SYDE 552 Assignment 2: Vision

Due Monday, February 26, 11:59pm

Value: 15% of total marks for the course

This assignment covers the mammilian vision system, including both questions about the biology itself and constructing computational models based on Regression and Convolutional Neural Networks.

You can work in groups to do the assignment, but your answers and code must be original to you. Your submission will be a filled-out copy of this notebook (cells for code and written answers provided).

1. The Vision System

The purpose of this part of the assignment is to test your knowledge of the brain's visual system and the relationship between neurobiological features and computational properties. The best answers will discuss both function and anatomy, and will draw on specific anatomical examples to support theoretical claims. You are encouraged to discuss answers with your classmates, consult the slides notes, or use external resources -- but your answers must be your own! In particular, read the Kandel et al. chapters listed on the slides. Expect to write around 5 sentences for each 1 point.

1.a) [2 marks] The neurons in different parts of the brain are sensitive to different things, and can be thought of as different feature detectors. For each of the types of neurons listed below, describe what feature they detect, their receptive fields, and how their connectivity to other neurons and/or their internal neural processes helps them to do this feature detection:

- Cones
- Sustained Ganglion Cells
- Transient Ganglion Cells
- Simple Cells

- Cones

Cones are most sensitive to a specific range of frequencies of light and sample the visual information coming into the retina. There are 3 main types, each with different ranges of peak sensitivity. Cones have the smallest receptive field and are densely packed in the fovea, contributing primarily to acquity of daytime vision. These cells project to horizontal and bipolar cells in the plexiform layer of the retina [1].

Phototransduction converts the incoming light signal into a chemical one that modulates neural activity. Retinal molecules (rhodopsin) are converted to an active state in the presence of light, which turns opsin into metarhodopsin II. The resulting activation of transducin molecules causes cGMP-gated channels to close, resulting in hyperpolarization of the cell and corresponding decrease in glutamate release [1]. Lower concentrations of glutamate allow cation channels in 'on' bipolar cells to remain open, thus resulting in depolarization and triggering a neural signal.

- Sustained Ganglion Cells

Ganglion cells form the most anterior layer of the retina, each receiving inputs from multiple bipolar cells. The receptive fields of ganglion cells are organized in an 'on' region surrounded by an 'off' region (or vice versa). They fire most rapidly in response to a stimulus that is concentrated in the 'on' region, but absent from the 'off' region.

Sustained ganglion cells in particular respond to the presence or absence of a stimulus. If a stimulus is presented in a sustained cell's receptive field, it will fire as long as the stimulus exists (up to several seconds) [1]. These cells are good at detecting edges of objects (i.e., differences in illumination) due to the center-surround organization of the receptive field—areas of higher contrast evoke a stronger response than areas that are evenly lit [1]. Having both 'on' and 'off' types allow ganglion cells to detect both rapid increases and decreases in light intensity.

- Transient Ganglion Cells

Transient ganglion cells respond most prominently to *changes* in input. Unlike sustained cells, transient ganglion cells will produce a burst of spikes when a stimulus is first presented, but then stop firing until another change is detected [1]. Therefore, these cells are sensitive to temporal changes in input.

- Simple Cells

Simple cells, located in V1, receive input from the optic radiation from the LGN [2]. These cells receive inputs from groups of ganglion cells that are organized in a particular spatial arrangement. Like ganglion cells in the retina, simple cells also have receptive fields that are organized into distinct 'on' and 'off' regions. However, since these cells receive inputs from multiple ganglion cells, their 'on' and 'off' regions are arranged into rectangular regions (as opposed to centre-surround) resulting in receptive field that responds to a particular stimulus orientation (for example, a bar of light oriented vertically) [2].

This orientation specificity is partly due to the hierarchical organization of cells in visual processing and the retinotopic mapping that is preserved between areas of the visual pathway, allowing cells that are physically close to one another in the retina to be used in detecting higher-level features. These cells are also highly selective for a particular location within the visual field, so they detect both the orientation and spatial position of objects [2].

References

[1] E. R. Kandel, J. H. Schwartz, T. M. Jessell, S. A. Siegelbaum, and A. J. Hudspeth, "Low-Level Visual Processing: The Retina," in *Principles of Neural Science, 5th Ed.* New York: McGraw Hill, 2013, ch. 26, pp. 577-601.

[2] E. R. Kandel, J. H. Schwartz, T. M. Jessell, S. A. Siegelbaum, and A. J. Hudspeth, "Intermediate-Level Visual Processing and Visual Primitives," in *Principles of Neural Science, 5th Ed.* New York: McGraw Hill, 2013, ch. 27, pp. 602-620.

1. b) [1 marks] Describe two instances where retinotopic organization facilitates visual processing. For each example, be sure to mention its anatomical location and discuss how retinitopy contributes to the feature detection.

Retinotopic organization is present in the retina, allowing photoreceptors that are spatially close to one another to detect incoming information from similar areas of the visual field. This is an important characteristic of the retina allowing coherent images to be interpreted, since the mapping of images on the retina corresponds to mapping of real objects in the physical environment.

Retinotopic organization also exists in V1, and allows the communication of visual information to V1 from the retina. A visuotopic map in V1 preserves the spatial organizaiton of photoreceptive cells in the retina, allowing efficient conduction of visual information. This is an example of serial processing between sequential areas in the visual pathway.

The specialized nature of visual processing requires cells that respond to different features, achieved through columnar organization of cells in the visual system – orientation columns, occular dominance columns, and blobs (colour sensitivity) are repeated at different frequencies such that a 1 mm x 1 mm slice of cortex can function as a processing unit [3]. This allows for parallel processing of information from different receptive fields to occur.

References

[3] E. R. Kandel, J. H. Schwartz, T. M. Jessell, S. A. Siegelbaum, and A. J. Hudspeth, "The Constructive Nature of Visual Processing," in *Principles of Neural Science, 5th Ed.* New York: McGraw Hill, 2013, ch. 25, pp. 556-576.

1. c) [2 marks] Discuss the similarities and differences between convolutional neural networks and the visual system.

- Similarities

- Similarities

Similarities include that neurons closer to the input (low-level) have small receptive fields (as in rods and cones in the retina) and those farther from the input layer have larger receptive fields [4]. There is a clear hierarchical structure in the organization of a CNN that mimics that of the primate visual system. There is also translational invariance that is present in both artifiical and biological systems, which means that features are detected the same way regardless of where they appear in the visual field [4]. Another similarity is the use of dropout in ANNs to mimic the vesicle release failure that can occur in biological networks.

- Differences

A major difference is that the visual system does not do convolution – biological neurons cannot share weights in the same way that a CNN would (we know this because of the connections that exist between neurons) [4]. This means that although convolution might result in similaritires in structure, it is not the mechanism underlying vision.

References

[4] T. Stewart and M. Furlong. SYDE 552. Class Lecture, Computational Neuroscience: "Lecture 10: Artificial Neural Networks." Systems Design Engineering, University of Waterloo, Waterloo, Canada, Feb. 7, 2024.

2. Classifying Stimuli Using Regression

The retina transforms the light entering an eye into a particular set of features, which are then sent to the rest of the brain for further processing. In this section we look at how neurons might detect patterns, and how that detection changes with different feature detectors.

The data we will use for this is the classic MNIST dataset

In [1]:

```
import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt
```

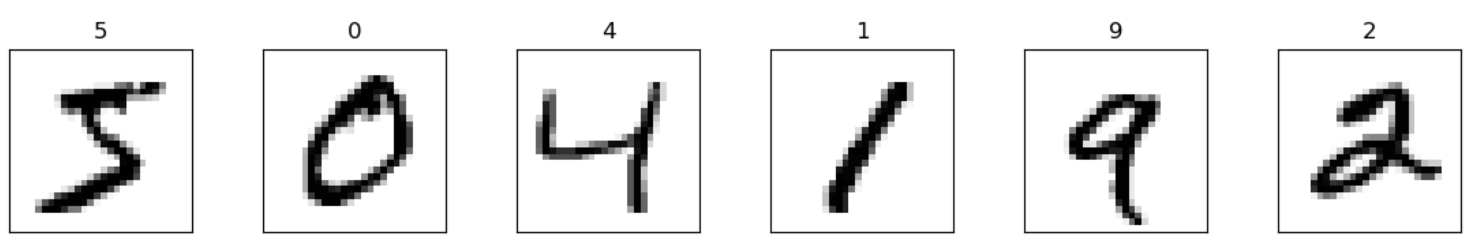
In [2]:

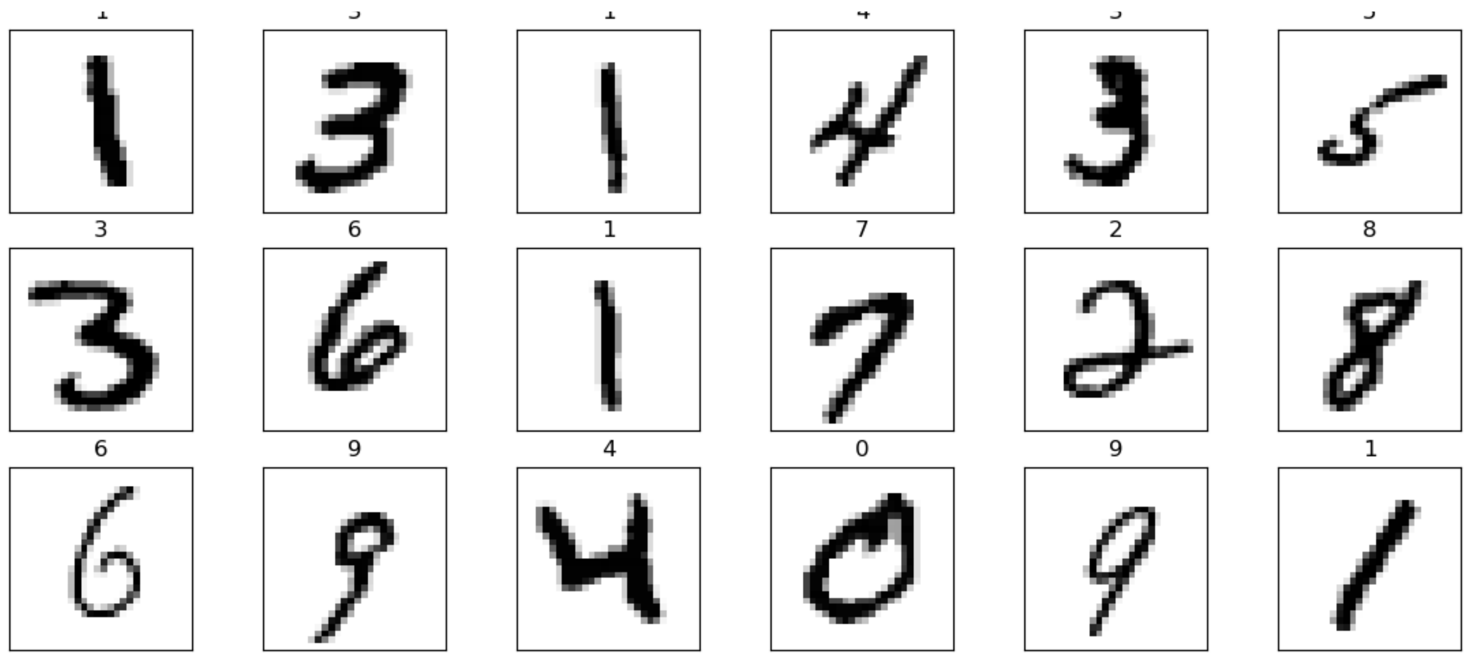
```
# load MNIST dataset
mnist = torchvision.datasets.MNIST(root='.', download=True)
```

The MNIST digits are 28x28 pixels each, each pixel is a value from 0 to 255, and there are 60,000 of them. The raw data is in `mnist.data` and the target value (i.e. the actual digit) is in `mnist.targets`. Here are the first 24 of each:

In [3]:

```
%matplotlib inline
plt.figure(figsize=(14,8))
for i in range(24):
    plt.subplot(4, 6, i+1)
    plt.imshow(mnist.data[i], vmin=0, vmax=255, cmap='gray_r')
    plt.xticks([])
    plt.yticks([])
    plt.title(int(mnist.targets[i]))
```





2.a) [1 mark] We can imagine the MNIST digits as 784 (28 times 28) input neurons. We want to connect these 784 neurons to 10 output neurons, one for each digit, and see how accurately we can classify the digits.

To find the weights in this question, we will use Ridge Regression. `X` is the MNIST input data, divided by 255 to rescale it to between 0 and 1, and then reshaped to be a 60000x784 matrix

```
X = mnist.data.reshape((60000,28*28)).float()/255
```

The target data T is a "one-hot" representation of our outputs. That is, instead of the desired output to be `5`, the output should be `[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]` and if the desired output should be `0`, that would be `[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]`.

```
T = torch.nn.functional.one_hot(mnist.targets).float()
```

If our output is `Y=X @ W`, we need to find `W` such that `Y` is as close as possible to `T`. For Ridge Regression, this is computed as

```
W = torch.inverse(X.T @ X + lambd*I) @ (X.T @ T)
```

where `I` is an identity matrix of the correct size (`torch.eye(784).float()`) and `lambd` is the λ parameter that stops the regression from overfitting.

When building any sort of classifier model, we generally want to create the model using one set of data, and then test it on another set of data. Here, we will use the first 5,000 data points for creating ("training") the model, and the other 55,000 for testing:

```
N = 5000
X_train, X_test = X[:N], X[N:]  # split X into two parts for training and testing
T_train, T_test = T[:N], T[N:]  # split T into two parts for training and testing
```

Given this data, you should find `W` using *only* the `X_train` and `T_train` data. Once you find `W` you can apply it to the `X_train` and `X_test` to get `Y_train` and `Y_test`

```
Y_train = X_train @ W
Y_test = X_test @ W
```

Finally, you can compute the accuracy by determining when the output is the correct category. Here we will do this by counting when the largest output value in each row in `Y` is at the same spot as the largest output value in each row in `T`:

```
accuracy_train = torch.sum(torch.argmax(Y_train, axis=1)==torch.argmax(T_train, axis=
```

```
1))/len(Y_train)
accuracy_test = torch.sum(torch.argmax(Y_test, axis=1)==torch.argmax(T_test, axis=1))
/len(Y_test)
```

- **Compute the training and testing accuracy when $\lambda = 1$ and we use the first 5,000 data points as for training (and test on the remaining 55,000). Report both numbers.**
- **Do we expect the testing accuracy to be larger or smaller than the training accuracy? Why?**

In [4]:

```
# load MNIST data and convert to PyTorch tensors
X = mnist.data.reshape((60000,28*28)).float()/255 # collapses the image pixel data into a s
ingle dimension (1x784)
T = torch.nn.functional.one_hot(mnist.targets).float()  # gets one-hot encoding of target la
bels (true labels)
T[:5] # print the first 5 one-hot encoded labels
```

Out[4]:

```
tensor([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

In [5]:

```
# split up data into training and testing sets
N = 5000 # size of training set
X_train, X_test = X[:N], X[N:]  # split X into two parts for training and testing
T_train, T_test = T[:N], T[N:]  # split T into two parts for training and testing
```

In [6]:

```
# calculate the weight matrix using the Ridge regression formula:
lambd = 1 # set lambda value for regularization
I = torch.eye(784).float() # identity matrix of size 784x784
W = torch.inverse(X_train.T @ X_train + lambd*I) @ (X_train.T @ T_train)
```

In [7]:

```
# calculate predicted labels for training and testing sets using the weight matrix W
Y_train = X_train @ W
Y_test = X_test @ W

# calculate model accuracy using the predicted labels and true labels
accuracy_train = torch.sum(torch.argmax(Y_train, axis=1)==torch.argmax(T_train, axis=1))/le
n(Y_train) # divide by len to get percentage accuracy
accuracy_test = torch.sum(torch.argmax(Y_test, axis=1)==torch.argmax(T_test, axis=1))/len(Y
_test)
```

In [8]:

```
print(f"Train accuracy: {accuracy_train:.4f}") # ~90% accuracy
print(f"Test accuracy: {accuracy_test:.4f}") # ~82% accuracy
```

```
Train accuracy: 0.9034
Test accuracy: 0.8192
```

Discussion Question:

- Do we expect the testing accuracy to be larger or smaller than the training accuracy? Why?

It is expected that the testing accuracy will be slightly lower than the training accuracy, since the classifier has

already seen all of the training data (but has not yet seen the test data). In other words, we can expect that the classification error will be slightly higher for the test set – this is consistent with the result above, where the accuracy in classifying training data is about 8 to 9% higher than the accuracy in classifying test data.

Intuitively, this makes sense because the test set might contain data samples that are completely different from anything the classifier has seen in the training set. Ideally, this would be avoided by training on a set that is (a) large enough, and (b) a representative sample of all data, so that the test set does not contain anything vastly different from what was encountered during training.

2. b) [2 marks] Repeat part a) but vary the value of `lambd` from 10^{-4} to 10^5. You can use a `for` loop such as `for lambd in np.logspace(-5, 5, 11):`.

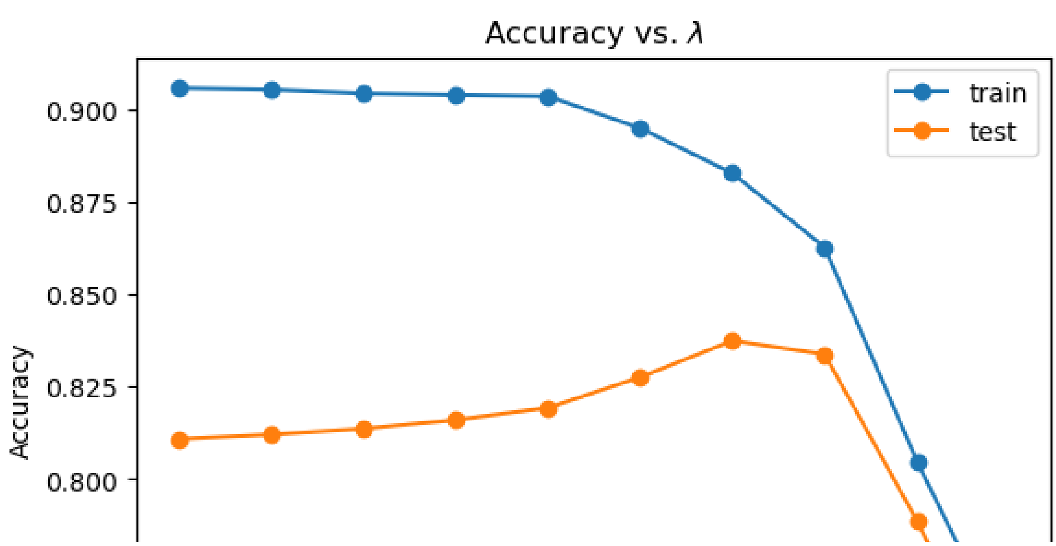
- **Generate a single plot that shows the training and testing accuracy. Make sure to label your axes and the lines on the plot.**
- **What is the best value for `lambd` (i.e. the value for which we get the best training accuracy).**
- **Why does changing `lambd` affect the accuracy?**
- **Why would having a large `lambd` value be good for making a biologically realistic model?**

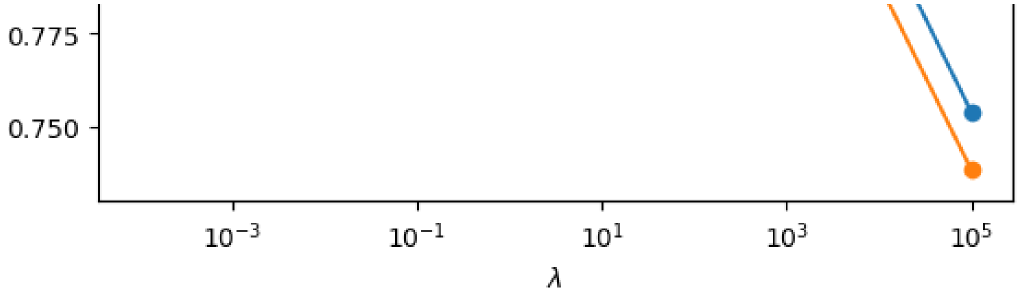
In [9]:

