In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [24]:
```
import matplotlib.pyplot as plt
import numpy as np
import torch
from torchvision.transforms.functional import to_tensor, normalize, affine

from solution import (
    NetworkConfiguration,
    Trainer,
)
```

Q2

Q2 Part 1

Generate a figure with the test Mean Absolute Error w.r.t. increasing epochs for values of learning rates in the set {0.01, 1 × 10−4, 1 × 10−8}. What are the effects of learning rates that are very small or very large?

In [3]:
```
mlp_net_config = NetworkConfiguration(
    dense_hiddens=(128, 128),
)

# Note: Only the dense_hiddens parameter is used for MLP, the rest will be ignored
print(mlp_net_config)
```

```
NetworkConfiguration(n_channels=(16, 32, 48), kernel_sizes=(3, 3, 3), strides=(1, 1, 1),
dense_hiddens=(128, 128))
```

In [15]:
```
lr_list = [0.01, 1e-4, 1e-8]
mlp_train_logs_dict = {}
```

In [16]:
```
for lr in lr_list:
    mlp_trainer = Trainer(network_type="mlp",
                          net_config=mlp_net_config,
                          lr=lr,
                          batch_size=128,
                          activation_name="relu",
                          )

    mlp_train_logs = mlp_trainer.train_loop(n_epochs=50)
    mlp_train_logs_dict[lr] = mlp_train_logs
```

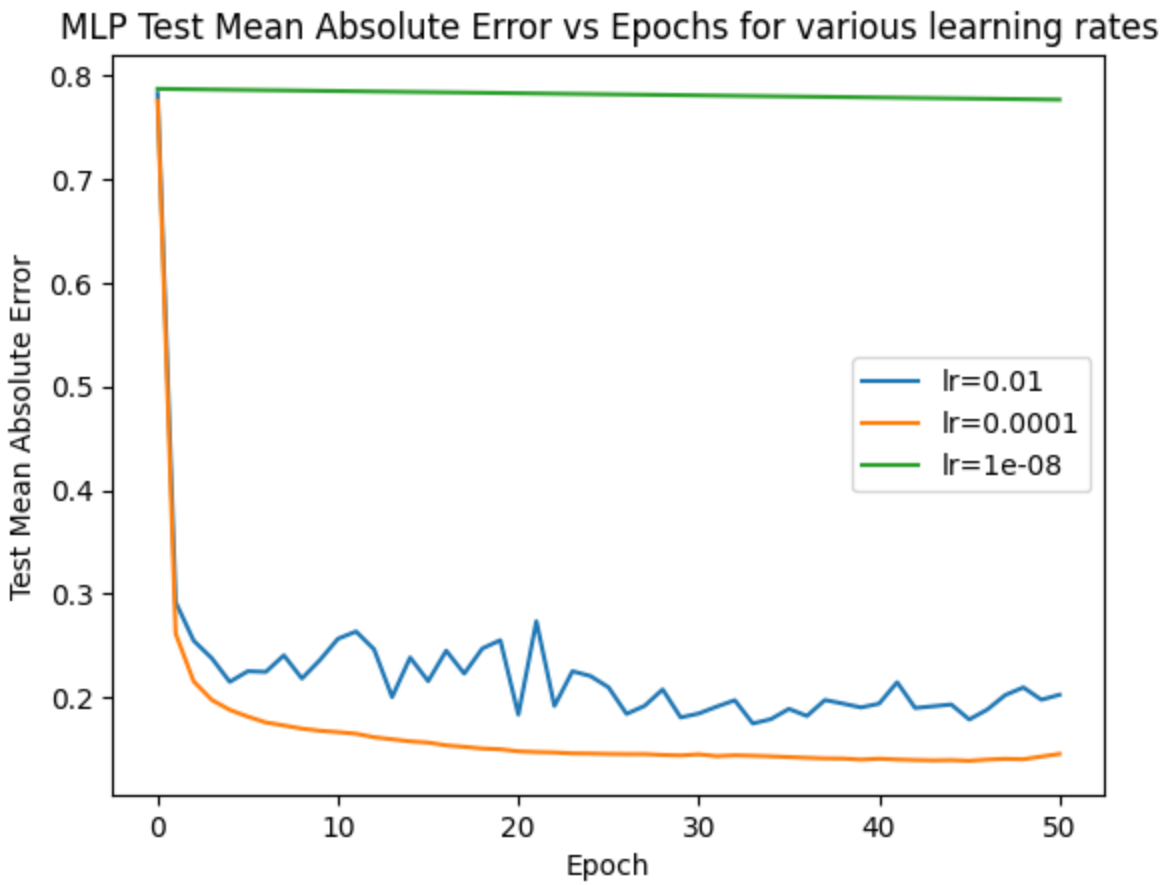
```
100%|██████████| 50/50 [00:52<00:00,  1.05s/it]
100%|██████████| 50/50 [00:40<00:00,  1.22it/s]
100%|██████████| 50/50 [00:41<00:00,  1.20it/s]
```

In [36]:
```
mlp_fig, mlp_ax = plt.subplots()

for lr, mlp_train_logs in mlp_train_logs_dict.items():
    test_mae_list = mlp_train_logs['test_mae']
    mlp_ax.plot(test_mae_list, label=f"lr={lr}")
```

```
mlp_ax.set_title("MLP Test Mean Absolute Error vs Epochs for various learning rates")
mlp_ax.set_ylabel("Test Mean Absolute Error")
mlp_ax.set_xlabel("Epoch")
mlp_ax.legend()
```

Out[36]: <matplotlib.legend.Legend at 0x7fa23b9812e0>



Discussion:

As can be seen from the figure above, when learning rate is very large (lr=0.01), the test mean absolute error can fluctuate wildly, since each training epoch's weight update is larger in magnitude and correspondingly has a larger effect on test mean absolute error. The large learning rate may also prevent the model from reaching a lower minima on the loss landscape, due to the larger weight updates (ie. "bouncing" around the local minima), which can be be seen from the fact that the model was only able to reach a test mean absolute error of around 0.2, whereas when the learning rate was smaller (lr=0.0001), the model had a test mean absolute error of around 0.1.

Conversely, when learning rate is very small (lr=1e-8), each training epoch's weight update is so small that learning happens very slowly, which can be seen in a very slow improvement (decrease) in test mean absolute error. Even at the epoch 50, the test mean absolute error was still near 0.8, which is much higher than when lr=0.01 and lr=0.001.

Q2 Part 2

Generate a figure with the test Mean Absolute Error w.r.t. increasing epochs for the above CNN

In [29]:
```
cnn_net_config = NetworkConfiguration(
    n_channels=(16, 32, 45),
    kernel_sizes=(3, 3, 3),
```

```
    strides=(1, 1, 1),
    dense_hiddens=(128,),
)

print(cnn_net_config)
```

```
NetworkConfiguration(n_channels=(16, 32, 45), kernel_sizes=(3, 3, 3), strides=(1, 1, 1),
dense_hiddens=(128,))
```

In [30]:
```
cnn_trainer = Trainer(network_type="cnn",
                      net_config=cnn_net_config,
                      batch_size=128,
                      activation_name="relu",
                      )
```

In [31]:
```
cnn_train_logs = cnn_trainer.train_loop(n_epochs=50)
```

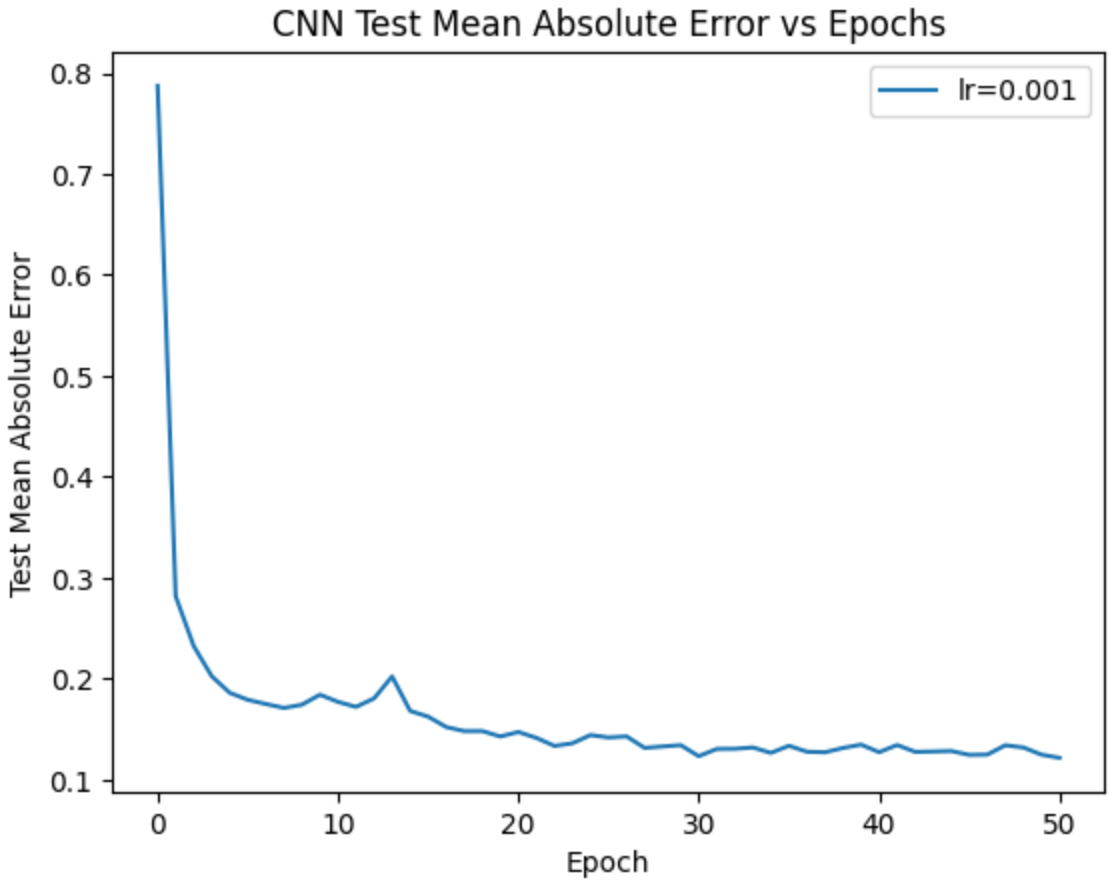
```
100%|██████████| 50/50 [11:49<00:00, 14.20s/it]
```

In [39]:
```
cnn_fig, cnn_ax = plt.subplots()

cnn_test_mae_list = cnn_train_logs['test_mae']
cnn_ax.plot(cnn_test_mae_list, label=f"lr=0.001")

cnn_ax.set_title("CNN Test Mean Absolute Error vs Epochs")
cnn_ax.set_ylabel("Test Mean Absolute Error")
cnn_ax.set_xlabel("Epoch")
cnn_ax.legend()
```

Out[39]:
```
<matplotlib.legend.Legend at 0x7fa22aeb2ca0>
```



Discussion:

As can be seen from the figure above, as epoch increases, the CNN test mean absolute error decreases sharply from 0.8 to around 0.15 and stabilises. This shows that the model has been fitting well on the data, and overfitting has not happened yet. The capacity of the model could still be increased (e.g. more convolutional layers) which might possibly lead to an even lower test mean absolute error, or the model could simply be trained for many more epochs to see if the test mean absolute error continues decreasing.

Q3

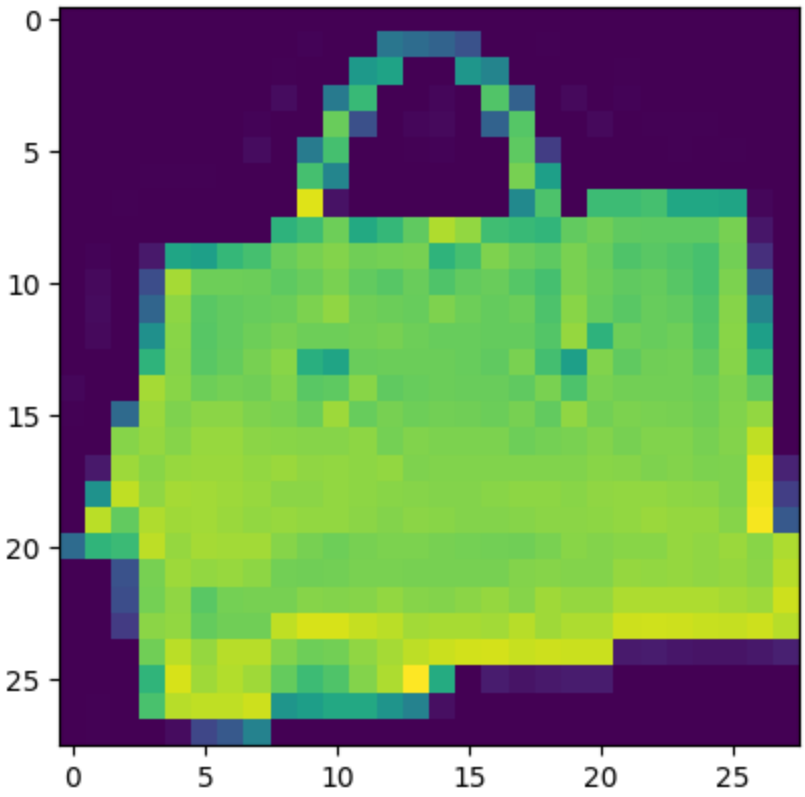
In [392…
```
equivariance_trainer = Trainer()
```

In [393…
```
img_np, model_output_np, shifted_difference_np, rotated_difference_np = equivariance_tra
```

A figure showing the original image, which is the first image self.train[0][0] of the training set;

In [394…
```
plt.imshow(img_np[0])
```

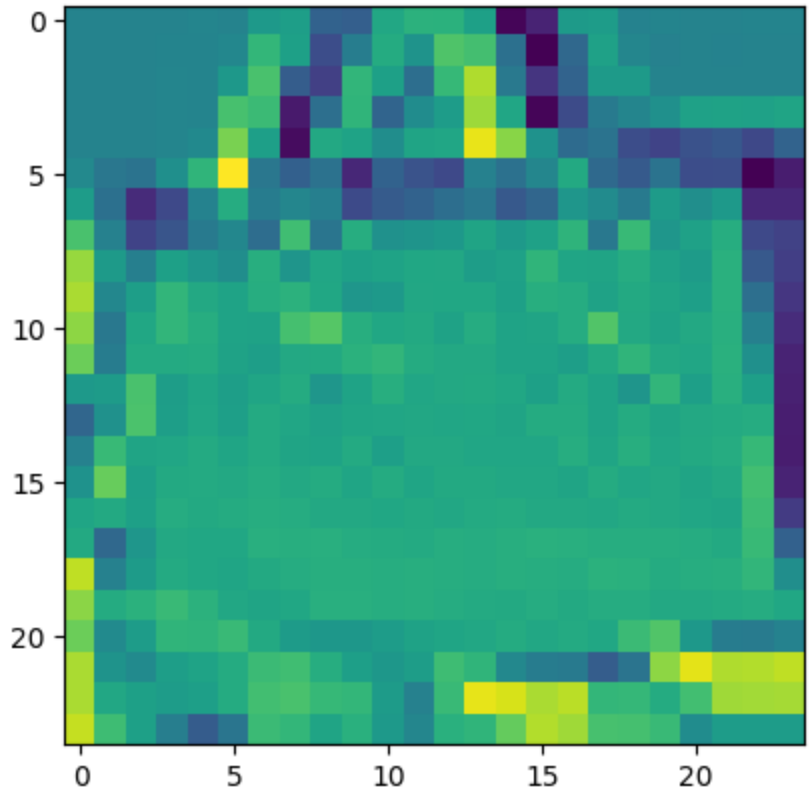
Out[394]:
```
<matplotlib.image.AxesImage at 0x7ff4ad60b9a0>
```



A figure showing the output features of the small CNN model when given that first image as an input;

In [389…
```
plt.imshow(model_output_np[0])
```

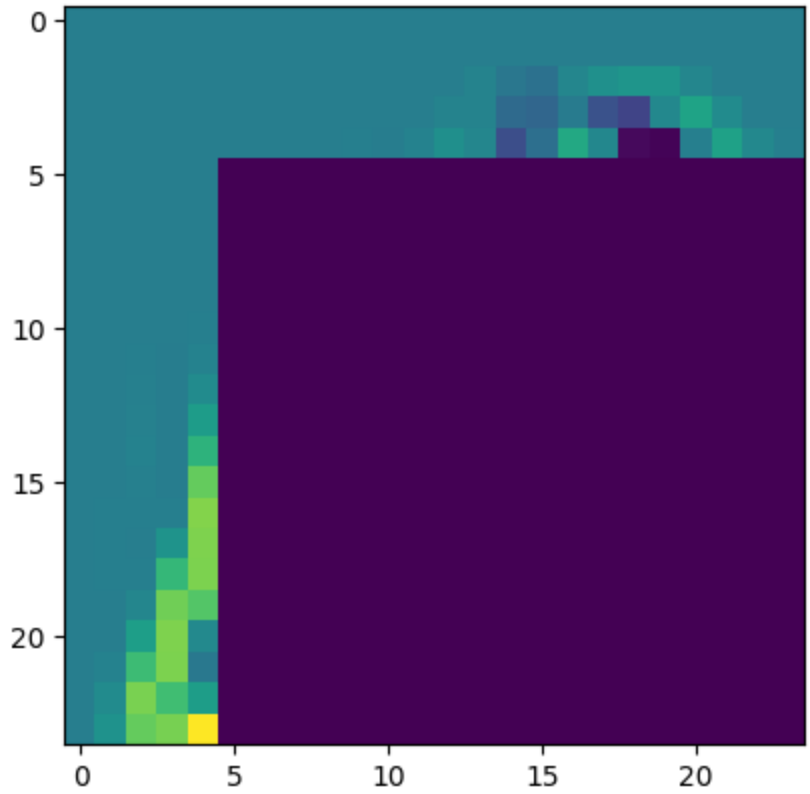
Out[389]:
```
<matplotlib.image.AxesImage at 0x7ff4ad0c5130>
```



A figure showing the image of the pixel-by-pixel absolute difference between (1) the shifted output features given the unshifted input, and (2) the unshifted output of the CNN given the shifted input. Shifts should be performed by using the provided shift transformation;

