EdgeDetection_Student

October 15, 2022

1 HW5: Exploring Inductive Bias of Convolutional Neural Networks and Systematic Experimentation in Machine Learning

In this homework, we will study 1) what is inductive bias and how it affects the learning process, and 2) how to conduct systematic experiments in machine learning. We will compare convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs) extensively as an example to study these two topics.

1.1 1. Inductive Bias

What is inductive bias? It is the assumption that the learning algorithm makes about the problem domain. Suppose that we build a machine learning system. We want to leverage the specific knowledge about the problem domain to make the learning process **more efficient** and the system **generalize much better** with fewer parameters. Let's be more precise. What do exactly **more efficient** and **generalize much better** mean? The learning process is more efficient 1) if we can learn the model with fewer parameters, 2) if we can learn the model with fewer data, and 3) if we can learn the model with fewer iterations. And the system generalizes much better if the model can generalize to the unseen data well.

We have already observed the power of inductive bias. We know that CNN generalizes better than MLP even with the same number of parameters. We partially concluded that is because CNN has the inductive bias that the model is translation invariant. We will study the inductive bias of CNN in more detail in this homework.

In this homework, we will use the edge detection task as an example to study the inductive bias of CNN. We will compare CNN and MLP extensively. And we will see when CNN can fail.

1.2 2. Systematic Experimentation in Machine Learning

How can we prove our hypothesis that CNN has the inductive bias that the model is translation invariant? We conduct extensive experiments in machine learning research (and other fields) to prove our hypothesis. In this context, systematic experimentation refers to running a series of experiments to prove our hypothesis. In this homework, we will study how to conduct systematic experimentation in machine learning.

Let's take a step back and think about 1) what our hypothesis is and 2) what experiments are needed to conduct to prove our hypothesis. The first question is easy. The hypothesis is that CNN

has the inductive biases of locality and translational invariance. It is not enough to show that CNN performs better than MLP with the same number of parameters. Then, how do we design the experiments to prove our hypothesis? In this homework, we will design the experiments, conduct the experiments, analyze the results, and draw a conclusion.

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as T

import numpy as np
import random
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from tqdm import tqdm
from copy import deepcopy
from torch.utils.data import DataLoader
from dataset.edge_detection_dataset import EdgeDetectionDataset
from models.cnn import *
from models.mlp import *
from helpers.train_helper import train_one_epoch, evaluate
from helpers.model_helper import *
from helpers.vis_helper import *
from helpers.reproduce_helper import set_seed

seed = 7
set_seed(seed)

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
  ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

1.3 Generate Dataset

What would be an excellent dataset to study the inductive bias of CNN? First, have to start with the problem as simple as possible. The complex problem makes it hard to understand the underlying mechanism and is challenging to debug in experimental settings. Hence, we choose the

edge detection task as an example to study the inductive bias of CNN. Because 1. Edge detection is a straightforward task, 2. It is easy to generate the dataset,

3. The edge of the image is a very fundamental low-level feature useful to every computer vision task such as object detection and finally,

4. Edge detection is an excellent example of studying the inductive bias of CNN.

We will generate the dataset for this toy problem. The dataset consists of 10 images of size 28x28 per class, which are all grey scales. Each image contains a vertical edge, a horizontal edge, or nothing. The labels are 0 for vertical edges, 1 for horizontal edges, and 2 for nothing.

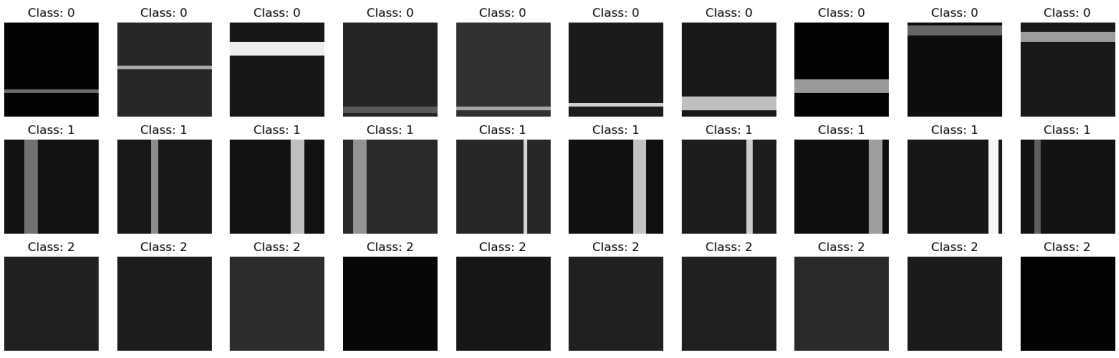
`EdgeDetectionDataset` class is a dataset class that generates and loads the dataset. The dataset inherits `torch.utils.data.Dataset`, and it generates data when it is initialized. This class takes two arguments: `domain_config` and `transform`. `domain_config` is a dictionary that specifies the domain information of train/valid dataset, such as the number of images per class and the size of the image. `transform` is a function that transforms the image. In this homework, we will use `torchvision.transforms.ToTensor()` to convert the image to a tensor.

We highly recommend you read the implementation of `EdgeDetectionDataset` class in `dataset/edge_detection_dataset.py` to understand how the dataset is generated.

[2]:
```
# Define the domain configuration of the dataset
set_seed(seed)

visualize_data_config = dict(
    data_per_class=10,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
)

visualize_dataset = EdgeDetectionDataset(visualize_data_config, mode='train',␣
↪transform=None)
```

1.3.1 Visualize Dataset

[3]:
```
vis_dataset(visualize_dataset, num_classes=3, num_show_per_class=10)
```



1.3.2 Q1. Overfitting Models to Small Dataset

In this problem, we will make our models overfit the small dataset to test the model architecture and our synthetic dataset. We use the same dataset for both models. Let's generate a small dataset with ten images per class.

[4]:
```
set_seed(seed)

small_dataset_config = None
small_dataset = None
transforms = T.Compose([T.ToTensor()])

################################################################################
# TODO: Generate dataset with 10 images per class                              #
# Hint: Refer visualize_data_config                                            #
################################################################################
small_dataset_config = dict(
    data_per_class=10,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
)

small_dataset = EdgeDetectionDataset(small_dataset_config, mode='train',␣
 ↪transform=transforms)
################################################################################
#                               END OF YOUR CODE                               #
################################################################################
```

In this notebook, we will use pytorch dataloader to load the dataset. We will use `torch.utils.data.DataLoader` to load the dataset. `DataLoader` takes two arguments: `dataset` and `batch_size`. `dataset` is the dataset that we want to load. Note that `batch_size` is one of important hyperparameters. We will use `batch_size=32` for this problem.

[5]:
```
small_dataset_loader = None

################################################################################
# TODO: Implement dataloader                                                   #
# Hint: You should flag shuffle = True for training data loader                #
# This flag makes huge difference in training                                  #
################################################################################
batch_size = 32

small_dataset_loader = torch.utils.data.DataLoader(small_dataset,batch_size)
################################################################################
```

```
#                                 END OF YOUR CODE                             #
################################################################################
```

1.3.3 Model Architecture

MLP has two hidden layer with 10 hidden units and 10 hidden units. The input size is 28x28=784 and the output size is 3. We use ReLU as the activation function. We use cross entropy loss as the loss function.

MLP architecture: FC(784, 10) -> ReLU -> FC(10, 10) -> ReLU -> FC(10, 3)

CNN has two convolutional layers followed by global average pooling and one fully connected layer. Both convolutional layers have 3 filters whose kernel size is 7. We use ReLU as the activation function. We use cross entropy loss as the loss function.

CNN arhitecture is as follows: CONV - RELU - MAXPOOL - CONV - RELU - MAXPOOL - FC

1.3.4 Fitting on Small Dataset

Now let's train the model on the small dataset. The final tranining loss should be around 100% for both models.

[6]:
```
set_seed(seed)

lr = 0.01
num_epochs = 500
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

cnn_model = SimpleCNN(kernel_size=7)
cnn_model.to(device)
untrained_cnn_model = deepcopy(cnn_model)

mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
mlp_model.to(device)

mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

criterion = nn.CrossEntropyLoss()
print("CNN Model has {} parameters".format(count_parameters(cnn_model,␣
 ↪only_trainable=True)))
print("MLP Model has {} parameters".format(count_parameters(mlp_model,␣
 ↪only_trainable=True)))

for epoch in tqdm(range(num_epochs)):
    train_one_epoch(cnn_model, cnn_optimizer, criterion, small_dataset_loader,␣
 ↪device, epoch, verbose=False)
```

```
    train_one_epoch(mlp_model, mlp_optimizer, criterion, small_dataset_loader,␣
 ↪device, epoch, verbose=False)

    _, cnn_acc, _ = evaluate(cnn_model, criterion, small_dataset_loader,␣
 ↪device, verbose=False)
    _, mlp_acc, _ = evaluate(mlp_model, criterion, small_dataset_loader,␣
 ↪device, verbose=False)

print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
```

```
CNN Model has 606 parameters
MLP Model has 39793 parameters
```

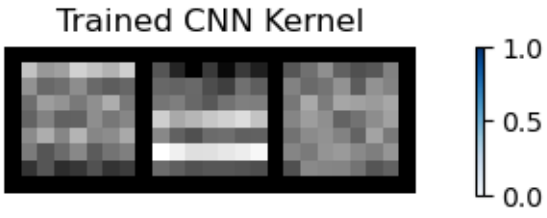
```
100%|                        | 500/500 [00:09<00:00, 55.37it/s]
```

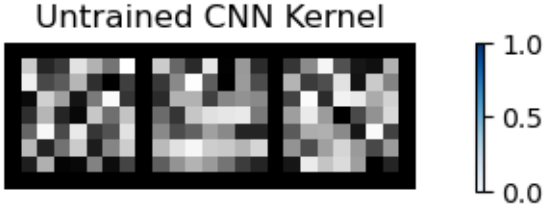
```
CNN Acc: 100.0, MLP Acc: 100.0
```

We checked that both models can overfit the small dataset. This is one of the most important sanity check. If the model cannot overfit the small dataset, the model is not powerful enough to learn the dataset. In this case, we need to increase the size of the model.

1.3.5 Visualize Learned Filters

[7]:
```
cnn_kernel = cnn_model.conv1.weight.data.clone().cpu()
untrained_kernel = untrained_cnn_model.conv1.weight.data.clone().cpu()

vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel')
vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN␣
 ↪Kernel')
```





Q. Can you find any interesting patterns in the learned filters?

A. Yes, it seems the CNN has overfitted and thus learned the top line for the horizonal images and two horizontal lines for the hoirontal images. Thus a horizontal line in the top row wouldn't be recognized by the CNN

1.3.6 Q2. Sweeping the Number of Training Images

We understood the given task and checked that both models had enough expressive power. We will compare the performance of MLP and CNN by changing the number of data per class. We expect that the model with proper inductive biases on this task will fit with **fewer training examples**. And let's see which one has inductive biases. In this problem, we will use the same dataset for both models. We sweep the number of training images from 10 to 50. The validation set will be the same for all the experiments.

[8]:
```
set_seed(seed)

train_loader_dict = dict()
num_images_list = [10, 20, 30, 40, 50]
valid_loader = None

transforms = T.Compose([T.ToTensor()])
train_batch_size = 10
valid_batch_size = 256
################################################################################
# TODO: Implement train_loader_dict for each number of training images.      #
# Key: The number of training images (10, 50, 100, and 500)                  #
# Value: The corresponding dataloader                                        #
# The validation set size is 50 images per class                             #
################################################################################
for num_images in num_images_list:
    train_config = dict(
    data_per_class=num_images,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    )

    train_dataset = EdgeDetectionDataset(train_config, mode='train',␣
 ↪transform=transforms)

    train_loader_dict[num_images] = torch.utils.data.
 ↪DataLoader(train_dataset,train_batch_size)

valid_config = dict(
    data_per_class=50,
```

```
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    )

valid_dataset = EdgeDetectionDataset(valid_config, mode='train',␣
 ↪transform=transforms)

valid_loader = torch.utils.data.DataLoader(valid_dataset,valid_batch_size)

################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

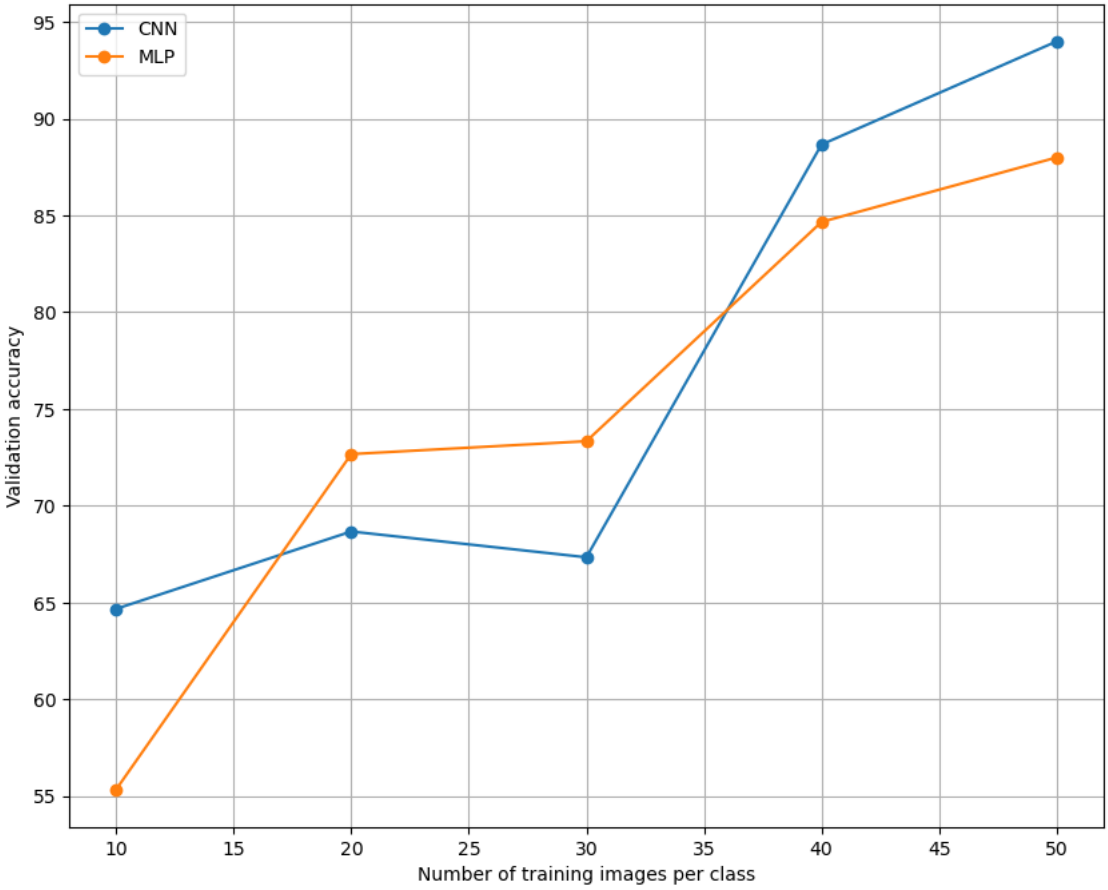
[9]:
```
lr = 5e-3
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_valid_acc_list = []
    mlp_valid_acc_list = []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
 ↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model,␣
 ↪mlp_optimizer, criterion, train_loader, device, epoch, verbose=False)
```

```
        cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion,␣
↪valid_loader, device, verbose=False)
        mlp_valid_loss, mlp_valid_acc, _ = evaluate(mlp_model, criterion,␣
↪valid_loader, device, verbose=False)

        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)

    cnn_kernel_dict[num_image] = deepcopy(cnn_model.conv1.weight.cpu().detach())
    untrained_cnn_kernel_dict[num_image] = deepcopy(untrained_cnn_model.conv1.
↪weight.cpu().detach())

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
    mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images
100%|                    | 100/100 [00:04<00:00, 24.28it/s]
CNN Acc: 64.66666666666667, MLP Acc: 55.333333333333336
Training with 20 images
100%|                    | 100/100 [00:06<00:00, 15.91it/s]
CNN Acc: 68.66666666666667, MLP Acc: 72.66666666666667
Training with 30 images
100%|                    | 100/100 [00:06<00:00, 14.34it/s]
CNN Acc: 67.33333333333333, MLP Acc: 73.33333333333333
Training with 40 images
100%|                    | 100/100 [00:08<00:00, 11.45it/s]
CNN Acc: 88.66666666666667, MLP Acc: 84.66666666666667
Training with 50 images
100%|                    | 100/100 [00:09<00:00, 10.17it/s]
CNN Acc: 94.0, MLP Acc: 88.0
```

[10]:
```
## Plot the validation accuracy
plt.plot(num_images_list, cnn_acc_list, marker='o', label='CNN')
plt.plot(num_images_list, mlp_acc_list, marker='o', label='MLP')
plt.xlabel('Number of training images per class')
plt.ylabel('Validation accuracy')
```

```
plt.legend()
plt.grid()
plt.show()
```



OK, in most cases, CNN looks like it is performing better than MLP. So can we conclude that CNN has the inductive biases of locality and translational invariance? Not yet. We need to conduct a series of other experiments to show that CNN has such inductive biases.

Seemingly, the experiment result is odd. First, the performance of the low data regime `num_train_images_per_class=10` is very bad, considering the task is straightforward. Second, some students will observe that the performance of MLP is better than CNN at some point. At least, CNN should be much better even in a small data regime if it is translational equivariant. How do we debug the model? We will study how to debug the model in the following problem.

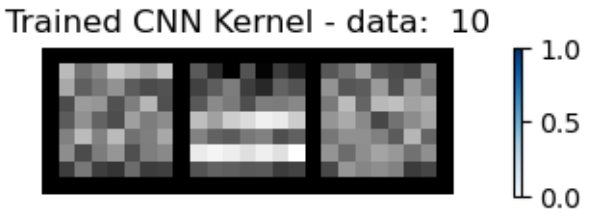
Here are some checklists that you can do to debug the problem.

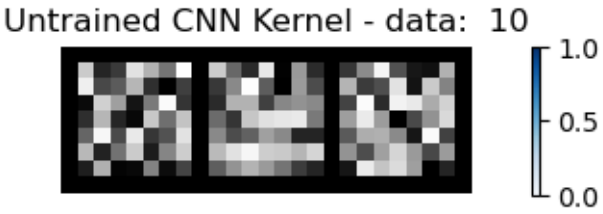
1. Did you check the dataset? For example, is the dataset balanced? Is the dataset noisy? Is the dataset too small?
2. Did you check the model architecture? For example, is the model architecture powerful enough to learn the dataset? Is the model architecture too complex? Is the model architecture too simple?

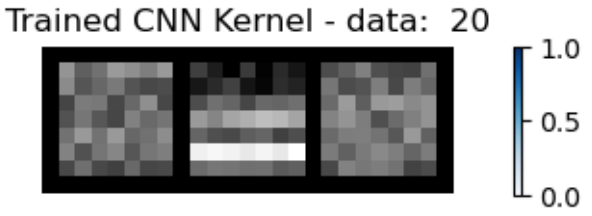
3. Did you check the model initialization? For example, is the model initialized properly? Is the model initialized randomly? Is the model initialized with the pre-trained weights?
4. Did you check that the model is trained correctly? For example, does the kernel look like an edge detector? What would be the performance of CNN if kernels were initialized with edge detectors?
5. Did you check the training procedure? For example, is the training procedure correct? Is the training procedure stable? Is the training procedure too slow?
6. Did you optimize the hyperparameters? For example, learning rate, batch size, and the number of epochs.

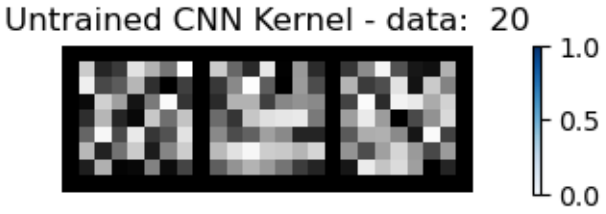
Note that we already checked the dataset, initialization, and model architecture. But we didn't check the step after 3. Let's step 4 first. We will first see what the learned weights look like, initialize the kernels with edge detectors, and see what happens.

[11]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel -␣
 ↪data:  {}'.format(num_image))
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN␣
 ↪Kernel - data:  {}'.format(num_image))
```

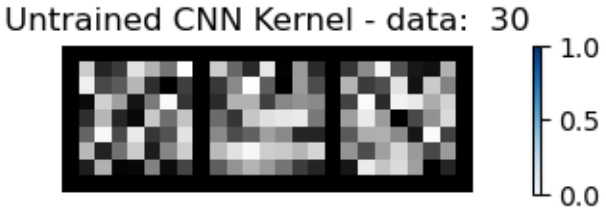


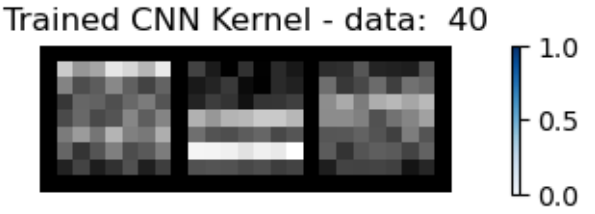


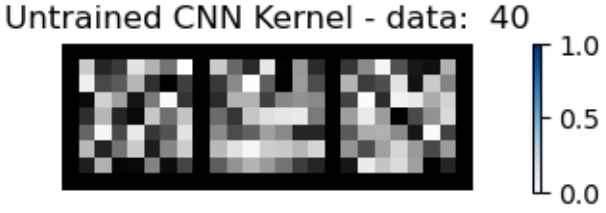


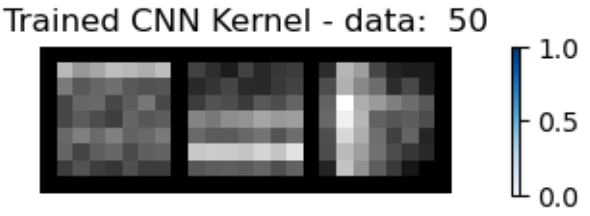

Untrained CNN Kernel - data: 20
1.0
0.5
0.0

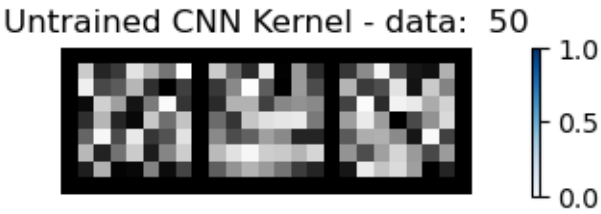

Trained CNN Kernel - data: 30
1.0
0.5
0.0


Untrained CNN Kernel - data: 30
1.0
0.5
0.0


Trained CNN Kernel - data: 40
1.0
0.5
0.0


Untrained CNN Kernel - data: 40
1.0
0.5
0.0





Q. Compare the learned kernels, untrained kernels, and edge-detector kernels. What do you observe?

A. As we increase the sample size the untrained kernels are unaffected, as they are randomly initialized, the trained kernels all resemble the trained kernels from small data set. Though, the kernel that represents the no edge detection class seems to change dramtically across sample sizes. Specifically it takes on a cross shape that must be to eliminate erroneous edges.

Visualized kernels seem very odd. Some kernels look randomly generated. Think about the data generating process. The factor determining this dataset is the edge location, edge width, and the intensities of background and edges. Therefore, we might be able to get kernels that look like edge detectors. Then, the next logical question should be, what if kernels are initialized with edge detectors? How would the performance change? Because we inject the additional inductive biases into the model. We expect the validation accuracy to be much better and with fewer training examples. Let's try it.

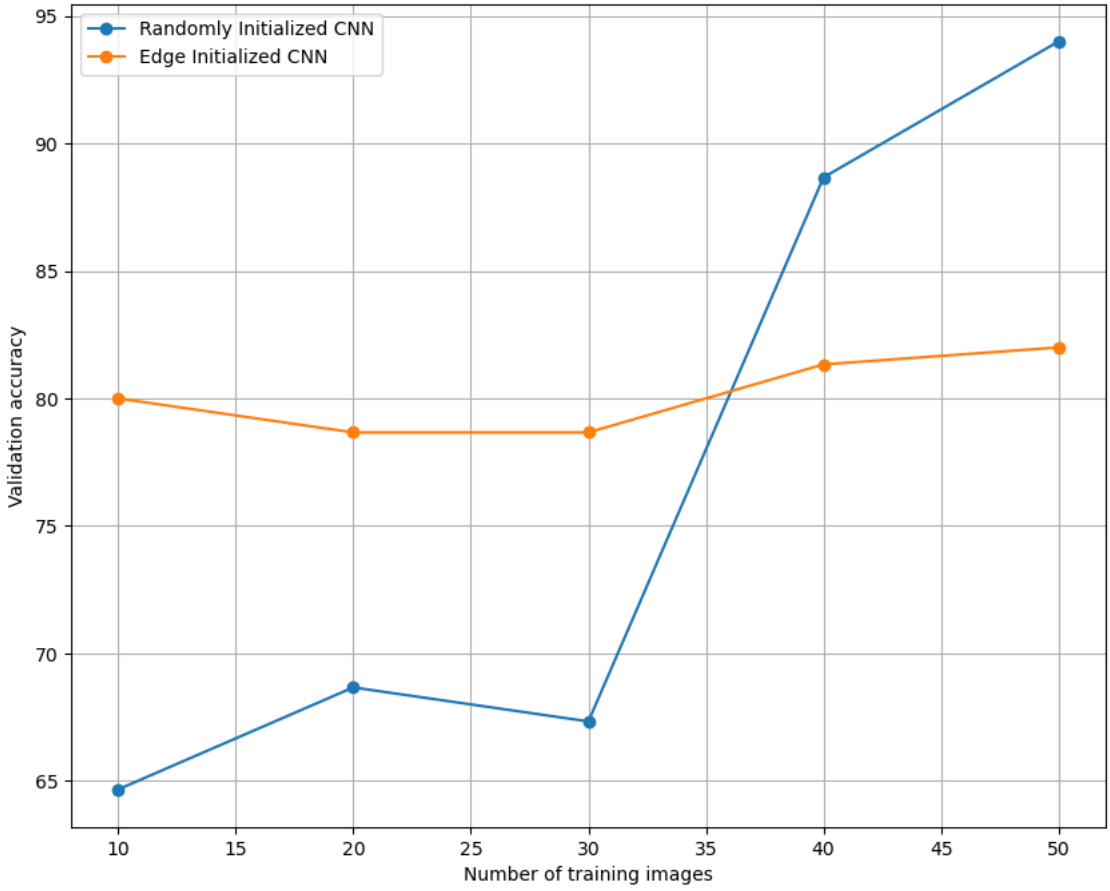
1.3.7 Injecting Inductive Bias: Initialize Kernels with Edge Detectors

[12]:
```
lr = 0.05
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

edge_init_cnn_acc_list = list()
```

```
for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    cnn_model = SimpleCNN(kernel_size=2)
    init_conv_kernel_with_edge_detector(cnn_model)
    freeze_conv_layer(cnn_model)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)


    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    edge_init_cnn_valid_acc_list = []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)
        cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion,␣
↪valid_loader, device, verbose=False)
        edge_init_cnn_valid_acc_list.append(cnn_valid_acc)

    cnn_acc = edge_init_cnn_valid_acc_list[-1]

    print("CNN Acc: {}".format(cnn_acc))
    edge_init_cnn_acc_list.append(cnn_acc)
```

```
Training with 10 images
100%|                    | 100/100 [00:01<00:00, 52.65it/s]
CNN Acc: 80.0
Training with 20 images
100%|                    | 100/100 [00:02<00:00, 35.41it/s]
CNN Acc: 78.66666666666667
Training with 30 images
100%|                    | 100/100 [00:02<00:00, 34.73it/s]
CNN Acc: 78.66666666666667
Training with 40 images
100%|                    | 100/100 [00:02<00:00, 34.00it/s]
CNN Acc: 81.33333333333333
Training with 50 images
100%|                    | 100/100 [00:03<00:00, 25.36it/s]
CNN Acc: 82.0
```

[13]:
```
## Plot the validation accuracy
plt.plot(num_images_list, cnn_acc_list, marker='o', label='Randomly Initialized␣
↪CNN')
plt.plot(num_images_list, edge_init_cnn_acc_list, marker='o', label='Edge␣
↪Initialized CNN')
plt.xlabel('Number of training images')
plt.ylabel('Validation accuracy')
plt.legend()
plt.grid()
plt.show()
```



As you can see in the above graph, the performance of CNN initialized with edge detectors is much better than CNN initialized with random weights. It is a significant observation, especially in a low data regime. Now we have to check the training procedure.

Q. We freeze the convolutional layer and train only final layer (classifier) in this experiment. For a high data regime, the performance of CNN initialized with edge detectors is worse than CNN initialized with random weights. Why do you think this happens?

A. The superior performance of the fully learned high data CNN must come from learned parameters

not in the final layer. The inductive bias from initilizing with edge detectors is enough to surpass the performacne of fully learned low data models, but the added data and flexibility of the high data model with fully learned parameters surpasses the accuracy of the initilization model. In Bayesian terms, when there is little data the prior (i.e. initilization) has a strong impact on posterior. As data increases its better to allow the posterior to change all its parameters.

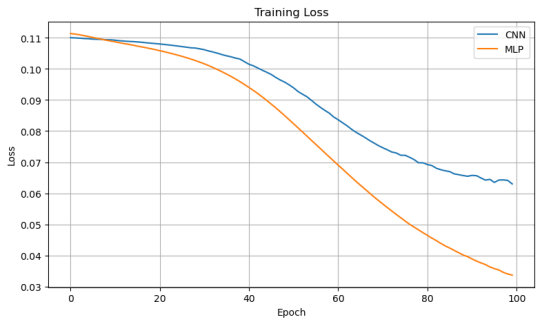
1.3.8 Q3. Checking the Training Procedure

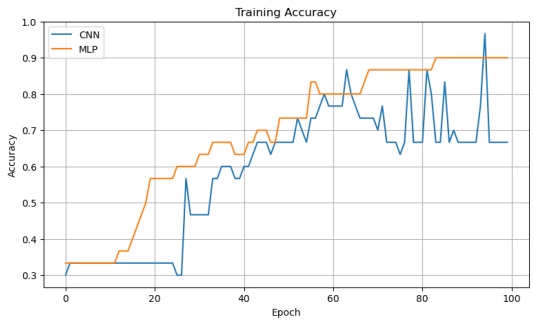
Checking the training procedure is very important. We must log at least training loss, training accuracy, validation loss, and validation accuracy. Let's log such training signals and find out what is going on.

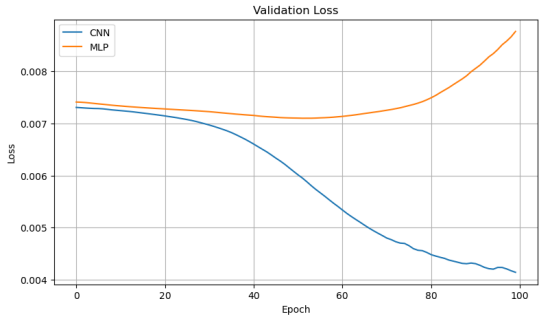
[14]:
```
lr = 5e-3
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
 ↪cnn_valid_loss_list = [], [], [], []
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list,␣
 ↪mlp_valid_loss_list = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
 ↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model,␣
 ↪mlp_optimizer, criterion, train_loader, device, epoch, verbose=False)
```

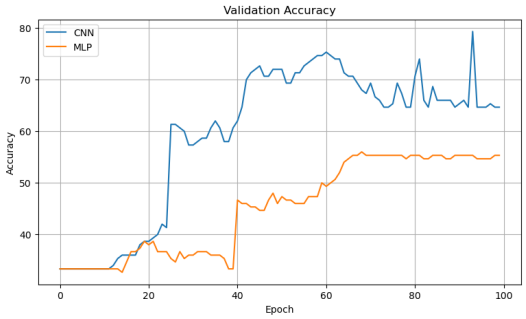
```
        cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion,␣
↪valid_loader, device, verbose=False)
        mlp_valid_loss, mlp_valid_acc, _ = evaluate(mlp_model, criterion,␣
↪valid_loader, device, verbose=False)

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list,␣
↪mlp_train_loss_list, mlp_train_acc_list)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list,␣
↪mlp_valid_loss_list, mlp_valid_acc_list)

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.data.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.
↪data.detach().cpu()

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
    mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images
100%|                    | 100/100 [00:04<00:00, 23.22it/s]
```

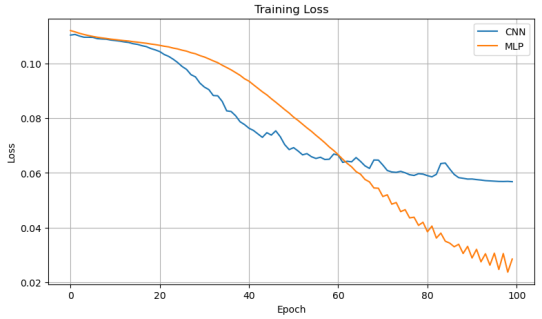


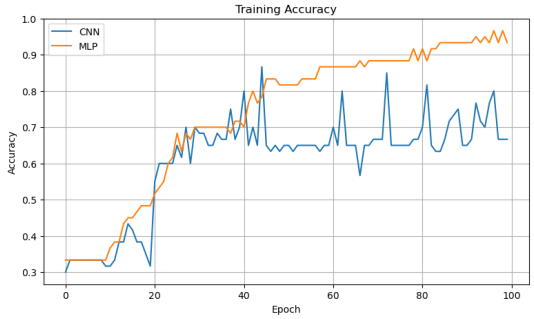


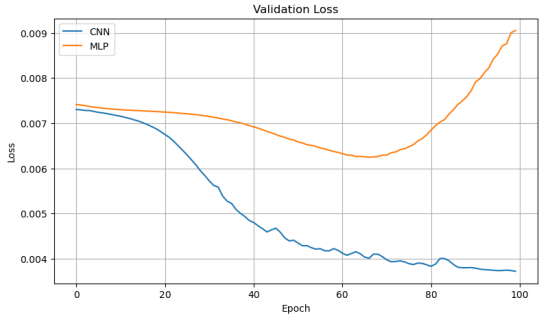


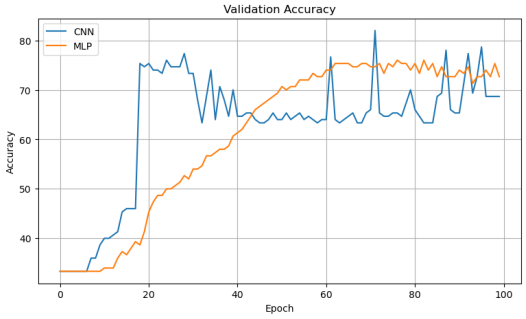



```
CNN Acc: 64.66666666666667, MLP Acc: 55.333333333333336
Training with 20 images
100%|                          | 100/100 [00:05<00:00, 17.78it/s]
```

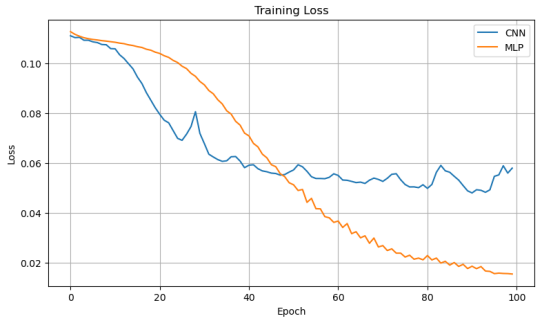


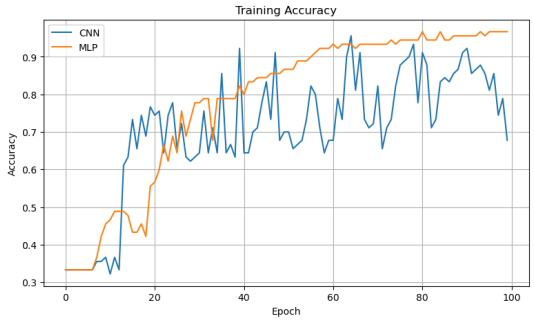


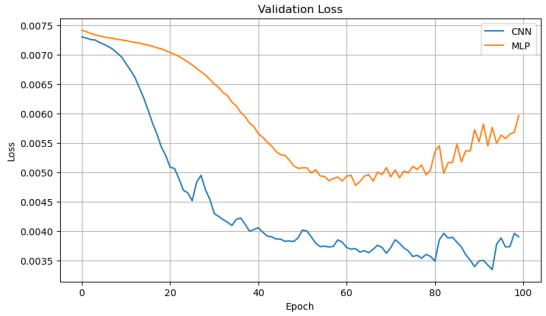


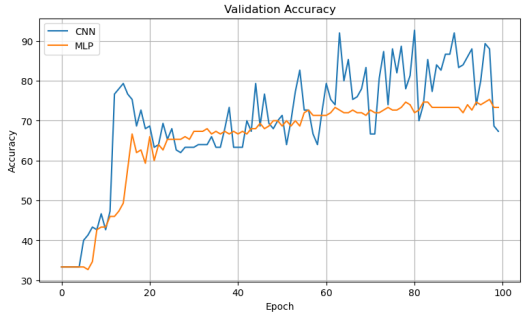



```
CNN Acc: 68.66666666666667, MLP Acc: 72.66666666666667
Training with 30 images
100%|                          | 100/100 [00:06<00:00, 15.14it/s]
```

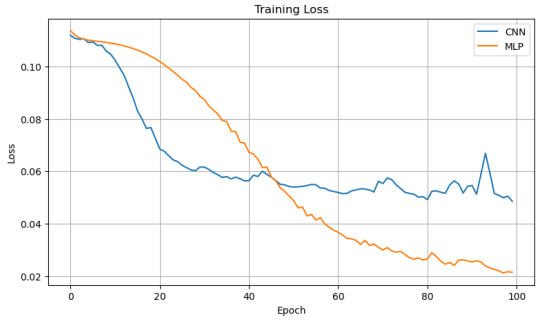


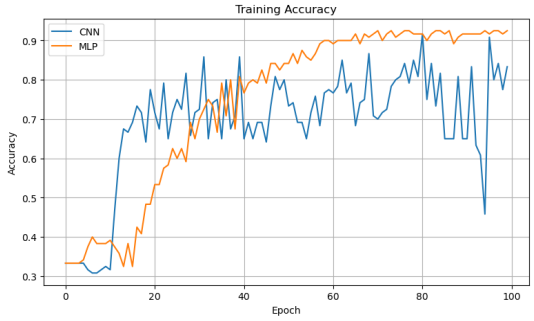


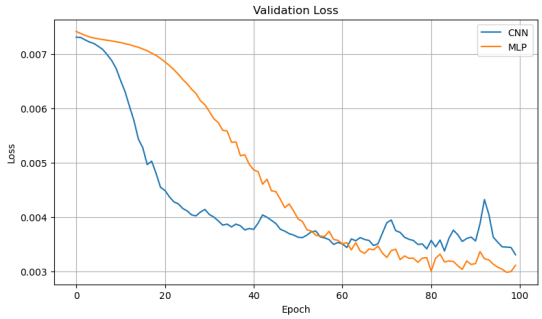


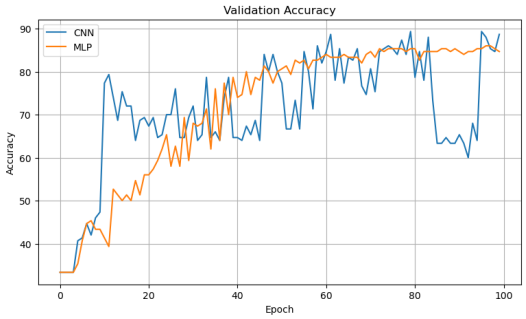



```
CNN Acc: 67.33333333333333, MLP Acc: 73.33333333333333
Training with 40 images
100%|                                   | 100/100 [00:08<00:00, 11.43it/s]
```

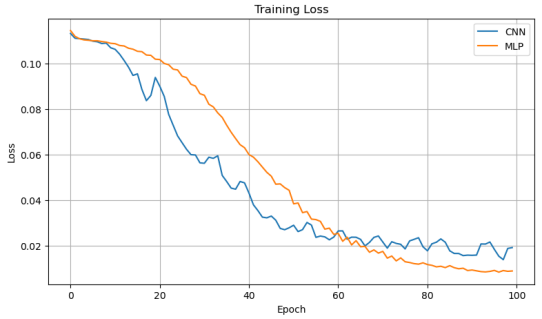


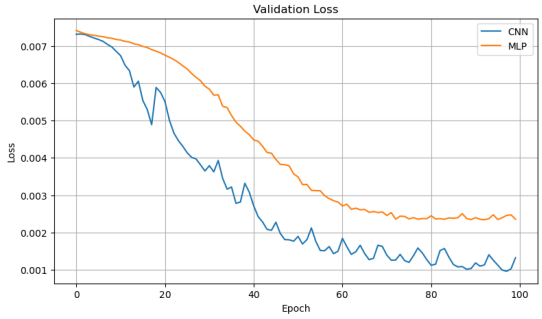


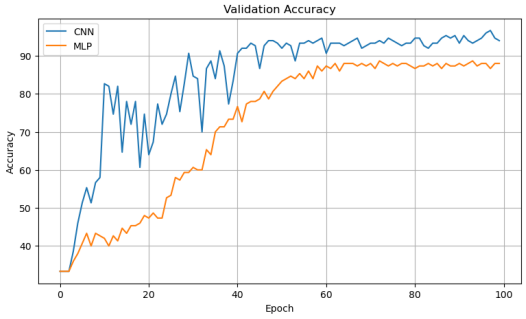





```
CNN Acc: 88.66666666666667, MLP Acc: 84.66666666666667
Training with 50 images
100%|                                   | 100/100 [00:10<00:00,  9.17it/s]
```








```
CNN Acc: 94.0, MLP Acc: 88.0
```

What is going on here? Validation loss and validation accuracy are not flat at the end. It means that the model is not converged. We need to train the model more. Let's train the model with the higher number of epochs. Increase the number of epochs until the validation loss and accuracy are flat.

Q. List every epochs that you trained the model. Final accuracy of CNN should be at least 90%

for 20 images per class. Final accuracy of MLP should be at least 80% for 10 images per class.

A. 300: CNN is 80% accurate at 20 images, MLP is 52% accurate with 10 images 500: CNN is 97% accurate at 20 images, MLP is 51% accurate with 10 images

Q. Check the learned kernels. What do you observe?

A. Again with 50 images and 500 epoch we get similiar kernels to the oroignal model. Speficailly, a top bar on the vertical model, two horizontal lines on the horizontal kernel, and a cross shape on the no edge kernel.

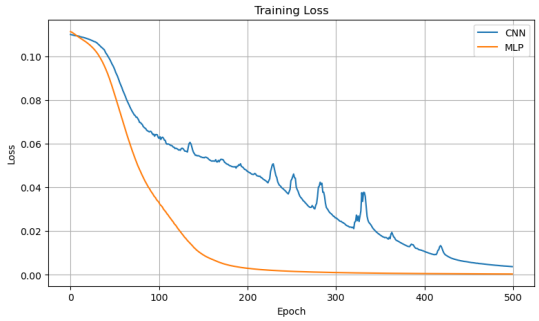
Q. (optional) You might find that with the high number of epochs, validation loss of MLP is increasing whild validation accuracy increasing. How can we interpret this? (Hint: Refer to this paper)

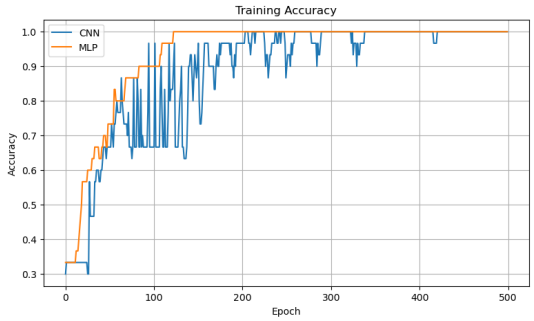
Q. (optional) Do hyperparameter tuning. And list the best hyperparameter setting that you found and report the final accuracy of CNN and MLP.

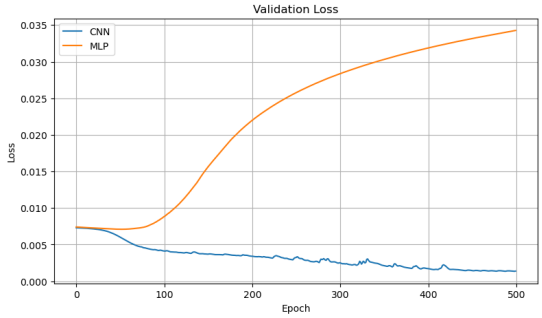
[15]:
```
################################################################################
# TODO: Try other num_epochs. Final accuracy of CNN should be at least      #
# 90% for 10 images per class.                                               #
################################################################################
num_epochs = 500
################################################################################
#                                END OF YOUR CODE                            #
################################################################################
lr = 5e-3
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
```

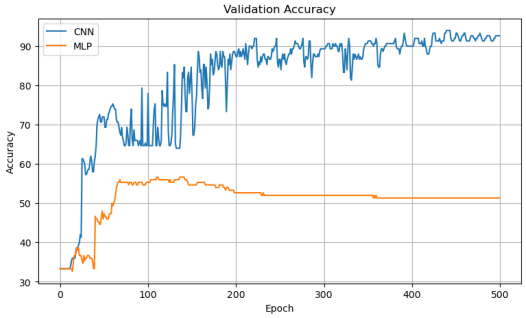
```
  cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
↪cnn_valid_loss_list = [], [], [], []
  mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list,␣
↪mlp_valid_loss_list = [], [], [], []
  for epoch in tqdm(range(num_epochs)):
      cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)
      mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model,␣
↪mlp_optimizer, criterion, train_loader, device, epoch, verbose=False)

      cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion,␣
↪valid_loader, device, verbose=False)
      mlp_valid_loss, mlp_valid_acc, _ = evaluate(mlp_model, criterion,␣
↪valid_loader, device, verbose=False)

      cnn_train_acc_list.append(cnn_train_acc)
      cnn_valid_acc_list.append(cnn_valid_acc)
      mlp_train_acc_list.append(mlp_train_acc)
      mlp_valid_acc_list.append(mlp_valid_acc)
      cnn_train_loss_list.append(cnn_train_loss)
      cnn_valid_loss_list.append(cnn_valid_loss)
      mlp_train_loss_list.append(mlp_train_loss)
      mlp_valid_loss_list.append(mlp_valid_loss)

  vis_training_curve(cnn_train_loss_list, cnn_train_acc_list,␣
↪mlp_train_loss_list, mlp_train_acc_list)
  vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list,␣
↪mlp_valid_loss_list, mlp_valid_acc_list)

  cnn_kernel_dict[num_image] = deepcopy(cnn_model.conv1.weight.data.detach().
↪cpu())
  untrained_cnn_kernel_dict[num_image] = deepcopy(untrained_cnn_model.conv1.
↪weight.data.detach().cpu())

  cnn_acc = cnn_valid_acc_list[-1]
  mlp_acc = mlp_valid_acc_list[-1]

  print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
  cnn_acc_list.append(cnn_acc)
  mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images
100%|                    | 500/500 [00:22<00:00, 22.49it/s]
```

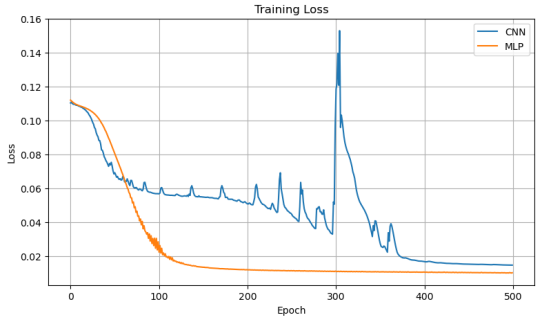


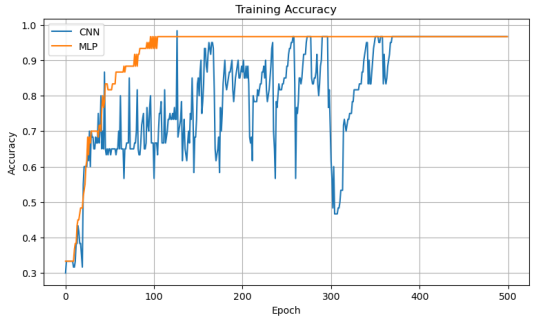


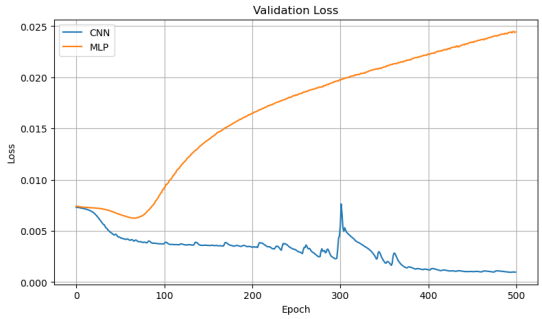


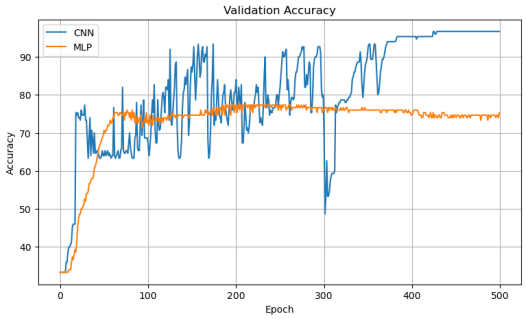



```
CNN Acc: 92.66666666666667, MLP Acc: 51.333333333333336
Training with 20 images
100%|                              | 500/500 [00:28<00:00, 17.35it/s]
```

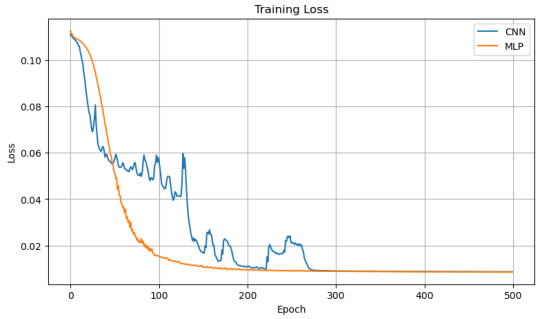


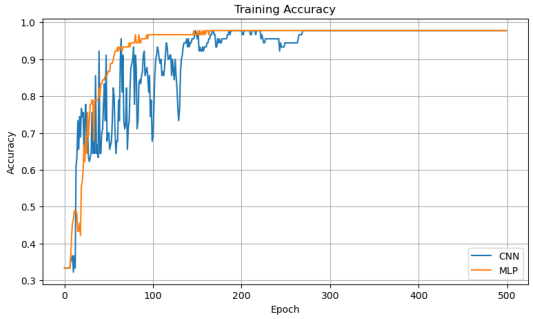


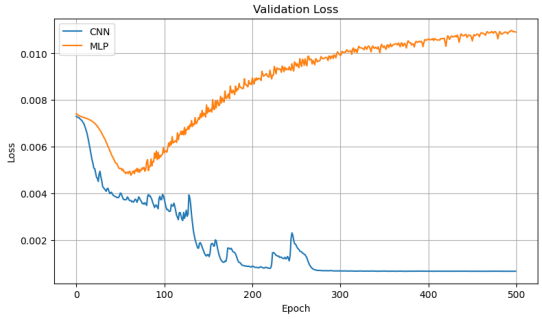


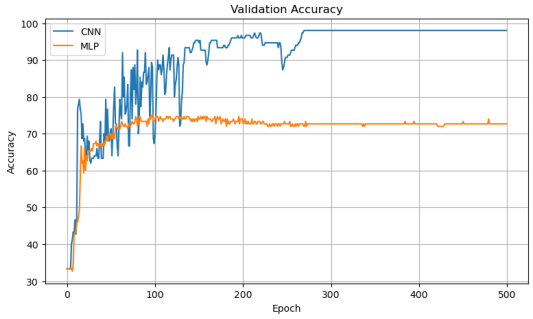



```
CNN Acc: 96.66666666666667, MLP Acc: 75.33333333333333
Training with 30 images
100%|                          | 500/500 [00:34<00:00, 14.56it/s]
```

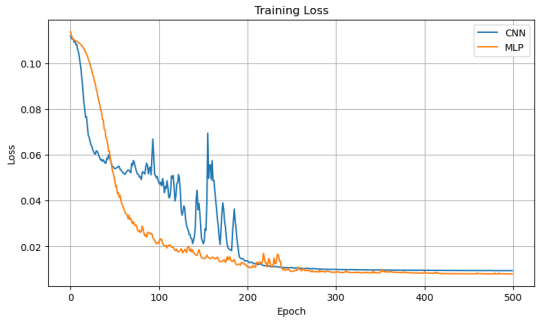


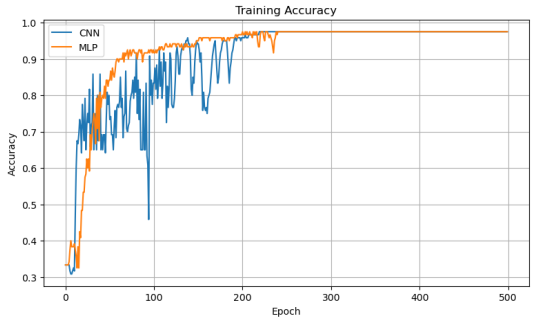


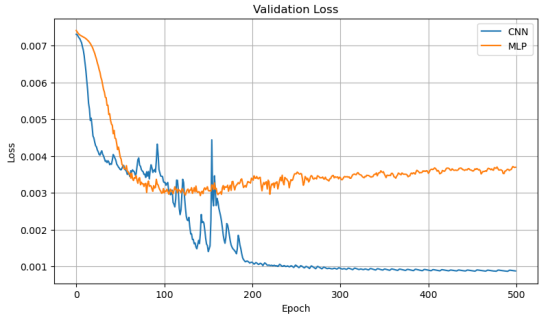


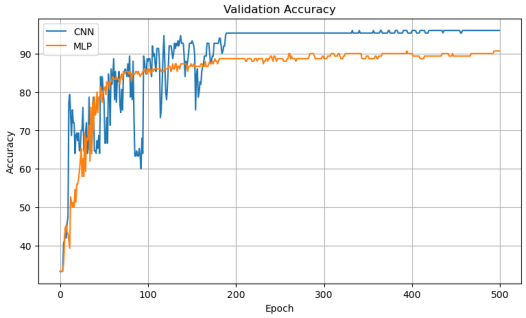



```
CNN Acc: 98.0, MLP Acc: 72.66666666666667
Training with 40 images
100%|                          | 500/500 [00:41<00:00, 11.92it/s]
```

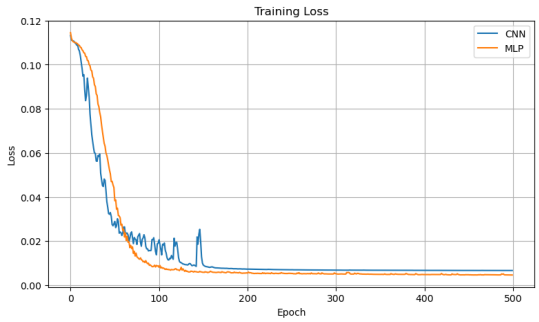


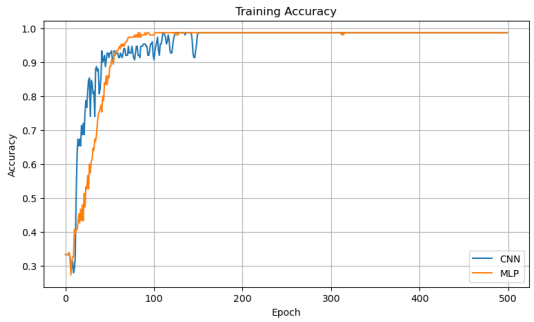


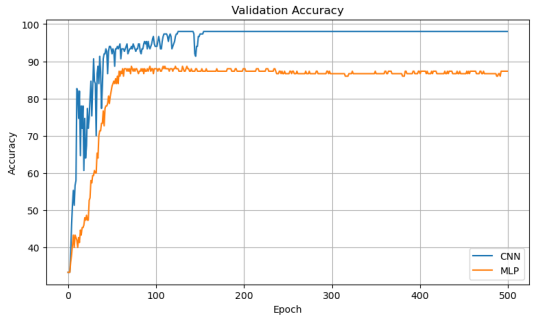





```
CNN Acc: 96.0, MLP Acc: 90.66666666666667
Training with 50 images
100%|                              | 500/500 [00:47<00:00, 10.52it/s]
```

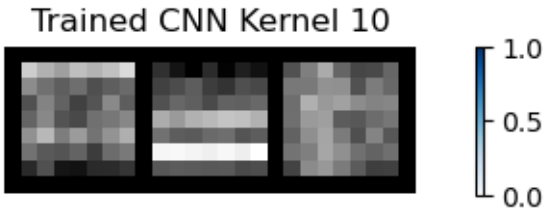


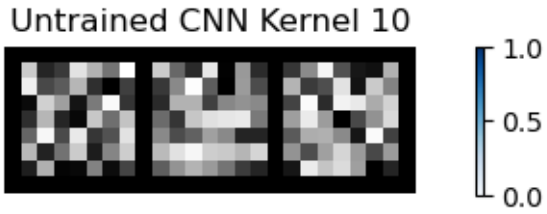


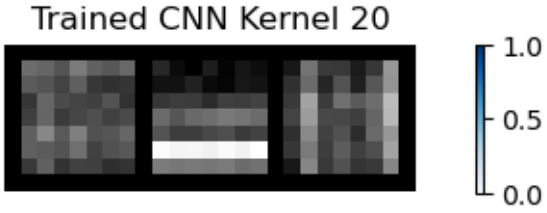


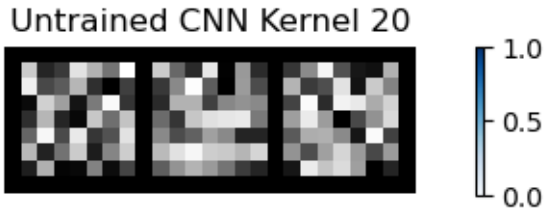

```
CNN Acc: 98.0, MLP Acc: 87.33333333333333
```

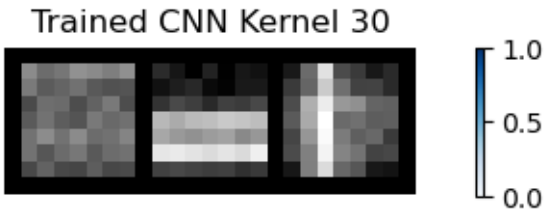
[16]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel␣
 ↪{}'.format(num_image))
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN␣
 ↪Kernel {}'.format(num_image))
```

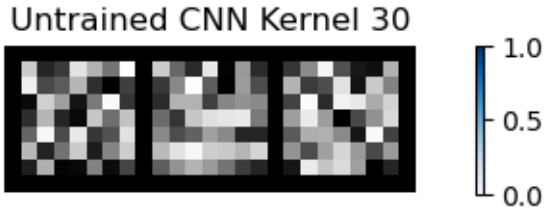


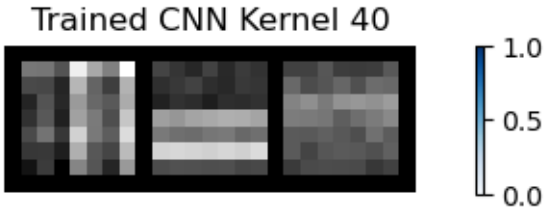


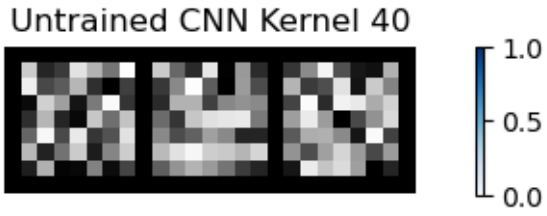


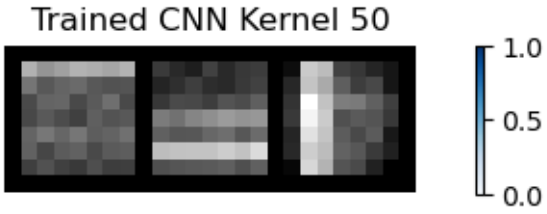

Untrained CNN Kernel 20
1.0
0.5
0.0

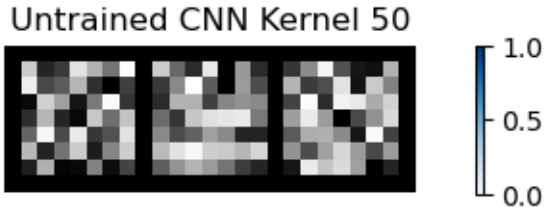

Trained CNN Kernel 30
1.0
0.5
0.0


Untrained CNN Kernel 30
1.0
0.5
0.0


Trained CNN Kernel 40
1.0
0.5
0.0







Q. How much more data is needed for MLP to get a competitive performance with CNN? Does MLP really generalize or memorize?

A. For instance here, with 50 images the MLP model achieves a validation accuracy of 90% while the CNN model only needs 10 images to get to 92% accuracy. Therefore the MLP models needs at least 5 times as many images. Moreover, the MLP models training accuracy is not always indictative of validation accuracy, therefore it seems that the MLP memorizes not generalizes.

1.3.9 Q4. Domain Shift between Training and Validation Set

In this problem, we will see how the model performance changes when the domain of the training set and that of the validation set are different. We will generate training set images with edges that locate only half of the image and validation set images with edges that locate only the other half of the image. Let's repeat the same experiment as the previous problem.

[17]:
```
set_seed(seed)
train_loader_dict = dict()
num_train_images_list = [10, 20, 30, 40, 50]
```

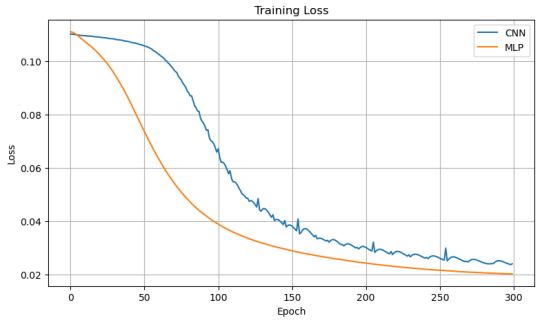
```
possible_edge_location_ratio = 0.5
valid_loader = None

transforms = T.Compose([T.ToTensor()])
batch_size = 10
################################################################################
# TODO: Implement train_loader_dict for each number of training images.      #
# Key: The number of training images (10, 50, 100, and 500)                  #
# Value: The corresponding dataloader                                        #
# The validation set size is 50 images per class                             #
# Hint: You can use the same code as above                                   #
# Hint: Pass possible_edge_location_ratio arguments to domain_config         #
# Hint: possible_edge_location_ratio is 0.5                                  #
################################################################################

for num_images in num_train_images_list:
    train_config = dict(
    data_per_class=num_images,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    possible_edge_location_ratio = possible_edge_location_ratio
    )

    train_dataset = EdgeDetectionDataset(train_config, mode='train',␣
 ↪transform=transforms)

    train_loader_dict[num_images] = torch.utils.data.
 ↪DataLoader(train_dataset,train_batch_size)

valid_config = dict(
    data_per_class=50,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    )

valid_dataset = EdgeDetectionDataset(valid_config, mode='train',␣
 ↪transform=transforms)

valid_loader = torch.utils.data.DataLoader(valid_dataset,valid_batch_size)

################################################################################
#                                END OF YOUR CODE                            #
################################################################################
```

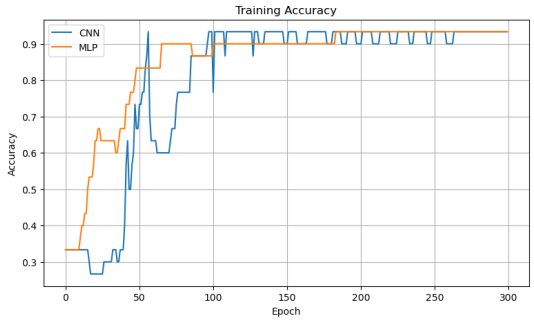
[18]:
```
lr = 3e-3
num_epochs = 300
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

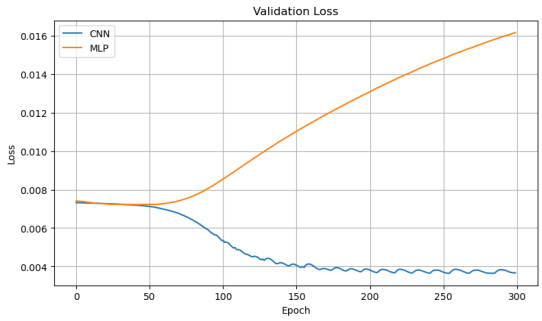
```
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()
mlp_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
 ↪cnn_valid_loss_list = [], [], [], []
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list,␣
 ↪mlp_valid_loss_list = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
 ↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model,␣
 ↪mlp_optimizer, criterion, train_loader, device, epoch, verbose=False)

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix =␣
 ↪evaluate(cnn_model, criterion, valid_loader, device, verbose=False)
        mlp_valid_loss, mlp_valid_acc, mlp_confusion_matrix =␣
 ↪evaluate(mlp_model, criterion, valid_loader, device, verbose=False)

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
```

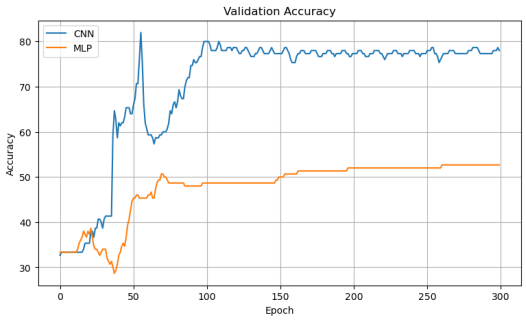
```
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list,␣
↪mlp_train_loss_list, mlp_train_acc_list)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list,␣
↪mlp_valid_loss_list, mlp_valid_acc_list)

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = deepcopy(cnn_model.conv1.weight.detach().cpu())
    untrained_cnn_kernel_dict[num_image] = deepcopy(untrained_cnn_model.conv1.
↪weight.detach().cpu())

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix
    mlp_confusion_matrix_dict[num_image] = mlp_confusion_matrix

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
    mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images
100%|                    | 300/300 [00:15<00:00, 19.00it/s]
```

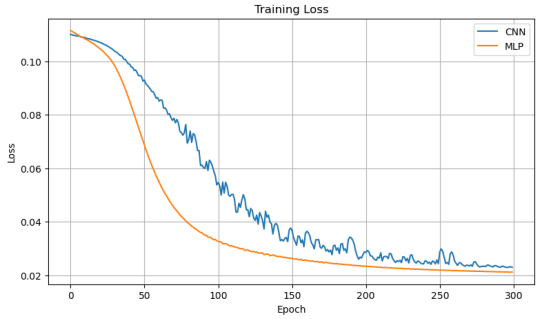


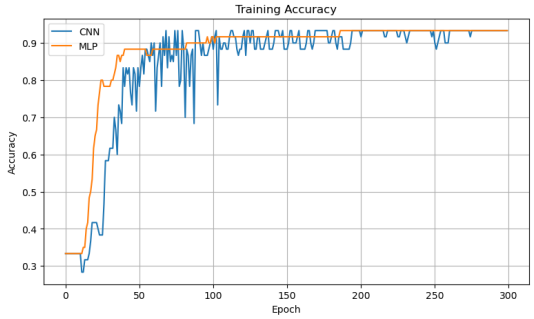


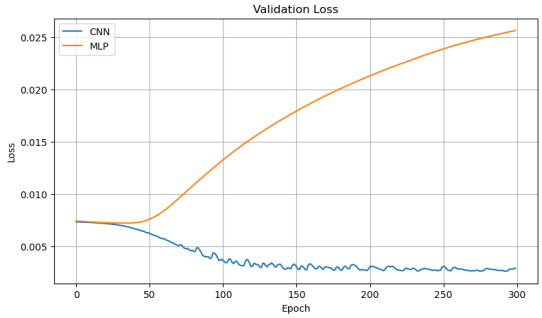


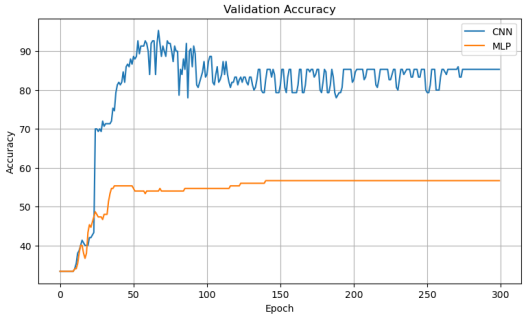


```
CNN Acc: 78.0, MLP Acc: 52.666666666666664
Training with 20 images
100%|                                   | 300/300 [00:17<00:00, 17.33it/s]
```

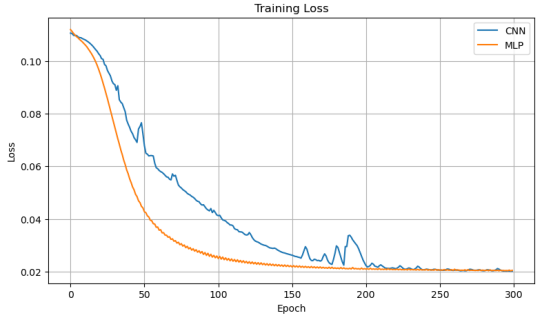


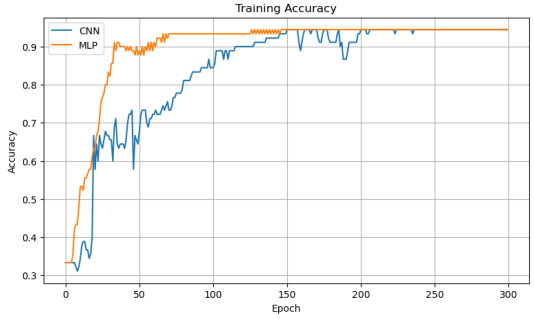


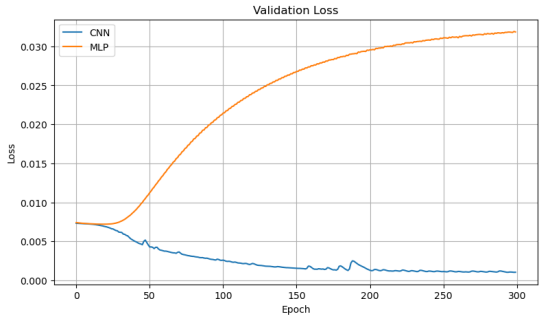


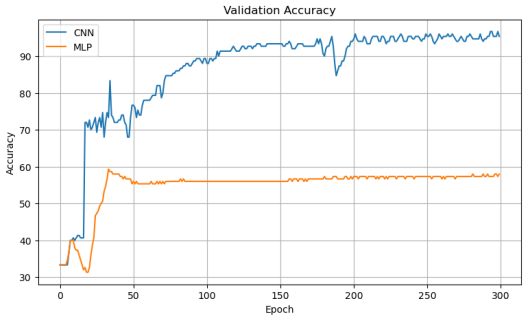



```
CNN Acc: 85.33333333333333, MLP Acc: 56.666666666666664
Training with 30 images
100%|                                   | 300/300 [00:24<00:00, 12.20it/s]
```

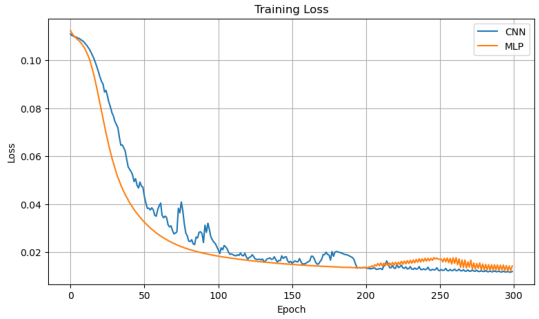


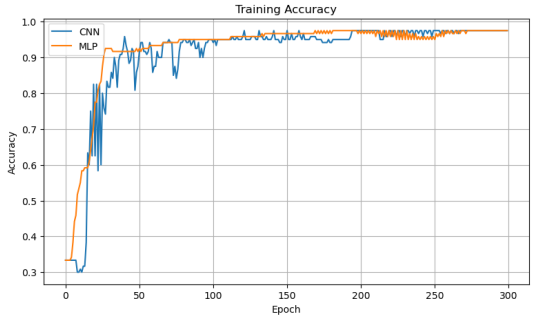


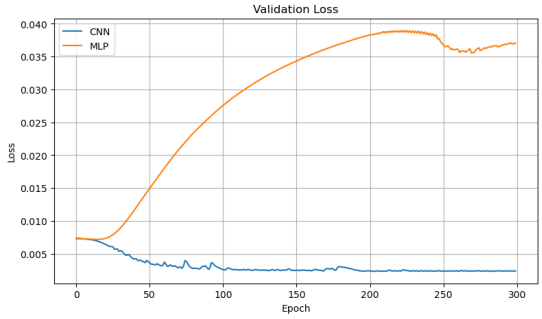


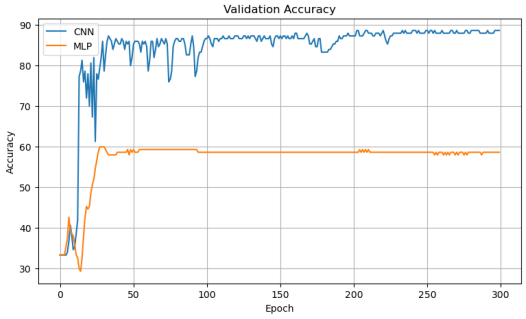



```
CNN Acc: 95.33333333333333, MLP Acc: 58.0
Training with 40 images
100%|                          | 300/300 [00:24<00:00, 12.27it/s]
```

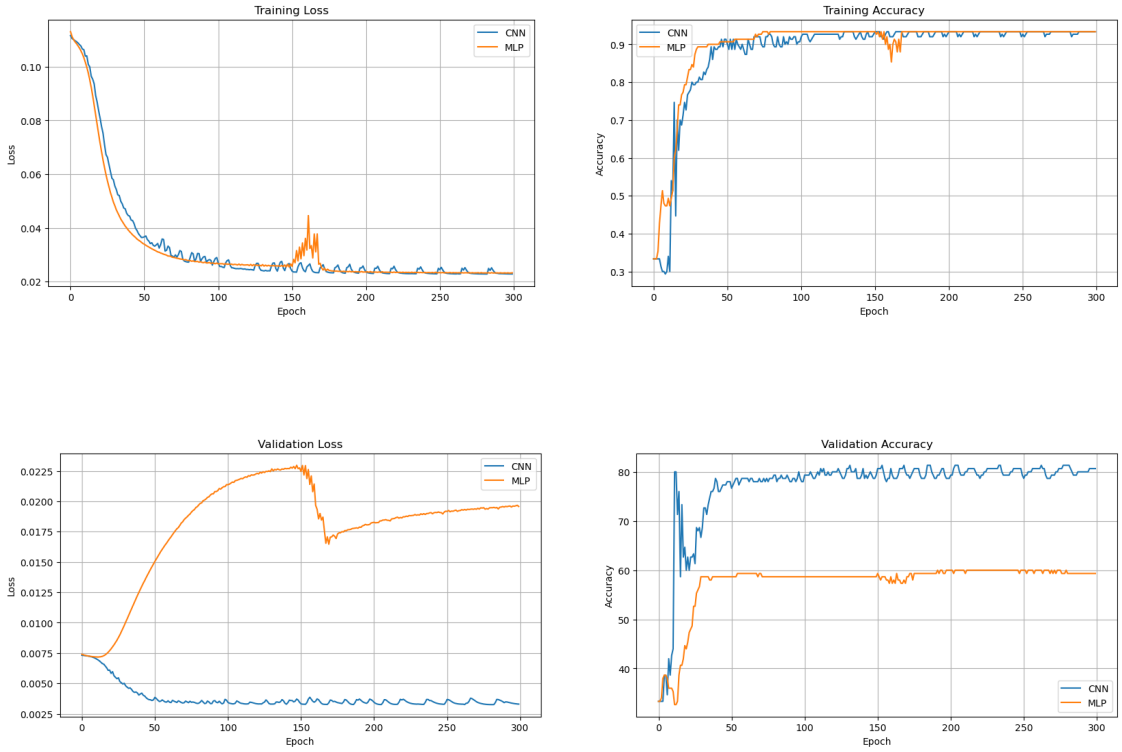






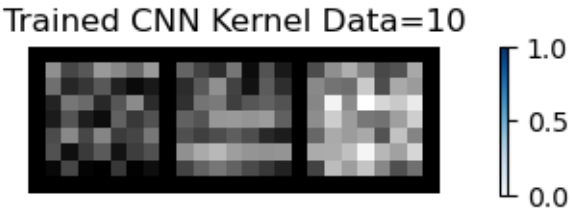


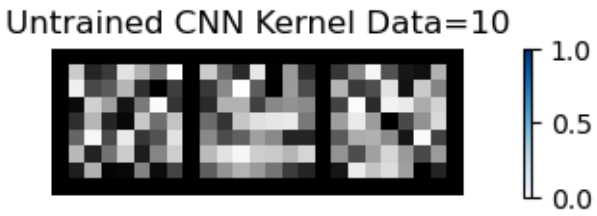

```
CNN Acc: 88.66666666666667, MLP Acc: 58.666666666666664
Training with 50 images
100%|                          | 300/300 [00:29<00:00, 10.02it/s]
```

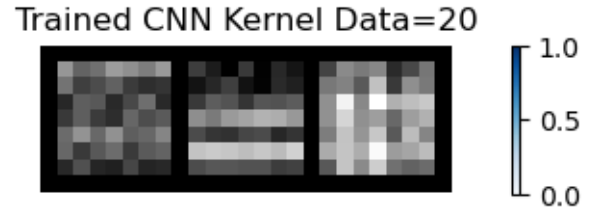


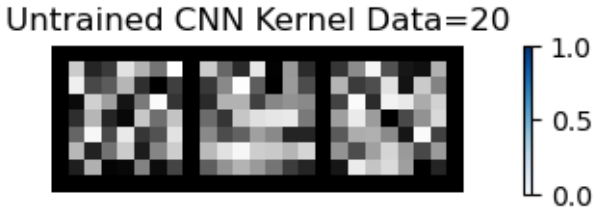

```
CNN Acc: 80.66666666666667, MLP Acc: 59.333333333333336
```

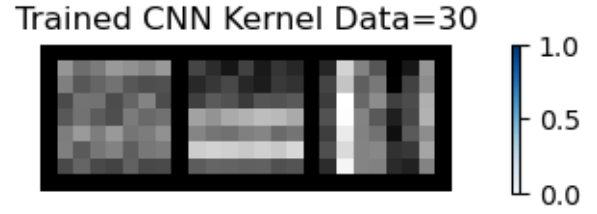
[19]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel␣
↪Data={}'.format(num_image))
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN␣
↪Kernel Data={}'.format(num_image))
```

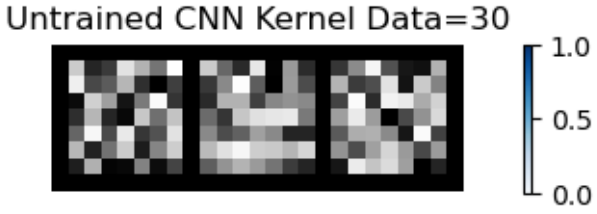


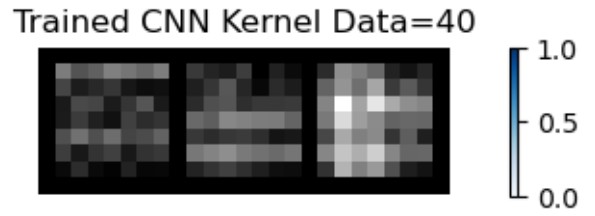

Untrained CNN Kernel Data=10
1.0
0.5
0.0

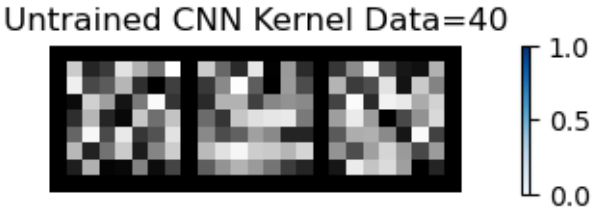

Trained CNN Kernel Data=20
1.0
0.5
0.0

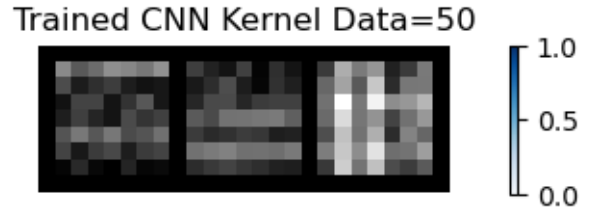

Untrained CNN Kernel Data=20
1.0
0.5
0.0

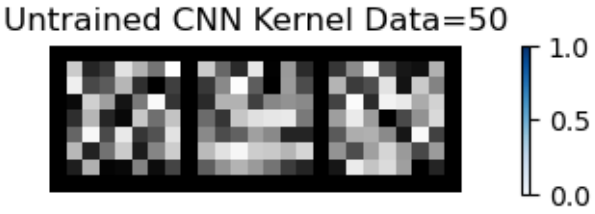

Trained CNN Kernel Data=30
1.0
0.5
0.0


Untrained CNN Kernel Data=30
1.0
0.5
0.0









In this example, you will see that both CNN and MLP performance are worse than those in the previous question. If two models learn how to extract edges, they should be able to classify the images with edges even though the edges locate in the other half of the images. However, both models fail to do so. What would be the problem? To investigate this, let's first look at the confusion matrices for both models link.

[24]:
```
## Plot the confusion matrix
for num_image, cnn_confusion_matrix in cnn_confusion_matrix_dict.items():
    mlp_confusion_matrix = mlp_confusion_matrix_dict[num_image]
    vis_confusion_matrix(cnn_confusion_matrix, ['horizontal', 'vertical',␣
 ↪'none'], 'CNN-{}-images'.format(num_image))
    #vis_confusion_matrix(mlp_confusion_matrix, ['horizontal', 'vertical',␣
 ↪'none'], 'MLP-{}-images'.format(num_image))
```

```
/Users/Ethan/Dropbox (Personal)/CS 282/HW5/helpers/vis_helper.py:154:
RuntimeWarning: More than 20 figures have been opened. Figures created through
the pyplot interface (`matplotlib.pyplot.figure`) are retained until explicitly
closed and may consume too much memory. (To control this warning, see the
rcParam `figure.max_open_warning`).
  fig = plt.figure(figsize=(10, 10))
```

Q. Why do you think the confusion matrix looks like this? Why does CNN misclassify the images with edge to those without edge? Why does MLP misclassify the images with vertical edge to those with horizontal edges and vice versa? (Hint: Visualize some of the images in the training and validation set. And we are using kernel_size=7, which is large relative to the image size.)

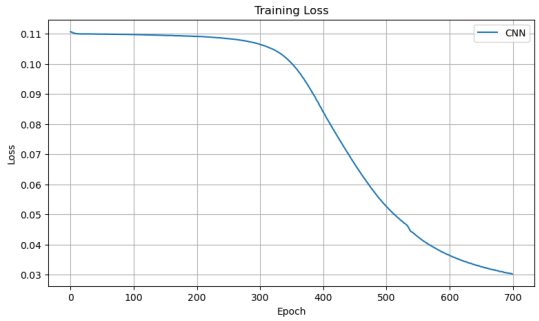
A. The CNN is classifying the images with edges in the region not covered in the training set as not having edges because there were no training inputs with edges in that region. The same reasoning explains why vertical edges are classified as horizontal - the algorithm is finding an edge in the region that was not covered by the training sample and subsequently misidentifying.

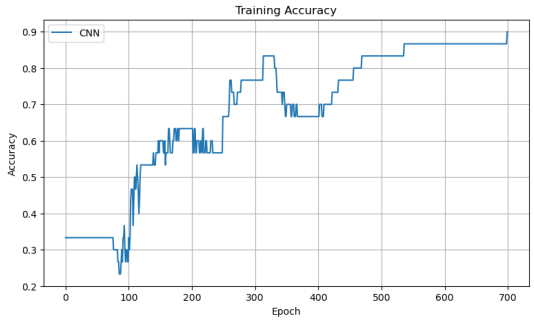
We can do better than this. We didn't explore hyperparameter space yet. Let's search hyperparameters that can generalize well to the validation set. We will change the learning rate, the number of epochs, and kernel size for CNN.

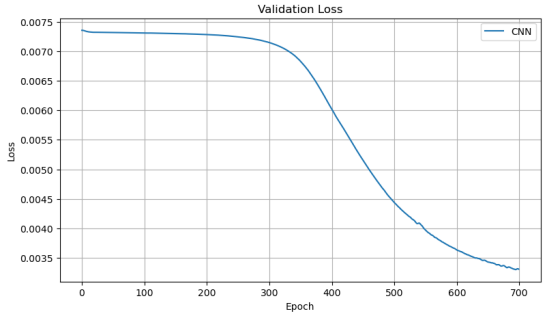
[27]:
```
################################################################################
# TODO: Try other num_epochs, lr, kernel_size. The validation accuracy      #
# should achieve 80% for 10 images per class.                               #
################################################################################
lr = 4e-3
num_epochs = 700
kernel_size = 3
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_valid_acc_list = list()

cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
```

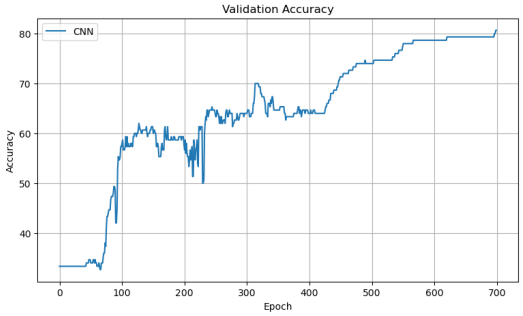
```
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=kernel_size)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
↪cnn_valid_loss_list = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix =␣
↪evaluate(cnn_model, criterion, valid_loader, device, verbose=False)

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, None, None)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, None, None)

    cnn_acc = cnn_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.
↪detach().cpu()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix

    print("CNN Acc: {}".format(cnn_acc))
    cnn_acc_list.append(cnn_acc)
```

```
Training with 10 images
100%|                    | 700/700 [00:15<00:00, 43.82it/s]
```

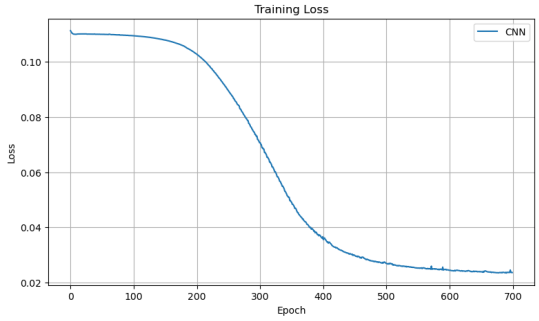


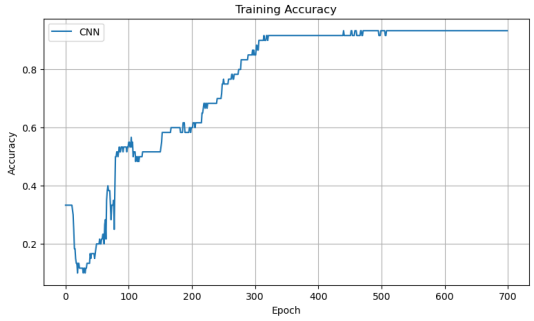


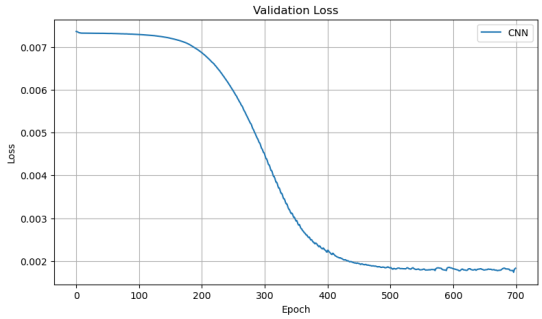


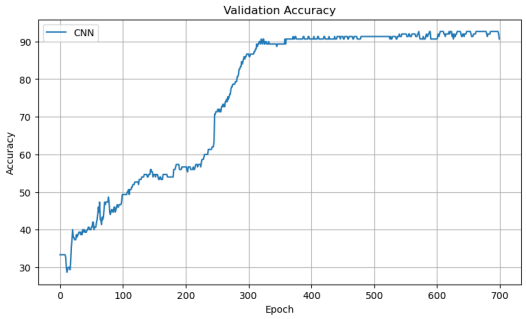



```
CNN Acc: 80.66666666666667
Training with 20 images
100%|                          | 700/700 [00:22<00:00, 31.10it/s]
```

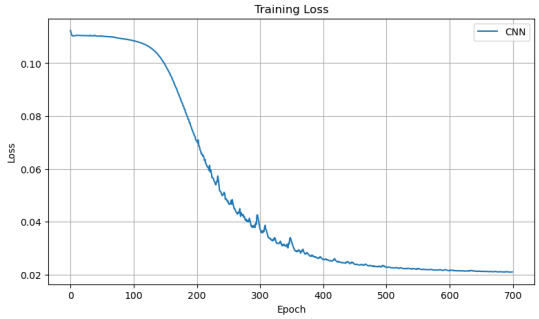


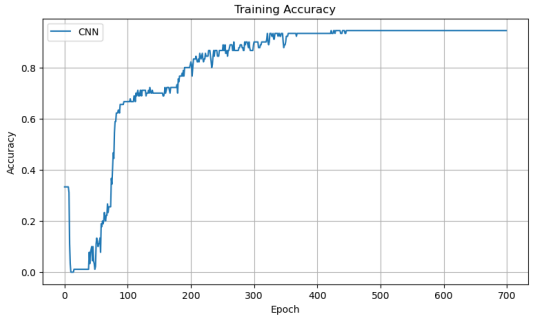


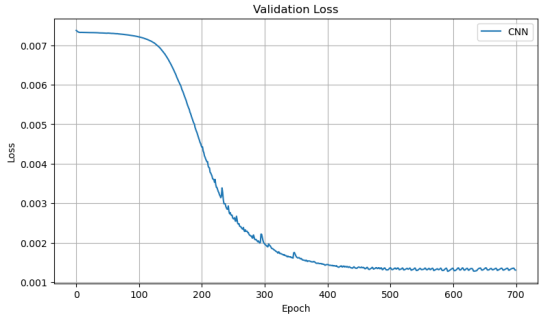


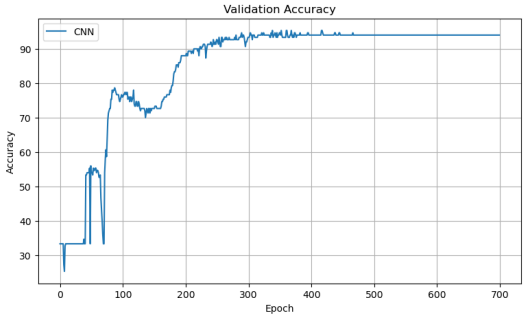



```
CNN Acc: 90.66666666666667
Training with 30 images
100%|                    | 700/700 [00:26<00:00, 26.06it/s]
```

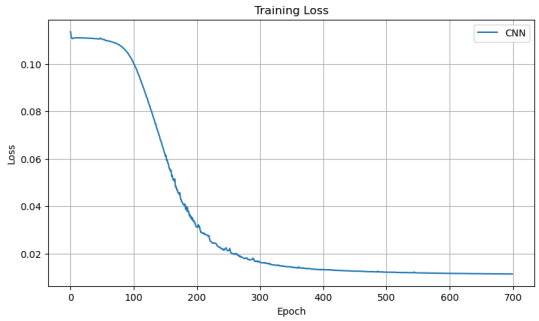


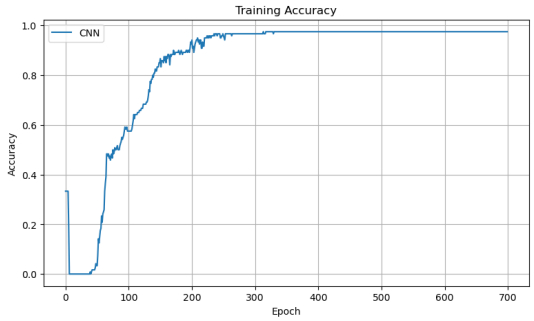


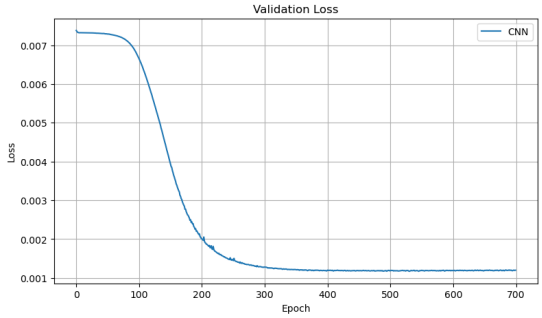


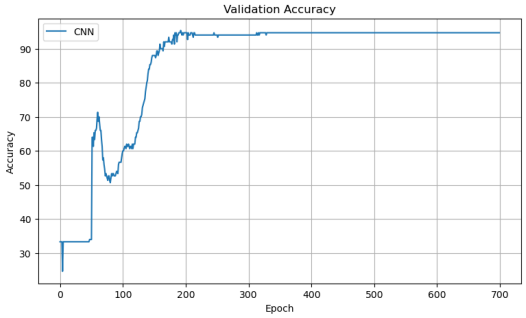



```
CNN Acc: 94.0
Training with 40 images
100%|                    | 700/700 [00:31<00:00, 21.99it/s]
```

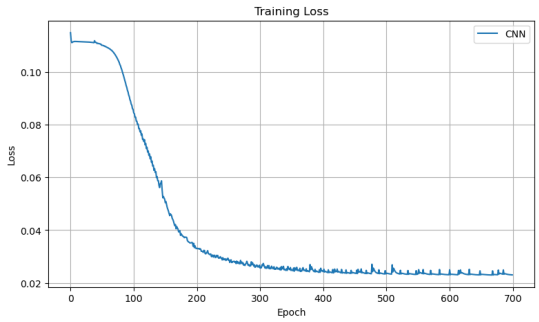


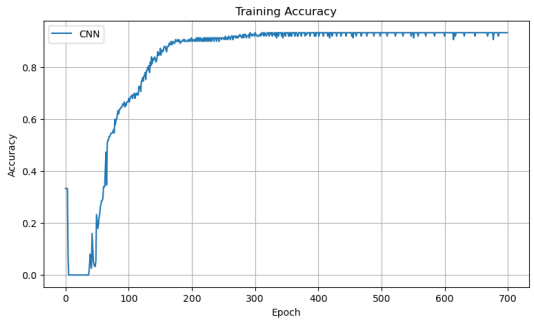


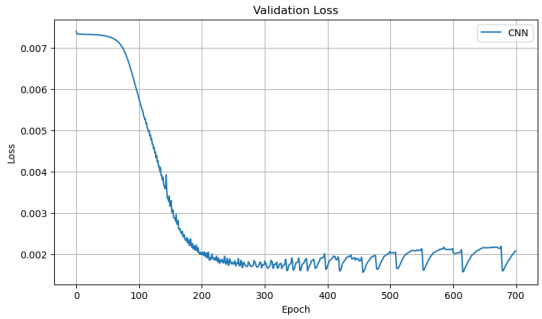


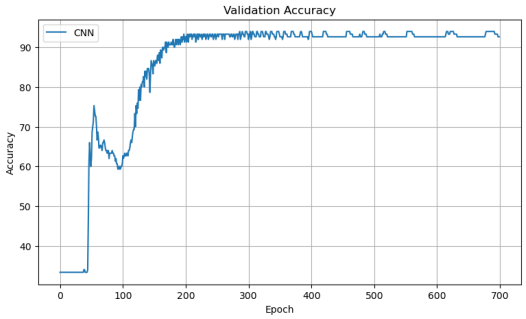



```
CNN Acc: 94.66666666666667
Training with 50 images
100%|                          | 700/700 [00:43<00:00, 15.92it/s]
```







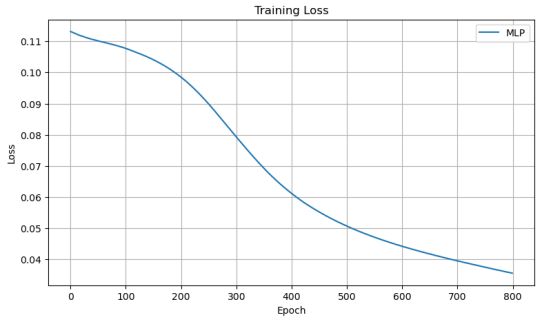


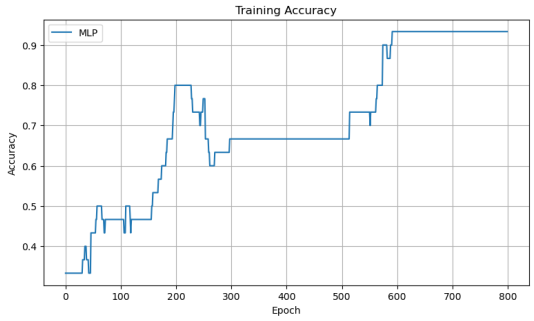

```
CNN Acc: 92.66666666666667
```

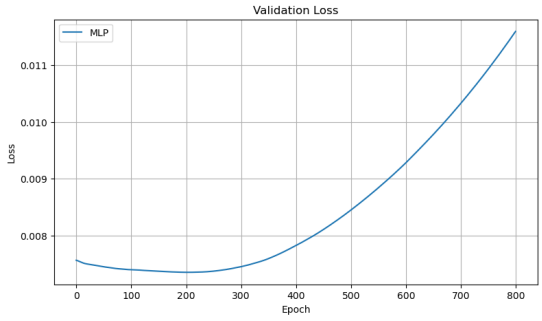
[31]:
```
################################################################################
# TODO: Try other num_epochs, lr, kernel_size. The validation accuracy      #
# should achieve 40% for 10 images per class.                               #
# len(hidden_dims) should be 2.                                             #
################################################################################
lr = 5e-4
num_epochs = 800
hidden_dims = [4, 4]
################################################################################
#                              END OF YOUR CODE                             #
################################################################################

assert len(hidden_dims) == 2, "len(hidden_dims) should be 2."

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

mlp_valid_acc_list = list()

mlp_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)

    mlp_model = ThreeLayerMLP(hidden_dims=hidden_dims)
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list,␣
 ↪mlp_valid_loss_list = [], [], [], []
```

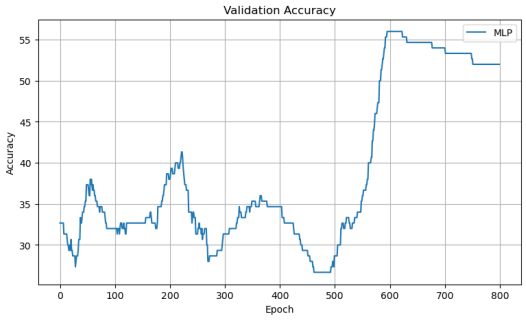
```
    for epoch in tqdm(range(num_epochs)):
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model,␣
↪mlp_optimizer, criterion, train_loader, device, epoch, verbose=False)

        mlp_valid_loss, mlp_valid_acc, mlp_confusion_matrix =␣
↪evaluate(mlp_model, criterion, valid_loader, device, verbose=False)

        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve([], [], mlp_train_loss_list, mlp_train_acc_list)
    vis_validation_curve([], [], mlp_valid_loss_list, mlp_valid_acc_list)

    mlp_acc = mlp_valid_acc_list[-1]

    mlp_confusion_matrix_dict[num_image] = mlp_confusion_matrix

    print("MLP Acc: {}".format(mlp_acc))
    mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images
100%|                    | 800/800 [00:12<00:00, 62.86it/s]
```

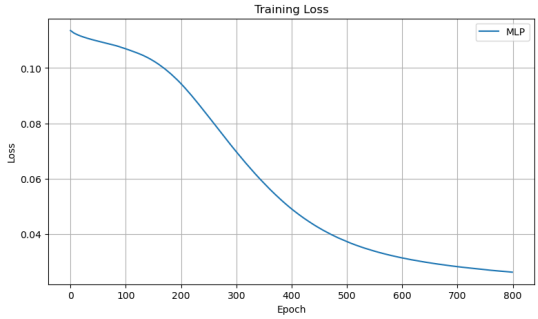


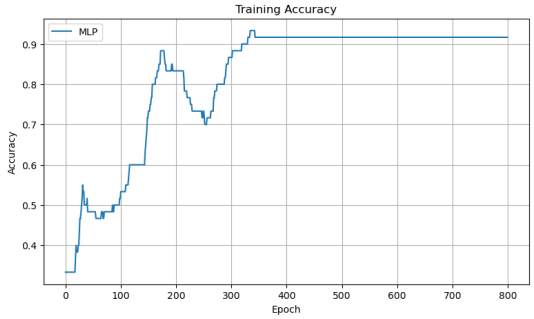


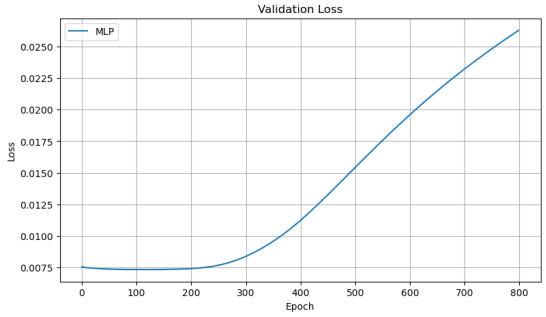


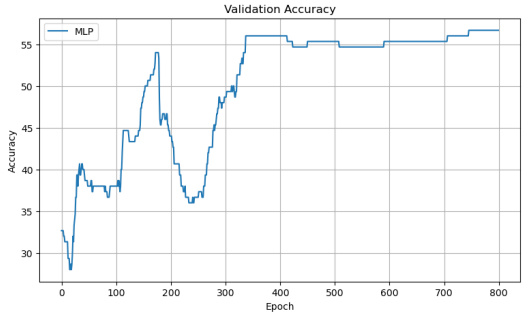


```
MLP Acc: 52.0
Training with 20 images
100%|                                   | 800/800 [00:15<00:00, 50.29it/s]
```

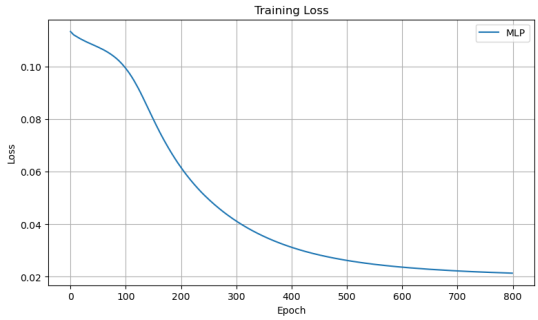


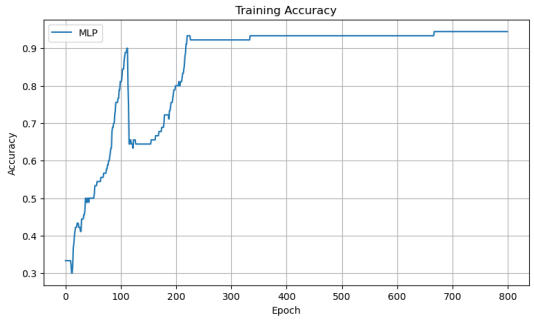


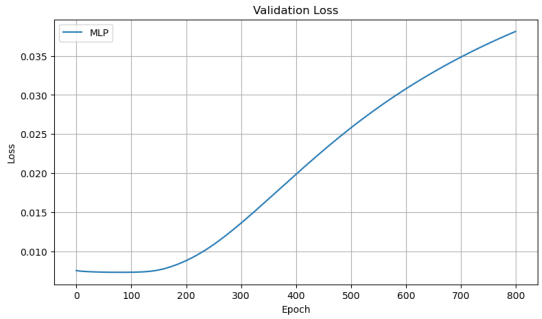


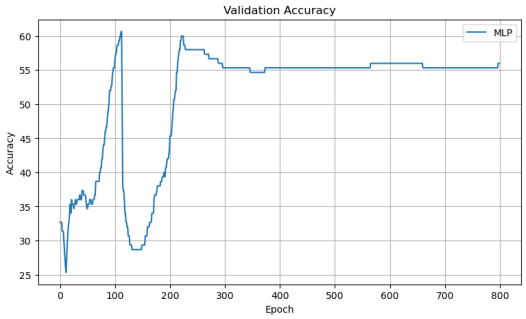



```
MLP Acc: 56.666666666666664
Training with 30 images
100%|                                   | 800/800 [00:17<00:00, 45.38it/s]
```

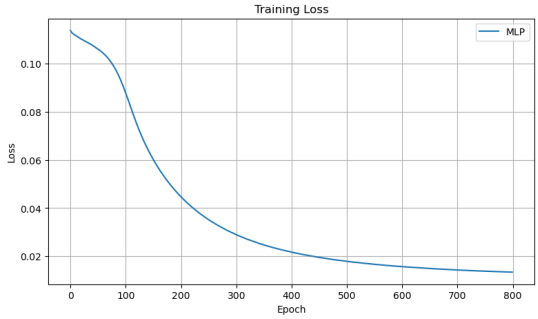


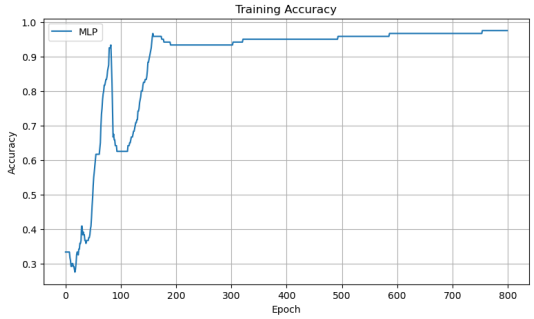


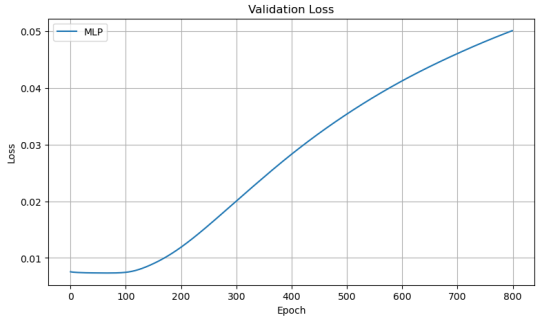


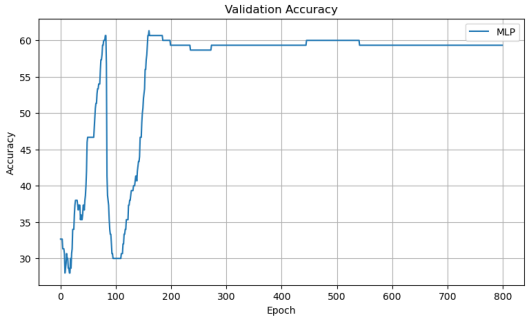



```
MLP Acc: 56.0
Training with 40 images
100%|                          | 800/800 [00:24<00:00, 32.84it/s]
```

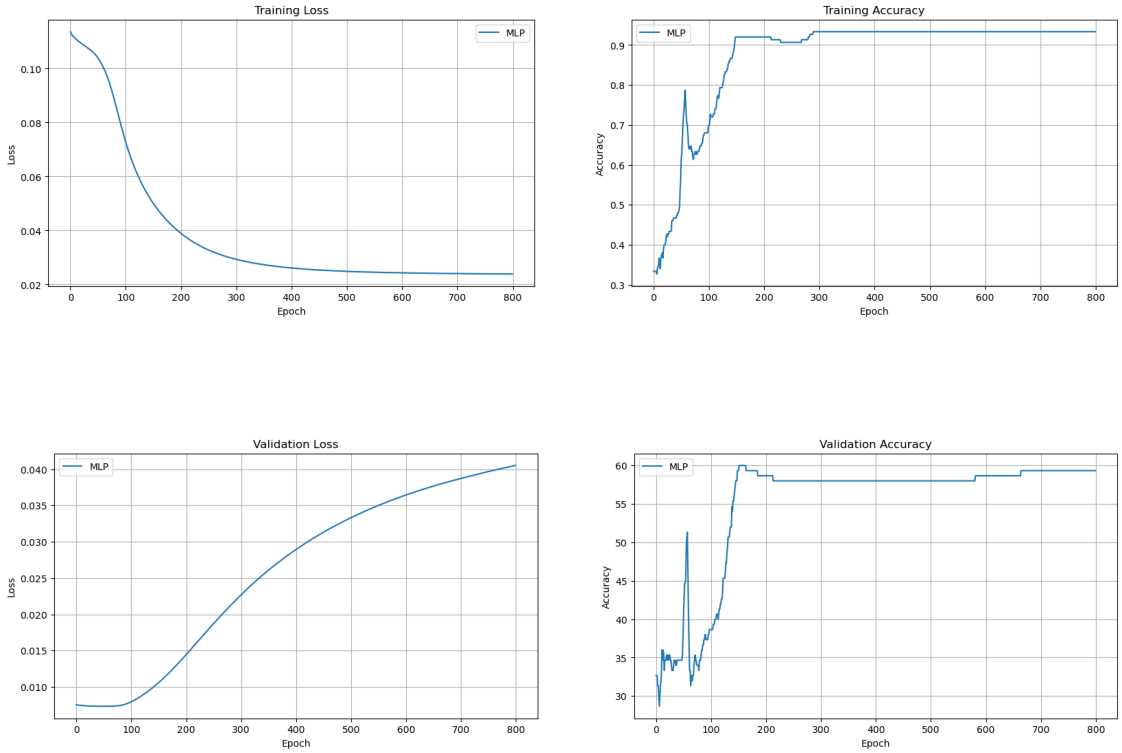









```
MLP Acc: 59.333333333333336
Training with 50 images
100%|                          | 800/800 [00:24<00:00, 32.39it/s]
```




```
MLP Acc: 59.333333333333336
```

Q. Why do you think MLP fails to learn the task while CNN can learn the task? (Hint: Think about the model architecture.)

A. The CNN architecture, with a small kernel size, can learn the local features of an edge and translate that to areas outside the training edge region. This translational property of CNNs allows them to better learn edges and identify them in new images.

1.3.10 Q5. When CNN is Worse than MLP

In this problem, we will see that CNN is not always better than MLP in the image domain. Using CNN assumes that the data has locally correlated, whatever data looks. We can manually 'whiten' or remove such local correlation simply by applying random permutation to the images. A random permutation matrix is a matrix that has the same number of rows and columns. Each row and column has the same number of 1s. The rest of the elements are 0s. For example, the following is a random permutation matrix.

```
[[0, 1, 0, 0],
 [0, 0, 0, 1],
 [1, 0, 0, 0],
 [0, 0, 1, 0]]
```

This matrix randomly reorders the elements of the vector. For example, if we apply this matrix to the vector `[1, 2, 3, 4]`, we will get `[2, 4, 1, 3]`. If we apply this matrix to the image, we will

get the image with the same content, but the pixels are randomly shuffled. One property of the random permutation matrix is that it is invertible. It means that we can recover the original image by simply applying the inverse matrix to the shuffled image. From the information-theoretical perspective, the random permutation matrix preserves the mutual information of the image and the label.

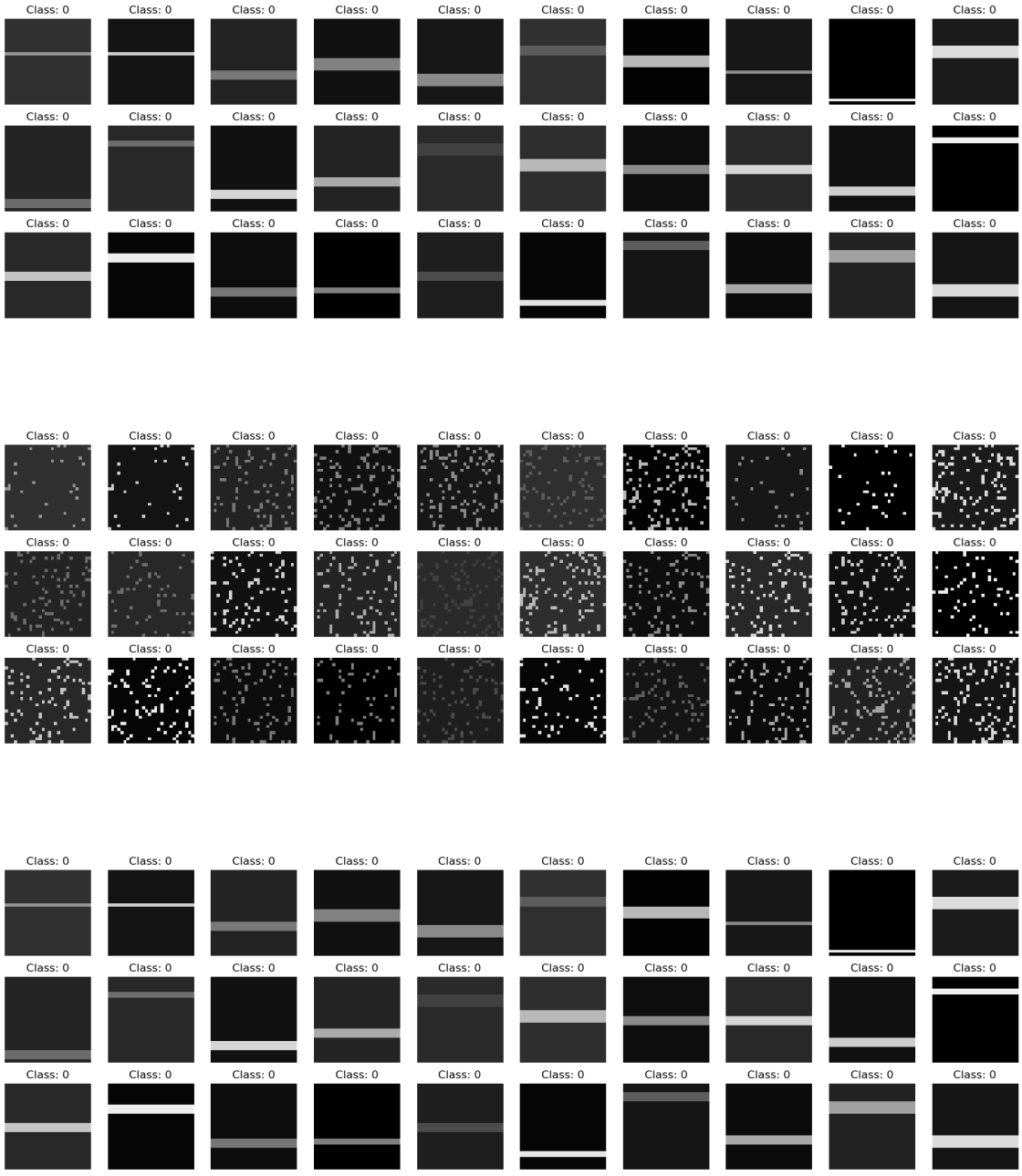
We will repeat the same experiment as the previous problem. Visualize the dataset first.

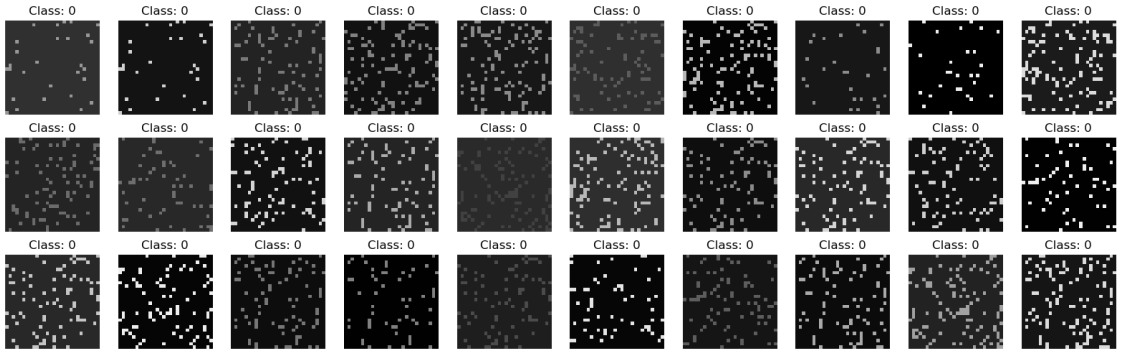
[33]:
```
set_seed(seed)
visual_domain_config = None
use_permutation = True

permutater = np.arange(28 * 28,  dtype=np.int32)
np.random.shuffle(permutater)
unpermutater = np.argsort(permutater)

visual_dataset = None

transforms = T.Compose([T.ToTensor()])
################################################################################
# TODO: Implement visual_dataset for this new domain                           #
# Hint: If you read docstring of EdgeDetectionDataset, you will find           #
# 'use_permutation' args. Pass True to this args.                              #
# Also pass permutator to EdgeDetectionDataset                                 #
################################################################################

visual_domain_config = dict(
    data_per_class=50,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    use_permutation = use_permutation,
    permutater = permutater,
    unpermutater = unpermutater
    )

visual_dataset = EdgeDetectionDataset(visual_domain_config, mode='train',␣
 ↪transform=transforms)

################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

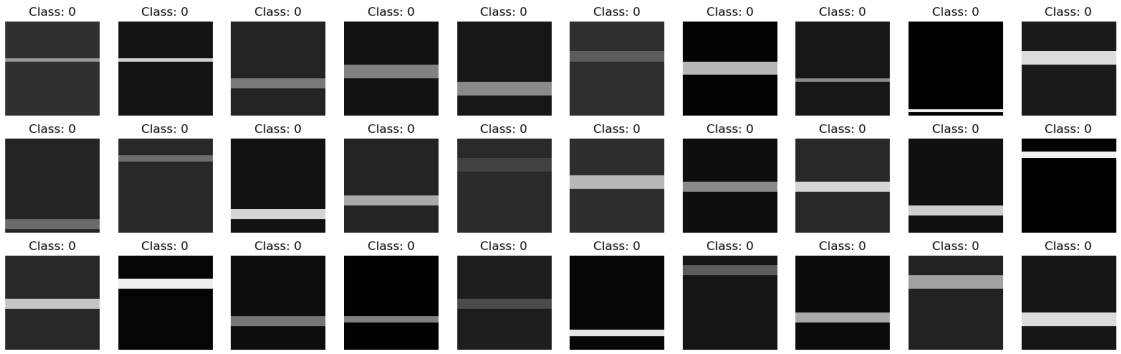
[36]:
```
## Visualize the images
unpermutator = visual_dataset.get_unpermutater()
print('Dataset Image before permutation')
vis_unpermuted_dataset(visual_dataset, num_classes=3, num_show_per_class=10,␣
 ↪unpermutator=unpermutator)
```

```
print('Dataset Image after permutation')
vis_dataset(visual_dataset, num_classes=3, num_show_per_class=10)
```

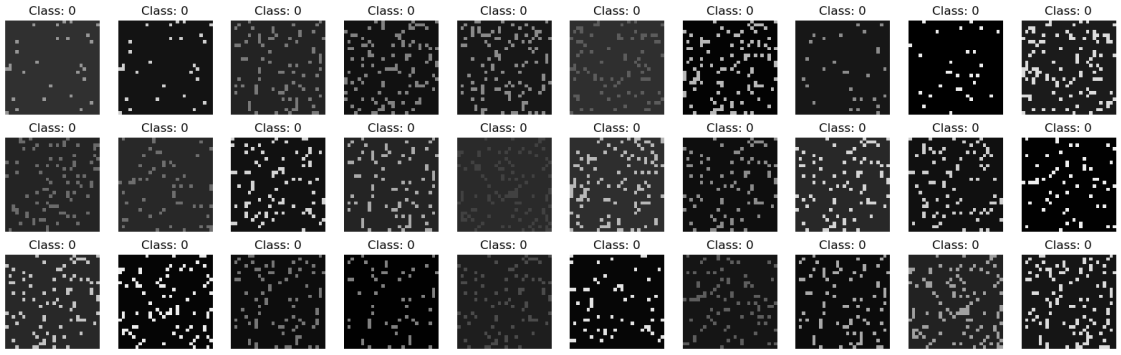
Dataset Image before permutation







Dataset Image after permutation



Now let's train CNN and MLP on the permuted dataset.

[37]:
```
set_seed(seed)

train_loader_dict = dict()
num_train_images_list = [30, 40, 50, 60, 70]
use_permutation = True
```

```
valid_loader = None

permutater = np.arange(28 * 28,  dtype=np.int32)
np.random.shuffle(permutater)
unpermutater = np.argsort(permutater)

transforms = T.Compose([T.ToTensor()])

batch_size = 10
################################################################################
# TODO: Implement train_loader_dict for each number of training images.      #
# Key: The number of training images (30, 40, 50, 60 and 70)                 #
# Value: The corresponding dataloader                                        #
# The validation set size is 50 images per class                             #
# 'use_permutation' args. Pass True to this args.                            #
# Also pass permutator/unpermutator to EdgeDetectionDataset                  #
################################################################################
for num_images in num_train_images_list:
    train_config = dict(
    data_per_class=num_images,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    use_permutation = use_permutation,
    permutater = permutater,
    unpermutater = unpermutater
    )

    train_dataset = EdgeDetectionDataset(train_config, mode='train',␣
 ↪transform=transforms)

    train_loader_dict[num_images] = torch.utils.data.
 ↪DataLoader(train_dataset,batch_size)

valid_config = dict(
    data_per_class=50,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    )

valid_dataset = EdgeDetectionDataset(valid_config, mode='train',␣
 ↪transform=transforms)

valid_loader = torch.utils.data.DataLoader(valid_dataset,valid_batch_size)

################################################################################
#                                END OF YOUR CODE                            #
################################################################################
```

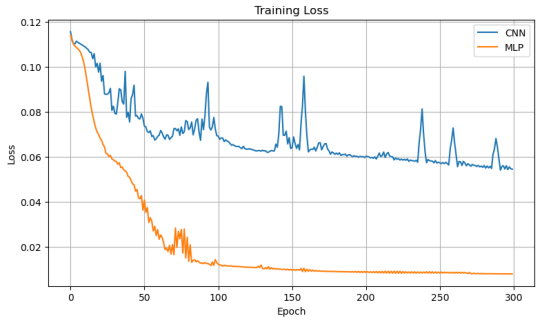
Note that kernel size is 3 in this experiment.

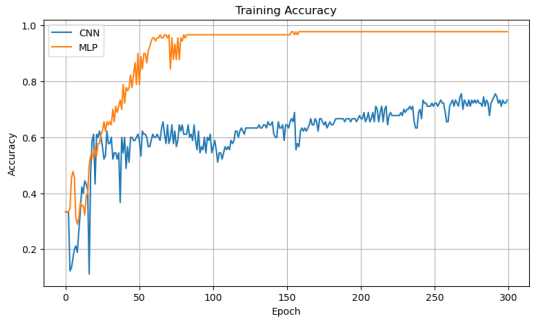
[40]:
```
lr = 1e-2
num_epochs = 300
kernel_size = 3
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()
mlp_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=kernel_size)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9,␣
↪weight_decay=5e-4)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9,␣
↪weight_decay=5e-4)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
↪cnn_valid_loss_list = [], [], [], []
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list,␣
↪mlp_valid_loss_list = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model,␣
↪mlp_optimizer, criterion, train_loader, device, epoch, verbose=False)

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix =␣
↪evaluate(cnn_model, criterion, valid_loader, device, verbose=False)
        mlp_valid_loss, mlp_valid_acc, mlp_confusion_matrix =␣
↪evaluate(mlp_model, criterion, valid_loader, device, verbose=False)
```

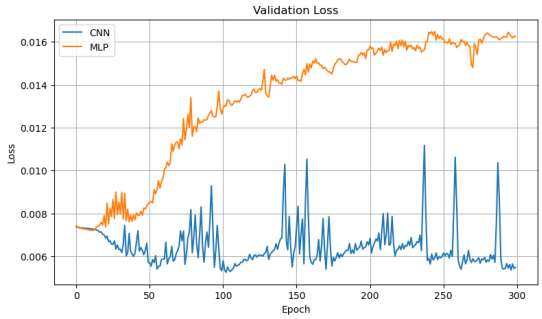
```
        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list,␣
↪mlp_train_loss_list, mlp_train_acc_list)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list,␣
↪mlp_valid_loss_list, mlp_valid_acc_list)

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.
↪detach().cpu()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix
    mlp_confusion_matrix_dict[num_image] = mlp_confusion_matrix

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
    mlp_acc_list.append(mlp_acc)
```

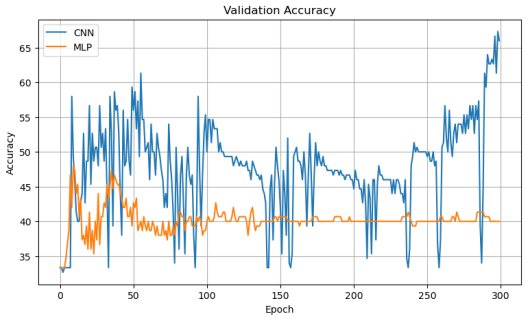
```
Training with 30 images
```

```
100%|                    | 300/300 [00:19<00:00, 15.76it/s]
```

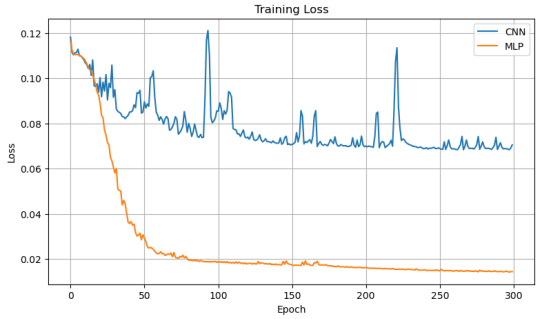


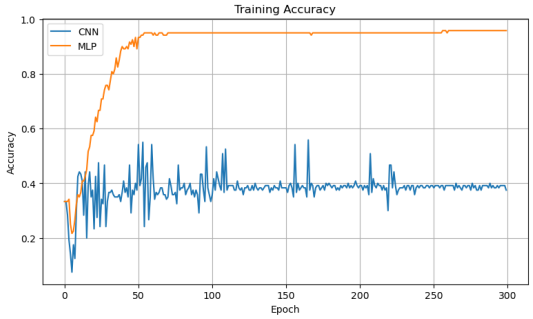


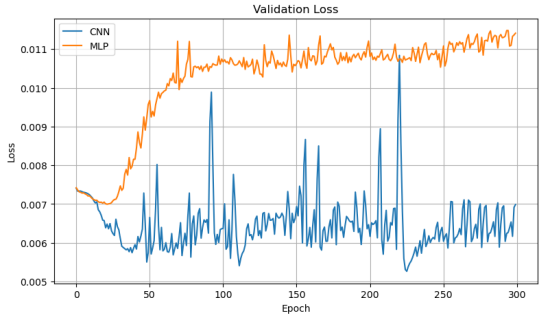


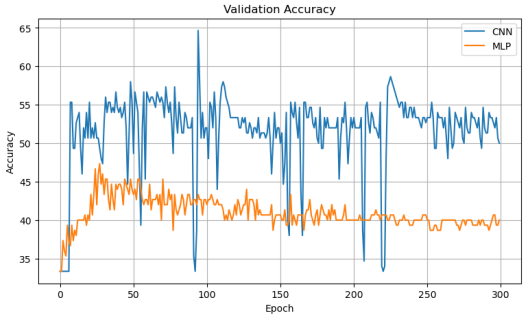



```
CNN Acc: 66.0, MLP Acc: 40.0
Training with 40 images
100%|                          | 300/300 [00:23<00:00, 12.94it/s]
```

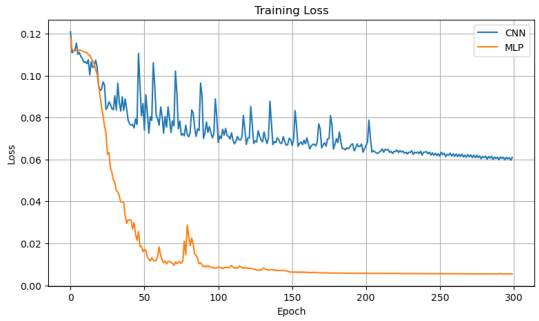


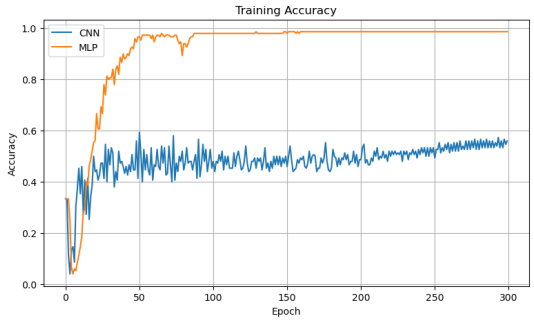


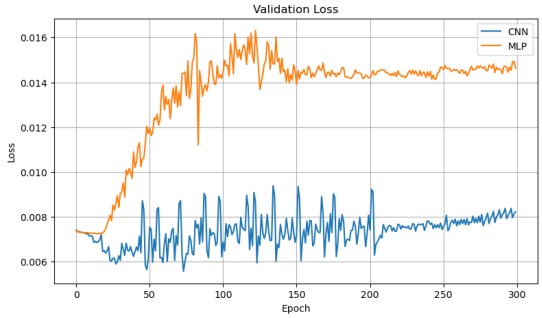


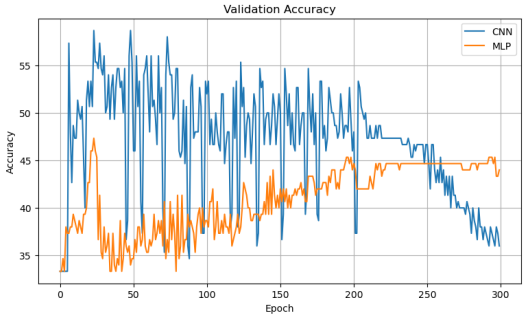



```
CNN Acc: 50.0, MLP Acc: 40.0
Training with 50 images
100%|                          | 300/300 [00:27<00:00, 10.88it/s]
```

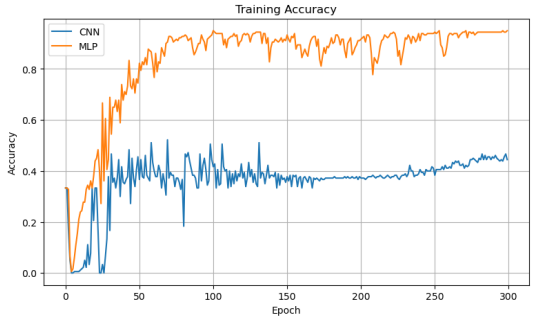


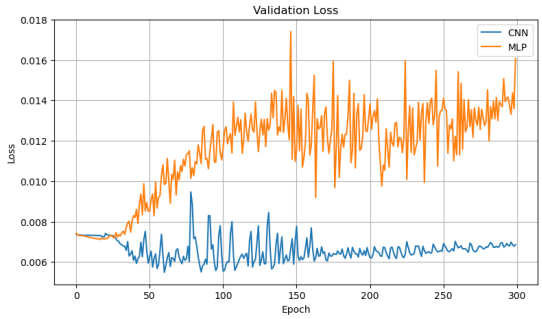


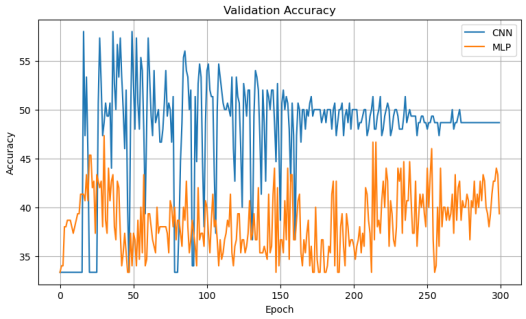




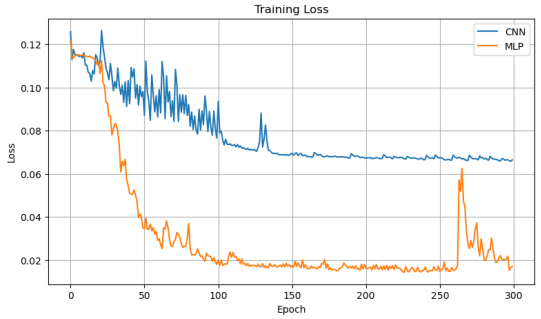

```
CNN Acc: 36.0, MLP Acc: 44.0
Training with 60 images
100%|                              | 300/300 [00:33<00:00,  8.83it/s]
```

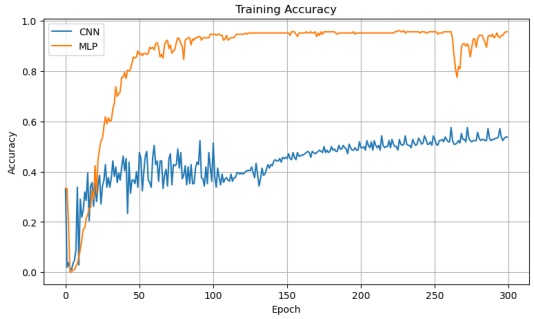


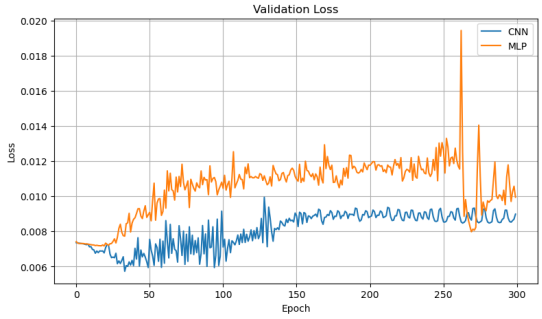


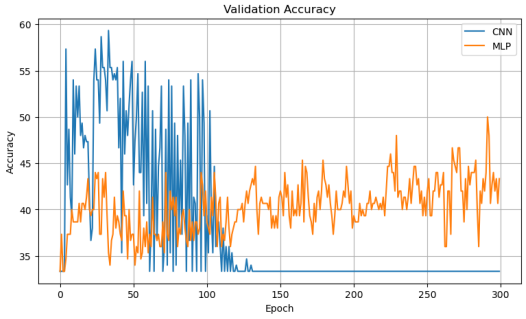



```
CNN Acc: 48.666666666666664, MLP Acc: 39.333333333333336
Training with 70 images
100%|                                | 300/300 [00:32<00:00,  9.12it/s]
```










```
CNN Acc: 33.333333333333336, MLP Acc: 43.333333333333336
```

Q. What do you observe? What is the reason that CNN is worse than MLP? (Hint: Think about the model architecture.)

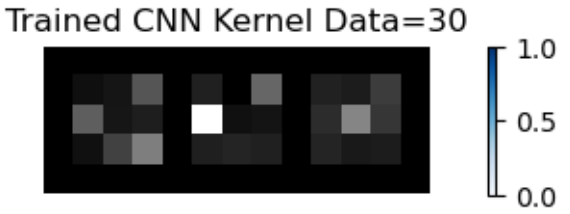
A. The translational weight-sharing property of the CNN is exactly what keeps it from learning the permuted data. The permuted data is outside the kernel space of the CNN.

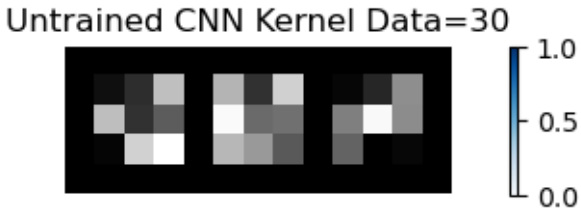
Q. Assuming we are increasing kernel size of CNN. Does the validation accuracy increase or decrease? Why?

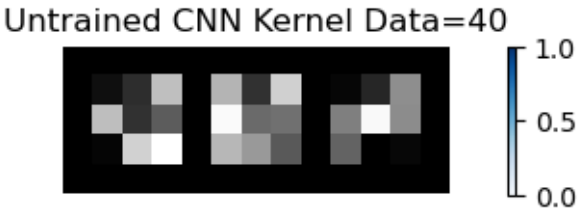
A. If we increase the kernel size it increases the validation accuracy becomes edges that had been permuted outside the original kernel are now inside the kernel and thus are learnable.

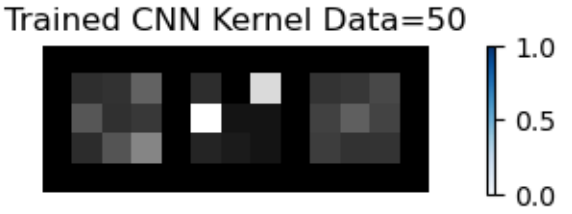
Now let's visualize CNN's learned kernel.

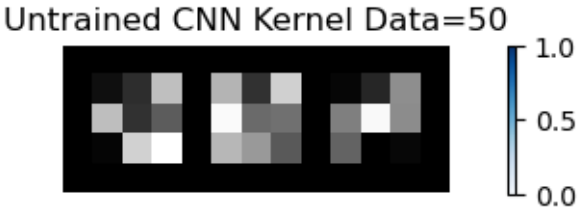
[41]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel␣
 ↪Data={}'.format(num_image))
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN␣
 ↪Kernel Data={}'.format(num_image))
```

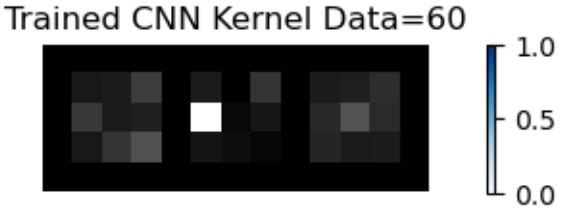


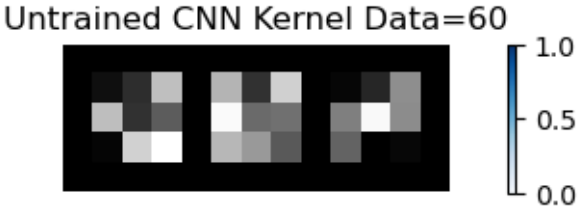


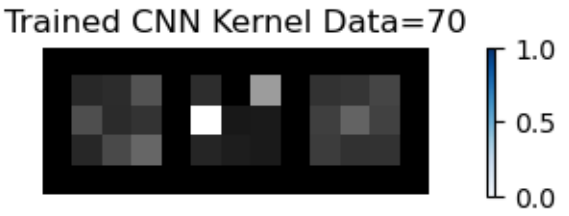

Untrained CNN Kernel Data=40
1.0
0.5
0.0

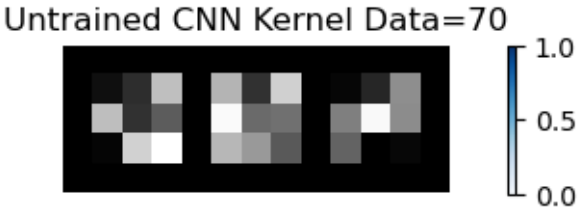

Trained CNN Kernel Data=50
1.0
0.5
0.0


Untrained CNN Kernel Data=50
1.0
0.5
0.0


Trained CNN Kernel Data=60
1.0
0.5
0.0


Untrained CNN Kernel Data=60
1.0
0.5
0.0





Q. How do the learned kernels look like? Explain why.

A. The kernels seemed to have learned diagnol lines. The permutations rotated the images so the kernels learned those lines.

From the above example, we can see that CNN is not always better than MLP. We have to think about the domain (or task) of the dataset and the model architecture to decide which model is better.

1.3.11 Q6. Increasing the Number of Classes

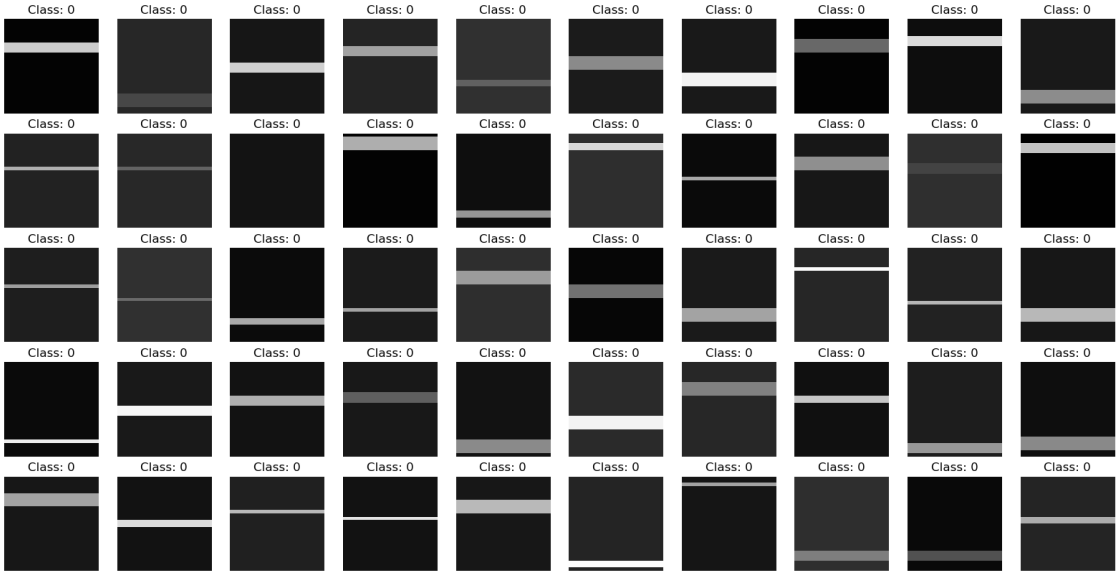
OK, can we conclude that CNN has the inductive bias that the model is translation invariant? Let's try other experiments. We make the task harder. In this problem, we increase the number of classes to 5. The new classes are 0 for horizontal edges, 1 for vertical edges, 2 for diagonal edges, 3 for vertical and horizontal, and 4 for nothing. Let's generate the dataset with 10 images per class and visualize the dataset.

[43]:
```
set_seed(seed)
visual_domain_config = None

visual_dataset = None

transforms = T.Compose([T.ToTensor()])
################################################################################
# TODO: Implement visual_dataset for this new domain                           #
# Hint: If you read docstring of EdgeDetectionDataset, you will find           #
# 'class_type' args. Pass ['horizontal', 'vertical', 'diagonal', 'both',       #
```

```
# 'none'] to 'class_type' args.                                                #
################################################################################
visual_domain_config = dict(
    data_per_class=50,
    num_classes=5,
    class_type=["horizontal", "vertical", "diagonal", "both", "none"],
    )

visual_dataset = EdgeDetectionDataset(visual_domain_config, mode='train',␣
 ↪transform=transforms)
################################################################################
#                                END OF YOUR CODE                              #
################################################################################
```

Let's visualize the dataset first.

[44]:
```
vis_dataset(visual_dataset, 5, 10)
```



Now let's make the new dataset. In this problem, we also see how the model performance changes as the number of images per class increases. Let's sweep the number of training images 10, 20, 30, 40, and 50. The validation set will be the same (50) for all the cases.

[45]:
```
set_seed(seed)

train_dataset_config = None
train_loader_dict = dict()
num_train_images_list = [10, 20, 30, 40, 50]
valid_loader = None
```

```
transforms = T.Compose([T.ToTensor()])
batch_size = 10
################################################################################
# TODO: Implement train_loader_dict for each number of training images.      #
# Key: The number of training images (10, 20, 30, 40 and 50)                 #
# Value: The corresponding dataloader                                        #
# The validation set size is 50 images per class                             #
# Hint: class_type = ['horizontal', 'vertical', 'diagonal', 'both', 'none'] #
# Hint: Be careful about the number of classes                               #
################################################################################


for num_images in num_train_images_list:
    train_dataset_config = dict(
    data_per_class=num_images,
    num_classes=5,
    class_type=["horizontal", "vertical", "diagonal", "both", "none"]
    )

    train_dataset = EdgeDetectionDataset(train_dataset_config, mode='train',␣
 ↪transform=transforms)

    train_loader_dict[num_images] = torch.utils.data.
 ↪DataLoader(train_dataset,batch_size)

valid_config = dict(
    data_per_class=50,
    num_classes=5,
    class_type=["horizontal", "vertical", "diagonal", "both", "none"]
    )

valid_dataset = EdgeDetectionDataset(valid_config, mode='train',␣
 ↪transform=transforms)

valid_loader = torch.utils.data.DataLoader(valid_dataset,valid_batch_size)

################################################################################
#                                END OF YOUR CODE                            #
################################################################################
```

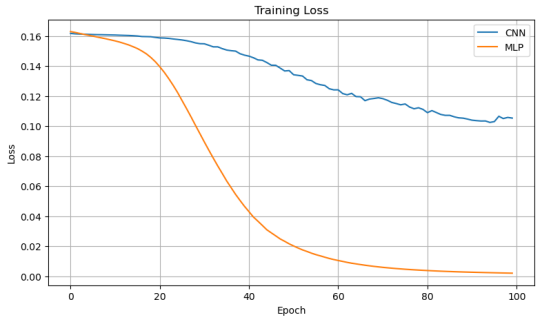
[46]:
```
lr = 1e-2
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
```

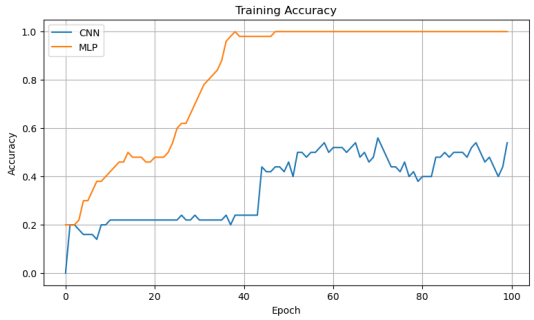
```
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()
mlp_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7, num_classes=5)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10], num_classes=5)
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
 ↪cnn_valid_loss_list = [], [], [], []
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list,␣
 ↪mlp_valid_loss_list = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
 ↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model,␣
 ↪mlp_optimizer, criterion, train_loader, device, epoch, verbose=False)

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix =␣
 ↪evaluate(cnn_model, criterion, valid_loader, device, verbose=False)
        mlp_valid_loss, mlp_valid_acc, mlp_confusion_matrix =␣
 ↪evaluate(mlp_model, criterion, valid_loader, device, verbose=False)

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)
```

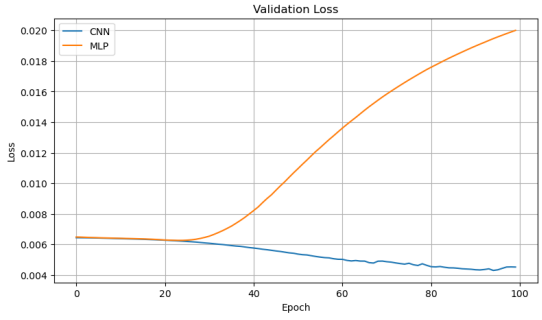
```
    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list,␣
↪mlp_train_loss_list, mlp_train_acc_list)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list,␣
↪mlp_valid_loss_list, mlp_valid_acc_list)

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.
↪detach().cpu()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix
    mlp_confusion_matrix_dict[num_image] = mlp_confusion_matrix

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
    mlp_acc_list.append(mlp_acc)
```

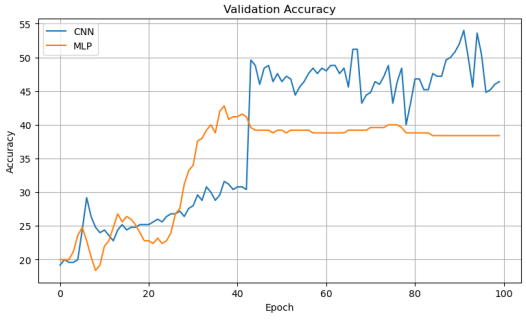
```
Training with 10 images
```

```
100%|                    | 100/100 [00:08<00:00, 12.45it/s]
```



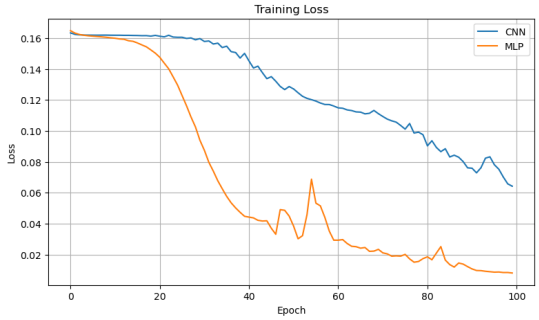


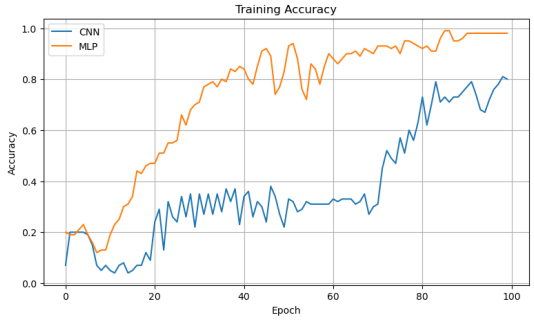


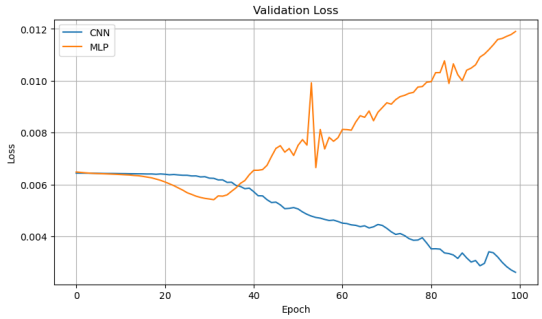


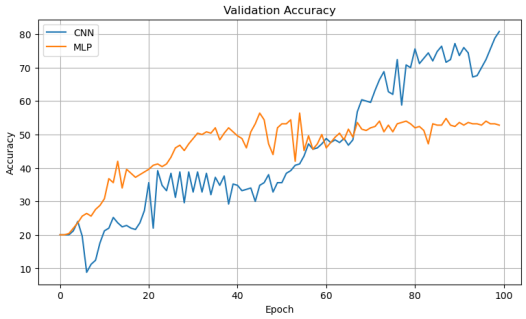

```
CNN Acc: 46.4, MLP Acc: 38.4
```

```
Training with 20 images
100%|                                   | 100/100 [00:09<00:00, 10.73it/s]
```

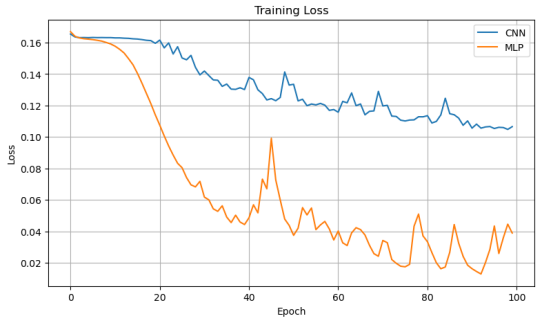


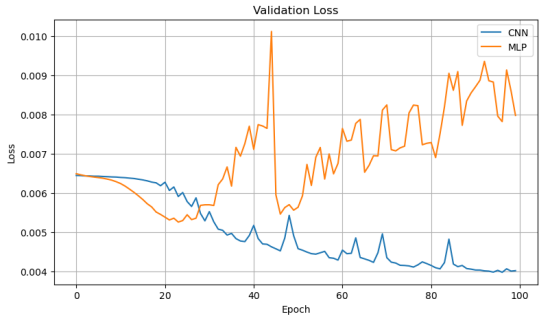


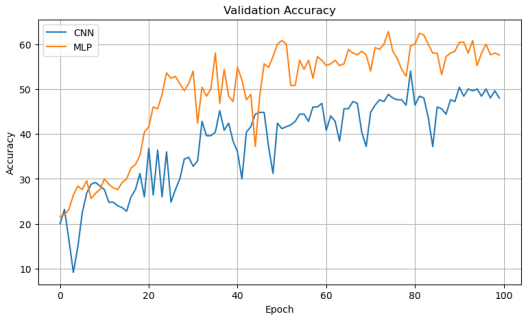




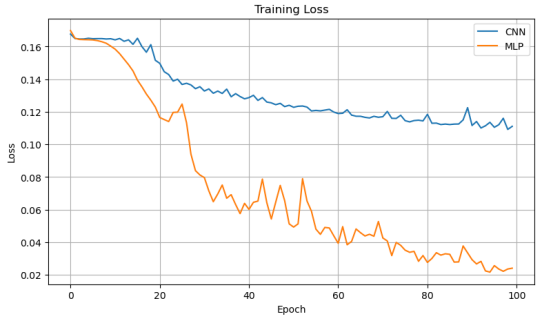

```
CNN Acc: 80.8, MLP Acc: 52.8
Training with 30 images
100%|                                   | 100/100 [00:11<00:00,  8.60it/s]
```

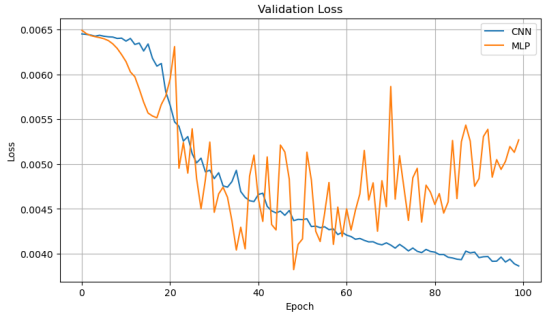


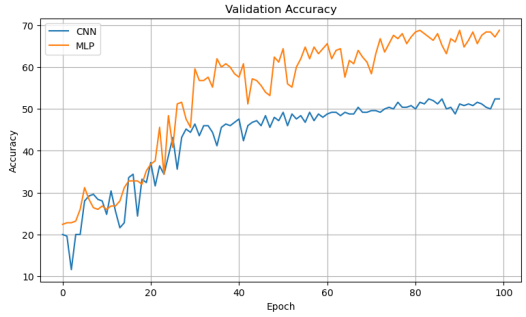




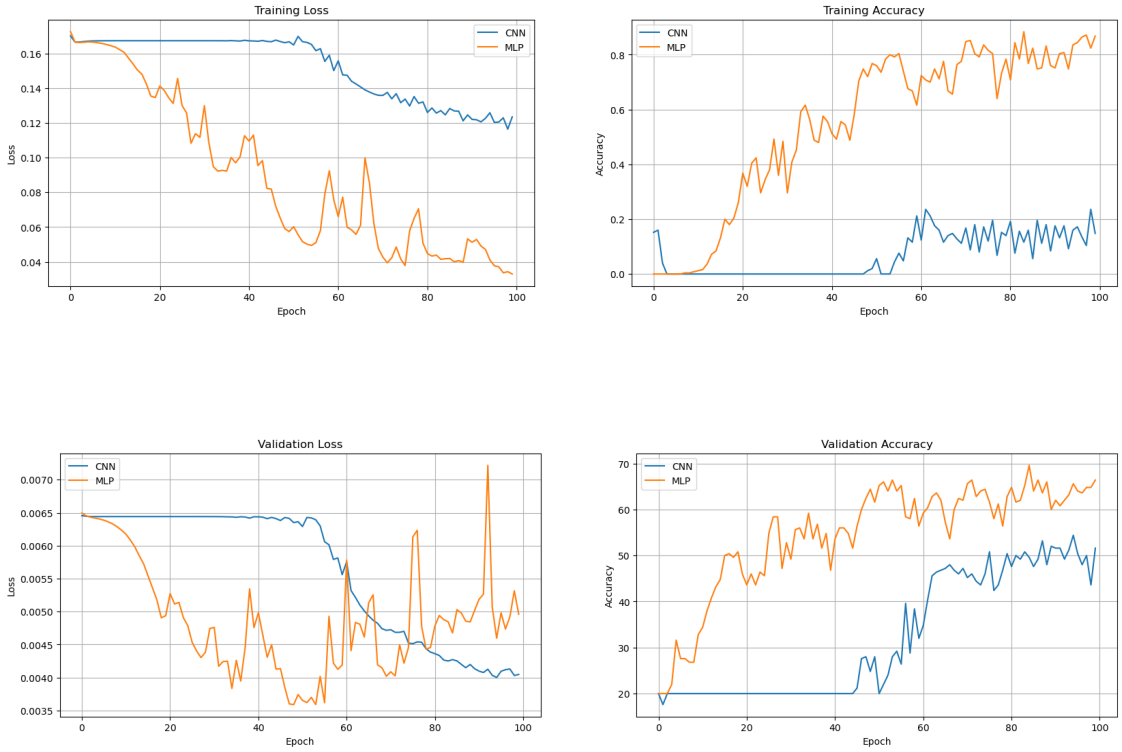

```
CNN Acc: 48.0, MLP Acc: 57.6
Training with 40 images
100%|                          | 100/100 [00:13<00:00,  7.27it/s]
```





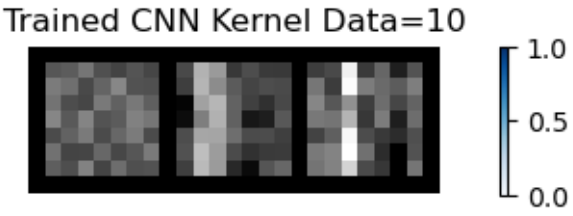


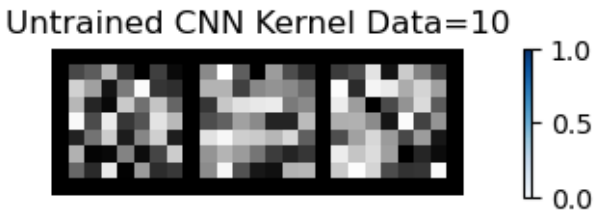

```
CNN Acc: 52.4, MLP Acc: 68.8
Training with 50 images
100%|                          | 100/100 [00:15<00:00,  6.33it/s]
```

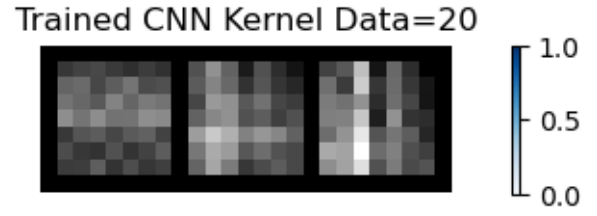


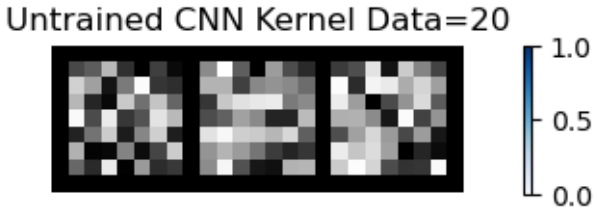

```
CNN Acc: 51.6, MLP Acc: 66.4
```

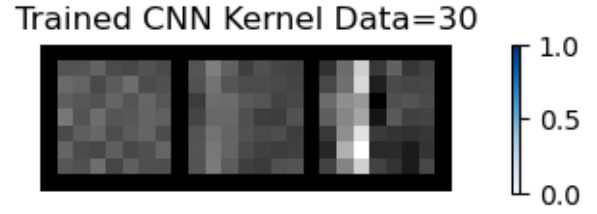
[47]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel␣
 ↪Data={}'.format(num_image))
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN␣
 ↪Kernel Data={}'.format(num_image))
```

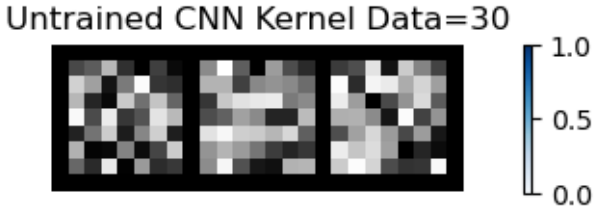


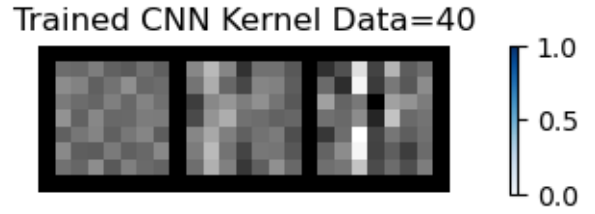

Untrained CNN Kernel Data=10
1.0
0.5
0.0

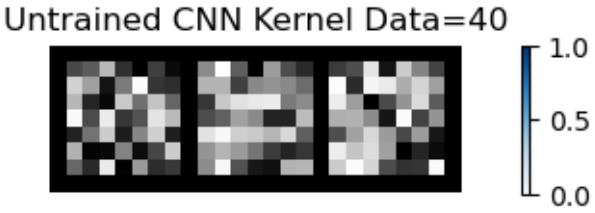

Trained CNN Kernel Data=20
1.0
0.5
0.0

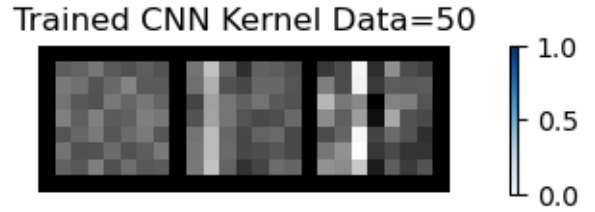

Untrained CNN Kernel Data=20
1.0
0.5
0.0

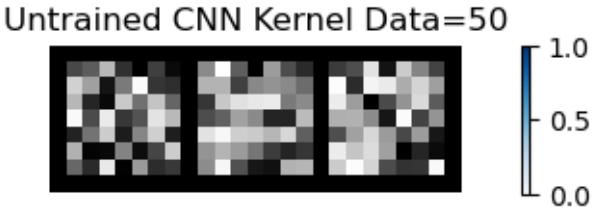

Trained CNN Kernel Data=30
1.0
0.5
0.0


Untrained CNN Kernel Data=30
1.0
0.5
0.0









(Optional) Ok, CNN performs pretty good. But what if we increase the width or the depth of CNN? The patterns that we have to detect are 5 but our kernels per layer are only 3. Intuitively, this is quite a suboptimal. Here, we will investigate the affect of increasing width and depth. Let's use the same dataset but we will use `DeeperCNN` and `WiderCNN` in `cnn.py`. `DeeperCNN` has 2 times more layers than `SimpleCNN` and `WiderCNN` has 2 times more kernels per layer than `SimpleCNN`. Let's train the models and visualize the validation accuracy.

[ ]:
```
################################################################################
# TODO: Training DeeperCNN and tuning hyperparameters Try other num_epochs,    #
# lr, kernel_size. The validation accuracy                                     #
################################################################################
lr = None
num_epochs = None
kernel_size = None
################################################################################
#                                END OF YOUR CODE                              #
################################################################################
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_valid_acc_list = list()

cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = DeeperCNN(kernel_size=kernel_size)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
 ↪cnn_valid_loss_list = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
 ↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix =␣
 ↪evaluate(cnn_model, criterion, valid_loader, device, verbose=False)

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, None, None)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, None, None)

    cnn_acc = cnn_valid_acc_list[-1]
```

```
    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.
 ↪detach().cpu()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix

    print("CNN Acc: {}".format(cnn_acc))
    cnn_acc_list.append(cnn_acc)
```

[ ]:
```
################################################################################
# TODO: Training WiderCNN and tuning hyperparameters Try other num_epochs,  #
# lr, kernel_size. The validation accuracy                                     #
################################################################################
lr = None
num_epochs = None
kernel_size = None
################################################################################
#                               END OF YOUR CODE                               #
################################################################################
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_valid_acc_list = list()

cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = WiderCNN(kernel_size=kernel_size)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list,␣
 ↪cnn_valid_loss_list = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model,␣
 ↪cnn_optimizer, criterion, train_loader, device, epoch, verbose=False)

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix =␣
 ↪evaluate(cnn_model, criterion, valid_loader, device, verbose=False)
```

```
        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, None, None)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, None, None)

    cnn_acc = cnn_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.
↪detach().cpu()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix

    print("CNN Acc: {}".format(cnn_acc))
    cnn_acc_list.append(cnn_acc)
```