```
lambda_values = np.logspace(-4, 5, 10) # create 11 values of lambda from 10^-4 to 10^5
acc_train = np.zeros(10)
acc_test = np.zeros(10)

# repeat steps above for different values of lambda:
for i, lambd in enumerate(lambda_values):
    W = torch.inverse(X_train.T @ X_train + lambd*I) @ (X_train.T @ T_train)
    Y_train = X_train @ W
    Y_test = X_test @ W
    acc_train[i] = torch.sum(torch.argmax(Y_train, axis=1)==torch.argmax(T_train, axis=1))/
len(Y_train)
    acc_test[i]  = torch.sum(torch.argmax(Y_test, axis=1)==torch.argmax(T_test, axis=1))/le
n(Y_test)
```

In [10]:

```
# plot train and test accuracy vs. lambda
fig, ax = plt.subplots()
ax.plot(lambda_values, acc_train, 'o-', label='train')
ax.plot(lambda_values, acc_test, 'o-', label='test')
ax.set_title('Accuracy vs. $\lambda$')
ax.set(xlabel='$\lambda$', ylabel='Accuracy')
ax.set(xscale='log')
ax.legend()
plt.show()
```





In [11]:

```
print(f"Optimal train lambda value: {lambda_values[np.argmax(acc_train)]}") # 10^-4
print(f"Optimal test lambda value: {lambda_values[np.argmax(acc_test)]}") # 10^2
```

```
Optimal train lambda value: 0.0001
Optimal test lambda value: 100.0
```

Discussion Questions:

- What is the best value for `lambd` (i.e. the value for which we get the best training accuracy?

The best *train* accuracy is seen when we use a `lambd` value of 10e-4 (highest point of blue curve). However, we would likely want to choose a `lambd` value that will give us the highest *test accuracy*, since this is more reflective of how the system will perform in practice. This value is comparatively much larger, at $\lambda = 100$. This indicates that a larger `lambd` value is required to prevent overfitting to the (relatively small) training dataset that we are using.

- Why does changing `lambd` affect the accuracy?

`lambd` is the regularization factor in the Ridge regression formula, and its purpose is to prevent the model from choosing overly large weights (i.e., overfitting).

- Why would having a large `lambd` value be good for making a biologically realistic model?

A larger value of `lambd` will increase the realism of the model because bioligical systems are trained on massive amounts of data (much more than any artificial neural network). Using the regression analogy, the `lambd` value of biological systems is large, since these systems are naturally biased against having large 'weights' in order to maximize generalizability.

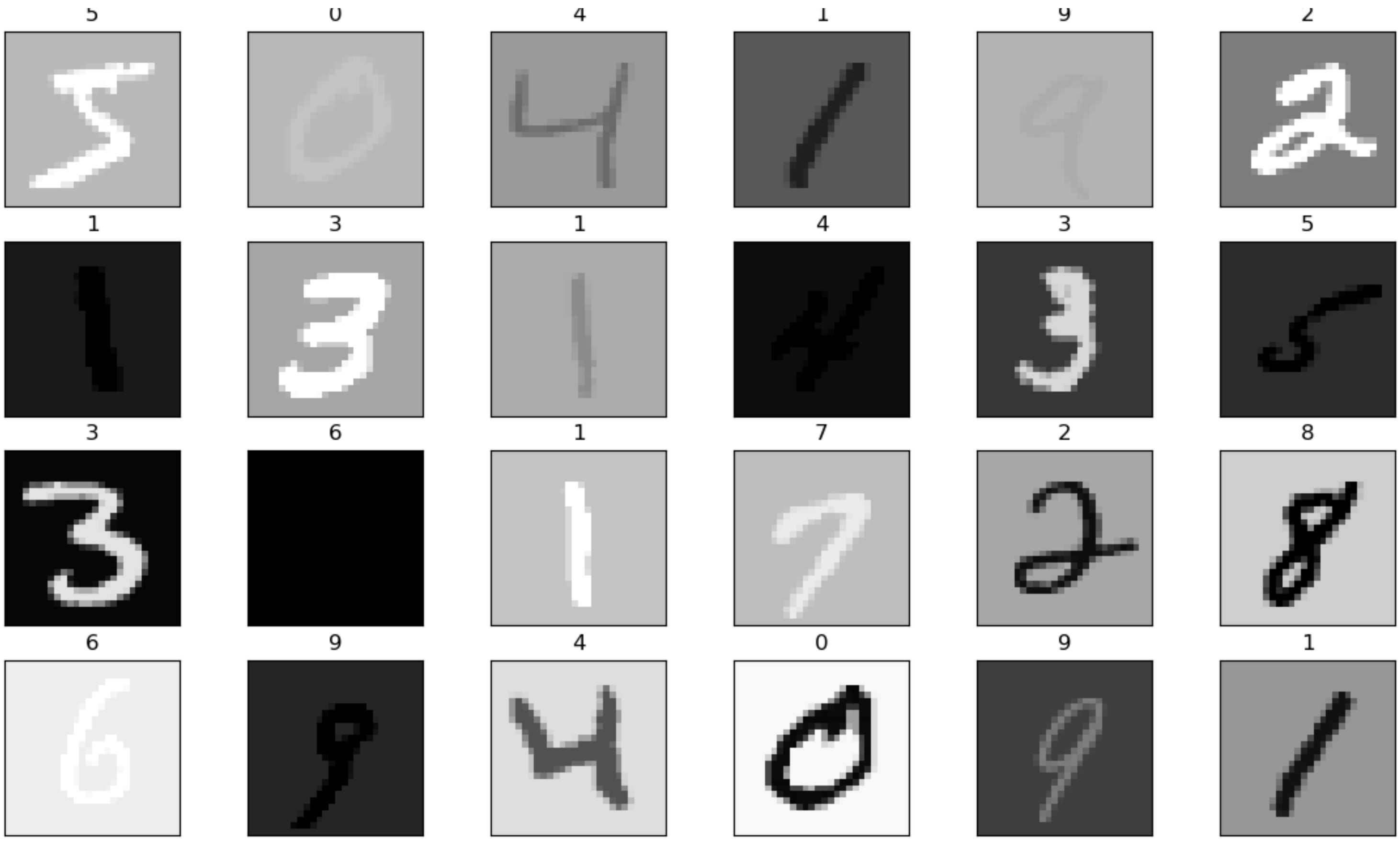
2. c) [1 mark] The input we have used so far is not very realistic. In real life, when we see written digits, they are under a wide range of lighting conditions. For this question, we change `X` by scaling it randomly and adding a random background brightness.

```
X = mnist.data.reshape((60000,28*28)).float()/255
X = X*(1-2*torch.rand(60000)[:,None]) + torch.rand(60000)[:,None]
```

To see what this looks like, here is how you can plot it:

In [12]:

```
# X = mnist.data.reshape((60000,28*28)).float()/255
X_noisy = X*(1-2*torch.rand(60000))[:,None] + torch.rand(60000)[:,None]

plt.figure(figsize=(14,8))
for i in range(24):
    plt.subplot(4, 6, i+1)
    plt.imshow(X_noisy[i].reshape(28,28), vmin=0, vmax=1, cmap='gray_r')
    plt.xticks([])
    plt.yticks([])
    plt.title(int(mnist.targets[i]))
```



In [13]:

```
X_corrected = X_noisy - torch.mean(X_noisy, axis=1)[:,None] # subtract the mean
X_corrected = torch.abs(X_corrected) # take the absolute value
X_corrected = X_corrected / torch.linalg.norm(X_corrected, axis=1)[:,None] # normalize the data
```

- **Generate the same plot as in 2b) but for this new dataset.**
- **Is this a harder or easier task than with the original dataset?**
- **Is this new dataset more like the data at the retina or like the data in the ganglion cells?**
- **Is the original dataset more like the data at the retina or like the data in the ganglion cells?**

In [14]:

```
# split up new data into training and testing sets
N = 5000 # keep train/test sets the same size
X_train_noisy, X_test_noisy = X_noisy[:N], X_noisy[N:]
T_train, T_test = T[:N], T[N:]
```

In [15]:

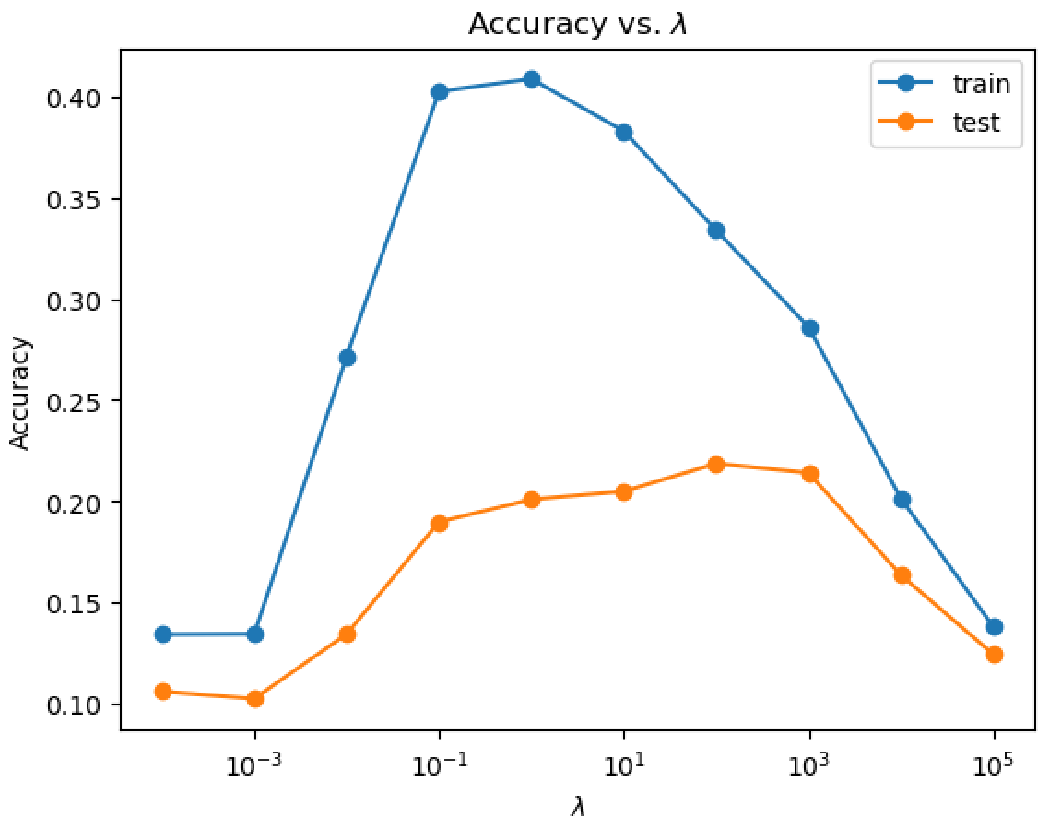
```
# repeat the steps above in 2b):
lambda_values = np.logspace(-4, 5, 10) # create 11 values of lambda from 10^-4 to 10^5
acc_train_noisy = np.zeros(10)
acc_test_noisy = np.zeros(10)

for i, lambd in enumerate(lambda_values):
    W = torch.inverse(X_train_noisy.T @ X_train_noisy + lambd*I) @ (X_train_noisy.T @ T_train)
    Y_train_noisy = X_train_noisy @ W
    Y_test_noisy = X_test_noisy @ W
    acc_train_noisy[i] = torch.sum(torch.argmax(Y_train_noisy, axis=1)==torch.argmax(T_train, axis=1))/len(Y_train_noisy)
    acc_test_noisy[i]  = torch.sum(torch.argmax(Y_test_noisy, axis=1)==torch.argmax(T_test, axis=1))/len(Y_test_noisy)
```

In [16]:

```
# plot train and test accuracy vs. lambda
fig, ax = plt.subplots()
ax.plot(lambda_values, acc_train_noisy, 'o-', label='train')
ax.plot(lambda_values, acc_test_noisy, 'o-', label='test')
ax.set_title('Accuracy vs. $\lambda$')
ax.set(xlabel='$\lambda$', ylabel='Accuracy')
ax.set(xscale='log')
ax.legend()
```

Out[16]:

```
<matplotlib.legend.Legend at 0x1240e75d0>
```



In [17]:

```
print(f"Optimal train lambda value: {lambda_values[np.argmax(acc_train_noisy)]}") # 10^-4
print(f"Optimal test lambda value: {lambda_values[np.argmax(acc_test_noisy)]}") # 10^2
```

```
Optimal train lambda value: 1.0
Optimal test lambda value: 100.0
```

Discussion Questions:

- Is this a harder or easier task than with the original dataset?

This task is much harder than with the original dataset, as seen by the reduced accuracy scores in both training and testing. This makes sense because there is significantly more variation in this data, so the classifier needs to be able to accommodate a wider range of inputs. These transformations represent noise in the incoming signal, which occurs in real-world scenarios where parts of the data migt be obscured or missing. Though the difficulty of this task is increased, training on non-ideal data ensures that the system is more likely to perform well (i.e. more robust) under a variety of circumstances.

- Is this new dataset more like the data at the retina or like the data in the ganglion cells?

The new data is more like the raw data coming into the retina, which has to accommodate varied levels of contrast

and different lighting conditions (*perceptual constancy*). The classifier now faces the additional challenge of accounting for these variations. In this way, it is not surprising that it performs worse, since we are expecting it to perform an additional function (that of the retina) with the same amount of resources as before.

- Is the original dataset more like the data at the retina or like the data in the ganglion cells?

In contrast, the original dataset is more like the data in the retinal ganglion cells. We can consider this data to have gone through a layer of preprocessing in the retina. Retinal processing allows visual inputs to be recognized even under variable lighting conditions, transforming the raw inputs to have standard brightness and contrast levels. Therefore, the data in 2c) is like the data coming into the retina, and the data in 2b) is more like the output of this processes.

2. d) [1 mark] **We can think of neurons in the visual system as transforming the data in various ways. Given the dataset in 2c), neurons might be able to transform it to look more like the origin data.**

Here are three data transformations that could be applied here:

Subtracting the Mean

```
X = X-torch.mean(X, axis=1)[:,None]
```

Absolute value

```
X = torch.abs(X)
```

Normalizing

```
X = X/torch.linalg.norm(X, axis=1)[:,None]
```

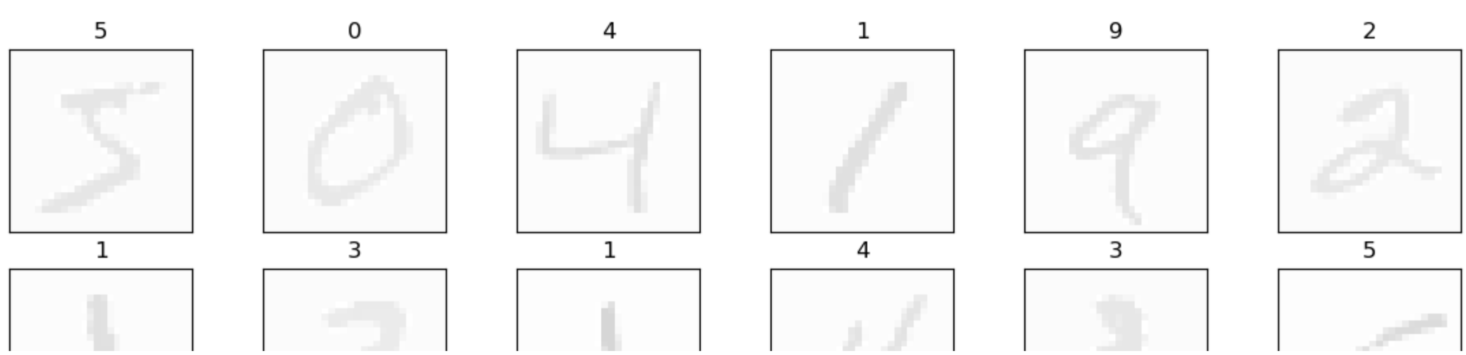
- **Apply all three of them (in the order shown above) to the dataset and generate the same graph as in 2b) and 2c).**
- **How does the performance of the network compare to that of 2b) and 2c)?**
- **Do any of the three transformations above correspond to processing that occurs in the eye before the signal is sent to the rest of the brain?**
- **Given this result, why does the eye transform the data between raw rods & cones and the ganglion cells?**

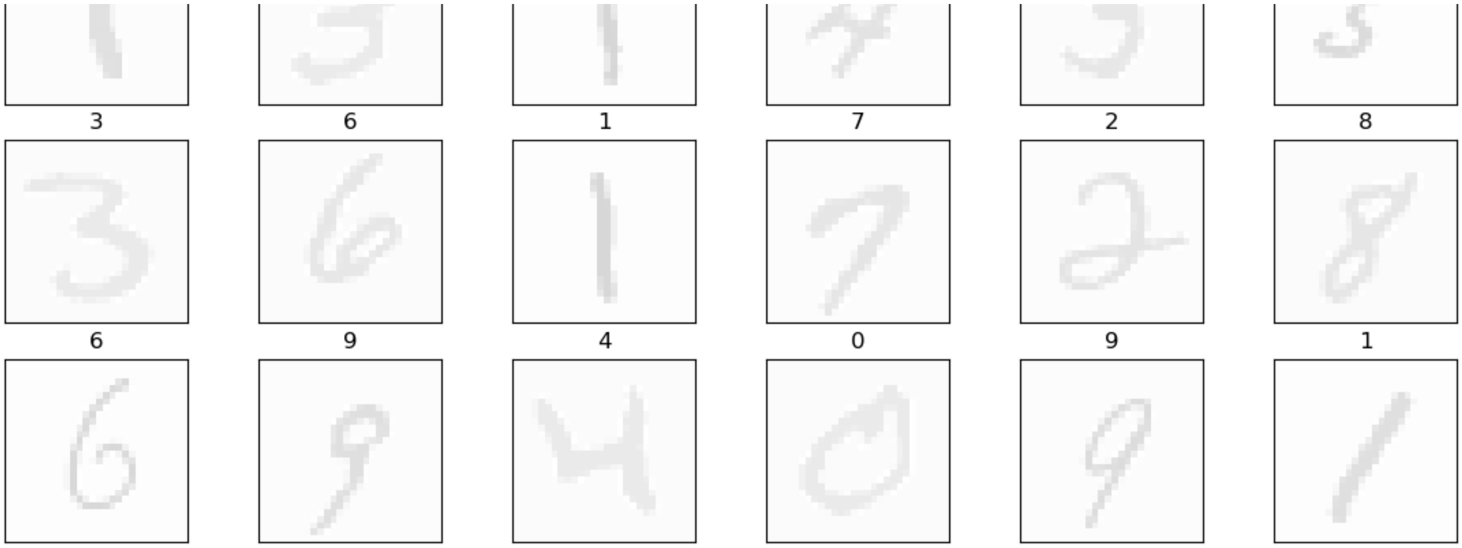
In [18]:

```
X_corrected = X_noisy - torch.mean(X_noisy, axis=1)[:,None] # subtract the mean
X_corrected = torch.abs(X_corrected) # take the absolute value
X_corrected = X_corrected / torch.linalg.norm(X_corrected, axis=1)[:,None] # normalize the data
```

In [19]:

```
# visualize the corrected images:
plt.figure(figsize=(14,8))
for i in range(24):
    plt.subplot(4, 6, i+1)
    plt.imshow(X_corrected[i].reshape(28,28), vmin=0, vmax=1, cmap='gray_r')
    plt.xticks([])
    plt.yticks([])
    plt.title(int(mnist.targets[i]))
```





In [20]:

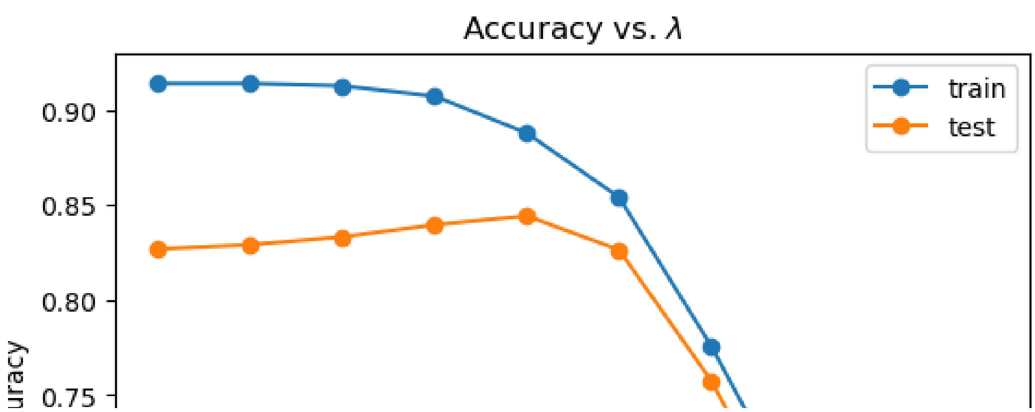
```
# apply the same network as in 2b/2c:
N = 5000
X_train_corrected, X_test_corrected = X_corrected[:N], X_corrected[N:]
T_train, T_test = T[:N], T[N:]
```

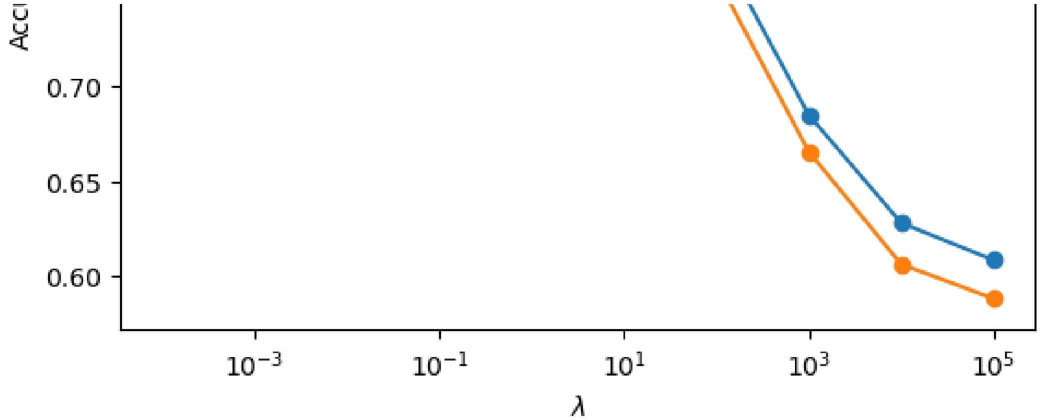
In [21]:

```
lambda_values = np.logspace(-4, 5, 10) # create 11 values of lambda from 10^-4 to 10^5
acc_train_corrected = np.zeros(10)
acc_test_corrected = np.zeros(10)

for i, lambd in enumerate(lambda_values):
    W = torch.pinverse(X_train_corrected.T @ X_train_corrected + lambd*I) @ (X_train_correc
ted.T @ T_train)
    Y_train_corrected = X_train_corrected @ W
    Y_test_corrected = X_test_corrected @ W
    acc_train_corrected[i] = torch.sum(torch.argmax(Y_train_corrected, axis=1)==torch.argma
x(T_train, axis=1))/len(Y_train_corrected)
    acc_test_corrected[i]  = torch.sum(torch.argmax(Y_test_corrected, axis=1)==torch.argmax
(T_test, axis=1))/len(Y_test_corrected)
```

In [22]:

```
# plot train and test accuracy vs. lambda
fig, ax = plt.subplots()
ax.plot(lambda_values, acc_train_corrected, 'o-', label='train')
ax.plot(lambda_values, acc_test_corrected, 'o-', label='test')
ax.set_title('Accuracy vs. $\lambda$')
ax.set(xlabel='$\lambda$', ylabel='Accuracy')
ax.set(xscale='log')
ax.legend()
plt.show()
```





In [23]:

```
print(f"Optimal train lambda value: {lambda_values[np.argmax(acc_train_corrected)]}")
print(f"Optimal test lambda value: {lambda_values[np.argmax(acc_test_corrected)]}")
```

```
Optimal train lambda value: 0.0001
Optimal test lambda value: 1.0
```

Discussion Questions:

- How does the performance of the network compare to that of 2b) and 2c)?

The network performs much better than in 2c) with the noisy data. The corrected dataset has lower contrast than the original dataset, but has much more consistency than the uncorrected data. From the plot above, we can also see that changes to the dataset have resulted in the optimal regularization parameter λ being much lower than before.

- Do any of the three transformations above correspond to processing that occurs in the eye before the signal is sent to the rest of the brain?

- **Subtracting the mean – this is similar to the low-level processing that occurs in the retina in the form of light adaptation [1].**
- **Absolute value – this operation takes any images that are inverted (i.e. where the background is darker than the text) and inverts them so that the opposite is true. This is like the transformation that is done by the ON and OFF phototransduction pathways of bipolar cells. ON pathways preserve the sign of the signal, whereas OFF pathways invert the sign.**
- **Normalizing – this operation rescales the data so that there is more consistency across images, increasing the contrast between the object and the background. A similar process is conducted by horizontal cells in the retina, which increases contrast between regions of an image by inhibiting neighbouring cells from firing, thus increasing contrast at object borders [1].**

- Given this result, why does the eye transform the data between raw rods & cones and the ganglion cells?

The eye transforms incoming data because this way, it is more able to reliably detect features of visual stimuli. The experiment above shows that the additional step of correcting (transforming) the data leads to significantly better performance, which is an important part of being aware of the surrounding environment, which would provide a significant biological/evolutionary advantage.

3. Classifying Stimuli Using Backpropogation

Regression is restricted to learning the layer of weights that produces the final output. If we want to also learn what features are most useful for producing that output, we need a more complex learning rule, and this is typically backpropogation. Here we will classify the same data as in question 2, and we will build up different network structures to do so.

Backpropogation tends to work best when learning on a bunch of data at the same time (a "batch"). The following

Backpropagation tends to work best when learning on a bunch of data at the same time (a "batch"). The following code will set up the same training and testing data as in question 2, but presented in randomized batches of 1000 at a time.

```
mnist = torchvision.datasets.MNIST(root='.', download=True, transform=torchvision.transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(torch.utils.data.Subset(mnist, np.arange(5000)),
                                            batch_size=1000, shuffle=True)
test_loader = torch.utils.data.DataLoader(torch.utils.data.Subset(mnist, np.arange(5000, 10000)),
                                           batch_size=1000, shuffle=True)
```

To create a neural network, we need to define what the weights are we will learn and we need to define the computation that the network will perform. Here is the definition of a simple network that has an input of 784 values (the MNIST inputs), which go to 50 "hidden"-layer neurons, and then to the output 10 neurons. So the network will learn to transform the 784 inputs into 50 new representations, and from those 50 features it will learn weights to create an output of 10 values (our 10 categories). This is known as a multi-layer perceptron, or a standard neural network with a single hidden layer.

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # these will be learned
        self.fc1 = nn.Linear(784, 50)    # the weights from the input to the new learned features (hidden layer)
        self.fc2 = nn.Linear(50, 10)     # the weights from the hidden layer to the output

    def forward(self, x):
        # the processing the network will do
        x = x.view(-1, 784)             # flatten the input from 28x28 to 784 values
        x = F.relu(self.fc1(x))         # apply the first set of weights, then apply the ReLU neuron model
        x = self.fc2(x)                 # apply the second set of weights
        return F.log_softmax(x)         # apply a softmax function as we just want one large output indicating category

network = Net()
```

Finally, we need to train our model. When training, it is useful to keep track of how well the model is doing on the testing data. Since testing the network takes time, we don't necessarily want to do it all the time. Instead, the following code trains the network 10 times, and then records how well the network does on the training data and on the testing data.

```
# create the learning rule
optimizer = optim.SGD(network.parameters(),
                      lr=0.1,   # learning rate
                      momentum=0.5)

# variables to keep track of the training and testing accuracy
accuracy_train = []
accuracy_test = []
```

```
accuracy_test = []

def continue_training():
    network.train()       # configure the network for training
    for i in range(10):   # train the network 10 times
        correct = 0
        for data, target in train_loader:       # working in batchs of 1000
            optimizer.zero_grad()               # initialize the learning system
            output = network(data)              # feed in the data
            loss = F.nll_loss(output, target)   # compute how wrong the output is
            loss.backward()                     # change the weights to reduce error
            optimizer.step()                    # update the learning rule

            pred = output.data.max(1, keepdim=True)[1]           # compute which output is largest
            correct += pred.eq(target.data.view_as(pred)).sum()   # compute the number of correct outputs
    # update the list of training accuracy values
    score = float(correct/len(train_loader.dataset))
    accuracy_train.append(score)
    print('Iteration', len(accuracy_train), 'Training accuracy:', score)

    correct = 0
    network.eval()
    for data, target in test_loader:    # go through the test data once (in groups of 1000)
        output = network(data)                               # feed in the data
        pred = output.data.max(1, keepdim=True)[1]           # compute which output is largest
        correct += pred.eq(target.data.view_as(pred)).sum()   # compute the number of correct outputs
    # update the list of testing accuracy values
    score = float(correct/len(test_loader.dataset))
    accuracy_test.append(score)
    print('Iteration', len(accuracy_test), 'Testing accuracy:', score)
```

Given the above code, you can train your network 10 times by doing

```
for i in range(10):
    continue_training()
```

If you want to continue training even more, you can just run that `for` loop again.

To plot the final accuracy results, you can use

```
plt.figure(figsize=(12,4))
plt.plot(accuracy_train, label='training')
plt.plot(accuracy_test, label='testing')
plt.legend()
plt.xlabel('training iterations')
plt.ylabel('accuracy')
plt.show()
```

3. a) [1 mark] Run the model above for 10 iterations (i.e. call `continue_training` 10 times).

- **Plot the training and testing accuracy.**

- **Is this model better or worse than the best models developed in question 2?**

In [24]:

```
# load the MNIST data in batches of 1000
mnist = torchvision.datasets.MNIST(root='.', download=True, transform=torchvision.transforms
.ToTensor())
train_loader = torch.utils.data.DataLoader(torch.utils.data.Subset(mnist, np.arange(5000)),
batch_size=1000, shuffle=True)
test_loader = torch.utils.data.DataLoader(torch.utils.data.Subset(mnist, np.arange(5000, 10
000)), batch_size=1000, shuffle=True)
```

In [25]:

```
import torch.nn as nn
import torch.nn.functional as F

# modify the Net class to include the hidden layer size as an argument (will be used in 3c)
class Net(nn.Module):
    def __init__(self, hidden_size=50):
        super(Net, self).__init__()
        # these will be learned
        self.fc1 = nn.Linear(784, hidden_size)    # the weights from the input to the new le
arned features (hidden layer)
        self.fc2 = nn.Linear(hidden_size, 10)     # the weights from the hidden layer to the
output

    def forward(self, x):
        # the processing the network will do
        x = x.view(-1, 784)             # flatten the input from 28x28 to 784 values
        x = F.relu(self.fc1(x))         # apply the first set of weights, then apply the Re
LU neuron model
        x = self.fc2(x)                 # apply the second set of weights
        return F.log_softmax(x)         # apply a softmax function as we just want one larg
e output indicating category
```

In [26]:

```
# create the model and learning rule:
LR = 0.1
MOMENTUM = 0.5
network = Net()
optimizer = torch.optim.SGD(network.parameters(), lr=LR, momentum=MOMENTUM)
```

In [27]:

```
def continue_training(train_list, test_list):
    network.train()       # configure the network for training
    for i in range(10):   # train the network 10 times
        correct = 0
        for data, target in train_loader:       # working in batchs of 1000
            optimizer.zero_grad()               # initialize the learning system
            output = network(data)              # feed in the data
            loss = F.nll_loss(output, target)   # compute how wrong the output is
            loss.backward()                     # change the weights to reduce error
            optimizer.step()                    # update the learning rule

            pred = output.data.max(1, keepdim=True)[1]            # compute which output is
largest
            correct += pred.eq(target.data.view_as(pred)).sum()   # compute the number of co
rrect outputs

    # update the list of training accuracy values
    train_score = float(correct/len(train_loader.dataset))
    train_list.append(train_score)
    print(f'Iteration {len(train_list)} - training accuracy: {train_score:.8f}')
```

```
    correct = 0
    network.eval()
    for data, target in test_loader:    # go through the test data once (in groups of 1000)
        output = network(data)                               # feed in the data
        pred = output.data.max(1, keepdim=True)[1]           # compute which output is largest
        correct += pred.eq(target.data.view_as(pred)).sum()  # compute the number of correct outputs

    # update the list of testing accuracy values
    test_score = float(correct/len(test_loader.dataset))
    test_list.append(test_score)
    print(f'Iteration {len(test_list)} - testing accuracy: {test_score:.8f}')
```

In [28]:

```
# variables to keep track of the training and testing accuracy
train_3a = []
test_3a = []

for i in range(10):
    continue_training(train_3a, test_3a)
```

```
/var/folders/rx/5_fd7v5s5dbc3yr3cx7bw9m40000gn/T/ipykernel_95115/3140384043.py:17: UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.
  return F.log_softmax(x)         # apply a softmax function as we just want one large output indicating category
```

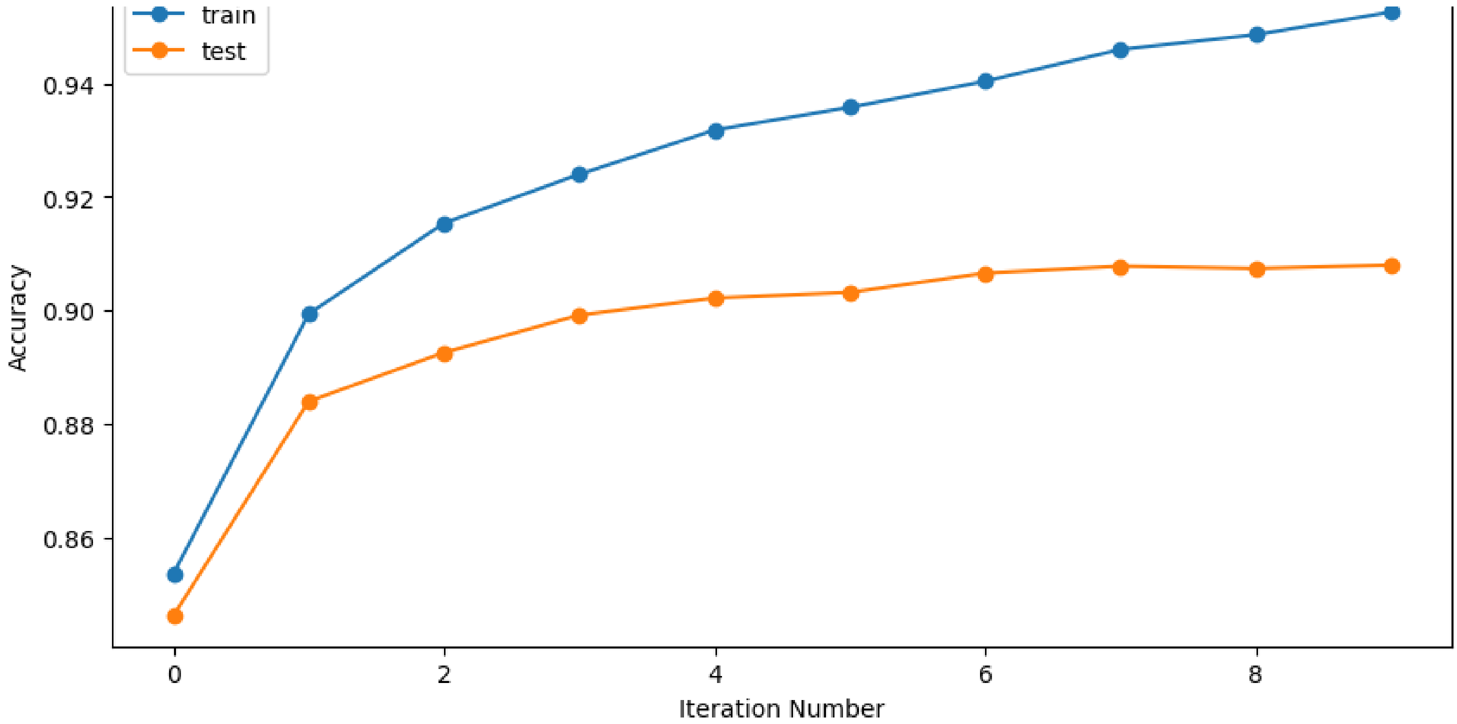
```
Iteration 1 - training accuracy: 0.85360003
Iteration 1 - testing accuracy: 0.84619999
Iteration 2 - training accuracy: 0.89940000
Iteration 2 - testing accuracy: 0.88400000
Iteration 3 - training accuracy: 0.91540003
Iteration 3 - testing accuracy: 0.89260000
Iteration 4 - training accuracy: 0.92400002
Iteration 4 - testing accuracy: 0.89920002
Iteration 5 - training accuracy: 0.93180001
Iteration 5 - testing accuracy: 0.90219998
Iteration 6 - training accuracy: 0.93580002
Iteration 6 - testing accuracy: 0.90319997
Iteration 7 - training accuracy: 0.94040000
Iteration 7 - testing accuracy: 0.90660000
Iteration 8 - training accuracy: 0.94599998
Iteration 8 - testing accuracy: 0.90780002
Iteration 9 - training accuracy: 0.94859999
Iteration 9 - testing accuracy: 0.90740001
Iteration 10 - training accuracy: 0.95260000
Iteration 10 - testing accuracy: 0.90799999
```

In [29]:

```
# plot the training and testing accuracy over 10 iterations
fig, ax = plt.subplots(figsize=(10,5))
ax.plot(train_3a, 'o-', label='train')
ax.plot(test_3a, 'o-', label='test')
ax.set_title('Accuracy Over 10 Iterations')
ax.set_xlabel('Iteration Number')
ax.set_ylabel('Accuracy')
ax.legend()
```

Out[29]:

```
<matplotlib.legend.Legend at 0x1239f6b50>
```



In [30]:

```
print(f"Max training accuracy: {max(train_3a):.4f}")
print(f"Max testing accuracy: {max(test_3a):.4f}")
```

```
Max training accuracy: 0.9526
Max testing accuracy: 0.9080
```

Discussion Question:

- Is this model better or worse than the best models developed in question 2?

The highest test accuracy achieved in this case is about 90%, whereas the best model from Q2 achieved an accuracy of about 85%. This shows that backpropagation provides a significant improvement over regression. This type of learning also has the advantage of not needing to tune the hyperparameter λ, which can result in poor performance if chosen incorrectly.

3. b) [1 mark] Repeat question 3a five times. This does not mean to run a single model for 50 iterations. Rather, you need to reset the model and train it again. The easiest way to do this is to recreate the network and the optimizer like this:

```
network = Net()
optimizer = optim.SGD(network.parameters(),
                      lr=0.1,
                      momentum=0.5)
```

- **Make a plot showing the 5 different training accuracies and 5 different testing accuracies**
- **Also show the average training and testing accuracy on the plot.**
- **Each of the 5 models should show slightly different accuracies. Why is this the case?**

In [31]:

```
# repeat 5 times and plot train and test accuracy:
fig, ax = plt.subplots(figsize=(10,5))
train_mean_3b = np.zeros(10)
test_mean_3b = np.zeros(10)

for i in range(5):
```

```
    accuracy_train = []
    accuracy_test = []
    network = Net()
    optimizer = torch.optim.SGD(network.parameters(), lr=LR, momentum=MOMENTUM)
    for j in range(10):
        continue_training(accuracy_train, accuracy_test)
    ax.plot(accuracy_train, linewidth=0.5, alpha=0.5)
    ax.plot(accuracy_test, linewidth=0.5, alpha=0.5)
    train_mean_3b += np.array(accuracy_train)
    test_mean_3b += np.array(accuracy_test)
train_mean_3b /= 5
test_mean_3b /= 5

ax.plot(train_mean_3b, 'o-', linewidth=2, label='train (mean)')
ax.plot(test_mean_3b, 'o-', linewidth=2, label='test (mean)')
ax.set_title('Accuracy Over 10 Iterations')
ax.set_xlabel('Iteration Number')
ax.set_ylabel('Accuracy')
ax.legend()
```

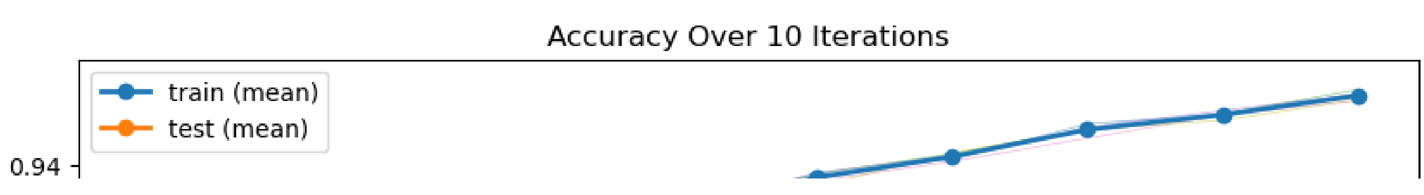
```
/var/folders/rx/5_fd7v5s5dbc3yr3cx7bw9m40000gn/T/ipykernel_95115/3140384043.py:17: UserWarni
ng: Implicit dimension choice for log_softmax has been deprecated. Change the call to includ
e dim=X as an argument.
  return F.log_softmax(x)         # apply a softmax function as we just want one large outpu
t indicating category
```

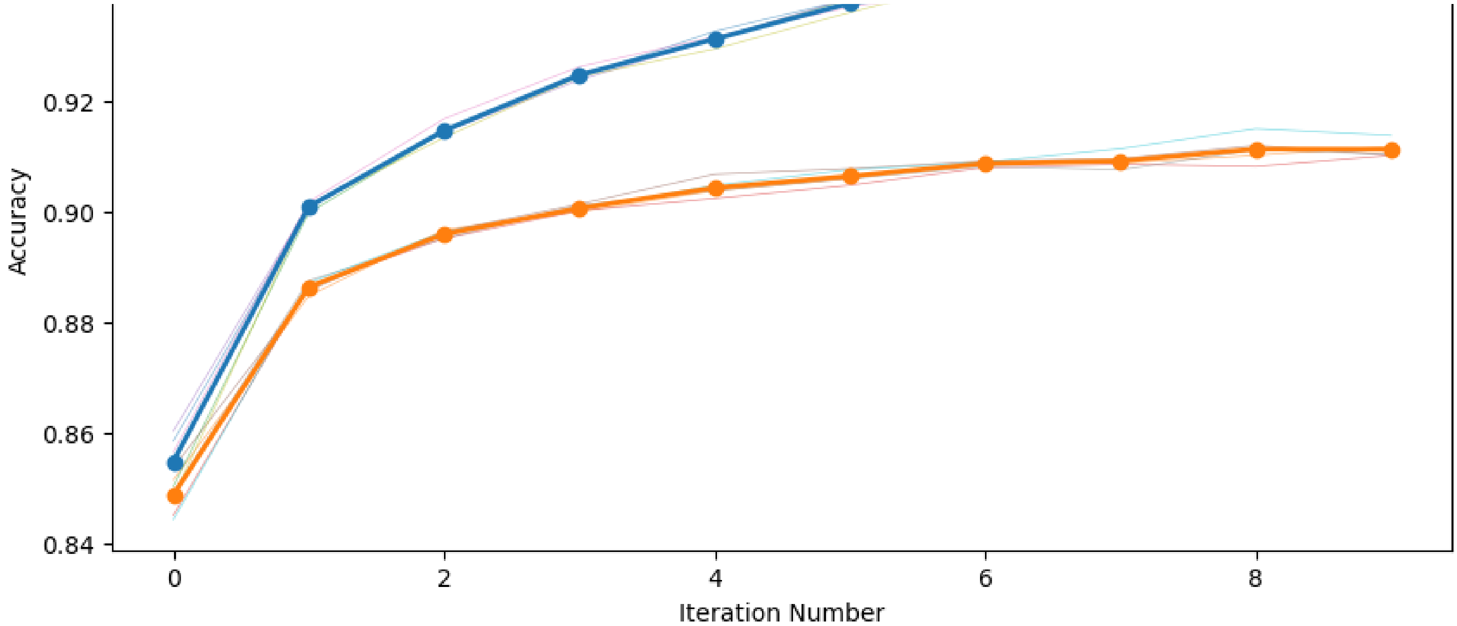
```
Iteration 1 - training accuracy: 0.85860002
Iteration 1 - testing accuracy: 0.85159999
Iteration 2 - training accuracy: 0.90079999
Iteration 2 - testing accuracy: 0.88480002
Iteration 3 - training accuracy: 0.91439998
Iteration 3 - testing accuracy: 0.89679998
Iteration 4 - training accuracy: 0.92439997
Iteration 4 - testing accuracy: 0.89999998
Iteration 5 - training accuracy: 0.93260002
Iteration 5 - testing accuracy: 0.90380001
Iteration 6 - training accuracy: 0.93839997
Iteration 6 - testing accuracy: 0.90600002
Iteration 7 - training accuracy: 0.94099998
Iteration 7 - testing accuracy: 0.90920001
Iteration 8 - training accuracy: 0.94739997
Iteration 8 - testing accuracy: 0.90880001
Iteration 9 - training accuracy: 0.94919997
Iteration 9 - testing accuracy: 0.91020000
Iteration 10 - training accuracy: 0.95279998
Iteration 10 - testing accuracy: 0.91180003
Iteration 1 - training accuracy: 0.85039997
Iteration 1 - testing accuracy: 0.84520000
Iteration 2 - training accuracy: 0.89980000
Iteration 2 - testing accuracy: 0.88660002
Iteration 3 - training accuracy: 0.91439998
Iteration 3 - testing accuracy: 0.89520001
Iteration 4 - training accuracy: 0.92479998
Iteration 4 - testing accuracy: 0.90020001
Iteration 5 - training accuracy: 0.93120003
Iteration 5 - testing accuracy: 0.90240002
Iteration 6 - training accuracy: 0.93839997
Iteration 6 - testing accuracy: 0.90480000
Iteration 7 - training accuracy: 0.94199997
Iteration 7 - testing accuracy: 0.90799999
Iteration 8 - training accuracy: 0.94639999
Iteration 8 - testing accuracy: 0.90859997
Iteration 9 - training accuracy: 0.94919997
Iteration 9 - testing accuracy: 0.90820003
Iteration 10 - training accuracy: 0.95359999
Iteration 10 - testing accuracy: 0.91020000
Iteration 1 - training accuracy: 0.86040002
Iteration 1 - testing accuracy: 0.85380000
```

```
Iteration 2 - training accuracy: 0.90120000
Iteration 2 - testing accuracy: 0.88580000
Iteration 3 - training accuracy: 0.91439998
Iteration 3 - testing accuracy: 0.89639997
Iteration 4 - training accuracy: 0.92379999
Iteration 4 - testing accuracy: 0.90140003
Iteration 5 - training accuracy: 0.93120003
Iteration 5 - testing accuracy: 0.90679997
Iteration 6 - training accuracy: 0.93860000
Iteration 6 - testing accuracy: 0.90780002
Iteration 7 - training accuracy: 0.94160002
Iteration 7 - testing accuracy: 0.90920001
Iteration 8 - training accuracy: 0.94679999
Iteration 8 - testing accuracy: 0.90960002
Iteration 9 - training accuracy: 0.94980001
Iteration 9 - testing accuracy: 0.91200000
Iteration 10 - training accuracy: 0.95279998
Iteration 10 - testing accuracy: 0.91020000
Iteration 1 - training accuracy: 0.85659999
Iteration 1 - testing accuracy: 0.84939998
Iteration 2 - training accuracy: 0.90179998
Iteration 2 - testing accuracy: 0.88760000
Iteration 3 - training accuracy: 0.91680002
Iteration 3 - testing accuracy: 0.89520001
Iteration 4 - training accuracy: 0.92619997
Iteration 4 - testing accuracy: 0.90120000
Iteration 5 - training accuracy: 0.93159997
Iteration 5 - testing accuracy: 0.90380001
Iteration 6 - training accuracy: 0.93699998
Iteration 6 - testing accuracy: 0.90600002
Iteration 7 - training accuracy: 0.94059998
Iteration 7 - testing accuracy: 0.90820003
Iteration 8 - training accuracy: 0.94480002
Iteration 8 - testing accuracy: 0.90759999
Iteration 9 - training accuracy: 0.94900000
Iteration 9 - testing accuracy: 0.91119999
Iteration 10 - training accuracy: 0.95160002
Iteration 10 - testing accuracy: 0.91039997
Iteration 1 - training accuracy: 0.84859997
Iteration 1 - testing accuracy: 0.84439999
Iteration 2 - training accuracy: 0.90100002
Iteration 2 - testing accuracy: 0.88720000
Iteration 3 - training accuracy: 0.91339999
Iteration 3 - testing accuracy: 0.89639997
Iteration 4 - training accuracy: 0.92439997
Iteration 4 - testing accuracy: 0.90039998
Iteration 5 - training accuracy: 0.92940003
Iteration 5 - testing accuracy: 0.90480000
Iteration 6 - training accuracy: 0.93599999
Iteration 6 - testing accuracy: 0.90759999
Iteration 7 - training accuracy: 0.94220001
Iteration 7 - testing accuracy: 0.90899998
Iteration 8 - training accuracy: 0.94639999
Iteration 8 - testing accuracy: 0.91140002
Iteration 9 - training accuracy: 0.94800001
Iteration 9 - testing accuracy: 0.91500002
Iteration 10 - training accuracy: 0.95179999
Iteration 10 - testing accuracy: 0.91380000
```

Out[31]:

```
<matplotlib.legend.Legend at 0x123c89e90>
```





In [32]:

```
print(f"Max training accuracy: {max(train_mean_3b):.4f}")
print(f"Max testing accuracy: {max(test_mean_3b):.4f}")
```

```
Max training accuracy: 0.9525
Max testing accuracy: 0.9113
```

Discussion Question:

- Each of the 5 models should show slightly different accuracies. Why is this the case?

The difference in accuracy between the 5 runs is due to slight differences in the dataset that each network is trained on. In the `train_loader` and `test_loader`, the `shuffle=True` parameter ensures that the images are presented in a different order each time when the images are batched (even though the same 5000 images are used). This process of training multiple networks and averaging the values results in more reliable accuracy scores, ensuring that a very high or low score was not achieved due to random chance.

3. c) [1 mark] Repeat question 3b varying the number of neurons in the hidden layer of the network. The current value is 50. Try it with 5, 10, 20, 50, and 100 neurons. For each number of neurons, repeat five times and take the average (like in question 3b).

- **Plot the final testing accuracy on the y-axis and the number of neurons on the x-axis. Note that to speed things up you can remove the testing computation from continue_training until the very end, since we only need the final testing score.**

In [33]:

```
# repeat 3b) using different hidden layer sizes:
hidden_sizes = [5, 10, 20, 50, 100]
train_3c = np.zeros((len(hidden_sizes), 5)) # save one value for each hidden layer size and
each of 5 trials
test_3c = np.zeros((len(hidden_sizes), 5))

for k in range(len(hidden_sizes)):
    print(f'Hidden layer size: {hidden_sizes[k]}')
    for i in range(5): # repeat the above 5 times for each hidden layer size
        print(f"Trial {i} of 5")
        accuracy_train = []
        accuracy_test = []
        network = Net(hidden_size=hidden_sizes[k])
```

```
        optimizer = torch.optim.SGD(network.parameters(), lr=LR, momentum=MOMENTUM)
        for j in range(10):
            continue_training(accuracy_train, accuracy_test)
        train_3c[k][i] += np.array(accuracy_train[-1]) # take last element of accuracy_train list (model accuracy after 10 iterations)
        test_3c[k][i] += np.array(accuracy_test[1])
```

```
Hidden layer size: 5
Trial 0 of 5
```

```
/var/folders/rx/5_fd7v5s5dbc3yr3cx7bw9m40000gn/T/ipykernel_95115/3140384043.py:17: UserWarning: Implicit dimension choice for log_softmax has been deprecated. Change the call to include dim=X as an argument.
  return F.log_softmax(x)         # apply a softmax function as we just want one large output indicating category
```

```
Iteration 1 - training accuracy: 0.73360002
Iteration 1 - testing accuracy: 0.73360002
Iteration 2 - training accuracy: 0.83099997
Iteration 2 - testing accuracy: 0.80960000
Iteration 3 - training accuracy: 0.86339998
Iteration 3 - testing accuracy: 0.84219998
Iteration 4 - training accuracy: 0.87779999
Iteration 4 - testing accuracy: 0.84700000
Iteration 5 - training accuracy: 0.88239998
Iteration 5 - testing accuracy: 0.85280001
Iteration 6 - training accuracy: 0.88840002
Iteration 6 - testing accuracy: 0.86059999
Iteration 7 - training accuracy: 0.89219999
Iteration 7 - testing accuracy: 0.85900003
Iteration 8 - training accuracy: 0.89740002
Iteration 8 - testing accuracy: 0.85820001
Iteration 9 - training accuracy: 0.89999998
Iteration 9 - testing accuracy: 0.85879999
Iteration 10 - training accuracy: 0.90319997
Iteration 10 - testing accuracy: 0.86240000
Current train mean: 0.9031999707221985
Current test mean: 0.8095999956130981
Trial 1 of 5
Iteration 1 - training accuracy: 0.62480003
Iteration 1 - testing accuracy: 0.62279999
Iteration 2 - training accuracy: 0.74379998
Iteration 2 - testing accuracy: 0.73900002
Iteration 3 - training accuracy: 0.80059999
Iteration 3 - testing accuracy: 0.78600001
Iteration 4 - training accuracy: 0.82419997
Iteration 4 - testing accuracy: 0.80479997
Iteration 5 - training accuracy: 0.83920002
Iteration 5 - testing accuracy: 0.81260002
Iteration 6 - training accuracy: 0.85360003
Iteration 6 - testing accuracy: 0.82400000
Iteration 7 - training accuracy: 0.87019998
Iteration 7 - testing accuracy: 0.83260000
Iteration 8 - training accuracy: 0.87720001
Iteration 8 - testing accuracy: 0.83859998
Iteration 9 - training accuracy: 0.88440001
Iteration 9 - testing accuracy: 0.84179997
Iteration 10 - training accuracy: 0.88720000
Iteration 10 - testing accuracy: 0.84359998
Current train mean: 0.8871999979019165
Current test mean: 0.7390000224113464
Trial 2 of 5
Iteration 1 - training accuracy: 0.74599999
Iteration 1 - testing accuracy: 0.74599999
Iteration 2 - training accuracy: 0.82359999
Iteration 2 - testing accuracy: 0.80599999
Iteration 3 - training accuracy: 0.85039997
Iteration 3 - testing accuracy: 0.82900000
```

```
Iteration 4 - training accuracy: 0.86839998
Iteration 4 - testing accuracy: 0.83759999
Iteration 5 - training accuracy: 0.87660003
Iteration 5 - testing accuracy: 0.85519999
Iteration 6 - training accuracy: 0.88499999
Iteration 6 - testing accuracy: 0.85820001
Iteration 7 - training accuracy: 0.89060003
Iteration 7 - testing accuracy: 0.86199999
Iteration 8 - training accuracy: 0.89579999
Iteration 8 - testing accuracy: 0.86479998
Iteration 9 - training accuracy: 0.90060002
Iteration 9 - testing accuracy: 0.86680001
Iteration 10 - training accuracy: 0.90200001
Iteration 10 - testing accuracy: 0.86979997
Current train mean: 0.9020000100135803
Current test mean: 0.8059999942779541
Trial 3 of 5
Iteration 1 - training accuracy: 0.70260000
Iteration 1 - testing accuracy: 0.70959997
Iteration 2 - training accuracy: 0.81220001
Iteration 2 - testing accuracy: 0.79180002
Iteration 3 - training accuracy: 0.85380000
Iteration 3 - testing accuracy: 0.82940000
Iteration 4 - training accuracy: 0.87059999
Iteration 4 - testing accuracy: 0.84660000
Iteration 5 - training accuracy: 0.88180000
Iteration 5 - testing accuracy: 0.84899998
Iteration 6 - training accuracy: 0.88720000
Iteration 6 - testing accuracy: 0.85579997
Iteration 7 - training accuracy: 0.89560002
Iteration 7 - testing accuracy: 0.85479999
Iteration 8 - training accuracy: 0.89980000
Iteration 8 - testing accuracy: 0.86360002
Iteration 9 - training accuracy: 0.90300000
Iteration 9 - testing accuracy: 0.86420000
Iteration 10 - training accuracy: 0.90619999
Iteration 10 - testing accuracy: 0.86240000
Current train mean: 0.9061999917030334
Current test mean: 0.7918000221252441
Trial 4 of 5
Iteration 1 - training accuracy: 0.77020001
Iteration 1 - testing accuracy: 0.76179999
Iteration 2 - training accuracy: 0.84320003
Iteration 2 - testing accuracy: 0.81660002
Iteration 3 - training accuracy: 0.86860001
Iteration 3 - testing accuracy: 0.83440000
Iteration 4 - training accuracy: 0.88020003
Iteration 4 - testing accuracy: 0.84600002
Iteration 5 - training accuracy: 0.88700002
Iteration 5 - testing accuracy: 0.85100001
Iteration 6 - training accuracy: 0.89520001
Iteration 6 - testing accuracy: 0.85879999
Iteration 7 - training accuracy: 0.89859998
Iteration 7 - testing accuracy: 0.85960001
Iteration 8 - training accuracy: 0.90300000
Iteration 8 - testing accuracy: 0.86420000
Iteration 9 - training accuracy: 0.90719998
Iteration 9 - testing accuracy: 0.86379999
Iteration 10 - training accuracy: 0.91140002
Iteration 10 - testing accuracy: 0.86580002
Current train mean: 0.9114000201225281
Current test mean: 0.8166000247001648
Hidden layer size: 10
Trial 0 of 5
Iteration 1 - training accuracy: 0.83719999
Iteration 1 - testing accuracy: 0.83120000
Iteration 2 - training accuracy: 0.88880002
Iteration 2 - testing accuracy: 0.87199998
```

```
Iteration 3 - training accuracy: 0.90840000
Iteration 3 - testing accuracy: 0.88400000
Iteration 4 - training accuracy: 0.92140001
Iteration 4 - testing accuracy: 0.89240003
Iteration 5 - training accuracy: 0.92860001
Iteration 5 - testing accuracy: 0.89639997
Iteration 6 - training accuracy: 0.93279999
Iteration 6 - testing accuracy: 0.89880002
Iteration 7 - training accuracy: 0.93580002
Iteration 7 - testing accuracy: 0.90079999
Iteration 8 - training accuracy: 0.94080001
Iteration 8 - testing accuracy: 0.90259999
Iteration 9 - training accuracy: 0.94139999
Iteration 9 - testing accuracy: 0.90259999
Iteration 10 - training accuracy: 0.94459999
Iteration 10 - testing accuracy: 0.90380001
Current train mean: 0.944599986076355
Current test mean: 0.871999979019165
Trial 1 of 5
Iteration 1 - training accuracy: 0.81459999
Iteration 1 - testing accuracy: 0.80440003
Iteration 2 - training accuracy: 0.88800001
Iteration 2 - testing accuracy: 0.87059999
Iteration 3 - training accuracy: 0.90640002
Iteration 3 - testing accuracy: 0.88599998
Iteration 4 - training accuracy: 0.91839999
Iteration 4 - testing accuracy: 0.88959998
Iteration 5 - training accuracy: 0.92479998
Iteration 5 - testing accuracy: 0.89399999
Iteration 6 - training accuracy: 0.92919999
Iteration 6 - testing accuracy: 0.89719999
Iteration 7 - training accuracy: 0.93540001
Iteration 7 - testing accuracy: 0.89960003
Iteration 8 - training accuracy: 0.93699998
Iteration 8 - testing accuracy: 0.90120000
Iteration 9 - training accuracy: 0.94080001
Iteration 9 - testing accuracy: 0.90100002
Iteration 10 - training accuracy: 0.94199997
Iteration 10 - testing accuracy: 0.90399998
Current train mean: 0.9419999718666077
Current test mean: 0.8705999851226807
Trial 2 of 5
Iteration 1 - training accuracy: 0.79740000
Iteration 1 - testing accuracy: 0.79759997
Iteration 2 - training accuracy: 0.87519997
Iteration 2 - testing accuracy: 0.85699999
Iteration 3 - training accuracy: 0.90399998
Iteration 3 - testing accuracy: 0.87739998
Iteration 4 - training accuracy: 0.91520000
Iteration 4 - testing accuracy: 0.88580000
Iteration 5 - training accuracy: 0.92479998
Iteration 5 - testing accuracy: 0.89120001
Iteration 6 - training accuracy: 0.92940003
Iteration 6 - testing accuracy: 0.89740002
Iteration 7 - training accuracy: 0.93360001
Iteration 7 - testing accuracy: 0.89780003
Iteration 8 - training accuracy: 0.93720001
Iteration 8 - testing accuracy: 0.89819998
Iteration 9 - training accuracy: 0.94019997
Iteration 9 - testing accuracy: 0.89980000
Iteration 10 - training accuracy: 0.94319999
Iteration 10 - testing accuracy: 0.89819998
Current train mean: 0.9431999921798706
Current test mean: 0.8569999933242798
Trial 3 of 5
Iteration 1 - training accuracy: 0.81779999
Iteration 1 - testing accuracy: 0.81019998
Iteration 2 - training accuracy: 0.87840003
```

```
Iteration 2 - testing accuracy: 0.85799998
Iteration 3 - training accuracy: 0.89819998
Iteration 3 - testing accuracy: 0.87199998
Iteration 4 - training accuracy: 0.91320002
Iteration 4 - testing accuracy: 0.88139999
Iteration 5 - training accuracy: 0.92379999
Iteration 5 - testing accuracy: 0.89020002
Iteration 6 - training accuracy: 0.93000001
Iteration 6 - testing accuracy: 0.89600003
Iteration 7 - training accuracy: 0.93260002
Iteration 7 - testing accuracy: 0.89560002
Iteration 8 - training accuracy: 0.93800002
Iteration 8 - testing accuracy: 0.89840001
Iteration 9 - training accuracy: 0.94139999
Iteration 9 - testing accuracy: 0.90079999
Iteration 10 - training accuracy: 0.94419998
Iteration 10 - testing accuracy: 0.90060002
Current train mean: 0.9441999793052673
Current test mean: 0.8579999804496765
Trial 4 of 5
Iteration 1 - training accuracy: 0.84460002
Iteration 1 - testing accuracy: 0.85000002
Iteration 2 - training accuracy: 0.89620000
Iteration 2 - testing accuracy: 0.88059998
Iteration 3 - training accuracy: 0.91159999
Iteration 3 - testing accuracy: 0.88779998
Iteration 4 - training accuracy: 0.91939998
Iteration 4 - testing accuracy: 0.89260000
Iteration 5 - training accuracy: 0.92760003
Iteration 5 - testing accuracy: 0.89340001
Iteration 6 - training accuracy: 0.93120003
Iteration 6 - testing accuracy: 0.89840001
Iteration 7 - training accuracy: 0.93360001
Iteration 7 - testing accuracy: 0.89840001
Iteration 8 - training accuracy: 0.93900001
Iteration 8 - testing accuracy: 0.90079999
Iteration 9 - training accuracy: 0.94080001
Iteration 9 - testing accuracy: 0.90240002
Iteration 10 - training accuracy: 0.94300002
Iteration 10 - testing accuracy: 0.90300000
Current train mean: 0.9430000185966492
Current test mean: 0.8805999755859375
Hidden layer size: 20
Trial 0 of 5
Iteration 1 - training accuracy: 0.86699998
Iteration 1 - testing accuracy: 0.86260003
Iteration 2 - training accuracy: 0.90160000
Iteration 2 - testing accuracy: 0.88639998
Iteration 3 - training accuracy: 0.91520000
Iteration 3 - testing accuracy: 0.89760000
Iteration 4 - training accuracy: 0.92519999
Iteration 4 - testing accuracy: 0.90179998
Iteration 5 - training accuracy: 0.93080002
Iteration 5 - testing accuracy: 0.90399998
Iteration 6 - training accuracy: 0.93640000
Iteration 6 - testing accuracy: 0.90719998
Iteration 7 - training accuracy: 0.93919998
Iteration 7 - testing accuracy: 0.90759999
Iteration 8 - training accuracy: 0.94220001
Iteration 8 - testing accuracy: 0.90719998
Iteration 9 - training accuracy: 0.94639999
Iteration 9 - testing accuracy: 0.90799999
Iteration 10 - training accuracy: 0.94700003
Iteration 10 - testing accuracy: 0.90740001
Current train mean: 0.9470000267028809
Current test mean: 0.8863999843597412
Trial 1 of 5
```

```
Iteration 1 - training accuracy: 0.84600002
Iteration 1 - testing accuracy: 0.84259999
Iteration 2 - training accuracy: 0.89960003
Iteration 2 - testing accuracy: 0.88200003
Iteration 3 - training accuracy: 0.91500002
Iteration 3 - testing accuracy: 0.89340001
Iteration 4 - training accuracy: 0.92479998
Iteration 4 - testing accuracy: 0.90039998
Iteration 5 - training accuracy: 0.93220001
Iteration 5 - testing accuracy: 0.90300000
Iteration 6 - training accuracy: 0.93640000
Iteration 6 - testing accuracy: 0.90399998
Iteration 7 - training accuracy: 0.94059998
Iteration 7 - testing accuracy: 0.90439999
Iteration 8 - training accuracy: 0.94239998
Iteration 8 - testing accuracy: 0.90640002
Iteration 9 - training accuracy: 0.94679999
Iteration 9 - testing accuracy: 0.90700001
Iteration 10 - training accuracy: 0.95060003
Iteration 10 - testing accuracy: 0.90759999
Current train mean: 0.9506000280380249
Current test mean: 0.8820000290870667
Trial 2 of 5
Iteration 1 - training accuracy: 0.83600003
Iteration 1 - testing accuracy: 0.83999997
Iteration 2 - training accuracy: 0.89780003
Iteration 2 - testing accuracy: 0.88020003
Iteration 3 - training accuracy: 0.91119999
Iteration 3 - testing accuracy: 0.89179999
Iteration 4 - training accuracy: 0.92119998
Iteration 4 - testing accuracy: 0.89560002
Iteration 5 - training accuracy: 0.92919999
Iteration 5 - testing accuracy: 0.89880002
Iteration 6 - training accuracy: 0.93400002
Iteration 6 - testing accuracy: 0.90219998
Iteration 7 - training accuracy: 0.93879998
Iteration 7 - testing accuracy: 0.90319997
Iteration 8 - training accuracy: 0.94260001
Iteration 8 - testing accuracy: 0.90200001
Iteration 9 - training accuracy: 0.94419998
Iteration 9 - testing accuracy: 0.90600002
Iteration 10 - training accuracy: 0.94760001
Iteration 10 - testing accuracy: 0.90640002
Current train mean: 0.9476000070571899
Current test mean: 0.8802000284194946
Trial 3 of 5
Iteration 1 - training accuracy: 0.84520000
Iteration 1 - testing accuracy: 0.84079999
Iteration 2 - training accuracy: 0.89520001
Iteration 2 - testing accuracy: 0.87860000
Iteration 3 - training accuracy: 0.91219997
Iteration 3 - testing accuracy: 0.89139998
Iteration 4 - training accuracy: 0.92259997
Iteration 4 - testing accuracy: 0.89620000
Iteration 5 - training accuracy: 0.93019998
Iteration 5 - testing accuracy: 0.89980000
Iteration 6 - training accuracy: 0.93379998
Iteration 6 - testing accuracy: 0.90340000
Iteration 7 - training accuracy: 0.93820000
Iteration 7 - testing accuracy: 0.90499997
Iteration 8 - training accuracy: 0.94180000
Iteration 8 - testing accuracy: 0.90619999
Iteration 9 - training accuracy: 0.94520003
Iteration 9 - testing accuracy: 0.90499997
Iteration 10 - training accuracy: 0.94819999
Iteration 10 - testing accuracy: 0.90420002
Current train mean: 0.948199987411499
Current test mean: 0.878600001335144
```

```
Trial 4 of 5
Iteration 1 - training accuracy: 0.83219999
Iteration 1 - testing accuracy: 0.83960003
Iteration 2 - training accuracy: 0.89600003
Iteration 2 - testing accuracy: 0.87919998
Iteration 3 - training accuracy: 0.91479999
Iteration 3 - testing accuracy: 0.89359999
Iteration 4 - training accuracy: 0.92379999
Iteration 4 - testing accuracy: 0.89899999
Iteration 5 - training accuracy: 0.92979997
Iteration 5 - testing accuracy: 0.90420002
Iteration 6 - training accuracy: 0.93339998
Iteration 6 - testing accuracy: 0.90520000
Iteration 7 - training accuracy: 0.93980002
Iteration 7 - testing accuracy: 0.90619999
Iteration 8 - training accuracy: 0.94319999
Iteration 8 - testing accuracy: 0.90560001
Iteration 9 - training accuracy: 0.94739997
Iteration 9 - testing accuracy: 0.90679997
Iteration 10 - training accuracy: 0.94919997
Iteration 10 - testing accuracy: 0.90480000
Current train mean: 0.9491999745368958
Current test mean: 0.8791999816894531
Hidden layer size: 50
Trial 0 of 5
Iteration 1 - training accuracy: 0.84520000
Iteration 1 - testing accuracy: 0.84759998
Iteration 2 - training accuracy: 0.89999998
Iteration 2 - testing accuracy: 0.88540000
Iteration 3 - training accuracy: 0.91600001
Iteration 3 - testing accuracy: 0.89539999
Iteration 4 - training accuracy: 0.92600000
Iteration 4 - testing accuracy: 0.90359998
Iteration 5 - training accuracy: 0.93220001
Iteration 5 - testing accuracy: 0.90340000
Iteration 6 - training accuracy: 0.93839997
Iteration 6 - testing accuracy: 0.90740001
Iteration 7 - training accuracy: 0.94220001
Iteration 7 - testing accuracy: 0.90799999
Iteration 8 - training accuracy: 0.94499999
Iteration 8 - testing accuracy: 0.90960002
Iteration 9 - training accuracy: 0.94880003
Iteration 9 - testing accuracy: 0.91020000
Iteration 10 - training accuracy: 0.95260000
Iteration 10 - testing accuracy: 0.90960002
Current train mean: 0.9526000022888184
Current test mean: 0.8853999972343445
Trial 1 of 5
Iteration 1 - training accuracy: 0.86220002
Iteration 1 - testing accuracy: 0.85140002
Iteration 2 - training accuracy: 0.89999998
Iteration 2 - testing accuracy: 0.88660002
Iteration 3 - training accuracy: 0.91520000
Iteration 3 - testing accuracy: 0.89420003
Iteration 4 - training accuracy: 0.92699999
Iteration 4 - testing accuracy: 0.89980000
Iteration 5 - training accuracy: 0.93220001
Iteration 5 - testing accuracy: 0.90520000
Iteration 6 - training accuracy: 0.93839997
Iteration 6 - testing accuracy: 0.90579998
Iteration 7 - training accuracy: 0.94480002
Iteration 7 - testing accuracy: 0.90799999
Iteration 8 - training accuracy: 0.94760001
Iteration 8 - testing accuracy: 0.91060001
Iteration 9 - training accuracy: 0.95240003
Iteration 9 - testing accuracy: 0.91079998
Iteration 10 - training accuracy: 0.95539999
Iteration 10 - testing accuracy: 0.91140002
```

```
Current train mean: 0.9553999900817871
Current test mean: 0.8866000175476074
Trial 2 of 5
Iteration 1 - training accuracy: 0.86100000
Iteration 1 - testing accuracy: 0.85100001
Iteration 2 - training accuracy: 0.90100002
Iteration 2 - testing accuracy: 0.88319999
Iteration 3 - training accuracy: 0.91320002
Iteration 3 - testing accuracy: 0.89359999
Iteration 4 - training accuracy: 0.92280000
Iteration 4 - testing accuracy: 0.89999998
Iteration 5 - training accuracy: 0.93260002
Iteration 5 - testing accuracy: 0.90319997
Iteration 6 - training accuracy: 0.93620002
Iteration 6 - testing accuracy: 0.90340000
Iteration 7 - training accuracy: 0.94199997
Iteration 7 - testing accuracy: 0.90560001
Iteration 8 - training accuracy: 0.94660002
Iteration 8 - testing accuracy: 0.90700001
Iteration 9 - training accuracy: 0.94959998
Iteration 9 - testing accuracy: 0.90880001
Iteration 10 - training accuracy: 0.95279998
Iteration 10 - testing accuracy: 0.90820003
Current train mean: 0.9527999758720398
Current test mean: 0.8831999897956848
Trial 3 of 5
Iteration 1 - training accuracy: 0.85839999
Iteration 1 - testing accuracy: 0.85479999
Iteration 2 - training accuracy: 0.90359998
Iteration 2 - testing accuracy: 0.88980001
Iteration 3 - training accuracy: 0.91860002
Iteration 3 - testing accuracy: 0.89639997
Iteration 4 - training accuracy: 0.92400002
Iteration 4 - testing accuracy: 0.90280002
Iteration 5 - training accuracy: 0.93159997
Iteration 5 - testing accuracy: 0.90539998
Iteration 6 - training accuracy: 0.93640000
Iteration 6 - testing accuracy: 0.90579998
Iteration 7 - training accuracy: 0.94199997
Iteration 7 - testing accuracy: 0.90740001
Iteration 8 - training accuracy: 0.94660002
Iteration 8 - testing accuracy: 0.90939999
Iteration 9 - training accuracy: 0.94940001
Iteration 9 - testing accuracy: 0.91240001
Iteration 10 - training accuracy: 0.95340002
Iteration 10 - testing accuracy: 0.91299999
Current train mean: 0.9534000158309937
Current test mean: 0.8898000121116638
Trial 4 of 5
Iteration 1 - training accuracy: 0.86180001
Iteration 1 - testing accuracy: 0.85519999
Iteration 2 - training accuracy: 0.90140003
Iteration 2 - testing accuracy: 0.88480002
Iteration 3 - training accuracy: 0.91700000
Iteration 3 - testing accuracy: 0.89160001
Iteration 4 - training accuracy: 0.92479998
Iteration 4 - testing accuracy: 0.89980000
Iteration 5 - training accuracy: 0.93339998
Iteration 5 - testing accuracy: 0.90319997
Iteration 6 - training accuracy: 0.93580002
Iteration 6 - testing accuracy: 0.90619999
Iteration 7 - training accuracy: 0.94040000
Iteration 7 - testing accuracy: 0.90579998
Iteration 8 - training accuracy: 0.94459999
Iteration 8 - testing accuracy: 0.90600002
Iteration 9 - training accuracy: 0.94980001
Iteration 9 - testing accuracy: 0.90859997
```

```
Iteration 10 - training accuracy: 0.95060003
Iteration 10 - testing accuracy: 0.91000003
Current train mean: 0.9506000280380249
Current test mean: 0.8848000168800354
Hidden layer size: 100
Trial 0 of 5
Iteration 1 - training accuracy: 0.85939997
Iteration 1 - testing accuracy: 0.85000002
Iteration 2 - training accuracy: 0.90319997
Iteration 2 - testing accuracy: 0.88440001
Iteration 3 - training accuracy: 0.91740000
Iteration 3 - testing accuracy: 0.89480001
Iteration 4 - training accuracy: 0.92619997
Iteration 4 - testing accuracy: 0.90179998
Iteration 5 - training accuracy: 0.93159997
Iteration 5 - testing accuracy: 0.90560001
Iteration 6 - training accuracy: 0.93820000
Iteration 6 - testing accuracy: 0.90640002
Iteration 7 - training accuracy: 0.94459999
Iteration 7 - testing accuracy: 0.91000003
Iteration 8 - training accuracy: 0.94859999
Iteration 8 - testing accuracy: 0.91060001
Iteration 9 - training accuracy: 0.95340002
Iteration 9 - testing accuracy: 0.91280001
Iteration 10 - training accuracy: 0.95639998
Iteration 10 - testing accuracy: 0.91259998
Current train mean: 0.9563999772071838
Current test mean: 0.8844000101089478
Trial 1 of 5
Iteration 1 - training accuracy: 0.86420000
Iteration 1 - testing accuracy: 0.85699999
Iteration 2 - training accuracy: 0.90259999
Iteration 2 - testing accuracy: 0.88679999
Iteration 3 - training accuracy: 0.91900003
Iteration 3 - testing accuracy: 0.89539999
Iteration 4 - training accuracy: 0.92820001
Iteration 4 - testing accuracy: 0.90120000
Iteration 5 - training accuracy: 0.93180001
Iteration 5 - testing accuracy: 0.90380001
Iteration 6 - training accuracy: 0.93820000
Iteration 6 - testing accuracy: 0.90679997
Iteration 7 - training accuracy: 0.94319999
Iteration 7 - testing accuracy: 0.90759999
Iteration 8 - training accuracy: 0.94779998
Iteration 8 - testing accuracy: 0.90920001
Iteration 9 - training accuracy: 0.95080000
Iteration 9 - testing accuracy: 0.91079998
Iteration 10 - training accuracy: 0.95520002
Iteration 10 - testing accuracy: 0.91380000
Current train mean: 0.9552000164985657
Current test mean: 0.8867999911308289
Trial 2 of 5
Iteration 1 - training accuracy: 0.86119998
Iteration 1 - testing accuracy: 0.85680002
Iteration 2 - training accuracy: 0.90160000
Iteration 2 - testing accuracy: 0.88679999
Iteration 3 - training accuracy: 0.91720003
Iteration 3 - testing accuracy: 0.89740002
Iteration 4 - training accuracy: 0.92460001
Iteration 4 - testing accuracy: 0.90319997
Iteration 5 - training accuracy: 0.93320000
Iteration 5 - testing accuracy: 0.90600002
Iteration 6 - training accuracy: 0.94040000
Iteration 6 - testing accuracy: 0.91060001
Iteration 7 - training accuracy: 0.94279999
Iteration 7 - testing accuracy: 0.91240001
Iteration 8 - training accuracy: 0.94919997
Iteration 8 - testing accuracy: 0.91399997
```

```
Iteration 9 - training accuracy: 0.95359999
Iteration 9 - testing accuracy: 0.91399997
Iteration 10 - training accuracy: 0.95719999
Iteration 10 - testing accuracy: 0.91520000
Current train mean: 0.9571999907493591
Current test mean: 0.8867999911308289
Trial 3 of 5
Iteration 1 - training accuracy: 0.86100000
Iteration 1 - testing accuracy: 0.85060000
Iteration 2 - training accuracy: 0.90359998
Iteration 2 - testing accuracy: 0.88900000
Iteration 3 - training accuracy: 0.91780001
Iteration 3 - testing accuracy: 0.89660001
Iteration 4 - training accuracy: 0.92659998
Iteration 4 - testing accuracy: 0.90300000
Iteration 5 - training accuracy: 0.93140000
Iteration 5 - testing accuracy: 0.90600002
Iteration 6 - training accuracy: 0.93660003
Iteration 6 - testing accuracy: 0.90700001
Iteration 7 - training accuracy: 0.94220001
Iteration 7 - testing accuracy: 0.90619999
Iteration 8 - training accuracy: 0.94620001
Iteration 8 - testing accuracy: 0.91020000
Iteration 9 - training accuracy: 0.95039999
Iteration 9 - testing accuracy: 0.91020000
Iteration 10 - training accuracy: 0.95440000
Iteration 10 - testing accuracy: 0.91240001
Current train mean: 0.9544000029563904
Current test mean: 0.8889999985694885
Trial 4 of 5
Iteration 1 - training accuracy: 0.86519998
Iteration 1 - testing accuracy: 0.85680002
Iteration 2 - training accuracy: 0.90200001
Iteration 2 - testing accuracy: 0.88440001
Iteration 3 - training accuracy: 0.91799998
Iteration 3 - testing accuracy: 0.89480001
Iteration 4 - training accuracy: 0.92760003
Iteration 4 - testing accuracy: 0.89960003
Iteration 5 - training accuracy: 0.93199998
Iteration 5 - testing accuracy: 0.90280002
Iteration 6 - training accuracy: 0.93779999
Iteration 6 - testing accuracy: 0.90579998
Iteration 7 - training accuracy: 0.94559997
Iteration 7 - testing accuracy: 0.90600002
Iteration 8 - training accuracy: 0.94779998
Iteration 8 - testing accuracy: 0.90700001
Iteration 9 - training accuracy: 0.95099998
Iteration 9 - testing accuracy: 0.90979999
Iteration 10 - training accuracy: 0.95400000
Iteration 10 - testing accuracy: 0.91100001
Current train mean: 0.9539999961853027
Current test mean: 0.8844000101089478
```

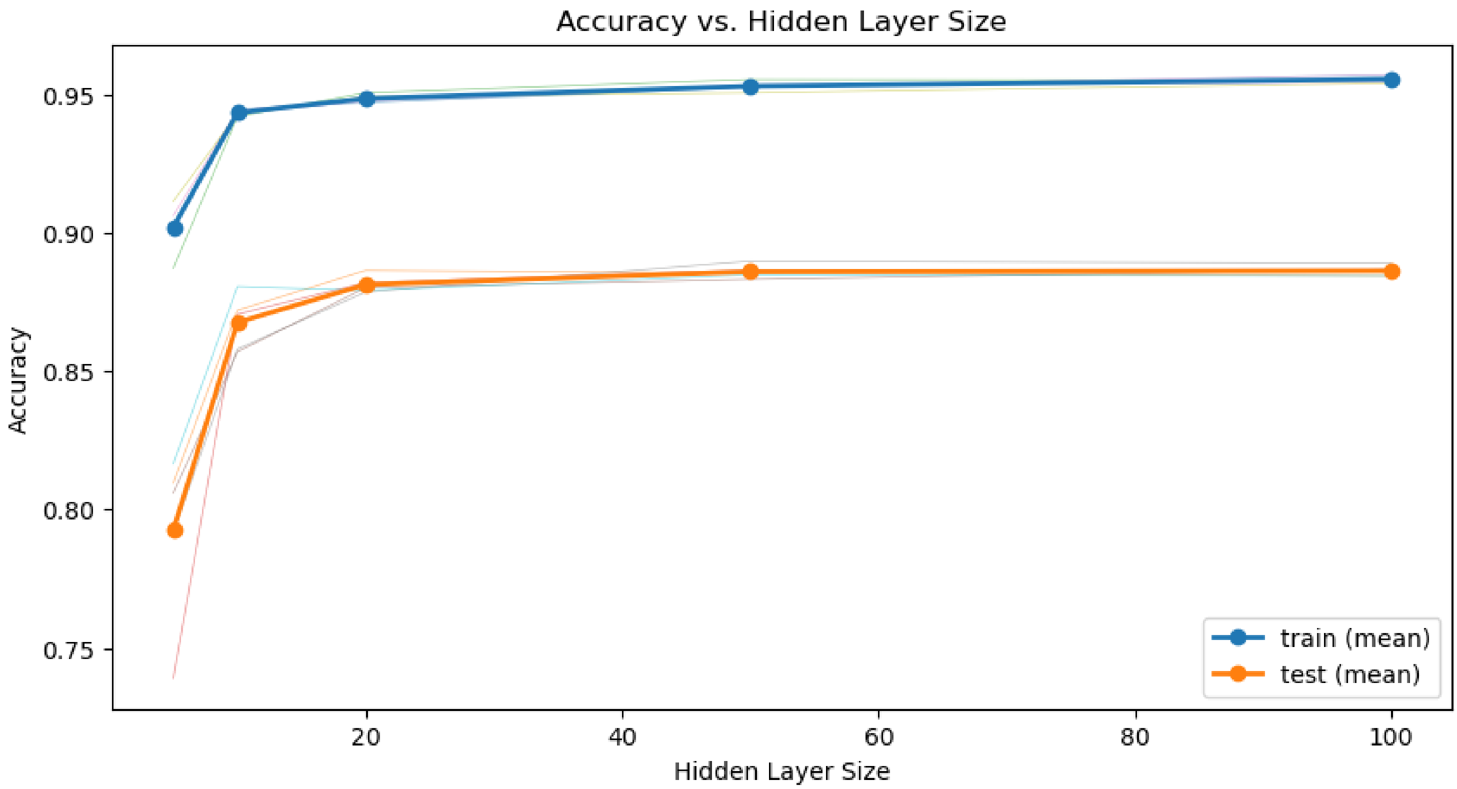
In [34]:

```
# plot accuracy vs. hidden layer size
fig, ax = plt.subplots(figsize=(10,5))

# plot each trial
for i in range(5):
    ax.plot(hidden_sizes, train_3c[:,i], linewidth=0.5, alpha=0.5)
    ax.plot(hidden_sizes, test_3c[:,i], linewidth=0.5, alpha=0.5)

# plot mean of all trials
ax.plot(hidden_sizes, np.mean(train_3c, axis=1), 'o-', linewidth=2, label='train (mean)')
ax.plot(hidden_sizes, np.mean(test_3c, axis=1), 'o-', linewidth=2, label='test (mean)')
ax.set_title('Accuracy vs. Hidden Layer Size')
```

```python
ax.set_xlabel('Hidden Layer Size')
ax.set_ylabel('Accuracy')
ax.legend()
```

Out[34]:

```
<matplotlib.legend.Legend at 0x123c05810>
```



In [47]:

```python
print(f"Max training accuracy: {max(np.mean(train_3c, axis=1)):.4f}")
print(f"Max testing accuracy: {max(np.mean(test_3c, axis=1)):.4f}")
```

```
Max training accuracy: 0.9554
Max testing accuracy: 0.8863
```

3. d) [2 marks] Now we will add a convolution layer to our network. The following network adds two convolution layers before two normal neural network layers.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 12, kernel_size=5)   # set the size of the convolution to 5x5, and have 12 of them
        self.conv2 = nn.Conv2d(12, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))   # make sure to do max pooling after the convolution layers
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)
```

The following code can be used to plot the learned features in the first layer:

```
plt.figure(figsize=(12,5))
for i in range(12):
    plt.subplot(3, 4, i+1)
    plt.imshow(network.conv1.weight[i][0].detach().numpy(), cmap='gray', interpolatio
n='nearest')
    plt.xticks([])
    plt.yticks([])
plt.show()
```

- **Train the model through 40 iterations and generate a plot of training and testing accuracy over time.**
- **Does this perform better or worse than the previous models in this assignment?**
- **What advantages and disadvantages do you see with this approach (in comparison to the previous parts of the assignment)?**
- **Plot the features learned by the first convolution layer. How do they compare to real features detected in the V1 area of the brain?**

In [36]:

```
# define convolutional nnet class:
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 12, kernel_size=5)  # set the size of the convolution to 
5x5, and have 12 of them
        self.conv2 = nn.Conv2d(12, 20, kernel_size=5)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))  # make sure to do max pooling after the 
convolution layers
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)
```

In [37]:

```
# train for 40 iterations, plot train and test accuracy over time:
network = ConvNet()
optimizer = torch.optim.SGD(network.parameters(), lr=LR, momentum=MOMENTUM)
train_3d = []
test_3d = []

for i in range(40):
    continue_training(train_3d, test_3d)
```

```
/var/folders/rx/5_fd7v5s5dbc3yr3cx7bw9m40000gn/T/ipykernel_95115/1114110666.py:16: UserWarni
ng: Implicit dimension choice for log_softmax has been deprecated. Change the call to includ
e dim=X as an argument.
  return F.log_softmax(x)
```

```
Iteration 1 - training accuracy: 0.51740003
Iteration 1 - testing accuracy: 0.64319998
Iteration 2 - training accuracy: 0.91579998
Iteration 2 - testing accuracy: 0.90480000
Iteration 3 - training accuracy: 0.95300001
Iteration 3 - testing accuracy: 0.93919998
Iteration 4 - training accuracy: 0.96700001
Iteration 4 - testing accuracy: 0.94279999
Iteration 5 - training accuracy: 0.97700000
Iteration 5 - testing accuracy: 0.94919997
```

```
Iteration 6 - training accuracy: 0.98159999
Iteration 6 - testing accuracy: 0.95420003
Iteration 7 - training accuracy: 0.98740000
Iteration 7 - testing accuracy: 0.96020001
Iteration 8 - training accuracy: 0.99159998
Iteration 8 - testing accuracy: 0.95999998
Iteration 9 - training accuracy: 0.99320000
Iteration 9 - testing accuracy: 0.95899999
Iteration 10 - training accuracy: 0.99400002
Iteration 10 - testing accuracy: 0.95880002
Iteration 11 - training accuracy: 0.99720001
Iteration 11 - testing accuracy: 0.95999998
Iteration 12 - training accuracy: 0.99720001
Iteration 12 - testing accuracy: 0.96120000
Iteration 13 - training accuracy: 0.99940002
Iteration 13 - testing accuracy: 0.96420002
Iteration 14 - training accuracy: 0.99900001
Iteration 14 - testing accuracy: 0.96359998
Iteration 15 - training accuracy: 0.99980003
Iteration 15 - testing accuracy: 0.96359998
Iteration 16 - training accuracy: 0.99959999
Iteration 16 - testing accuracy: 0.96420002
Iteration 17 - training accuracy: 1.00000000
Iteration 17 - testing accuracy: 0.96359998
Iteration 18 - training accuracy: 1.00000000
Iteration 18 - testing accuracy: 0.96539998
Iteration 19 - training accuracy: 1.00000000
Iteration 19 - testing accuracy: 0.96439999
Iteration 20 - training accuracy: 1.00000000
Iteration 20 - testing accuracy: 0.96520001
Iteration 21 - training accuracy: 1.00000000
Iteration 21 - testing accuracy: 0.96579999
Iteration 22 - training accuracy: 1.00000000
Iteration 22 - testing accuracy: 0.96480000
Iteration 23 - training accuracy: 1.00000000
Iteration 23 - testing accuracy: 0.96460003
Iteration 24 - training accuracy: 1.00000000
Iteration 24 - testing accuracy: 0.96539998
Iteration 25 - training accuracy: 1.00000000
Iteration 25 - testing accuracy: 0.96600002
Iteration 26 - training accuracy: 1.00000000
Iteration 26 - testing accuracy: 0.96539998
Iteration 27 - training accuracy: 1.00000000
Iteration 27 - testing accuracy: 0.96460003
Iteration 28 - training accuracy: 1.00000000
Iteration 28 - testing accuracy: 0.96480000
Iteration 29 - training accuracy: 1.00000000
Iteration 29 - testing accuracy: 0.96520001
Iteration 30 - training accuracy: 1.00000000
Iteration 30 - testing accuracy: 0.96560001
Iteration 31 - training accuracy: 1.00000000
Iteration 31 - testing accuracy: 0.96560001
Iteration 32 - training accuracy: 1.00000000
Iteration 32 - testing accuracy: 0.96499997
Iteration 33 - training accuracy: 1.00000000
Iteration 33 - testing accuracy: 0.96539998
Iteration 34 - training accuracy: 1.00000000
Iteration 34 - testing accuracy: 0.96480000
Iteration 35 - training accuracy: 1.00000000
Iteration 35 - testing accuracy: 0.96600002
Iteration 36 - training accuracy: 1.00000000
Iteration 36 - testing accuracy: 0.96539998
Iteration 37 - training accuracy: 1.00000000
Iteration 37 - testing accuracy: 0.96520001
Iteration 38 - training accuracy: 1.00000000
Iteration 38 - testing accuracy: 0.96579999
Iteration 39 - training accuracy: 1.00000000
Iteration 39 - testing accuracy: 0.96560001
```

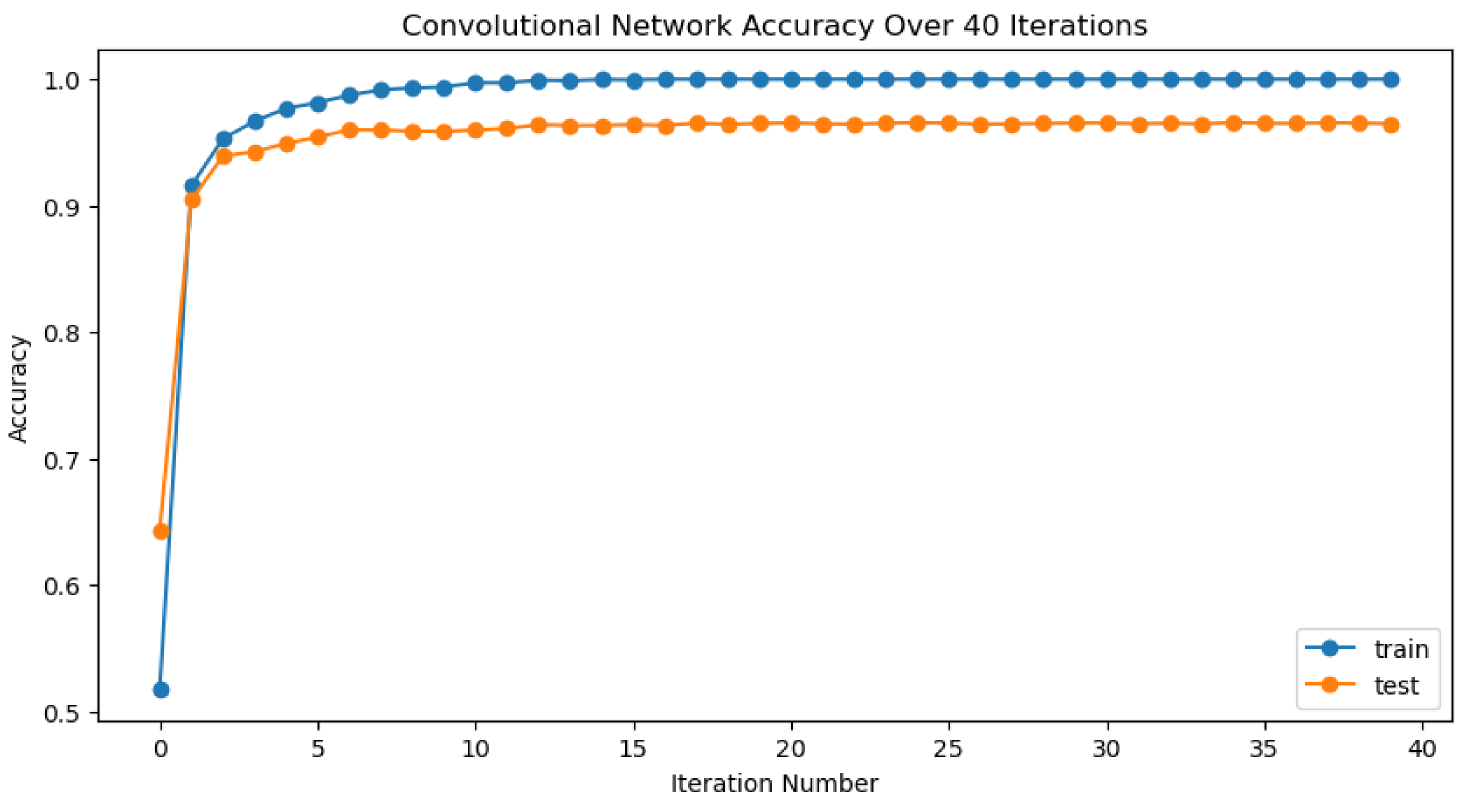
```
Iteration 39 - testing accuracy: 0.96300001
Iteration 40 - training accuracy: 1.00000000
Iteration 40 - testing accuracy: 0.96499997
```

In [38]:

```python
# plot train and test accuracy vs. iteration:
fig, ax = plt.subplots(figsize=(10,5))
ax.plot(train_3d, 'o-', label='train')
ax.plot(test_3d, 'o-', label='test')
ax.set_title('Convolutional Network Accuracy Over 40 Iterations')
ax.set_xlabel('Iteration Number')
ax.set_ylabel('Accuracy')
ax.legend()
```

Out[38]:

```
&lt;matplotlib.legend.Legend at 0x12445d810&gt;
```



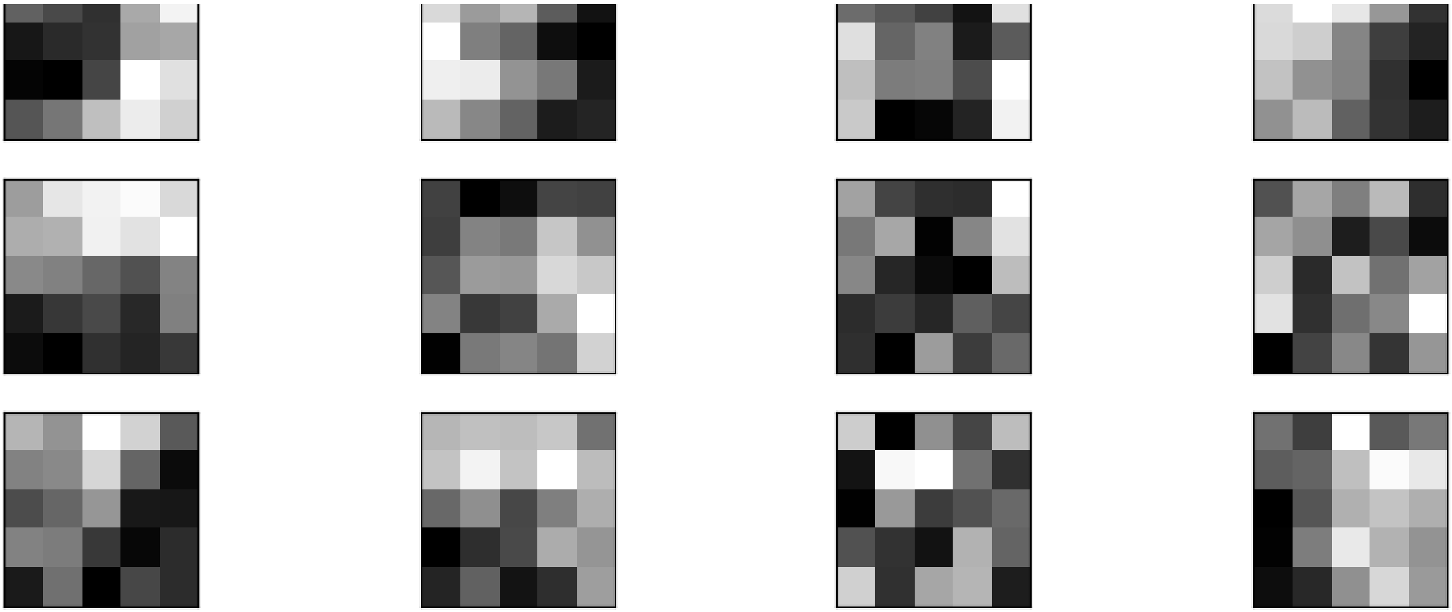
In [39]:

```python
print(f"Max training accuracy: {max(train_3d):.4f}")
print(f"Max testing accuracy: {max(test_3d):.4f}")
```

```
Max training accuracy: 1.0000
Max testing accuracy: 0.9660
```

In [40]:

```python
# plot the features learned by the first convolution layer:
plt.figure(figsize=(12,5))
for i in range(12):
    plt.subplot(3, 4, i+1)
    plt.imshow(network.conv1.weight[i][0].detach().numpy(), cmap='gray', interpolation='nea
rest')
    plt.xticks([])
    plt.yticks([])
plt.show()
```



Discussion Questions:

- Does this perform better or worse than the previous models in this assignment?

From the result above, this model performs much better than previous models from earlier in the assignment, with a test accuracy of about 96% (compared to ~ 90% in previous attempts).

- What advantages and disadvantages do you see with this approach (in comparison to the previous parts of the assignment)?

The obvious advantage of this approach is that it can achieve a much higher accuracy than before, so it does better at the task that we want to accomplish. This method also seems to work well on a much smaller dataset compared to regression (5,000 vs 60,000 samples?) This means that this method would be more applicable to tasks that have comparatively limited data available for training.

One potential disadvantage of this method is that it is slower and requires more resources compared to simpler networks. For this relatively trivial task, however, this not a major issue—a comparable accuracy can be achieved after training for only 10 iterations (rather than 40). For more complex tasks, this would not be the case, and longer training would be required. The complexity of convolutional networks for more difficult tasks is probably much greater, as well as computational resource requirements.

Another limitation is that the CNN does not offer transparency into the features that the model is using. In the visualization above, for example, the "features" have been identified are not clearly linked to the inputs, and more layers complicate this further. This makes for a very "black box" kind of solution, which cannot be easily interpreted by a human. In a case where the model is unsuccessful, this can make it difficult to understand what is going wrong.

- Plot the features learned by the first convolution layer. How do they compare to real features detected in the V1 area of the brain?

The features learned by the first layer are very different from the features detected in V1—there are no clear 'bars' of different orientations that we can see in these features, or other visual primitives that are used in the primate visual system. These features also look different each time the network is trained, which indicates that there is nothing "special" about these particular features.

BONUS [1 mark] Try to improve the neural network. You want to get the best testing accuracy you can. Try at least two different approaches and report your results.

1. Try a different optimizer

In [41]:

In [41]:

```
# try ADAM optimizer:
fig, ax = plt.subplots(figsize=(10,5))
train_adam = np.zeros(10)
test_adam = np.zeros(10)

for i in range(5):
    accuracy_train = []
    accuracy_test = []
    network = Net()
    optimizer = torch.optim.Adam(network.parameters(), lr=LR)
    for j in range(10):
        continue_training(accuracy_train, accuracy_test)
    ax.plot(accuracy_train, linewidth=0.5, alpha=0.5)
    ax.plot(accuracy_test, linewidth=0.5, alpha=0.5)
    train_adam += np.array(accuracy_train)
    test_adam += np.array(accuracy_test)
train_adam /= 5
test_adam /= 5

ax.plot(train_adam, 'o-', linewidth=2, label='train (mean)')
ax.plot(test_adam, 'o-', linewidth=2, label='test (mean)')
ax.set_title('Accuracy using Adam Optimizer')
ax.set_xlabel('Iteration Number')
ax.set_ylabel('Accuracy')
ax.legend()
```

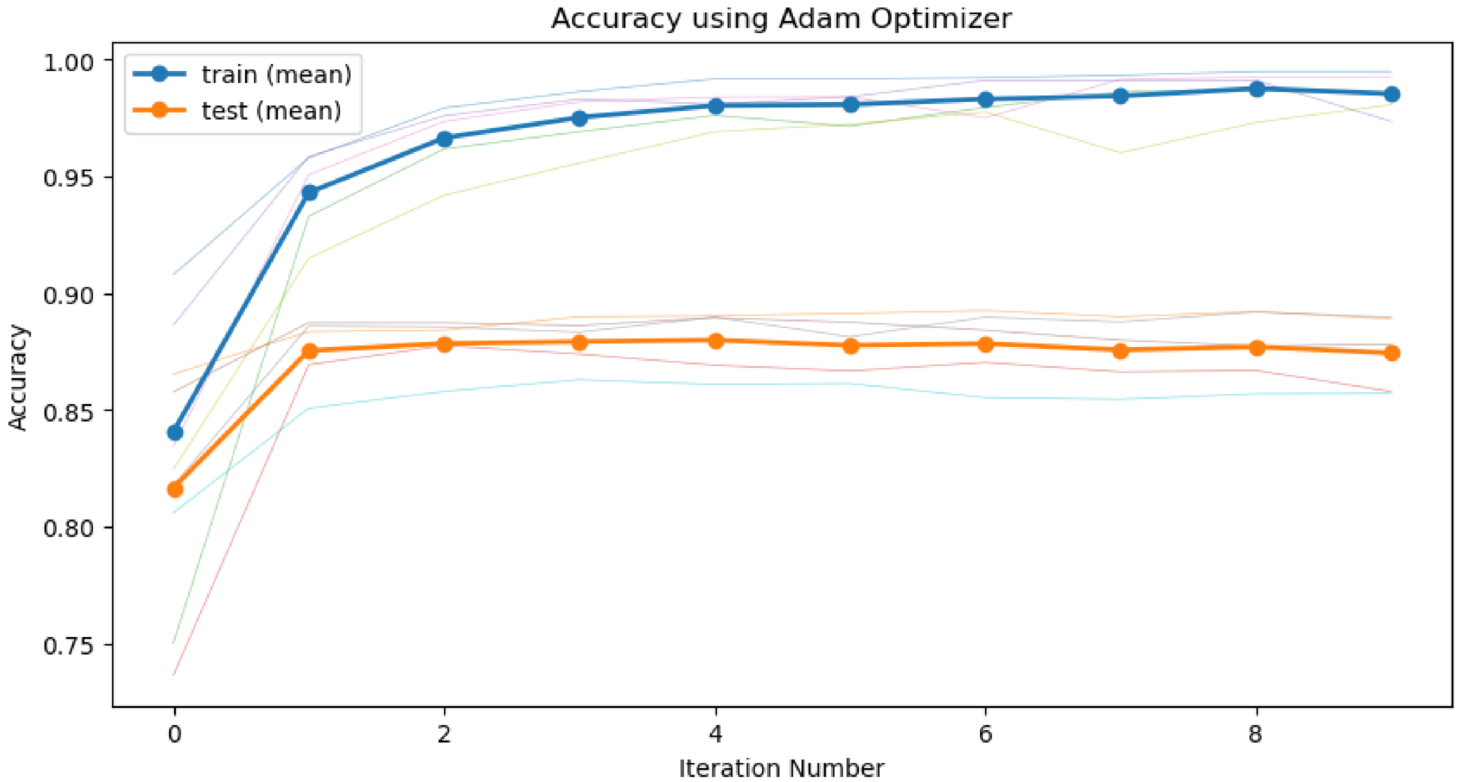
```
/var/folders/rx/5_fd7v5s5dbc3yr3cx7bw9m40000gn/T/ipykernel_95115/3140384043.py:17: UserWarni
ng: Implicit dimension choice for log_softmax has been deprecated. Change the call to includ
e dim=X as an argument.
  return F.log_softmax(x)         # apply a softmax function as we just want one large outpu
t indicating category
```

```
Iteration 1 - training accuracy: 0.90799999
Iteration 1 - testing accuracy: 0.86540002
Iteration 2 - training accuracy: 0.95800000
Iteration 2 - testing accuracy: 0.88360000
Iteration 3 - training accuracy: 0.97939998
Iteration 3 - testing accuracy: 0.88419998
Iteration 4 - training accuracy: 0.98640001
Iteration 4 - testing accuracy: 0.88999999
Iteration 5 - training accuracy: 0.99180001
Iteration 5 - testing accuracy: 0.89039999
Iteration 6 - training accuracy: 0.99180001
Iteration 6 - testing accuracy: 0.89139998
Iteration 7 - training accuracy: 0.99239999
Iteration 7 - testing accuracy: 0.89260000
Iteration 8 - training accuracy: 0.99339998
Iteration 8 - testing accuracy: 0.88999999
Iteration 9 - training accuracy: 0.99500000
Iteration 9 - testing accuracy: 0.89219999
Iteration 10 - training accuracy: 0.99500000
Iteration 10 - testing accuracy: 0.88900000
Iteration 1 - training accuracy: 0.75019997
Iteration 1 - testing accuracy: 0.73640001
Iteration 2 - training accuracy: 0.93300003
Iteration 2 - testing accuracy: 0.86940002
Iteration 3 - training accuracy: 0.96179998
Iteration 3 - testing accuracy: 0.87760001
Iteration 4 - training accuracy: 0.96920002
Iteration 4 - testing accuracy: 0.87400001
Iteration 5 - training accuracy: 0.97619998
Iteration 5 - testing accuracy: 0.86919999
Iteration 6 - training accuracy: 0.97160000
Iteration 6 - testing accuracy: 0.86680001
Iteration 7 - training accuracy: 0.97960001
Iteration 7 - testing accuracy: 0.87040001
Iteration 8 - training accuracy: 0.98640001
```

```
Iteration 8 - testing accuracy: 0.86640000
Iteration 9 - training accuracy: 0.98680001
Iteration 9 - testing accuracy: 0.86699998
Iteration 10 - training accuracy: 0.98540002
Iteration 10 - testing accuracy: 0.85799998
Iteration 1 - training accuracy: 0.88639998
Iteration 1 - testing accuracy: 0.85780001
Iteration 2 - training accuracy: 0.95859998
Iteration 2 - testing accuracy: 0.88739997
Iteration 3 - training accuracy: 0.97600001
Iteration 3 - testing accuracy: 0.88739997
Iteration 4 - training accuracy: 0.98320001
Iteration 4 - testing accuracy: 0.88620001
Iteration 5 - training accuracy: 0.98100001
Iteration 5 - testing accuracy: 0.88959998
Iteration 6 - training accuracy: 0.98420000
Iteration 6 - testing accuracy: 0.88760000
Iteration 7 - training accuracy: 0.99119997
Iteration 7 - testing accuracy: 0.88419998
Iteration 8 - training accuracy: 0.99100000
Iteration 8 - testing accuracy: 0.88000000
Iteration 9 - training accuracy: 0.99119997
Iteration 9 - testing accuracy: 0.87739998
Iteration 10 - training accuracy: 0.97359997
Iteration 10 - testing accuracy: 0.87819999
Iteration 1 - training accuracy: 0.83440000
Iteration 1 - testing accuracy: 0.81779999
Iteration 2 - training accuracy: 0.95080000
Iteration 2 - testing accuracy: 0.88599998
Iteration 3 - training accuracy: 0.97359997
Iteration 3 - testing accuracy: 0.88559997
Iteration 4 - training accuracy: 0.98199999
Iteration 4 - testing accuracy: 0.88340002
Iteration 5 - training accuracy: 0.98400003
Iteration 5 - testing accuracy: 0.88980001
Iteration 6 - training accuracy: 0.98420000
Iteration 6 - testing accuracy: 0.88139999
Iteration 7 - training accuracy: 0.97520000
Iteration 7 - testing accuracy: 0.88980001
Iteration 8 - training accuracy: 0.99159998
Iteration 8 - testing accuracy: 0.88779998
Iteration 9 - training accuracy: 0.99260002
Iteration 9 - testing accuracy: 0.89200002
Iteration 10 - training accuracy: 0.99280000
Iteration 10 - testing accuracy: 0.88980001
Iteration 1 - training accuracy: 0.82459998
Iteration 1 - testing accuracy: 0.80599999
Iteration 2 - training accuracy: 0.91500002
Iteration 2 - testing accuracy: 0.85079998
Iteration 3 - training accuracy: 0.94199997
Iteration 3 - testing accuracy: 0.85799998
Iteration 4 - training accuracy: 0.95580000
Iteration 4 - testing accuracy: 0.86299998
Iteration 5 - training accuracy: 0.96920002
Iteration 5 - testing accuracy: 0.86100000
Iteration 6 - training accuracy: 0.97240001
Iteration 6 - testing accuracy: 0.86140001
Iteration 7 - training accuracy: 0.97759998
Iteration 7 - testing accuracy: 0.85540003
Iteration 8 - training accuracy: 0.96020001
Iteration 8 - testing accuracy: 0.85460001
Iteration 9 - training accuracy: 0.97320002
Iteration 9 - testing accuracy: 0.85699999
Iteration 10 - training accuracy: 0.98079997
Iteration 10 - testing accuracy: 0.85720003
```

Out[41]:

```
<matplotlib.legend.Legend at 0x12420b0d0>
```



In [42]:

```
print(f"Max train accuracy: {max(train_adam)}")
print(f"Max test accuracy: {max(test_adam)}")
```

```
Max train accuracy: 0.9877600073814392
Max test accuracy: 0.8799999952316284
```

The results above show a plot that is much flatter compared to the one in 3b (where we produced the same plot using the SGD optimizer). A testing accuracy of about 88-89% is achieved, which is good, but not as good as the accuracy of 3b (> 90%) or 3d (> 96%). From some quick research, this is consistent with other experimental results using the Adam optimizer – it tends to converge quickly, but generalizes less well compared to SGD [5].

Reference

[5] P. Zhou, J. Feng, C. Ma, C. Xiong, S. Hoi, and E. Weinan, "Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning," *Conference on Neural Information Processing Systems (NeurIPS 2020)*, Vancouver, Canada. Available: https://proceedings.neurips.cc/paper/2020/file/f3f27a324736617f20abbf2ffd806f6d-Paper.pdf

2. Try different kernel sizes

In [43]:

```
# define convolutional nnet class:
class ConvNet2(nn.Module):
    def __init__(self, k=5, p=0):
        super(ConvNet2, self).__init__()
        self.conv1 = nn.Conv2d(1, 12, kernel_size=k, padding=p)   # change the padding to ac
count for the kernel dimension
        self.conv2 = nn.Conv2d(12, 20, kernel_size=k, padding=p) # this results in the same
output dimensions as before
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
```

```
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))  # make sure to do max pooling after the 
convolution layers
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x)
```

In [45]:

```
# train for 40 iterations, plot train and test accuracy over time:
kernel_sizes = [5, 7, 9]
padding = [0, 1, 2]
train = np.zeros((3, 20))
test = np.zeros((3, 20))

for k in range(len(kernel_sizes)):
    print(f'Kernel size: {kernel_sizes[k]}, Padding: {padding[k]}')
    temp_train = []
    temp_test = []
    network = ConvNet2(k=kernel_sizes[k], p=padding[k])
    optimizer = torch.optim.SGD(network.parameters(), lr=LR, momentum=MOMENTUM)
    for i in range(20):
        continue_training(temp_train, temp_test)

    train[k] += np.array(temp_train)
    test[k] += np.array(temp_test)
```

```
Kernel size: 5, Padding: 0
```

```
/var/folders/rx/5_fd7v5s5dbc3yr3cx7bw9m40000gn/T/ipykernel_95115/401323374.py:16: UserWarnin
g: Implicit dimension choice for log_softmax has been deprecated. Change the call to include
dim=X as an argument.
  return F.log_softmax(x)
```

```
Iteration 1 - training accuracy: 0.85020000
Iteration 1 - testing accuracy: 0.73879999
Iteration 2 - training accuracy: 0.94599998
Iteration 2 - testing accuracy: 0.93320000
Iteration 3 - training accuracy: 0.96259999
Iteration 3 - testing accuracy: 0.94760001
Iteration 4 - training accuracy: 0.97640002
Iteration 4 - testing accuracy: 0.95260000
Iteration 5 - training accuracy: 0.98339999
Iteration 5 - testing accuracy: 0.95779997
Iteration 6 - training accuracy: 0.98839998
Iteration 6 - testing accuracy: 0.96139997
Iteration 7 - training accuracy: 0.99019998
Iteration 7 - testing accuracy: 0.95859998
Iteration 8 - training accuracy: 0.99460000
Iteration 8 - testing accuracy: 0.96100003
Iteration 9 - training accuracy: 0.99360001
Iteration 9 - testing accuracy: 0.96499997
Iteration 10 - training accuracy: 0.99760002
Iteration 10 - testing accuracy: 0.96079999
Iteration 11 - training accuracy: 0.99800003
Iteration 11 - testing accuracy: 0.96520001
Iteration 12 - training accuracy: 0.99699998
Iteration 12 - testing accuracy: 0.96179998
Iteration 13 - training accuracy: 0.99879998
Iteration 13 - testing accuracy: 0.96499997
Iteration 14 - training accuracy: 0.99919999
Iteration 14 - testing accuracy: 0.96679997
Iteration 15 - training accuracy: 0.99980003
Iteration 15 - testing accuracy: 0.96640003
Iteration 16 - training accuracy: 0.99980003
Iteration 16 - testing accuracy: 0.96619999
```

```
Iteration 17 - training accuracy: 1.00000000
Iteration 17 - testing accuracy: 0.96660000
Iteration 18 - training accuracy: 1.00000000
Iteration 18 - testing accuracy: 0.96660000
Iteration 19 - training accuracy: 1.00000000
Iteration 19 - testing accuracy: 0.96619999
Iteration 20 - training accuracy: 1.00000000
Iteration 20 - testing accuracy: 0.96579999
Kernel size: 7, Padding: 1
Iteration 1 - training accuracy: 0.68159997
Iteration 1 - testing accuracy: 0.82139999
Iteration 2 - training accuracy: 0.93540001
Iteration 2 - testing accuracy: 0.93099999
Iteration 3 - training accuracy: 0.96319997
Iteration 3 - testing accuracy: 0.94620001
Iteration 4 - training accuracy: 0.97299999
Iteration 4 - testing accuracy: 0.94840002
Iteration 5 - training accuracy: 0.98040003
Iteration 5 - testing accuracy: 0.95700002
Iteration 6 - training accuracy: 0.98559999
Iteration 6 - testing accuracy: 0.96120000
Iteration 7 - training accuracy: 0.99080002
Iteration 7 - testing accuracy: 0.95880002
Iteration 8 - training accuracy: 0.99220002
Iteration 8 - testing accuracy: 0.95959997
Iteration 9 - training accuracy: 0.99400002
Iteration 9 - testing accuracy: 0.96319997
Iteration 10 - training accuracy: 0.99680001
Iteration 10 - testing accuracy: 0.96259999
Iteration 11 - training accuracy: 0.99839997
Iteration 11 - testing accuracy: 0.96399999
Iteration 12 - training accuracy: 0.99879998
Iteration 12 - testing accuracy: 0.96319997
Iteration 13 - training accuracy: 0.99940002
Iteration 13 - testing accuracy: 0.96359998
Iteration 14 - training accuracy: 0.99980003
Iteration 14 - testing accuracy: 0.96300000
Iteration 15 - training accuracy: 0.99959999
Iteration 15 - testing accuracy: 0.96460003
Iteration 16 - training accuracy: 1.00000000
Iteration 16 - testing accuracy: 0.96480000
Iteration 17 - training accuracy: 1.00000000
Iteration 17 - testing accuracy: 0.96359998
Iteration 18 - training accuracy: 1.00000000
Iteration 18 - testing accuracy: 0.96420002
Iteration 19 - training accuracy: 1.00000000
Iteration 19 - testing accuracy: 0.96399999
Iteration 20 - training accuracy: 1.00000000
Iteration 20 - testing accuracy: 0.96300000
Kernel size: 9, Padding: 2
Iteration 1 - training accuracy: 0.82679999
Iteration 1 - testing accuracy: 0.82539999
Iteration 2 - training accuracy: 0.94080001
Iteration 2 - testing accuracy: 0.92299998
Iteration 3 - training accuracy: 0.96719998
Iteration 3 - testing accuracy: 0.94720000
Iteration 4 - training accuracy: 0.97280002
Iteration 4 - testing accuracy: 0.94959998
Iteration 5 - training accuracy: 0.97979999
Iteration 5 - testing accuracy: 0.95300001
Iteration 6 - training accuracy: 0.99040002
Iteration 6 - testing accuracy: 0.95819998
Iteration 7 - training accuracy: 0.99419999
Iteration 7 - testing accuracy: 0.96100003
Iteration 8 - training accuracy: 0.99680001
Iteration 8 - testing accuracy: 0.95840001
Iteration 9 - training accuracy: 0.99820000
Iteration 9 - testing accuracy: 0.95760000
```

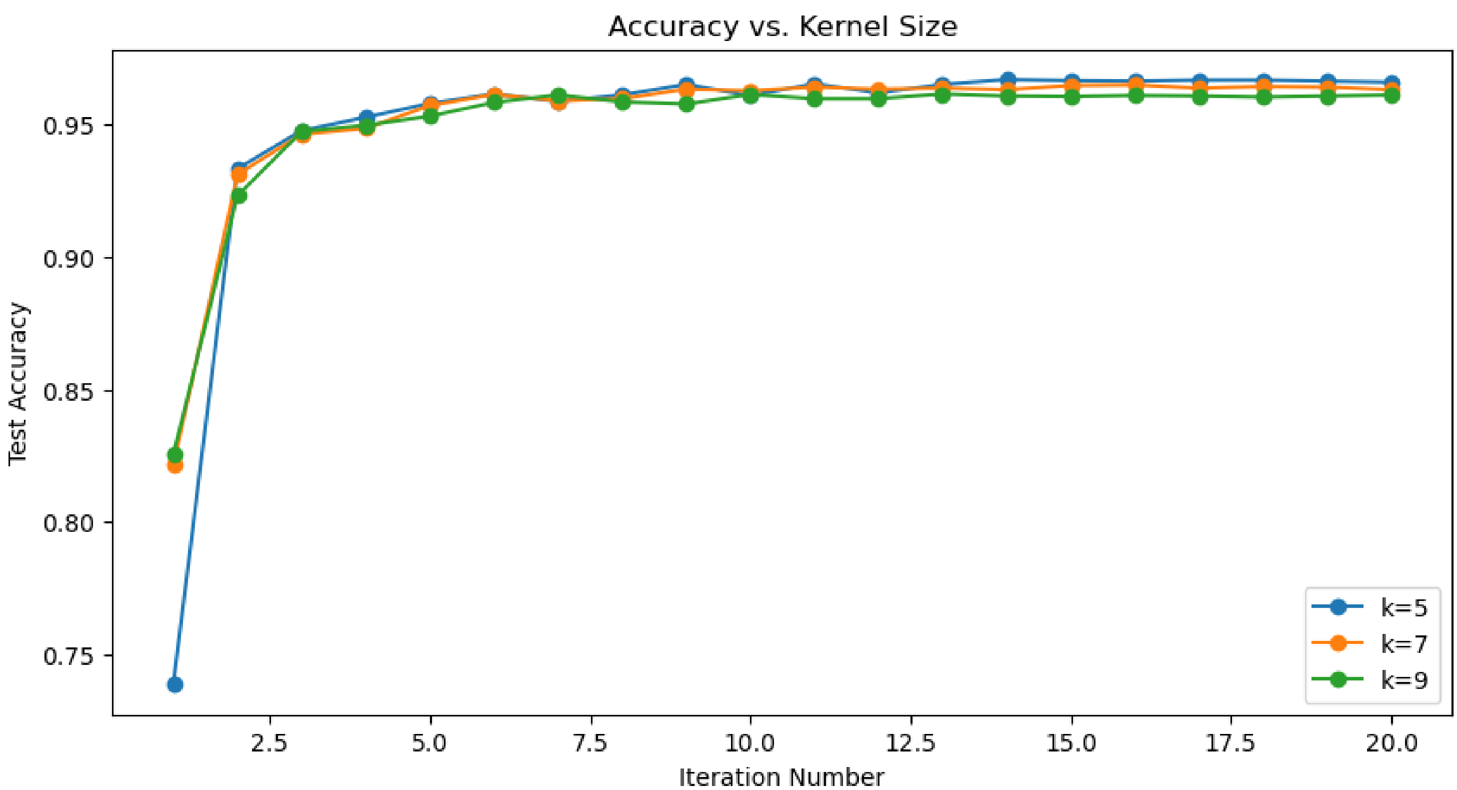
```
Iteration 10 - training accuracy: 0.99919999
Iteration 10 - testing accuracy: 0.96120000
Iteration 11 - training accuracy: 0.99940002
Iteration 11 - testing accuracy: 0.95959997
Iteration 12 - training accuracy: 0.99980003
Iteration 12 - testing accuracy: 0.95959997
Iteration 13 - training accuracy: 0.99980003
Iteration 13 - testing accuracy: 0.96139997
Iteration 14 - training accuracy: 0.99980003
Iteration 14 - testing accuracy: 0.96060002
Iteration 15 - training accuracy: 1.00000000
Iteration 15 - testing accuracy: 0.96039999
Iteration 16 - training accuracy: 1.00000000
Iteration 16 - testing accuracy: 0.96079999
Iteration 17 - training accuracy: 1.00000000
Iteration 17 - testing accuracy: 0.96060002
Iteration 18 - training accuracy: 1.00000000
Iteration 18 - testing accuracy: 0.96020001
Iteration 19 - training accuracy: 1.00000000
Iteration 19 - testing accuracy: 0.96060002
Iteration 20 - training accuracy: 1.00000000
Iteration 20 - testing accuracy: 0.96100003
```

In [46]:

```python
# plot train and test accuracy vs. iteration:
fig, ax = plt.subplots(figsize=(10,5))
iteration = np.linspace(1, 20, 20)
ax.plot(iteration, test[0], 'o-', label='k=5')
ax.plot(iteration, test[1], 'o-', label='k=7')
ax.plot(iteration, test[2], 'o-', label='k=9')
ax.set_title('Accuracy vs. Kernel Size')
ax.set_xlabel('Iteration Number')
ax.set_ylabel('Test Accuracy')
ax.legend()
```

Out[46]:

```
&lt;matplotlib.legend.Legend at 0x123f7eb50&gt;
```



The result here shows that the 5x5 kernel performs the best, and that the 7x7 and 9x9 kernels perform slightly worse. This experiment was run several times to compare results across runs, and there was no clear winner between the 5x5/7x7 kernels (likely because of some randomness in the training process and because they are both near the optimal size). I would guess that this optimal kernel size would change for different vision tasks, and that this would require some tuning depending on the type of data that is being processed. A kernel that's too small is inefficient and would need a very large network of neurons to perform well, but a kernel that's too large may obscure too many of the fine details of the image, leading to poorer performance.