In [390…
```
plt.imshow(shifted_difference_np[0])
```

Out[390]:
```
<matplotlib.image.AxesImage at 0x7ff4ad124fd0>
```

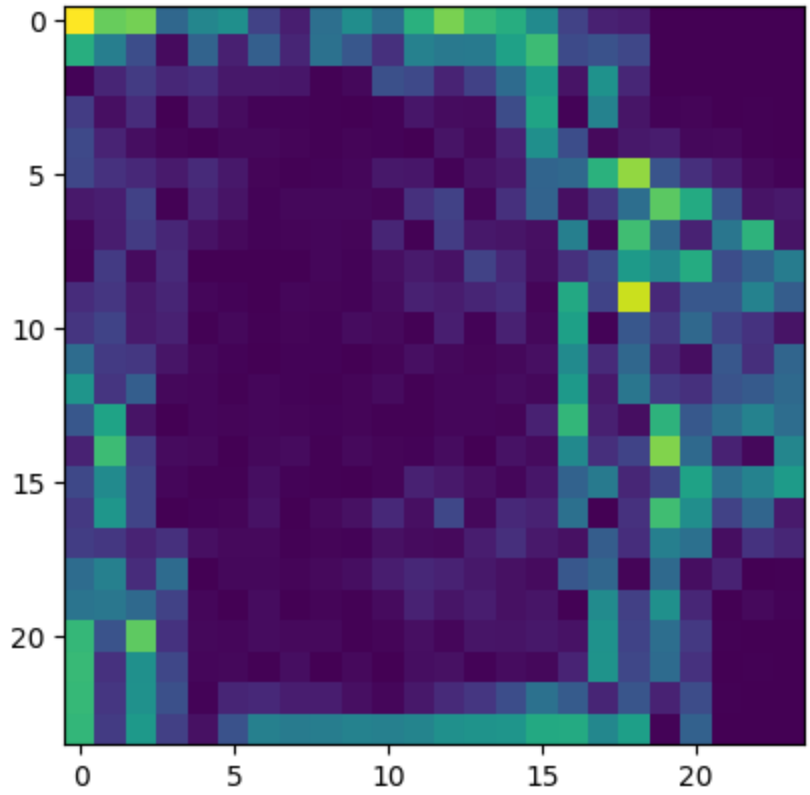


A similar figure, but this time with rotation. In other words, show the pixel-by-pixel absolute difference between the rotated output of the CNN given the original image and

the unprocessed output of the CNN given the rotated image. Use the provided rotation trans- formation, without any further processing.

In [391…
```
plt.imshow(rotated_difference_np[0])
```

Out[391]:
```
<matplotlib.image.AxesImage at 0x7ff4ad3ed8b0>
```



Discussion:

3.1. To which types of transformations are convolutional layers equivariant?

Answer:

Convolutional layers are shift equivariant, as can be seen from the third figure that shows the pixel-by-pixel absolute difference between the shifted output features given the unshifted input, and the unshifted output of the CNN given the shifted input. A large region of the plot extending from the bottom right corner has pixel values that are close to zero, which shows that the outcome of shifting the output features of the unshifted input is largely similar as the outcome of shifting the input before passing through the convolutional layer (in both cases, the convolutional layer detected the object in the image and output similar activations).

However, convolutional layers are not rotation equivariant, at least not to such large degrees of rotation (90 degrees). This is seen in the fourth figure, where there are many bright pixels indicating that there's significant difference between rotating the input before applying the convolution, versus applying the convolution on the unrotated input and then rotating the output.

3.2. Why should equivariance to some transformations be more useful than equivariance to other transformations for learning? Does it depend only on the training algorithm or on the data distribution and task as well?

Answer:

Equivariance to some transformations can be more useful than others for learning because some transformations are much more common in real-world data, such as shifting, scaling (zoom) and contrast / light intensity. Having equivariance to these transformations would mean that less data is needed for a model to learn to identify examples with these variations effectively.

The equivariance to transformations depends on the training algorithm - the type of convolution layer and associated kernels will affect how equivariant it is to transformations. In addition, equivariance on transformations depends on the data distribution and the task. For example, if the data has all the objects centered in every image, then having shift equivariance is redundant. Conversely, if the training data is augmented with all kinds of examples with the variations of shift, scale, contrast and rotation, then the algorithm itself does not need to be innately equivariant, but if it has fitted the data well it would have effectively learnt to be equivariant.

In [ ]: