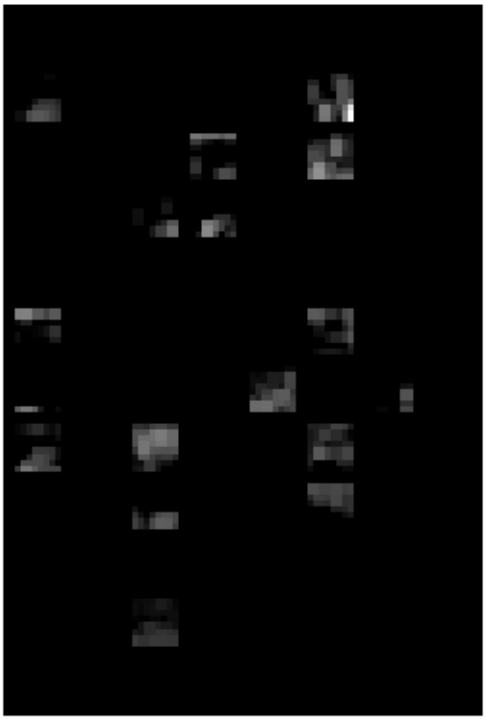
[11]:
```python
from torchvision.utils import make_grid
first_cifar_test_image = next(iter(test_loader))[0][0][None, :]
model = cnns[0]

# ------------------------------------------------
# TODO: Replace by your code
# ------------------------------------------------
#solution_4c_plot_feature_maps(model, first_cifar_test_image)
model.eval()

input_image = first_cifar_test_image[0].permute(1, 2, 0)

plt.figure()
plt.imshow(input_image)
plt.title("Input Image")
plt.show()


feature_maps_list = []
relu_layer_indices = []

# see https://medium.com/@deepeshdeepakdd2/
 ↪cnn-visualization-techniques-feature-maps-gradient-ascent-aec4f4aaf5bd

def hook(module, input, output):
    feature_maps_list.append(output)

hooks = []
for idx, layer in enumerate(model.modules()):
    if isinstance(layer, torch.nn.ReLU):
        relu_layer_indices.append(idx-2)
        hooks.append(layer.register_forward_hook(hook))  # rigister hook
```

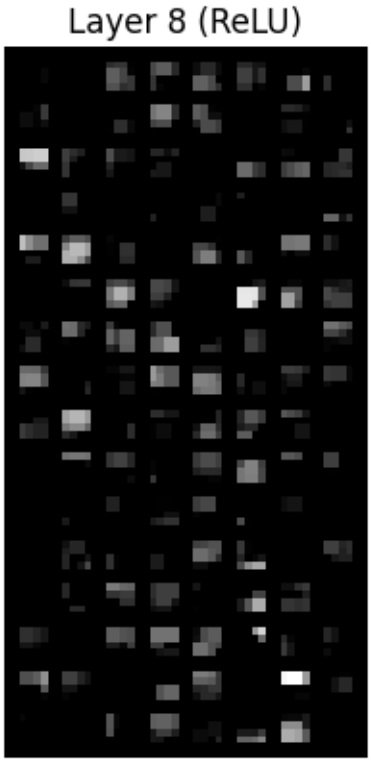
```
with torch.no_grad():
    _ = model(first_cifar_test_image.to(DEVICE)) # forward prad

for hook in hooks:
    hook.remove()

for idx, feature_maps in zip(relu_layer_indices, feature_maps_list):
    feature_maps = feature_maps.cpu()
    # (N, C, H, W)

    plt.figure()
    grid = make_grid(feature_maps[0].unsqueeze(1), normalize=True)
    plt.imshow(grid.permute(1, 2, 0))
    plt.title(f"Layer {idx} (ReLU)")
    plt.axis('off')
    plt.show()
# ------------------------------------------------
```



Layer 2 (ReLU)

Layer 5 (ReLU)





8 Robustness to translations of images

8.1 Task 5 (10 P)

1. Create a new test set with translated images. You may use the `transform` argument of the `CIFAR10` class to apply translations of images. Though you are free to choose the method you prefer.
2. As in task 4a, compute the accuracies of the models on the modified test set with translated images.

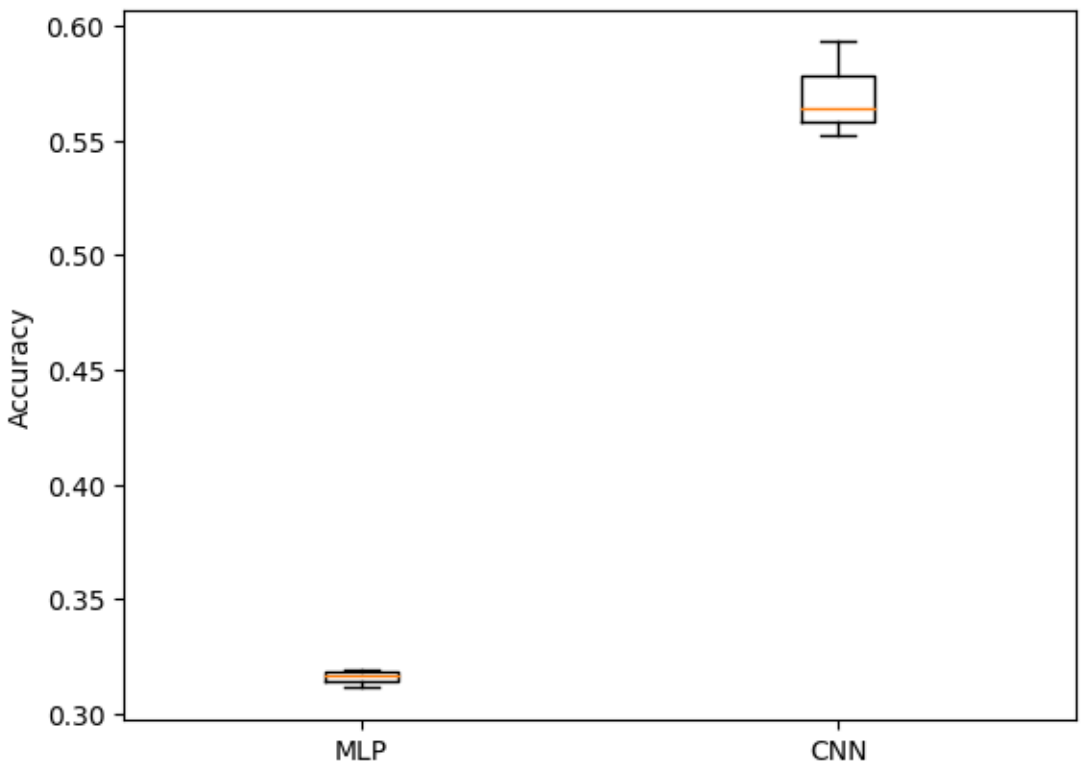
Translation simply refers to shifting the images a few pixels. As you can see in our example we have shifted the images by a maximum of 3/32 of its original size in both height and width (the `translate` argument), which results in a shift of max. 3 pixels.

You will note, that although the performance of both models is decreased, the MLPs suffer a higher decrease in accuracy.

[14]:
```
# ------------------------------------------------
# TODO: Replace by your code
# ------------------------------------------------
"""
```

```
solution_5_robustnes_to_translations(mlps=mlps,
                                     cnns=cnns,
                                     batch_size=BATCH_SIZE,
                                     get_accuracy_fn=get_accuracy,
                                     translate=(3 / 32, 3 / 32))
"""

from torchvision import transforms

translation_transform = transforms.Compose([
    transforms.RandomAffine(degrees=0, translate=(3/32, 3/32)),
    transforms.ToTensor()
])
translated_test_set = CIFAR10(root="sample_data", train=False,␣
 ↪transform=translation_transform, download=True)
translated_test_loader = DataLoader(translated_test_set, batch_size=BATCH_SIZE)

mlp_accuracies = []
cnn_accuracies = []

for i, mlp in enumerate(mlps):
    acc = get_accuracy(mlp, translated_test_loader)
    mlp_accuracies.append(acc)

for i, cnn in enumerate(cnns):
    acc = get_accuracy(cnn, translated_test_loader)
    cnn_accuracies.append(acc)

plt.figure()
plt.boxplot([mlp_accuracies, cnn_accuracies], labels=["MLP", "CNN"])
plt.ylabel("Accuracy")
plt.show()

print(f"MLP mean: {np.mean(mlp_accuracies)}")
print(f"CNN mean: {np.mean(cnn_accuracies)}")


# ------------------------------------------------
```

```
Files already downloaded and verified
```




```
MLP mean: 0.31586235761642456
CNN mean: 0.5697850584983826
```

[ ]: