EE 449
Homework 1 - Training Artificial Neural Network

Mustafa Erdi Kararmaz (2375194)
METU EEE
Ankara/Turkey

I. Introduction

In this assignment, we will conduct experiments involving the training of artificial neural networks (ANNs) and analyze the outcomes. We will develop and train classifiers using both multi-layer perceptron (MLP) and convolutional neural network (CNN) architectures. These implementations will be carried out in the Python programming language, utilizing PyTorch, NumPy, and scikit-learn as our main libraries.

II. Questions

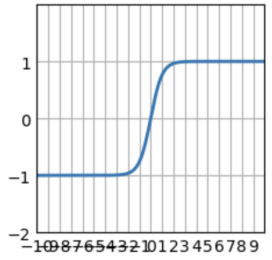
A. *Question 1*

1) Preliminaries: In this question, we are asked to calculate the derivatives of the three activation functions and plot the results. The derivative calculation can be seen in Appendix.

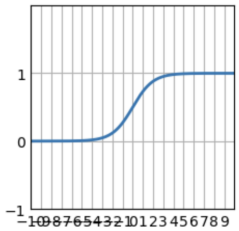
$$y_1 = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{1}$$

$$y_2 = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

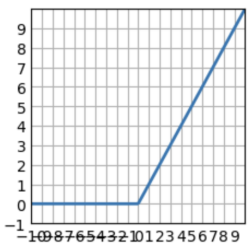
$$y_3 = \max(0, x) \tag{3}$$



(a) tanh(x)

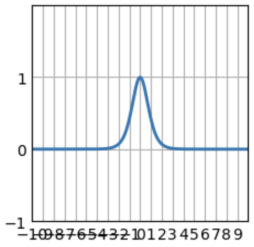


(b) sigmoid(x)

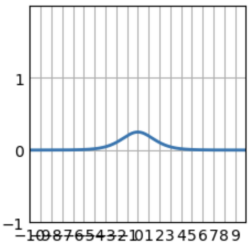


(c) ReLU

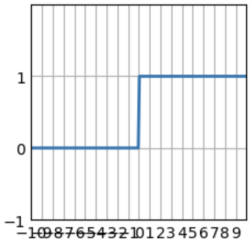
Fig. 1: Activation Functions



(a) $\frac{d}{dx}tanh(x)$



(b) $\frac{d}{dx}\sigma(x)$



(c) $\frac{d}{dx}ReLU$

Fig. 2: Derivatives of the activation functions

2) Implementation: The implementations of the functions can be seen the codes given in Appendix. The Figures 3,4 and 5 indicates the tanh, sigmoid and ReLU functions respectively.

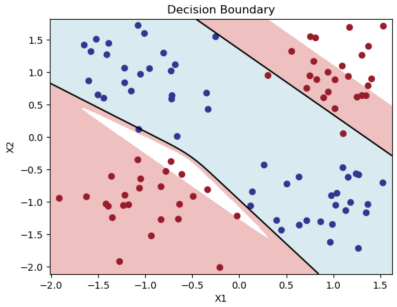


Fig. 3: 97% of test examples classified correctly.

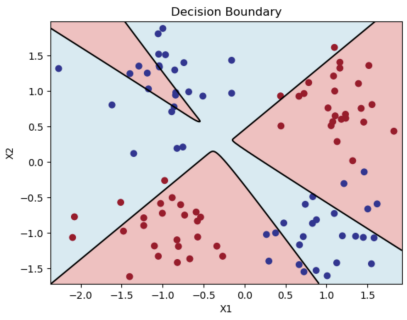


Fig. 4: 78% of test examples classified correctly.

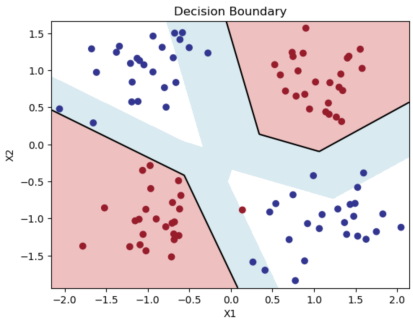


Fig. 5: 99% of test examples classified correctly.

3) Discussion: **1)**

Advantages and Disadvantages of tanh:

- Outputs values ranging between -1 and 1, which can help with data normalization.
- Suffers from vanishing gradient problem, which can slow down learning and make it difficult to deal with deep networks.
- Computationally more complex than ReLU.

Advantages and Disadvantages of sigmoid:

- S-shaped curve that can represent probabilities better since its output is in the range (0,1).
- Suffers from vanishing gradient problem, like tanh, only more severely.
- Outputs are not zero-centered, which can lead to gradients being pushed in one direction and possibly to slower convergence.(See the Question 4.)
- Can cause ”neuron saturation” where the neurons stop learning if the output is consistently near 0 or 1.

Advantages and Disadvantages of ReLU:

- Simple and computationally efficient
- Does not activate all neurons at the same time, leading to sparse activations which can be beneficial for efficiency and allows the network to learn complex patterns.
- Mitigates the vanishing gradient problem, allowing models to learn faster and perform better with deep networks.
- Can suffer from ”dying ReLU” where neurons can become inactive and only output zero if the inputs are negative, which can cause parts of the network to die during training.

Considering the experimental results in Figures 3,4 and 5, it can be seen that the ReLU function gives the highest accuracy and sigmoid function gives the worst.

2)

XOR problem is the problem that we cannot separate the classes with a single line. The problem is that we cannot use a single layer to separate these classes. However, using MLP, we can easily separate the classes.

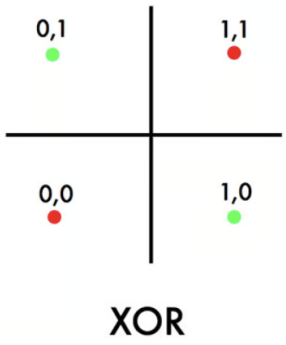


Fig. 6: XOR Problem

3)

The boundary changes in each run. The reason is that the generation of classes in each run is made by a random process. In each run, the points are generated using np.ramdom.randn() function from numpy library. Hence, the regions are changing in each run.

B. Question 2

1) Implementation: In this question, we are asked to implement a 2D convolution function. Using this function and the samples that is given to us (my ID ends with 4, so I am using samples_4) we will generate a result. The Figure 7. indicates this result.

Fig. 7: 2D convolution Result

The convolution function can be found in the Appendix.

2) Discussion:

1)

Convolutional Neural Networks (CNNs) are very important for working with images because they process pictures in a way that's similar to how our eyes do. Regular methods fail with the huge amount of information in pictures, but CNNs handle this by focusing on small parts at a time, like looking for lines, colors, or shapes. This helps them understand and organize the picture piece by piece. Because of this reason, CNNs are great at recognizing and categorizing images, making them useful for things like identifying faces or helping self-driving cars understand what they see.

2)

Kernel of a convolutional layer is like a filter of the input. We implement a 2D filter to do the convolution operation. The size of the window indicates the resolution of the filter. Smaller the filter size, harder to detect the tiny details. However, the size of the filter rises, the complexity of the algorithm rises as well.

3)

In these images, the high frequency content and low frequency content of the images of the number 4 are seen. The kernel (some kind of a filter) helps to reveal the edges (high frequency contents) of the image.

4)

5)

6)

C. *Question 3*

1) Dataset Discussion and Experiment: In this question, we are asked to implement different kind of ANN structures. To implement these structures, we will use pytorch library. The dataset that is provided to us is Fashion-MNIST dataset and it is widely used to train and test neural networks performance. In this question, there are 60000 data to train and 10000 to test the dataset. We will separate the training dataset into two part, one for training and one for validation. The dataset consists of 10 classes. Now, we will use different architectures to train the neural network to measure the performance of the different architectures.

The architectures that are used to train ANN are given as following.

- 'mlp_1': [FC-32, ReLU] + PredictionLayer
- 'mlp_2': [FC-32, ReLU, FC-64 (no bias)] + PredictionLayer
- 'cnn_3': [Conv-3x3x16, ReLU, Conv-5x5x8, ReLU, MaxPool-2x2, Conv-7x7x16, MaxPool-2x2] + PredictionLayer
- 'cnn_4': [Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, Conv-5x5x16, ReLU, MaxPool-2x2] + PredictionLayer
- 'cnn_5': [Conv-3x3x8, ReLU, Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, Conv-3x3x16, ReLU, Conv-3x3x8, ReLU, MaxPool-2x2] + PredictionLayer

where PredictionLayer = [FC10].

We will observe the loss curve, train accuracy curve, validation curve and the test accuracy. The experimental result can be seen in Figure 8.

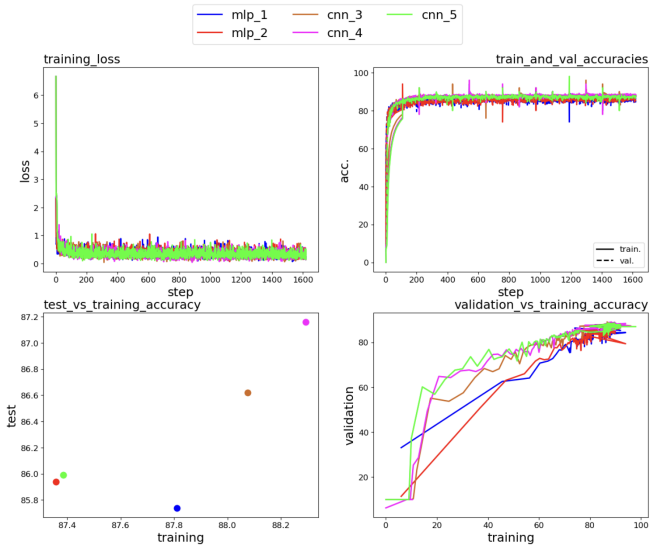


Fig. 8: Loss and accuracy plots of the Part 3

2) Discussion:

1)

The generalization performance of a classifier is the ability to perform unseen data—data that was not available or used during the training phase.

2)

Learning curves and validation curves are important to determine the generalization performance of the classifier.

3)

When we look at the learning curves of the architectures, we can see that all of the architectures performed similar behavior.

4)

The number of parameter may be good for training. Having more parameters typically means that a model can fit the training data better. However,the model may learn to memorize the training data, including noise, which doesn't generalize to new data, leading to *overfitting*.

5)

The deeper architectures can capture more complex features and may perform better. However, in some *deep learning* applications, the risk of overfitting is high due to the very deep networks.

6)

When we look at the mlp structures, we can see some patterns that shows the edges of the clothes. However, this might not the case every time. In the cnn architectures, we only see some 2D filters that will result in the classifier.

7)

In neural network, the units are not specialized to classes. The network adjusts the weights and biases of its units based on the input features and the desired output. This training

process allows the units to learn specialized understanding of how different kind of features contributes to the classification of classes.

8)

The weights of mlp are more interpretable. We can see some kind of pattern in the weights of the mlp class. Figure 9 indicates the fisrt layer of the mlp_1.

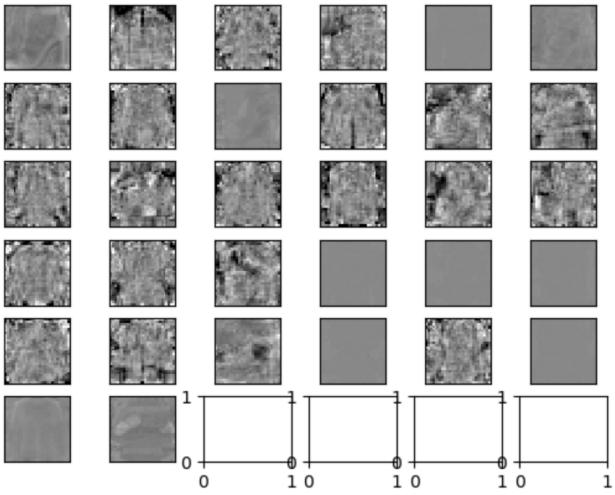


Fig. 9: Weights of the first layer of mlp_1

9)

10)

I would have choose the cnn_4 architecture since the training vs validation accuracy is higher in cnn_4.

D. *Question 4*

In this question, we are asked to compare the activation functions with different architectures.

1) Experiment: When we look at the sigmoid function and the architectures from the part 3, we can see that some of the architectures fail to converge. The reason is that the mathematics behind the sigmoid function. The sigmoid function fails at some point when the input gets closer to 1.

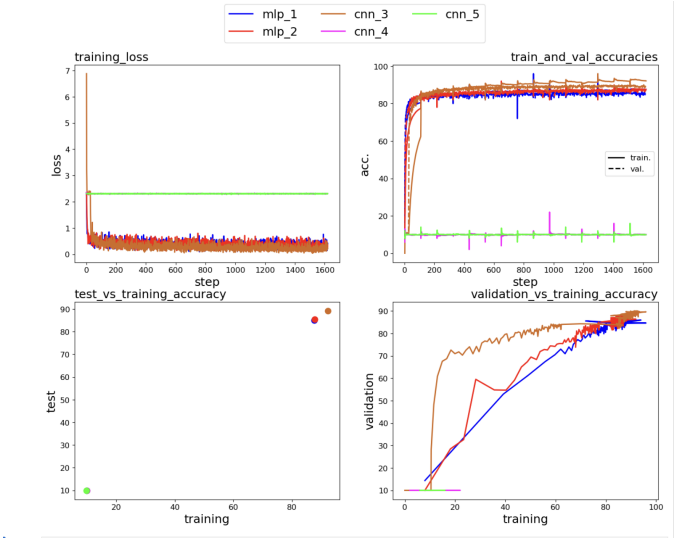


Fig. 10: Loss and accuracy plots of the Part 4

Unfortunately, I did not see the gradient calculation in the question, I trained it like in the question 3. So the saved data is in like part 3 and data plotted like part 3 question.

2) Discussion:

1)

2)

3)

When we look at the quesiton 1.2 and this question, we can see that the cnn_4 and cnn_5 fails in this question. The reason is that we use deeper layers and this causes the sigmoid function to fail.

4)

E. *Question 5*

In this question, we are asked to compare learning rates. Unfortunately, I did not have time to complete this task.

III. Conclusion

In this report, we explored how different types of Artificial Neural Networks (ANNs), like Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN), perform in various tasks. We found that the ReLU activation function works best for most cases because it helps avoid some common problems and makes the network learn faster. Experiments showed that how we set up the network can greatly affect its ability to learn from data, like in the XOR problem or when dealing with images using CNNs.

We also saw that while bigger, more complex networks can learn more detailed features, they can also easily overfit, meaning they memorize the training data but don't perform well on new, unseen data. The experiments with different architectures and the Fashion-MNIST dataset highlighted this balance between learning enough but not too much.

Overall, the work done here shows how important it is to choose the right setup for neural networks to work well. It helps us understand what makes some networks perform better and how to avoid common pitfalls.

IV. Appendix

Github Link of the Homework(You can use this link to reach the codes)

1.1. Preliminaries:

* $\frac{d}{dx}\left(\tanh(x)\right) = \frac{d}{dx}\left(\frac{e^{2x}-1}{e^{2x}+1}\right) = \frac{d}{dx}\left(\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}\right)$

$$= \frac{-\left(e^{x}-e^{-x}\right)\cdot\left(e^{x}-e^{-x}\right) + \left(e^{x}+e^{-x}\right)\cdot\left(e^{x}+e^{-x}\right)}{\left(e^{x}+e^{-x}\right)^{2}}$$

$$= \frac{-\left(e^{x}-e^{-x}\right)^{2} + \left(e^{x}+e^{-x}\right)^{2}}{\left(e^{x}+e^{-x}\right)^{2}} = \frac{\left(e^{x}+e^{-x}\right)^{2}\left(1-\left(\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}\right)^{2}\right)}{\left(e^{x}+e^{-x}\right)^{2}}$$

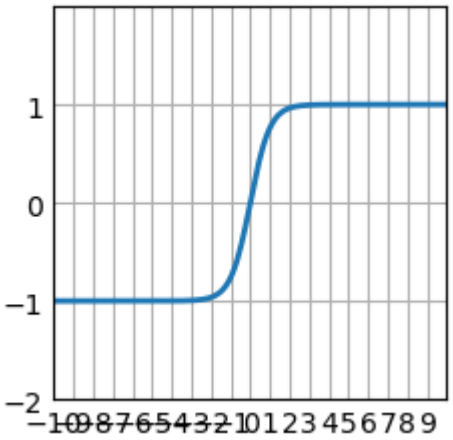
$$= \left(1-\left(\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}\right)^{2}\right) = \boxed{1-\left(\tanh(x)\right)^{2}}$$

* $\sigma(x) = \frac{1}{1+e^{-x}} \rightarrow \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{\left(1+e^{-x}\right)^{2}} =$

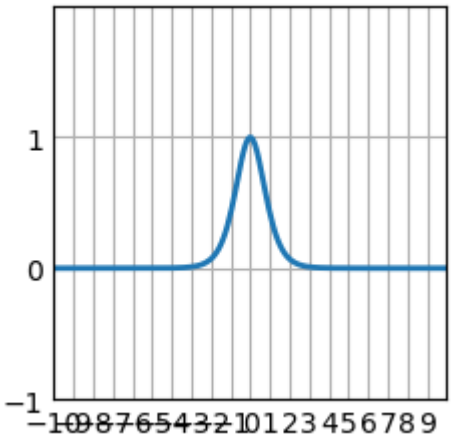
$$= \underbrace{\frac{e^{-x}}{1+e^{-x}}}_{1-\sigma(x)} \cdot \underbrace{\frac{1}{1+e^{-x}}}_{\sigma(x)} = \boxed{\sigma(x)\cdot\left(1-\sigma(x)\right)}$$

* $\frac{d}{dx}\left(\max(0,x)\right) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$

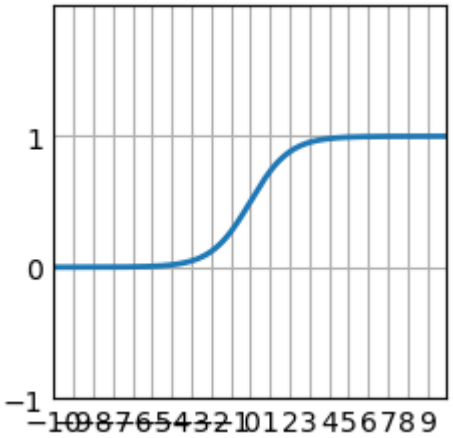
In [11]:
```
import matplotlib.pyplot as plt
import numpy as np

# tanh plot
x = np.linspace(-10, 10, 200)
y = np.tanh(x)

# plot
fig, ax = plt.subplots()

ax.plot(x, y, linewidth=2.0)

ax.set(xlim=(-10, 10), xticks=np.arange(-10, 10),
       ylim=(-2, 2), yticks=np.arange(-2, 2))

plt.show()
```



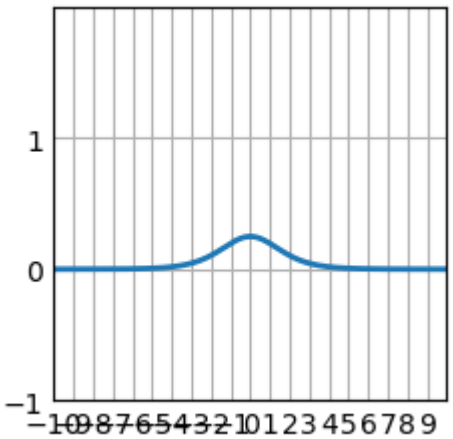
In [12]:
```
#derivative of tanh plot
y = 1 - np.square(np.tanh(x))

# plot
fig, ax = plt.subplots()

ax.plot(x, y, linewidth=2.0)

ax.set(xlim=(-10, 10), xticks=np.arange(-10, 10),
       ylim=(-1, 2), yticks=np.arange(-1, 2))

plt.show()
```



In [16]:
```
#sigmoid plot
y = 1/(1+np.exp(-x))

# plot
fig, ax = plt.subplots()

ax.plot(x, y, linewidth=2.0)

ax.set(xlim=(-10, 10), xticks=np.arange(-10, 10),
       ylim=(-1, 2), yticks=np.arange(-1, 2))

plt.show()
```



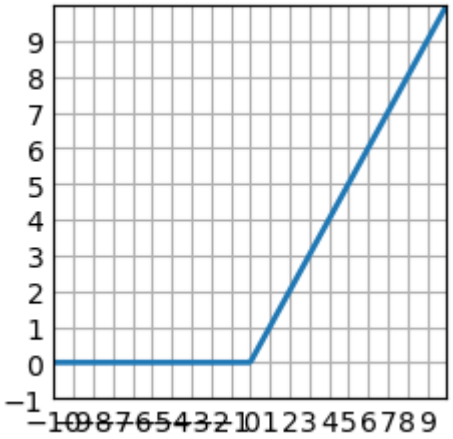
In [42]:
```
#derivative of sigmoid function
sigmoid = 1/(1+np.exp(-x))
y = sigmoid * (1 - sigmoid)

# plot
fig, ax = plt.subplots()

ax.plot(x, y, linewidth=2.0)

ax.set(xlim=(-10, 10), xticks=np.arange(-10, 10),
       ylim=(-1, 2), yticks=np.arange(-1, 2))

plt.show()
```



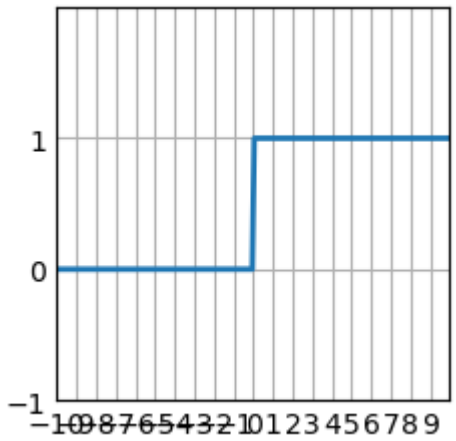

```
-9.899497487437186
```

In [48]:
```
#ReLU
```

```
y = np.zeros(200)
for ind in range(0,200):
    if(x[ind]>0):
        y[ind] = x[ind]
    else:
        y[ind] = 0


# plot
fig, ax = plt.subplots()

ax.plot(x, y, linewidth=2.0)

ax.set(xlim=(-10, 10), xticks=np.arange(-10, 10),
       ylim=(-1, 10), yticks=np.arange(-1, 10))

plt.show()
```



In [50]:
```
#derivative of ReLU

y = np.zeros(200)
for ind in range(0,200):
    if(x[ind]>0):
        y[ind] = 1
    else:
        y[ind] = 0


# plot
fig, ax = plt.subplots()

ax.plot(x, y, linewidth=2.0)

ax.set(xlim=(-10, 10), xticks=np.arange(-10, 10),
       ylim=(-1, 2), yticks=np.arange(-1, 2))

plt.show()
```



In [ ]:

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
import utils
```

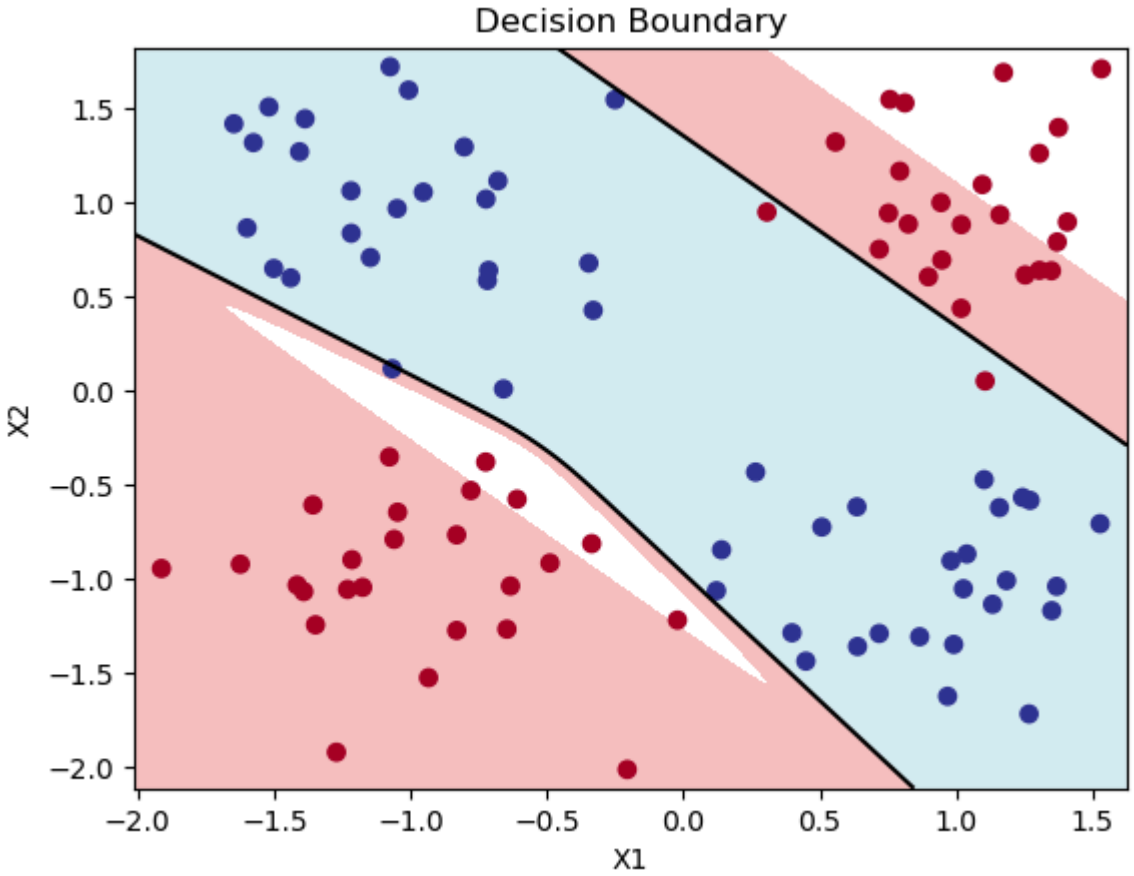
In [2]:
```
class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))


    def tanh(self, x):
        return np.tanh(x) # tanh function itself

    def tanh_derivative(self, x):
        return (1 - np.square(np.tanh(x))) #derivative of the tanh function

    def forward(self, inputs):
        # Forward pass through the network
        # We first calculate the dot product of the inputs and the weights(multiplying them and adding them)
        # after that, we add the bias parameter. The result of this calculation will be the input of the tanh funct
        tanh_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.tanh(tanh_input)

        #Now, we need to calculate the output parameter with the same way
        tanh_input2 = np.dot(self.hidden_output,self.weights_hidden_output) + self.bias_output
        self.output = self.tanh(tanh_input2)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network
        # Compute error
        # In here, the error will be the difference between the target and the output that we calculate.
        output_error = targets - self.output
```

```
        hidden_error = np.dot(output_error,self.weights_hidden_output.T)
        # Compute grads
        output_delta = output_error * self.tanh_derivative(self.output)
        hidden_delta = hidden_error * self.tanh_derivative(self.hidden_output)
        # Update weights and biases
        self.weights_hidden_output += np.dot(self.hidden_output.T,output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += np.dot(inputs.T,hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
```

In [3]:
```
x_train, y_train, x_val, y_val = utils.part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)
```

In [6]:
```
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = 0.001

# nn
nn = MLP(input_size, hidden_size, output_size)
# train
for epoch in range(10000):
    # Forward propagation
    output = nn.forward(x_train)
    # Backpropagation
    nn.backward(x_train,y_train,learning_rate)
    # Print the loss (MSE)
    if epoch % 1000 == 0:
        loss = np.mean((y_train-output)**2) #mse
        print(f'Epoch {epoch}: Loss = {loss}')
# test
y_predict = nn.forward(x_val) > 0.5

print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')
utils.part1PlotBoundary(x_val, y_val, nn)
```

```
Epoch 0: Loss = 0.8235713267875728
Epoch 1000: Loss = 0.04459445295815394
Epoch 2000: Loss = 0.04290561441994432
Epoch 3000: Loss = 0.036467881141081536
Epoch 4000: Loss = 0.0360738493291804
Epoch 5000: Loss = 0.03638034134452279
Epoch 6000: Loss = 0.037596937239886004
Epoch 7000: Loss = 0.042911420852054336
Epoch 8000: Loss = 0.04881378405267069
Epoch 9000: Loss = 0.050139562486487835
97.0 % of test examples classified correctly.
```



In [ ]:

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
import utils
```

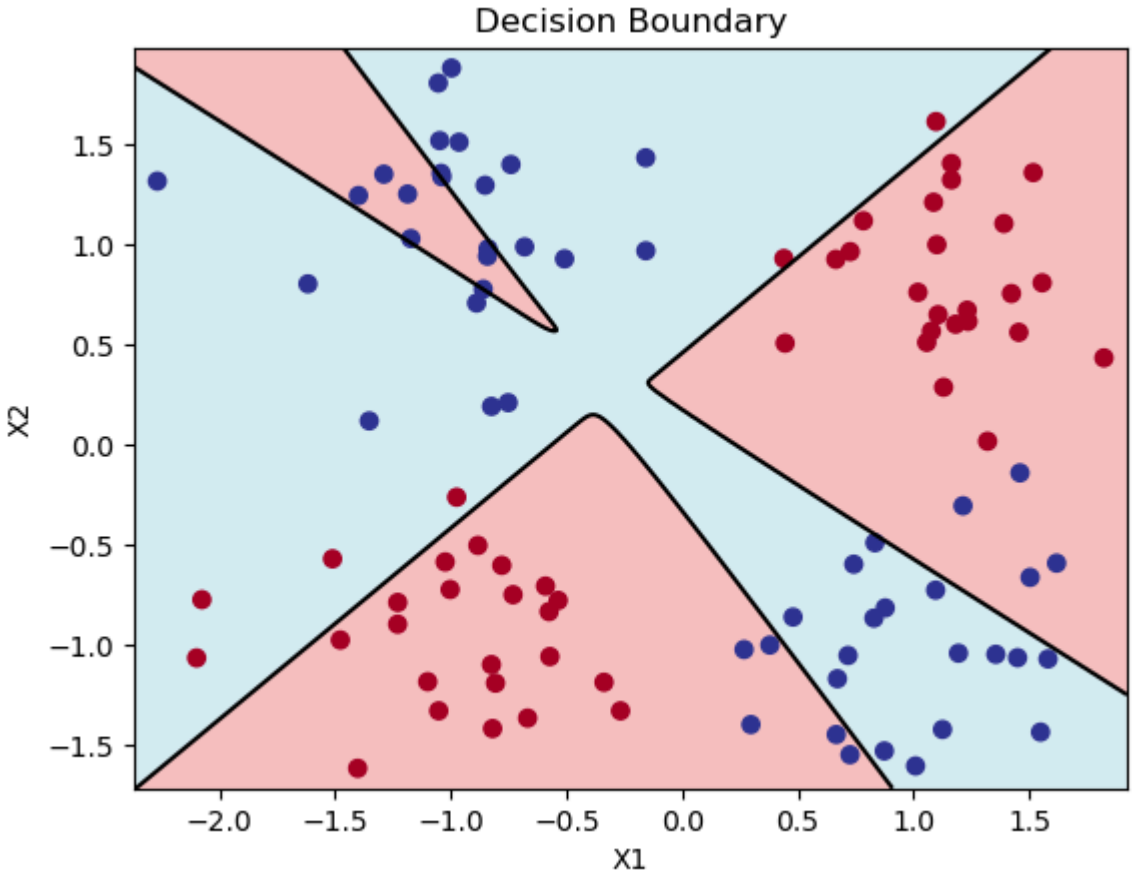
In [2]:
```
class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))


    def sigmoid(self, x):
        return (1/(1+np.exp(-x))) # sigmoid function itself

    def sigmoid_derivative(self, x):
        sigmoid = 1/(1+np.exp(-x))
        return (sigmoid * (1 - sigmoid)) #derivative of the sigmoid function

    def forward(self, inputs):
        # Forward pass through the network
        # We first calculate the dot product of the inputs and the weights(multiplying them and adding them)
        # after that, we add the bias parameter. The result of this calculation will be the input of the sigmoid fu
        sigmoid_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.sigmoid(sigmoid_input)

        #Now, we need to calculate the output parameter with the same way
        sigmoid_input2 = np.dot(self.hidden_output,self.weights_hidden_output) + self.bias_output
        self.output = self.sigmoid(sigmoid_input2)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network
        # Compute error
        # In here, the error will be the difference between the target and the output that we calculate.
```

```
        output_error = targets - self.output
        hidden_error = np.dot(output_error,self.weights_hidden_output.T)
        # Compute grads
        output_delta = output_error * self.sigmoid_derivative(self.output)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
        # Update weights and biases
        self.weights_hidden_output += np.dot(self.hidden_output.T,output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += np.dot(inputs.T,hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
```

In [3]:
```
x_train, y_train, x_val, y_val = utils.part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)
```

In [4]:
```
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = 0.001

# create nn with class
nn = MLP(input_size, hidden_size, output_size)
# train
for epoch in range(10000):
    # Forward propagation
    output = nn.forward(x_train)
    # Backpropagation
    nn.backward(x_train,y_train,learning_rate)
    # Print the loss (MSE)
    if epoch % 1000 == 0:
        loss = np.mean((y_train-output)**2) #mse
        print(f'Epoch {epoch}: Loss = {loss}')
# testing process
y_predict = nn.forward(x_val) > 0.5

print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')
utils.part1PlotBoundary(x_val, y_val, nn)
```

```
Epoch 0: Loss = 0.2588181869198883
Epoch 1000: Loss = 0.2153827172414274
Epoch 2000: Loss = 0.18566457465367278
Epoch 3000: Loss = 0.18436603523998252
Epoch 4000: Loss = 0.1914269201285468
Epoch 5000: Loss = 0.1974059175037798
Epoch 6000: Loss = 0.19891811004202758
Epoch 7000: Loss = 0.19873121929751378
Epoch 8000: Loss = 0.19840782911206525
Epoch 9000: Loss = 0.1987843851369886
78.0 % of test examples classified correctly.
```



In [ ]:

In [ ]:

In [16]:
```
import numpy as np
from matplotlib import pyplot as plt
import utils
```

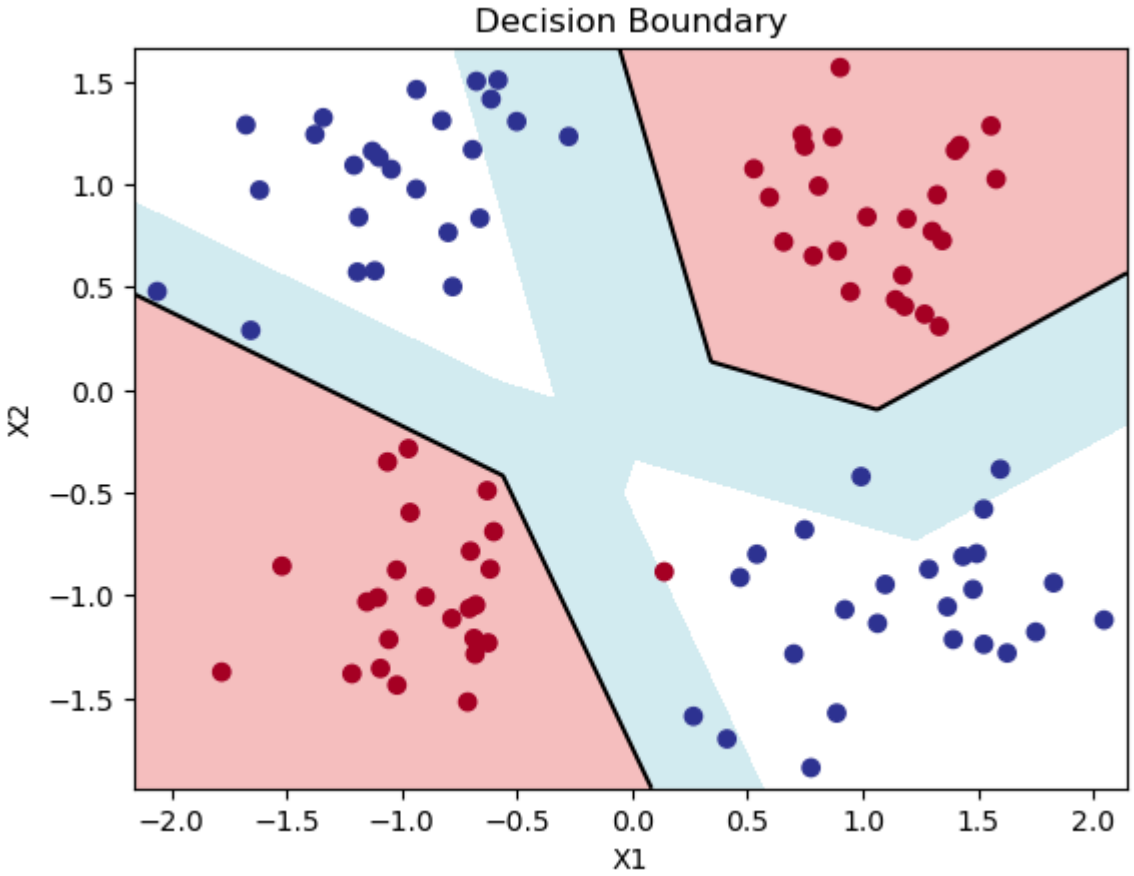
In [17]:
```
class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
        self.bias_output = np.zeros((1, self.output_size))


    def ReLu(self, x):
        return np.maximum(0,x) # ReLu function itself

    def ReLu_derivative(self, x):
        return np.where(x <= 0, 0, 1) #derivative of the ReLu function

    def forward(self, inputs):
        # Forward pass through the network
        # We first calculate the dot product of the inputs and the weights(multiplying them and adding them)
        # after that, we add the bias parameter. The result of this calculation will be the input of the ReLu funct
        ReLu_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.ReLu(ReLu_input)

        #Now, we need to calculate the output parameter with the same way
        ReLu_input2 = np.dot(self.hidden_output,self.weights_hidden_output) + self.bias_output
        self.output = self.ReLu(ReLu_input2)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        # Backward pass through the network
        # Compute error
        # In here, the error will be the difference between the target and the output that we calculate.
        output_error = targets - self.output
```

```
        hidden_error = np.dot(output_error,self.weights_hidden_output.T)
        # Compute gradients
        output_delta = output_error * self.ReLu_derivative(self.output)
        hidden_delta = hidden_error * self.ReLu_derivative(self.hidden_output)
        # Update weights and biases
        self.weights_hidden_output += np.dot(self.hidden_output.T,output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += np.dot(inputs.T,hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
```

In [18]:
```
x_train, y_train, x_val, y_val = utils.part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)
```

In [19]:
```
# Define neural network parameters
input_size = 2
hidden_size = 4
output_size = 1
learning_rate = 0.001

# initialize neural network using class MLP
nn = MLP(input_size, hidden_size, output_size)
# training process
for epoch in range(10000):
    # Forward propagation
    output = nn.forward(x_train)
    # Backpropagation
    nn.backward(x_train,y_train,learning_rate)
    # Print the loss (MSE)
    if epoch % 1000 == 0:
        loss = np.mean((y_train-output)**2) #mse
        print(f'Epoch {epoch}: Loss = {loss}')
# test the neural network
y_predict = nn.forward(x_val) > 0.5

print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')
utils.part1PlotBoundary(x_val, y_val, nn)
```

```
Epoch 0: Loss = 0.4933612571936991
Epoch 1000: Loss = 0.051046658181009466
Epoch 2000: Loss = 0.050527320797572356
Epoch 3000: Loss = 0.050409010542089926
Epoch 4000: Loss = 0.05031934195023968
Epoch 5000: Loss = 0.05027891502800111
Epoch 6000: Loss = 0.05028503989487604
Epoch 7000: Loss = 0.05025542196889612
Epoch 8000: Loss = 0.05030026394864986
Epoch 9000: Loss = 0.050261439883103706
99.0 % of test examples classified correctly.
```



In [ ]:

In [21]:
```
import numpy as np
import utils as ut

def my_conv2d(input, kernel):
    batch_size, input_channels, input_height, input_width = input.shape
    output_channels, _, filter_height, filter_width = kernel.shape

    # output matrix dimensions
    output_height = input_height - filter_height + 1
    output_width = input_width - filter_width + 1

    # allocating the output matrix
    output = np.zeros((batch_size, output_channels, output_height, output_width))

    # conv
    for batch in range(batch_size):
        for out_channel in range(output_channels):
            for in_channel in range(input_channels):
                for i in range(output_height):
                    for j in range(output_width):
                        region = input[batch, in_channel, i:i+filter_height, j:j+filter_width]
                        output[batch, out_channel, i, j] += np.sum(region * kernel[out_channel, in_channel])

    return output
```

In [22]:
```
# input shape: [batch size, input_channels, input_height, input_width]
input=np.load('samples_4.npy')
# input shape: [output_channels, input_channels, filter_height, filter width]
kernel=np.load('kernel.npy')
#print(input.shape)
out = my_conv2d(input, kernel)

ut.part2Plots(out)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

In [ ]:

In [235…
```
import torchvision
import torch
import torch.nn.functional as F
import numpy as np
import json
import utils as ut
```

In [226…
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [227…
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
train_subset = torch.utils.data.Subset(train_data, train_indices)
```

```
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [228…

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_mlp, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_mlp.fc1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [229…

```
class mlp_1(torch.nn.Module):
    def __init__(self, input_size):
        super(mlp_1, self).__init__()
        self.input_size = input_size
        # Adjust the hidden layer size to 32
        self.fc1 = torch.nn.Linear(input_size, 32)  # FC-32
        self.relu = torch.nn.ReLU()  # ReLU activation
        self.fc2 = torch.nn.Linear(32, 10)  # Prediction layer, size = num_classes

    def forward(self, x):
        x = x.view(-1, self.input_size)  # Flatten the input if needed
        hidden = self.fc1(x)  # Pass through the first fully connected layer
        relu = self.relu(hidden)  # Apply ReLU activation
        output = self.fc2(relu)  # Pass through the prediction layer
        return output

model_mlp = mlp_1(784)
```

In [230…

```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_mlp.parameters(), lr = 0.01, momentum = 0.0)
```

In [231…

```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_mlp,train_generator,val_generator)
```

```
Epoch:  1/15    Loss: 2.3526    Train Acc: 6.0000   Val Acc:33.1000
Epoch:  1/15    Loss: 0.9997    Train Acc: 45.6364   Val Acc:62.5667
Epoch:  1/15    Loss: 0.7040    Train Acc: 56.2857   Val Acc:64.0833
Epoch:  1/15    Loss: 0.6740    Train Acc: 60.3871   Val Acc:70.7500
Epoch:  1/15    Loss: 0.7387    Train Acc: 63.5122   Val Acc:71.6167
Epoch:  1/15    Loss: 0.5907    Train Acc: 65.4118   Val Acc:72.7667
Epoch:  1/15    Loss: 0.5353    Train Acc: 66.8197   Val Acc:74.7833
Epoch:  1/15    Loss: 0.6040    Train Acc: 68.0000   Val Acc:76.6833
Epoch:  1/15    Loss: 0.6459    Train Acc: 68.7654   Val Acc:76.4833
Epoch:  1/15    Loss: 0.3367    Train Acc: 69.5824   Val Acc:78.0667
Epoch:  1/15    Loss: 0.7026    Train Acc: 70.4554   Val Acc:79.0500
Epoch:  1/15    Loss: 0.4726    Train Acc: 71.2252   Val Acc:77.7833
Epoch:  1/15    Loss: 0.4398    Train Acc: 72.0661   Val Acc:75.3833
Epoch:  1/15    Loss: 0.6010    Train Acc: 72.0916   Val Acc:77.4333
Epoch:  1/15    Loss: 0.6683    Train Acc: 72.3688   Val Acc:75.4667
Epoch:  1/15    Loss: 0.7941    Train Acc: 72.6623   Val Acc:80.2667
Epoch:  1/15    Loss: 0.4404    Train Acc: 73.2298   Val Acc:80.5167
Epoch:  1/15    Loss: 0.4961    Train Acc: 73.6842   Val Acc:81.0500
Epoch:  1/15    Loss: 0.5134    Train Acc: 74.0331   Val Acc:80.3167
Epoch:  1/15    Loss: 0.3396    Train Acc: 74.5864   Val Acc:82.0000
Epoch:  1/15    Loss: 0.4488    Train Acc: 74.8159   Val Acc:80.6000
Epoch:  1/15    Loss: 0.4206    Train Acc: 74.9953   Val Acc:78.1333
Epoch:  1/15    Loss: 0.8305    Train Acc: 75.2398   Val Acc:81.4000
Epoch:  1/15    Loss: 0.3894    Train Acc: 75.6104   Val Acc:80.9500
Epoch:  1/15    Loss: 0.4128    Train Acc: 75.9087   Val Acc:79.3833
Epoch:  1/15    Loss: 0.4947    Train Acc: 75.9761   Val Acc:79.9167
Epoch:  1/15    Loss: 0.4632    Train Acc: 76.1533   Val Acc:81.4333
Epoch:  1/15    Loss: 0.8098    Train Acc: 76.2878   Val Acc:80.3000
Epoch:  1/15    Loss: 0.6716    Train Acc: 76.3701   Val Acc:80.8167
Epoch:  1/15    Loss: 0.5766    Train Acc: 76.5292   Val Acc:80.6000
Epoch:  1/15    Loss: 0.4477    Train Acc: 76.7575   Val Acc:81.2833
Epoch:  1/15    Loss: 0.5025    Train Acc: 76.9260   Val Acc:82.2167
Epoch:  1/15    Loss: 0.3183    Train Acc: 77.1277   Val Acc:82.5500
Epoch:  1/15    Loss: 0.4773    Train Acc: 77.3414   Val Acc:82.8833
Epoch:  1/15    Loss: 0.4176    Train Acc: 77.4194   Val Acc:79.5500
Epoch:  1/15    Loss: 0.5691    Train Acc: 77.5442   Val Acc:81.0333
Epoch:  1/15    Loss: 0.5117    Train Acc: 77.6565   Val Acc:82.5000
Epoch:  1/15    Loss: 0.5209    Train Acc: 77.7736   Val Acc:81.5000
Epoch:  1/15    Loss: 0.6112    Train Acc: 77.8530   Val Acc:78.0667
Epoch:  1/15    Loss: 0.3958    Train Acc: 77.9182   Val Acc:82.3167
Epoch:  1/15    Loss: 0.6888    Train Acc: 77.9850   Val Acc:82.8667
```

```
Epoch: 15/15    Loss: 0.5113    Train Acc: 87.7658   Val Acc:85.5000
Epoch: 15/15    Loss: 0.2738    Train Acc: 87.7503   Val Acc:85.5833
Epoch: 15/15    Loss: 0.3349    Train Acc: 87.7486   Val Acc:85.8500
Epoch: 15/15    Loss: 0.2474    Train Acc: 87.7336   Val Acc:85.7500
Epoch: 15/15    Loss: 0.3925    Train Acc: 87.7300   Val Acc:85.8500
Epoch: 15/15    Loss: 0.2731    Train Acc: 87.7524   Val Acc:86.3667
Epoch: 15/15    Loss: 0.2257    Train Acc: 87.7530   Val Acc:86.5500
Epoch: 15/15    Loss: 0.3586    Train Acc: 87.7726   Val Acc:85.8667
Epoch: 15/15    Loss: 0.3938    Train Acc: 87.7834   Val Acc:84.3333
Epoch: 15/15    Loss: 0.4601    Train Acc: 87.7752   Val Acc:86.2333
Epoch: 15/15    Loss: 0.1556    Train Acc: 87.7899   Val Acc:85.2333
Epoch: 15/15    Loss: 0.2990    Train Acc: 87.7900   Val Acc:85.7000
Epoch: 15/15    Loss: 0.2383    Train Acc: 87.8103   Val Acc:85.0500
Epoch: 15/15    Loss: 0.4464    Train Acc: 87.7862   Val Acc:86.1167
Epoch: 15/15    Loss: 0.4052    Train Acc: 87.7982   Val Acc:85.4667
Epoch: 15/15    Loss: 0.3015    Train Acc: 87.7884   Val Acc:84.8000
Epoch: 15/15    Loss: 0.3223    Train Acc: 87.7789   Val Acc:84.4833
Epoch: 15/15    Loss: 0.3781    Train Acc: 87.7887   Val Acc:85.6333
Epoch: 15/15    Loss: 0.4496    Train Acc: 87.7888   Val Acc:86.3500
Epoch: 15/15    Loss: 0.2529    Train Acc: 87.8040   Val Acc:85.8000
Epoch: 15/15    Loss: 0.3059    Train Acc: 87.8095   Val Acc:84.3500
```

In [232…

```
test_accuracy = validate_model(model_mlp,test_generator)
print(test_accuracy)
```

```
85.74
```

In [238…

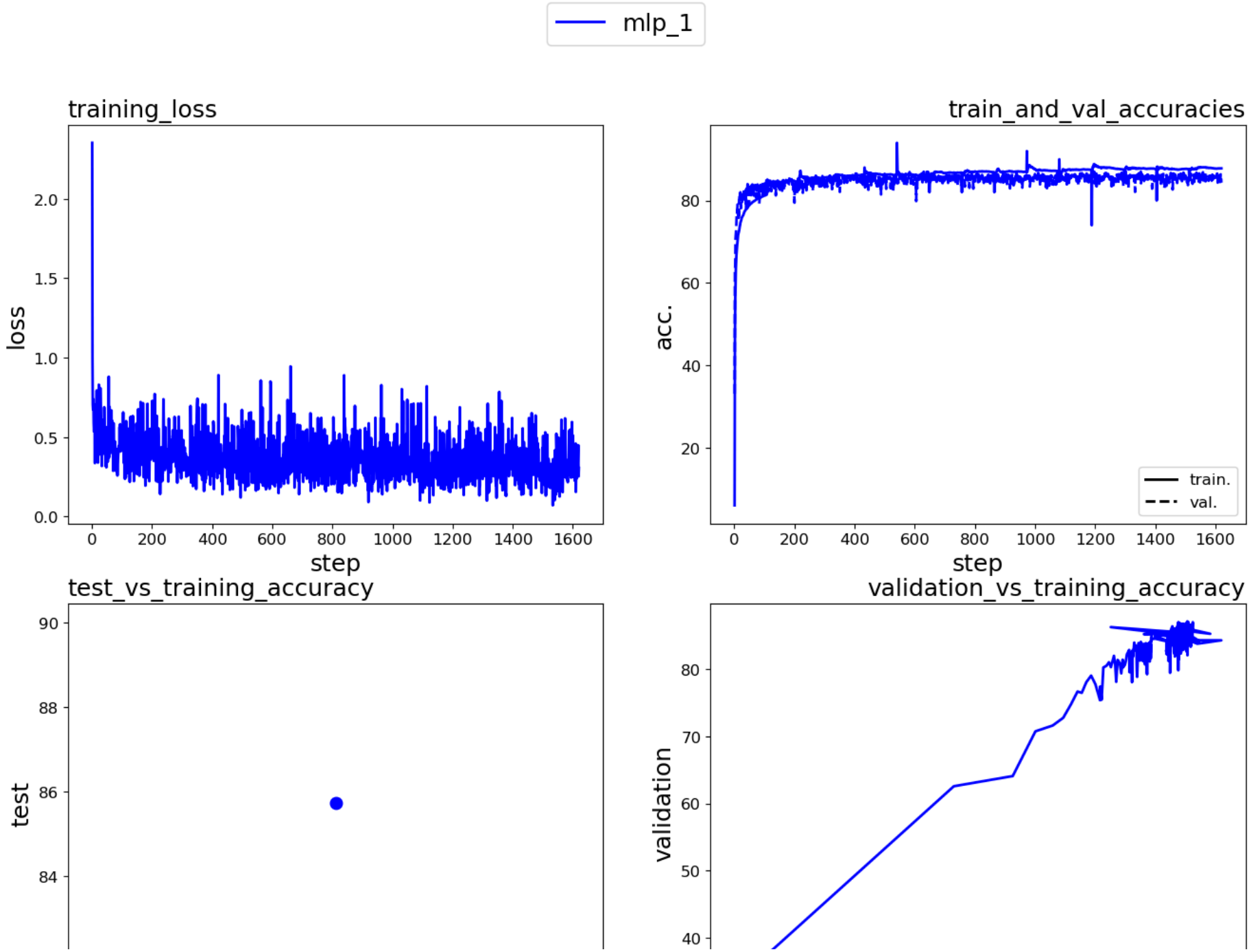
```
# Form the dictionary
architecture_name = "mlp_1"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

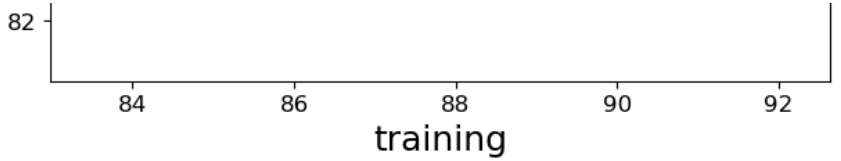
```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

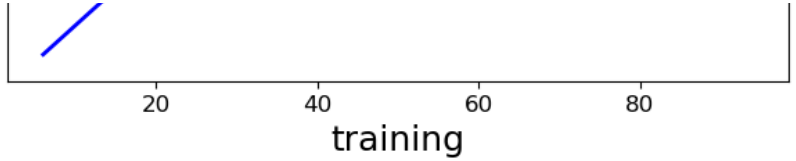
```
Experiment results saved to part3_mlp_1.json
```

In [240…

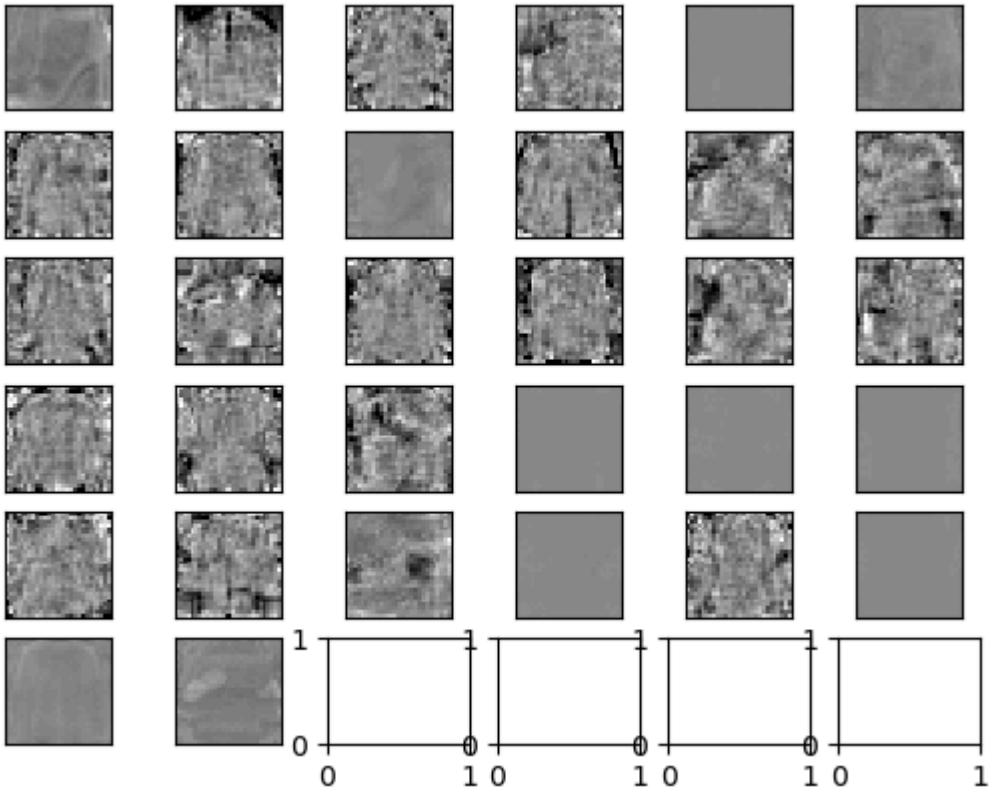
```
liste = [experiment_results]
ut.part3Plots(liste)
```


mlp_1
training_loss
loss
step
train_and_val_accuracies
acc.
step
train.
val.
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [245…
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```



In [ ]:

In [1]:
```
import torchvision
import torch
import torch.nn.functional as F
import numpy as np
import json
import utils as ut
```

In [2]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [3]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
train_subset = torch.utils.data.Subset(train_data, train_indices)
```

```
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [4]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_mlp, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_mlp.fc1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [5]:
```
class mlp_2(torch.nn.Module):
    def __init__(self, input_size):
        super(mlp_2, self).__init__()
        self.input_size = input_size
        self.fc1 = torch.nn.Linear(input_size, 32)  # FC-32
        self.relu = torch.nn.ReLU()  # ReLU activation
        self.fc2 = torch.nn.Linear(32, 64)
        self.fc3 = torch.nn.Linear(64,10)

    def forward(self, x):
        x = x.view(-1, self.input_size)  # Flatten the input if needed
        hidden = self.fc1(x)  # Pass through the first fully connected layer
        relu = self.relu(hidden)  # Apply ReLU activation
        hidden = self.fc2(relu)
        relu = self.relu(hidden)
        output = self.fc3(relu)  # Pass through the prediction layer
        return output

model_mlp = mlp_2(784)
```

In [6]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_mlp.parameters(), lr = 0.01, momentum = 0.0)
```

In [7]:
```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_mlp,train_generator,val_generator)
```

```
Epoch:  1/15    Loss: 2.3135    Train Acc: 6.0000   Val Acc:11.4333
Epoch:  1/15    Loss: 1.0686    Train Acc: 36.9091   Val Acc:50.7500
Epoch:  1/15    Loss: 0.9586    Train Acc: 47.0476   Val Acc:63.0833
Epoch:  1/15    Loss: 0.9803    Train Acc: 54.3226   Val Acc:66.0000
Epoch:  1/15    Loss: 0.7356    Train Acc: 58.6341   Val Acc:71.6333
Epoch:  1/15    Loss: 0.6863    Train Acc: 60.3137   Val Acc:72.8500
Epoch:  1/15    Loss: 0.5249    Train Acc: 62.0984   Val Acc:72.2500
Epoch:  1/15    Loss: 0.5593    Train Acc: 63.8028   Val Acc:72.5333
Epoch:  1/15    Loss: 0.8764    Train Acc: 64.9877   Val Acc:75.1000
Epoch:  1/15    Loss: 0.7172    Train Acc: 66.1099   Val Acc:77.6000
Epoch:  1/15    Loss: 0.4957    Train Acc: 67.0693   Val Acc:77.5500
Epoch:  1/15    Loss: 0.5940    Train Acc: 68.2523   Val Acc:79.3500
Epoch:  1/15    Loss: 0.4718    Train Acc: 69.0413   Val Acc:76.5500
Epoch:  1/15    Loss: 1.0141    Train Acc: 69.9695   Val Acc:77.4000
Epoch:  1/15    Loss: 0.4322    Train Acc: 70.5248   Val Acc:78.2500
Epoch:  1/15    Loss: 0.5664    Train Acc: 70.9404   Val Acc:78.9000
Epoch:  1/15    Loss: 0.7399    Train Acc: 71.5031   Val Acc:79.8833
Epoch:  1/15    Loss: 0.4772    Train Acc: 71.7895   Val Acc:79.3333
Epoch:  1/15    Loss: 0.8843    Train Acc: 72.2320   Val Acc:80.3333
Epoch:  1/15    Loss: 0.8328    Train Acc: 72.5654   Val Acc:81.2667
Epoch:  1/15    Loss: 0.5065    Train Acc: 72.9353   Val Acc:81.4833
Epoch:  1/15    Loss: 0.4047    Train Acc: 73.3270   Val Acc:81.8833
Epoch:  1/15    Loss: 0.6624    Train Acc: 73.7014   Val Acc:80.0667
Epoch:  1/15    Loss: 0.4644    Train Acc: 74.0606   Val Acc:81.9167
Epoch:  1/15    Loss: 0.6817    Train Acc: 74.4647   Val Acc:79.4000
Epoch:  1/15    Loss: 0.5290    Train Acc: 74.6295   Val Acc:81.3500
Epoch:  1/15    Loss: 0.6009    Train Acc: 74.8276   Val Acc:77.0667
Epoch:  1/15    Loss: 0.5826    Train Acc: 74.8782   Val Acc:78.2167
Epoch:  1/15    Loss: 0.6172    Train Acc: 75.1388   Val Acc:80.7667
Epoch:  1/15    Loss: 0.3169    Train Acc: 75.3196   Val Acc:80.4167
Epoch:  1/15    Loss: 0.3830    Train Acc: 75.5615   Val Acc:78.8500
Epoch:  1/15    Loss: 0.7258    Train Acc: 75.7685   Val Acc:80.7000
Epoch:  1/15    Loss: 0.5664    Train Acc: 75.9751   Val Acc:81.3000
Epoch:  1/15    Loss: 0.6707    Train Acc: 76.0483   Val Acc:80.8167
Epoch:  1/15    Loss: 0.3642    Train Acc: 76.2639   Val Acc:80.8167
Epoch:  1/15    Loss: 0.3976    Train Acc: 76.3305   Val Acc:81.0167
Epoch:  1/15    Loss: 0.5857    Train Acc: 76.4709   Val Acc:76.3333
Epoch:  1/15    Loss: 0.6715    Train Acc: 76.4582   Val Acc:76.6167
Epoch:  1/15    Loss: 0.6100    Train Acc: 76.4934   Val Acc:79.9333
Epoch:  1/15    Loss: 0.6850    Train Acc: 76.6650   Val Acc:80.8167
Epoch:  1/15    Loss: 0.4227    Train Acc: 76.7382   Val Acc:81.1833
```

```
Epoch: 15/15    Loss: 0.3505    Train Acc: 87.2377   Val Acc:86.6500
Epoch: 15/15    Loss: 0.1588    Train Acc: 87.2463   Val Acc:86.0833
Epoch: 15/15    Loss: 0.4061    Train Acc: 87.2548   Val Acc:85.8333
Epoch: 15/15    Loss: 0.1492    Train Acc: 87.2586   Val Acc:86.6000
Epoch: 15/15    Loss: 0.2048    Train Acc: 87.2975   Val Acc:86.6667
Epoch: 15/15    Loss: 0.1896    Train Acc: 87.2899   Val Acc:86.4833
Epoch: 15/15    Loss: 0.4594    Train Acc: 87.2653   Val Acc:85.8333
Epoch: 15/15    Loss: 0.2242    Train Acc: 87.2774   Val Acc:86.8167
Epoch: 15/15    Loss: 0.5686    Train Acc: 87.2808   Val Acc:85.6667
Epoch: 15/15    Loss: 0.3197    Train Acc: 87.3049   Val Acc:85.7333
Epoch: 15/15    Loss: 0.4983    Train Acc: 87.2997   Val Acc:86.5167
Epoch: 15/15    Loss: 0.3801    Train Acc: 87.3293   Val Acc:84.5833
Epoch: 15/15    Loss: 0.3409    Train Acc: 87.3401   Val Acc:85.2667
Epoch: 15/15    Loss: 0.5581    Train Acc: 87.3526   Val Acc:84.5833
Epoch: 15/15    Loss: 0.4191    Train Acc: 87.3591   Val Acc:85.7500
Epoch: 15/15    Loss: 0.2630    Train Acc: 87.3810   Val Acc:85.4167
Epoch: 15/15    Loss: 0.5687    Train Acc: 87.3598   Val Acc:85.8333
Epoch: 15/15    Loss: 0.3259    Train Acc: 87.3679   Val Acc:85.0500
Epoch: 15/15    Loss: 0.3133    Train Acc: 87.3663   Val Acc:85.5667
Epoch: 15/15    Loss: 0.3416    Train Acc: 87.3666   Val Acc:85.7667
Epoch: 15/15    Loss: 0.4073    Train Acc: 87.3576   Val Acc:85.1000
```

In [12]:
```
test_accuracy = validate_model(model_mlp,test_generator)
print(test_accuracy)
```

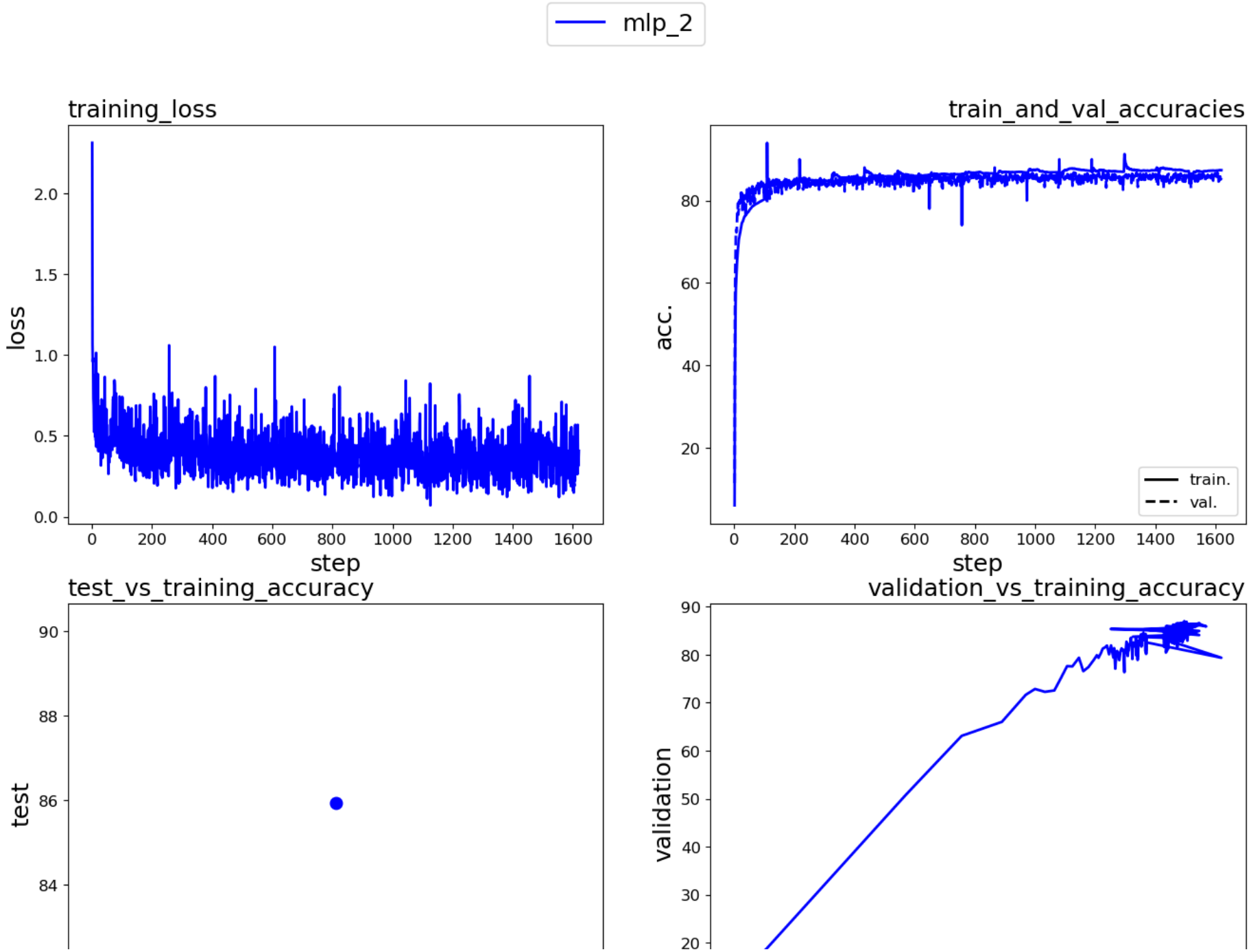
```
85.94
```

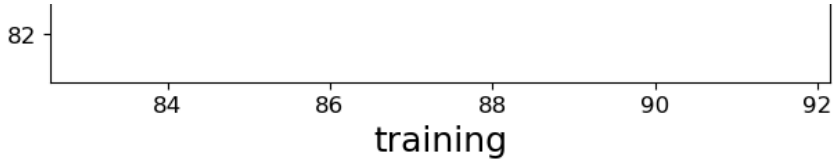
In [13]:
```
# Form the dictionary
architecture_name = "mlp_2"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

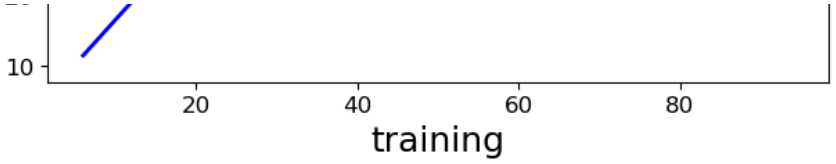
```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

```
Experiment results saved to part3_mlp_2.json
```

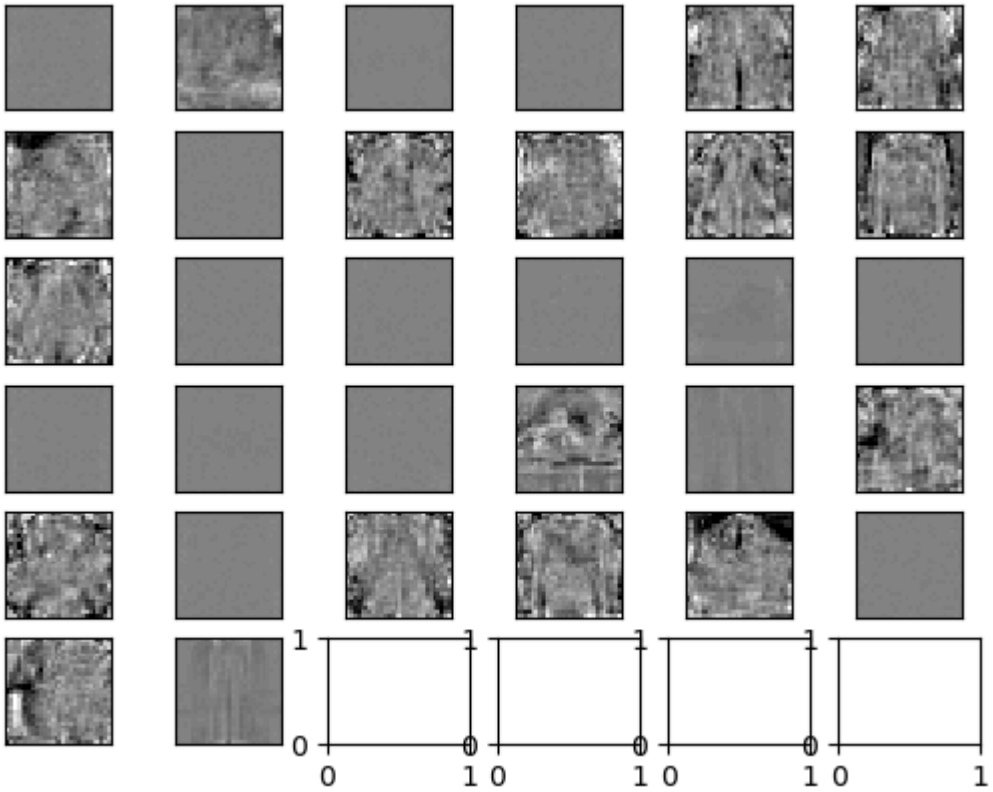
In [14]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```


mlp_2
training_loss
loss
step
train_and_val_accuracies
acc.
step
train.
val.
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [15]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```



In [ ]:
```
```

In [20]:
```
import torchvision
import torch
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import json
import utils as ut
```

In [21]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [22]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
```

```
train_subset = torch.utils.data.Subset(train_data, train_indices)
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [23]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_cnn, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_cnn.conv1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [24]:

```
class cnn_3(nn.Module):
    def __init__(self, num_classes=10):
        super(cnn_3, self).__init__()
        # First convolutional layer: Conv-3x3x16
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        # Second convolutional layer: Conv-5x5x8
        self.conv2 = nn.Conv2d(16, 8, kernel_size=5, stride=1, padding=2)
        # Third convolutional layer: Conv-7x7x16
        self.conv3 = nn.Conv2d(8, 16, kernel_size=7, stride=1, padding=3)
        # Max Pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Prediction layer
        # Assuming the input images are 28x28 pixels, the dimensionality needs to be calculated
        # considering the convolutions (with padding) and pooling operations applied.
        # After two pooling layers, the size of the image will be reduced to 7x7.
        self.fc = nn.Linear(16 * 7 * 7, num_classes)

    def forward(self, x):
        # Apply the first convolution, followed by ReLU
        x = F.relu(self.conv1(x))
        # Apply the second convolution, followed by ReLU
        x = F.relu(self.conv2(x))
        # Apply max pooling
        x = self.pool(x)
        # Apply the third convolution, followed by ReLU and then max pooling
        x = F.relu(self.conv3(x))
```

```
        x = self.pool(x)
        # Flatten the output for the fully connected layer
        x = x.view(-1, 16 * 7 * 7)
        # Pass through the prediction layer
        x = self.fc(x)
        return x


model_cnn = cnn_3(784)
```

In [25]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_cnn.parameters(), lr = 0.01, momentum = 0.0)
```

In [26]:
```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_cnn,train_generator,val_generator)
```

```
Epoch: 15/15    Loss: 0.3689    Train Acc: 88.3471   Val Acc:88.3833
Epoch: 15/15    Loss: 0.2666    Train Acc: 88.3185   Val Acc:87.7833
Epoch: 15/15    Loss: 0.2774    Train Acc: 88.3202   Val Acc:87.3000
Epoch: 15/15    Loss: 0.3846    Train Acc: 88.2566   Val Acc:88.2333
Epoch: 15/15    Loss: 0.4214    Train Acc: 88.2196   Val Acc:87.6167
Epoch: 15/15    Loss: 0.2496    Train Acc: 88.2309   Val Acc:87.6333
Epoch: 15/15    Loss: 0.4131    Train Acc: 88.2418   Val Acc:87.7167
Epoch: 15/15    Loss: 0.4432    Train Acc: 88.2072   Val Acc:87.3833
Epoch: 15/15    Loss: 0.2875    Train Acc: 88.2070   Val Acc:87.7667
Epoch: 15/15    Loss: 0.1859    Train Acc: 88.2287   Val Acc:87.7500
Epoch: 15/15    Loss: 0.1966    Train Acc: 88.2674   Val Acc:87.7500
Epoch: 15/15    Loss: 0.1918    Train Acc: 88.2872   Val Acc:87.8167
Epoch: 15/15    Loss: 0.4692    Train Acc: 88.2616   Val Acc:87.3833
Epoch: 15/15    Loss: 0.2899    Train Acc: 88.2944   Val Acc:86.6667
Epoch: 15/15    Loss: 0.7265    Train Acc: 88.2296   Val Acc:87.6167
Epoch: 15/15    Loss: 0.3366    Train Acc: 88.2324   Val Acc:87.3333
Epoch: 15/15    Loss: 0.2684    Train Acc: 88.2190   Val Acc:87.6000
Epoch: 15/15    Loss: 0.3029    Train Acc: 88.1997   Val Acc:87.6167
Epoch: 15/15    Loss: 0.3973    Train Acc: 88.1810   Val Acc:87.6167
Epoch: 15/15    Loss: 0.3606    Train Acc: 88.2089   Val Acc:87.9667
Epoch: 15/15    Loss: 0.1755    Train Acc: 88.1967   Val Acc:88.0000
Epoch: 15/15    Loss: 0.4552    Train Acc: 88.2027   Val Acc:87.9833
Epoch: 15/15    Loss: 0.3897    Train Acc: 88.2232   Val Acc:88.5500
Epoch: 15/15    Loss: 0.2674    Train Acc: 88.2431   Val Acc:88.6333
Epoch: 15/15    Loss: 0.1878    Train Acc: 88.2682   Val Acc:88.0833
Epoch: 15/15    Loss: 0.7278    Train Acc: 88.2616   Val Acc:87.7167
Epoch: 15/15    Loss: 0.2619    Train Acc: 88.2469   Val Acc:87.8833
Epoch: 15/15    Loss: 0.4497    Train Acc: 88.2654   Val Acc:87.3167
Epoch: 15/15    Loss: 0.4478    Train Acc: 88.2294   Val Acc:87.7167
Epoch: 15/15    Loss: 0.2006    Train Acc: 88.2024   Val Acc:88.4833
Epoch: 15/15    Loss: 0.2031    Train Acc: 88.2208   Val Acc:87.6667
Epoch: 15/15    Loss: 0.2811    Train Acc: 88.2283   Val Acc:87.1500
Epoch: 15/15    Loss: 0.2892    Train Acc: 88.2330   Val Acc:86.4667
Epoch: 15/15    Loss: 0.4322    Train Acc: 88.2351   Val Acc:88.3000
Epoch: 15/15    Loss: 0.2243    Train Acc: 88.2247   Val Acc:87.3500
Epoch: 15/15    Loss: 0.4475    Train Acc: 88.1973   Val Acc:87.2000
Epoch: 15/15    Loss: 0.2203    Train Acc: 88.1730   Val Acc:87.5167
Epoch: 15/15    Loss: 0.4493    Train Acc: 88.1540   Val Acc:87.3667
Epoch: 15/15    Loss: 0.3905    Train Acc: 88.1379   Val Acc:86.8333
Epoch: 15/15    Loss: 0.3973    Train Acc: 88.1199   Val Acc:87.8333
Epoch: 15/15    Loss: 0.3473    Train Acc: 88.1370   Val Acc:88.3500
```

```
Epoch: 15/15    Loss: 0.2796    Train Acc: 88.1378   Val Acc:87.9000
Epoch: 15/15    Loss: 0.1495    Train Acc: 88.1476   Val Acc:87.9500
Epoch: 15/15    Loss: 0.2730    Train Acc: 88.1661   Val Acc:87.4333
Epoch: 15/15    Loss: 0.2982    Train Acc: 88.1620   Val Acc:87.5333
Epoch: 15/15    Loss: 0.3804    Train Acc: 88.1822   Val Acc:87.7333
Epoch: 15/15    Loss: 0.2293    Train Acc: 88.1694   Val Acc:86.7833
Epoch: 15/15    Loss: 0.3705    Train Acc: 88.1740   Val Acc:87.5167
Epoch: 15/15    Loss: 0.2248    Train Acc: 88.1955   Val Acc:87.1500
Epoch: 15/15    Loss: 0.2486    Train Acc: 88.1640   Val Acc:86.3500
Epoch: 15/15    Loss: 0.2534    Train Acc: 88.1644   Val Acc:86.7000
Epoch: 15/15    Loss: 0.3789    Train Acc: 88.1359   Val Acc:87.8167
Epoch: 15/15    Loss: 0.2347    Train Acc: 88.1264   Val Acc:87.6333
Epoch: 15/15    Loss: 0.3161    Train Acc: 88.1292   Val Acc:87.7167
Epoch: 15/15    Loss: 0.2750    Train Acc: 88.1179   Val Acc:86.7500
Epoch: 15/15    Loss: 0.2806    Train Acc: 88.1286   Val Acc:87.8167
Epoch: 15/15    Loss: 0.3469    Train Acc: 88.0921   Val Acc:87.7167
Epoch: 15/15    Loss: 0.4424    Train Acc: 88.0485   Val Acc:86.9333
Epoch: 15/15    Loss: 0.2544    Train Acc: 88.0557   Val Acc:87.8667
Epoch: 15/15    Loss: 0.3541    Train Acc: 88.0742   Val Acc:88.3333
Epoch: 15/15    Loss: 0.2697    Train Acc: 88.0754   Val Acc:88.3167
Epoch: 15/15    Loss: 0.3619    Train Acc: 88.0747   Val Acc:87.8667
```

In [27]:
```
test_accuracy = validate_model(model_cnn,test_generator)
print(test_accuracy)
```

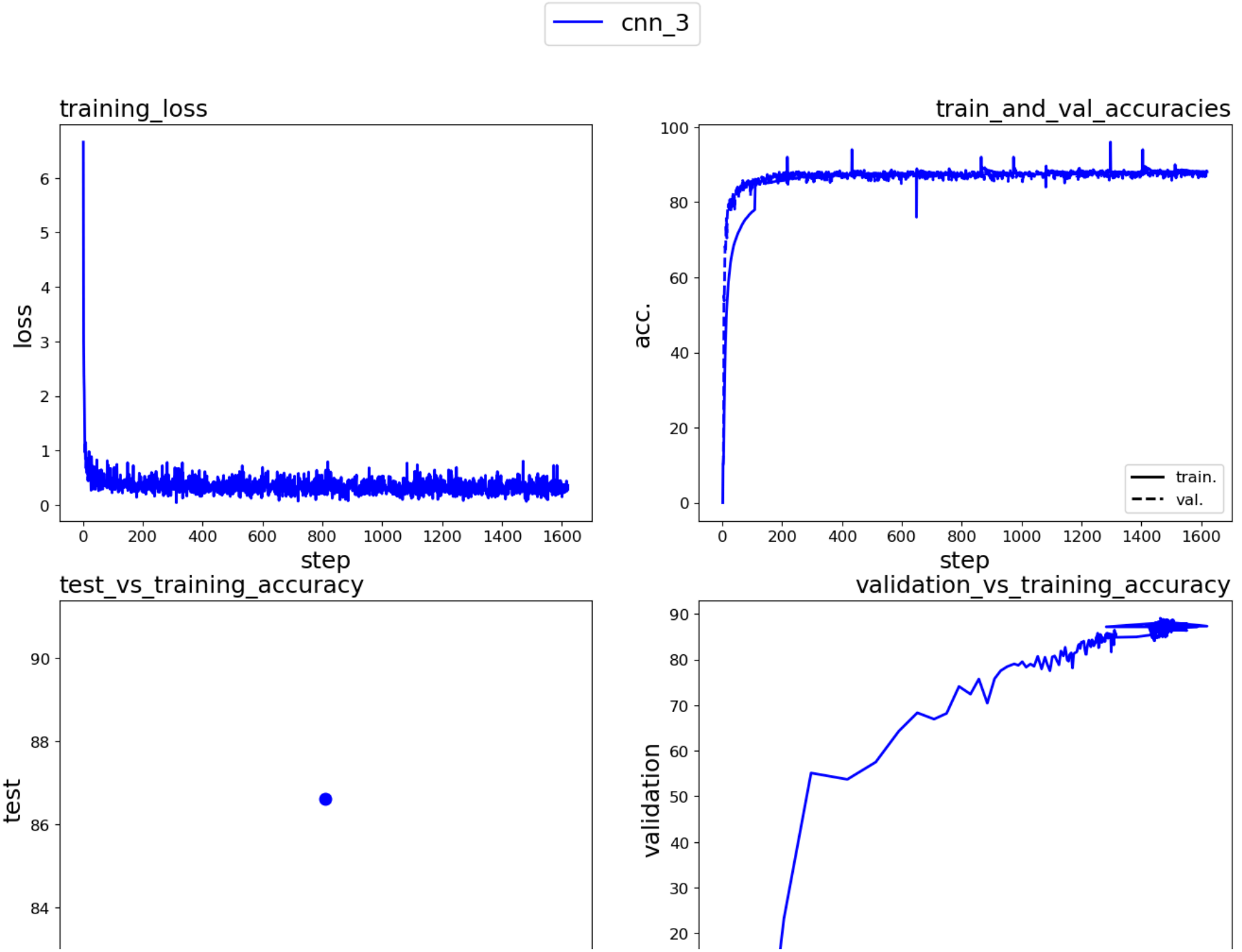
```
86.62
```

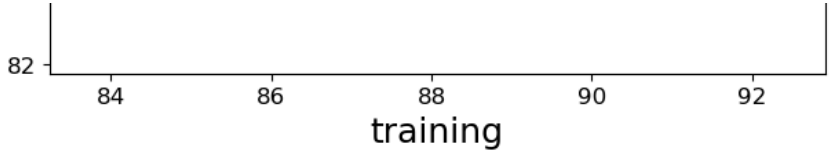
In [28]:
```
# Form the dictionary
architecture_name = "cnn_3"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

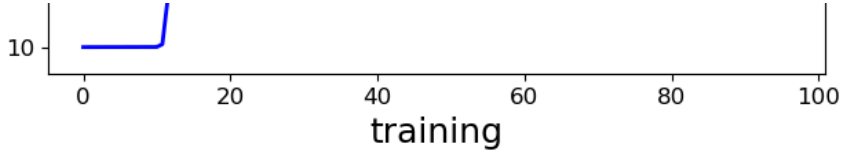
```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

```
Experiment results saved to part3_cnn_3.json
```

In [29]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```

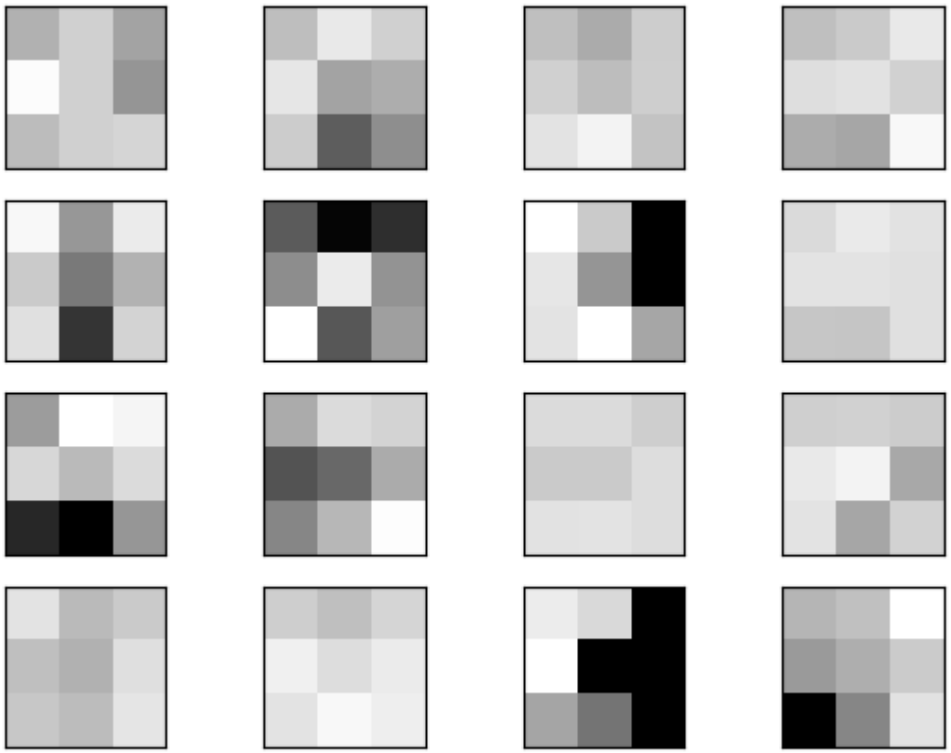

cnn_3
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [30]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```

```
/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1/Part3/utils.py:438: UserWarning: FigureCa
nvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



In [ ]:

In [1]:
```
import torchvision
import torch
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import json
import utils as ut
```

In [2]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [3]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
```

```
train_subset = torch.utils.data.Subset(train_data, train_indices)
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [4]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_cnn, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_cnn.conv1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [5]:

```
class cnn_4(nn.Module):
    def __init__(self, num_classes=10):
        super(cnn_4, self).__init__()
        # Convolutional and Pooling layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.conv3 = nn.Conv2d(8, 16, kernel_size=5, padding=2)  # Conv-5x5x16
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool-2x2
        self.conv4 = nn.Conv2d(16, 16, kernel_size=5, padding=2)  # Conv-5x5x16

        # The feature map size after convolutions and pooling will be 7x7.
        # This is because the max pooling layers halve the dimensions, and the convolutions are designed to preserv
        self.fc = nn.Linear(16 * 7 * 7, num_classes)  # Prediction layer

    def forward(self, x):
        x = F.relu(self.conv1(x))  # Conv-3x3x16 followed by ReLU
        x = F.relu(self.conv2(x))  # Conv-3x3x8 followed by ReLU
        x = F.relu(self.conv3(x))  # Conv-5x5x16 followed by ReLU
        x = self.pool(x)  # MaxPool-2x2
        x = F.relu(self.conv4(x))  # Conv-5x5x16 followed by ReLU
        x = self.pool(x)  # MaxPool-2x2 again

        # Flatten the output for the fully connected layer
        x = x.view(-1, 16 * 7 * 7)
        # Prediction layer
        x = self.fc(x)
```

```
        return x



model_cnn = cnn_4(784)
```

In [6]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_cnn.parameters(), lr = 0.01, momentum = 0.0)
```

In [7]:
```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_cnn,train_generator,val_generator)
```

```
Epoch: 15/15    Loss: 0.3511    Train Acc: 88.4946   Val Acc:88.6167
Epoch: 15/15    Loss: 0.3768    Train Acc: 88.4671   Val Acc:88.4333
Epoch: 15/15    Loss: 0.3102    Train Acc: 88.5031   Val Acc:87.4833
Epoch: 15/15    Loss: 0.2845    Train Acc: 88.4929   Val Acc:88.8500
Epoch: 15/15    Loss: 0.3515    Train Acc: 88.4870   Val Acc:88.4333
Epoch: 15/15    Loss: 0.2880    Train Acc: 88.5010   Val Acc:88.8667
Epoch: 15/15    Loss: 0.2749    Train Acc: 88.4568   Val Acc:87.5500
Epoch: 15/15    Loss: 0.3726    Train Acc: 88.4143   Val Acc:87.9167
Epoch: 15/15    Loss: 0.1889    Train Acc: 88.4584   Val Acc:88.3333
Epoch: 15/15    Loss: 0.3472    Train Acc: 88.4319   Val Acc:88.6333
Epoch: 15/15    Loss: 0.3119    Train Acc: 88.4670   Val Acc:88.4667
Epoch: 15/15    Loss: 0.2861    Train Acc: 88.5044   Val Acc:88.7833
Epoch: 15/15    Loss: 0.3446    Train Acc: 88.4750   Val Acc:87.9000
Epoch: 15/15    Loss: 0.5735    Train Acc: 88.4365   Val Acc:87.7500
Epoch: 15/15    Loss: 0.4150    Train Acc: 88.4027   Val Acc:86.4833
Epoch: 15/15    Loss: 0.2977    Train Acc: 88.4026   Val Acc:87.6667
Epoch: 15/15    Loss: 0.4502    Train Acc: 88.3639   Val Acc:87.5500
Epoch: 15/15    Loss: 0.4628    Train Acc: 88.3835   Val Acc:88.1333
Epoch: 15/15    Loss: 0.2749    Train Acc: 88.3963   Val Acc:88.1000
Epoch: 15/15    Loss: 0.3325    Train Acc: 88.4424   Val Acc:87.2167
Epoch: 15/15    Loss: 0.4239    Train Acc: 88.4539   Val Acc:87.8833
Epoch: 15/15    Loss: 0.1976    Train Acc: 88.4739   Val Acc:87.8833
Epoch: 15/15    Loss: 0.2177    Train Acc: 88.5257   Val Acc:88.4000
Epoch: 15/15    Loss: 0.4627    Train Acc: 88.4631   Val Acc:87.2833
Epoch: 15/15    Loss: 0.3009    Train Acc: 88.4337   Val Acc:87.7000
Epoch: 15/15    Loss: 0.1859    Train Acc: 88.4416   Val Acc:88.1167
Epoch: 15/15    Loss: 0.2940    Train Acc: 88.4078   Val Acc:87.5333
Epoch: 15/15    Loss: 0.2127    Train Acc: 88.4104   Val Acc:88.9000
Epoch: 15/15    Loss: 0.4992    Train Acc: 88.3887   Val Acc:86.9333
Epoch: 15/15    Loss: 0.3062    Train Acc: 88.3808   Val Acc:87.7500
Epoch: 15/15    Loss: 0.2045    Train Acc: 88.3469   Val Acc:86.7333
Epoch: 15/15    Loss: 0.2439    Train Acc: 88.3424   Val Acc:87.4167
Epoch: 15/15    Loss: 0.4266    Train Acc: 88.3150   Val Acc:87.7667
Epoch: 15/15    Loss: 0.3483    Train Acc: 88.3186   Val Acc:88.0500
Epoch: 15/15    Loss: 0.2029    Train Acc: 88.3321   Val Acc:87.5833
Epoch: 15/15    Loss: 0.3118    Train Acc: 88.3329   Val Acc:87.9667
Epoch: 15/15    Loss: 0.1837    Train Acc: 88.3484   Val Acc:88.2167
Epoch: 15/15    Loss: 0.1782    Train Acc: 88.3562   Val Acc:88.7000
Epoch: 15/15    Loss: 0.5759    Train Acc: 88.3520   Val Acc:88.2000
Epoch: 15/15    Loss: 0.1550    Train Acc: 88.3831   Val Acc:88.5000
Epoch: 15/15    Loss: 0.2201    Train Acc: 88.3833   Val Acc:88.1833
```

```
Epoch: 15/15    Loss: 0.4258    Train Acc: 88.3858   Val Acc:88.0000
Epoch: 15/15    Loss: 0.2574    Train Acc: 88.3814   Val Acc:87.7667
Epoch: 15/15    Loss: 0.1908    Train Acc: 88.3861   Val Acc:88.4833
Epoch: 15/15    Loss: 0.2883    Train Acc: 88.3596   Val Acc:88.3667
Epoch: 15/15    Loss: 0.2377    Train Acc: 88.3622   Val Acc:88.3833
Epoch: 15/15    Loss: 0.3665    Train Acc: 88.3388   Val Acc:86.8333
Epoch: 15/15    Loss: 0.3584    Train Acc: 88.3330   Val Acc:88.1333
Epoch: 15/15    Loss: 0.3291    Train Acc: 88.3358   Val Acc:88.5333
Epoch: 15/15    Loss: 0.3188    Train Acc: 88.3281   Val Acc:88.3500
Epoch: 15/15    Loss: 0.4527    Train Acc: 88.3080   Val Acc:88.1167
Epoch: 15/15    Loss: 0.2532    Train Acc: 88.2760   Val Acc:87.7000
Epoch: 15/15    Loss: 0.3963    Train Acc: 88.2671   Val Acc:87.5667
Epoch: 15/15    Loss: 0.2857    Train Acc: 88.2523   Val Acc:87.5333
Epoch: 15/15    Loss: 0.3575    Train Acc: 88.2478   Val Acc:87.9500
Epoch: 15/15    Loss: 0.1923    Train Acc: 88.2611   Val Acc:88.6667
Epoch: 15/15    Loss: 0.5638    Train Acc: 88.2703   Val Acc:87.8333
Epoch: 15/15    Loss: 0.2530    Train Acc: 88.2735   Val Acc:88.4167
Epoch: 15/15    Loss: 0.1904    Train Acc: 88.2786   Val Acc:88.1333
Epoch: 15/15    Loss: 0.3607    Train Acc: 88.2816   Val Acc:88.5000
Epoch: 15/15    Loss: 0.1311    Train Acc: 88.2922   Val Acc:86.9833
Epoch: 15/15    Loss: 0.2411    Train Acc: 88.2932   Val Acc:87.2167
```

In [8]:
```
test_accuracy = validate_model(model_cnn,test_generator)
print(test_accuracy)
```

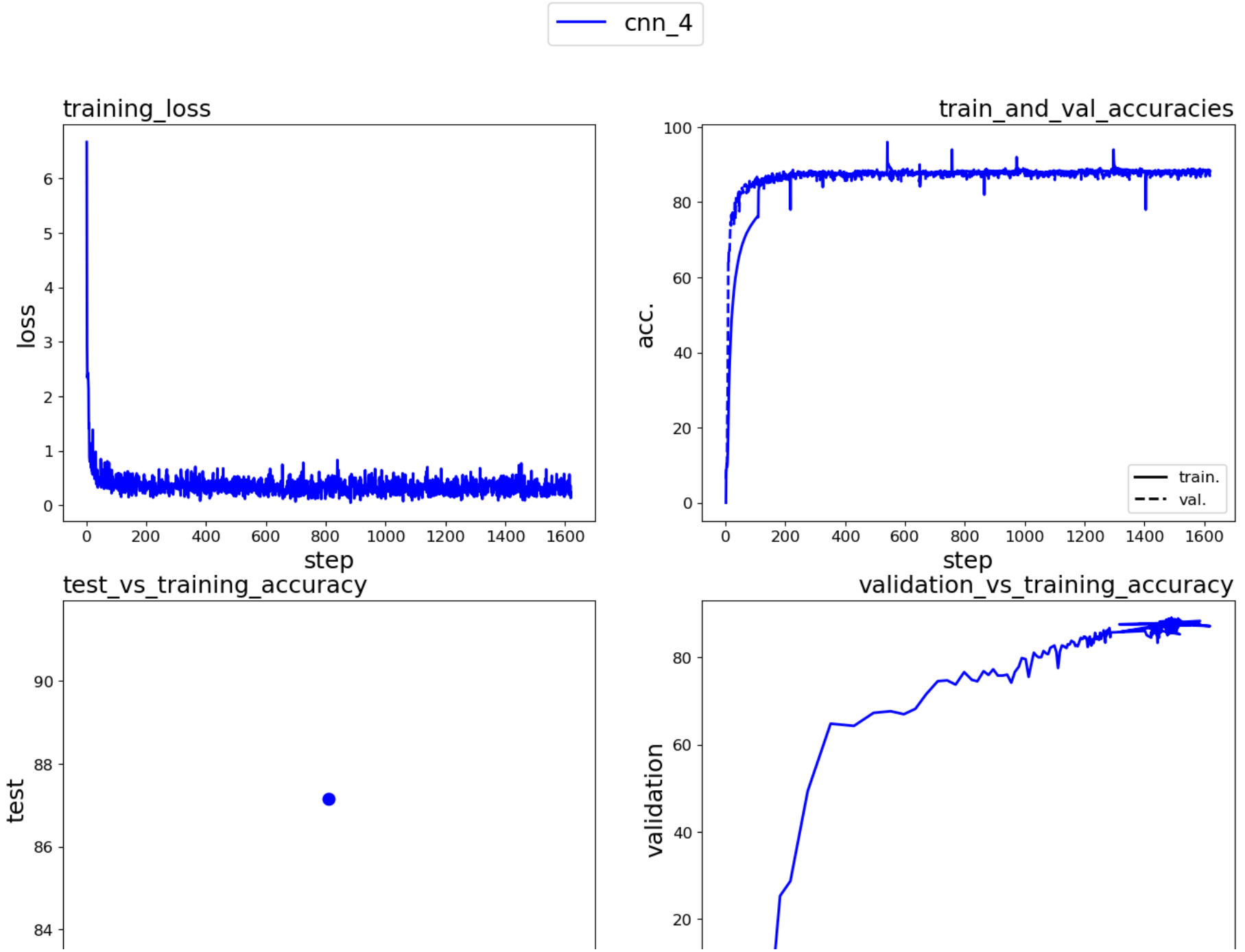
```
87.16
```

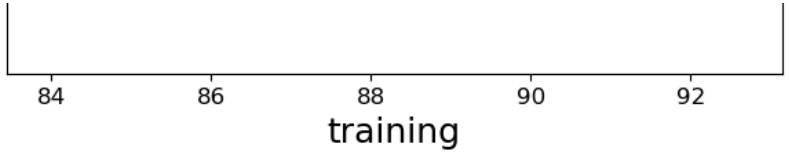
In [9]:
```
# Form the dictionary
architecture_name = "cnn_4"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

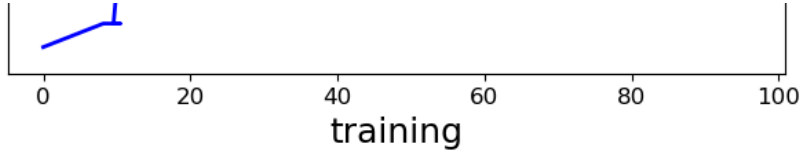
```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

```
Experiment results saved to part3_cnn_4.json
```

In [10]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```

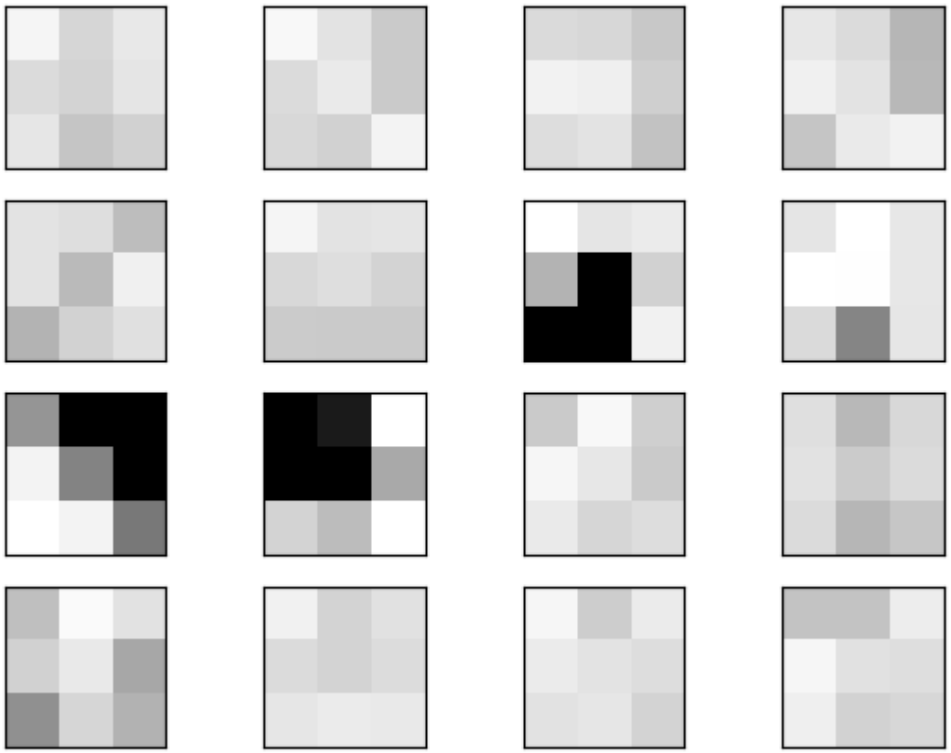

cnn_4
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [11]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```

```
/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1/Part3/utils.py:438: UserWarning: FigureCa
nvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



In [ ]:

In [1]:
```
import torchvision
import torch
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import json
import utils as ut
```

In [2]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [3]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
```

```
train_subset = torch.utils.data.Subset(train_data, train_indices)
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [4]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_cnn, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_cnn.conv1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [5]:
```
import torch.nn as nn
import torch.nn.functional as F

class cnn_5(nn.Module):
    def __init__(self, num_classes=10):
        super(cnn_5, self).__init__()
        # Convolutional layers with ReLU activations
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.conv3 = nn.Conv2d(16, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.conv4 = nn.Conv2d(8, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.pool1 = nn.MaxPool2d(2, 2)  # MaxPool-2x2 after the fourth conv layer
        self.conv5 = nn.Conv2d(16, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.conv6 = nn.Conv2d(16, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.pool2 = nn.MaxPool2d(2, 2)  # MaxPool-2x2 after the sixth conv layer

        # Prediction layer
        # After two pooling layers, the size of the feature map will be 7x7
        self.fc = nn.Linear(8 * 7 * 7, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))  # Conv-3x3x8 + ReLU
        x = F.relu(self.conv2(x))  # Conv-3x3x16 + ReLU
        x = F.relu(self.conv3(x))  # Conv-3x3x8 + ReLU
        x = F.relu(self.conv4(x))  # Conv-3x3x16 + ReLU
        x = self.pool1(x)  # MaxPool-2x2
```

```
        x = F.relu(self.conv5(x))  # Conv-3x3x16 + ReLU
        x = F.relu(self.conv6(x))  # Conv-3x3x8 + ReLU
        x = self.pool2(x)  # MaxPool-2x2
        x = x.view(-1, 8 * 7 * 7)  # Flatten the tensor
        x = self.fc(x)  # Prediction layer
        return x


model_cnn = cnn_5(784)
```

In [6]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_cnn.parameters(), lr = 0.01, momentum = 0.0)
```

In [7]:
```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_cnn,train_generator,val_generator)
```

```
Epoch:  1/15    Loss: 6.6676    Train Acc: 0.0000   Val Acc:10.0000
Epoch:  1/15    Loss: 2.4765    Train Acc: 7.4545   Val Acc:10.0000
Epoch:  1/15    Loss: 2.4217    Train Acc: 7.3333   Val Acc:10.0000
Epoch:  1/15    Loss: 2.5003    Train Acc: 7.8710   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3515    Train Acc: 8.0488   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2811    Train Acc: 8.5490   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2700    Train Acc: 9.0164   Val Acc:10.0000
Epoch:  1/15    Loss: 2.0245    Train Acc: 9.9718   Val Acc:37.1833
Epoch:  1/15    Loss: 1.1296    Train Acc: 14.2963   Val Acc:60.1500
Epoch:  1/15    Loss: 1.2183    Train Acc: 19.1429   Val Acc:56.8667
Epoch:  1/15    Loss: 0.8431    Train Acc: 23.1287   Val Acc:63.4000
Epoch:  1/15    Loss: 0.9819    Train Acc: 27.0991   Val Acc:67.6167
Epoch:  1/15    Loss: 1.0640    Train Acc: 30.0496   Val Acc:68.2167
Epoch:  1/15    Loss: 0.7769    Train Acc: 33.2061   Val Acc:73.5833
Epoch:  1/15    Loss: 0.7938    Train Acc: 35.8723   Val Acc:69.3667
Epoch:  1/15    Loss: 0.8563    Train Acc: 38.1987   Val Acc:73.1500
Epoch:  1/15    Loss: 0.4698    Train Acc: 40.3727   Val Acc:76.8333
Epoch:  1/15    Loss: 0.8737    Train Acc: 42.3743   Val Acc:72.3833
Epoch:  1/15    Loss: 0.7441    Train Acc: 44.0000   Val Acc:73.1333
Epoch:  1/15    Loss: 0.4639    Train Acc: 45.6230   Val Acc:73.5167
Epoch:  1/15    Loss: 0.8570    Train Acc: 46.9453   Val Acc:75.4500
Epoch:  1/15    Loss: 0.6882    Train Acc: 48.3223   Val Acc:77.0167
Epoch:  1/15    Loss: 0.6770    Train Acc: 49.5837   Val Acc:74.9500
Epoch:  1/15    Loss: 0.6427    Train Acc: 50.6320   Val Acc:74.2500
Epoch:  1/15    Loss: 0.7956    Train Acc: 51.4938   Val Acc:71.7000
Epoch:  1/15    Loss: 0.4628    Train Acc: 52.3984   Val Acc:76.7833
Epoch:  1/15    Loss: 0.6067    Train Acc: 53.4023   Val Acc:80.0167
Epoch:  1/15    Loss: 0.4638    Train Acc: 54.2878   Val Acc:75.6667
Epoch:  1/15    Loss: 0.3990    Train Acc: 55.1601   Val Acc:73.7333
Epoch:  1/15    Loss: 0.7557    Train Acc: 56.0069   Val Acc:80.2500
Epoch:  1/15    Loss: 0.6263    Train Acc: 56.7508   Val Acc:79.0500
Epoch:  1/15    Loss: 0.6094    Train Acc: 57.4791   Val Acc:78.7500
Epoch:  1/15    Loss: 0.7804    Train Acc: 58.1869   Val Acc:80.1000
Epoch:  1/15    Loss: 0.4871    Train Acc: 58.8640   Val Acc:80.5667
Epoch:  1/15    Loss: 0.7212    Train Acc: 59.4721   Val Acc:80.4167
Epoch:  1/15    Loss: 0.4960    Train Acc: 60.0342   Val Acc:77.3333
Epoch:  1/15    Loss: 0.6320    Train Acc: 60.5429   Val Acc:81.4167
Epoch:  1/15    Loss: 0.5370    Train Acc: 61.0566   Val Acc:79.3000
Epoch:  1/15    Loss: 0.4441    Train Acc: 61.5276   Val Acc:81.1667
Epoch:  1/15    Loss: 0.7616    Train Acc: 61.9847   Val Acc:81.1000
Epoch:  1/15    Loss: 0.4460    Train Acc: 62.5337   Val Acc:78.5500
```

```
Epoch: 15/15    Loss: 0.3498    Train Acc: 87.3731   Val Acc:87.6833
Epoch: 15/15    Loss: 0.1841    Train Acc: 87.3417   Val Acc:88.1000
Epoch: 15/15    Loss: 0.3480    Train Acc: 87.3154   Val Acc:87.4667
Epoch: 15/15    Loss: 0.2290    Train Acc: 87.2986   Val Acc:87.3000
Epoch: 15/15    Loss: 0.3844    Train Acc: 87.2997   Val Acc:87.1000
Epoch: 15/15    Loss: 0.2657    Train Acc: 87.3029   Val Acc:87.5000
Epoch: 15/15    Loss: 0.3868    Train Acc: 87.3233   Val Acc:87.1833
Epoch: 15/15    Loss: 0.2086    Train Acc: 87.3262   Val Acc:87.6500
Epoch: 15/15    Loss: 0.2162    Train Acc: 87.3396   Val Acc:87.7167
Epoch: 15/15    Loss: 0.4171    Train Acc: 87.3507   Val Acc:87.8500
Epoch: 15/15    Loss: 0.1939    Train Acc: 87.3450   Val Acc:87.0500
Epoch: 15/15    Loss: 0.1646    Train Acc: 87.3802   Val Acc:87.8500
Epoch: 15/15    Loss: 0.3341    Train Acc: 87.3946   Val Acc:87.2667
Epoch: 15/15    Loss: 0.5271    Train Acc: 87.3646   Val Acc:87.7833
Epoch: 15/15    Loss: 0.4056    Train Acc: 87.3867   Val Acc:86.7667
Epoch: 15/15    Loss: 0.4042    Train Acc: 87.3771   Val Acc:87.2333
Epoch: 15/15    Loss: 0.2095    Train Acc: 87.3928   Val Acc:86.5167
Epoch: 15/15    Loss: 0.3453    Train Acc: 87.3775   Val Acc:86.4833
Epoch: 15/15    Loss: 0.2697    Train Acc: 87.3758   Val Acc:87.8500
Epoch: 15/15    Loss: 0.5245    Train Acc: 87.3572   Val Acc:87.5667
Epoch: 15/15    Loss: 0.1450    Train Acc: 87.3856   Val Acc:87.4500
```

In [8]:
```
test_accuracy = validate_model(model_cnn,test_generator)
print(test_accuracy)
```

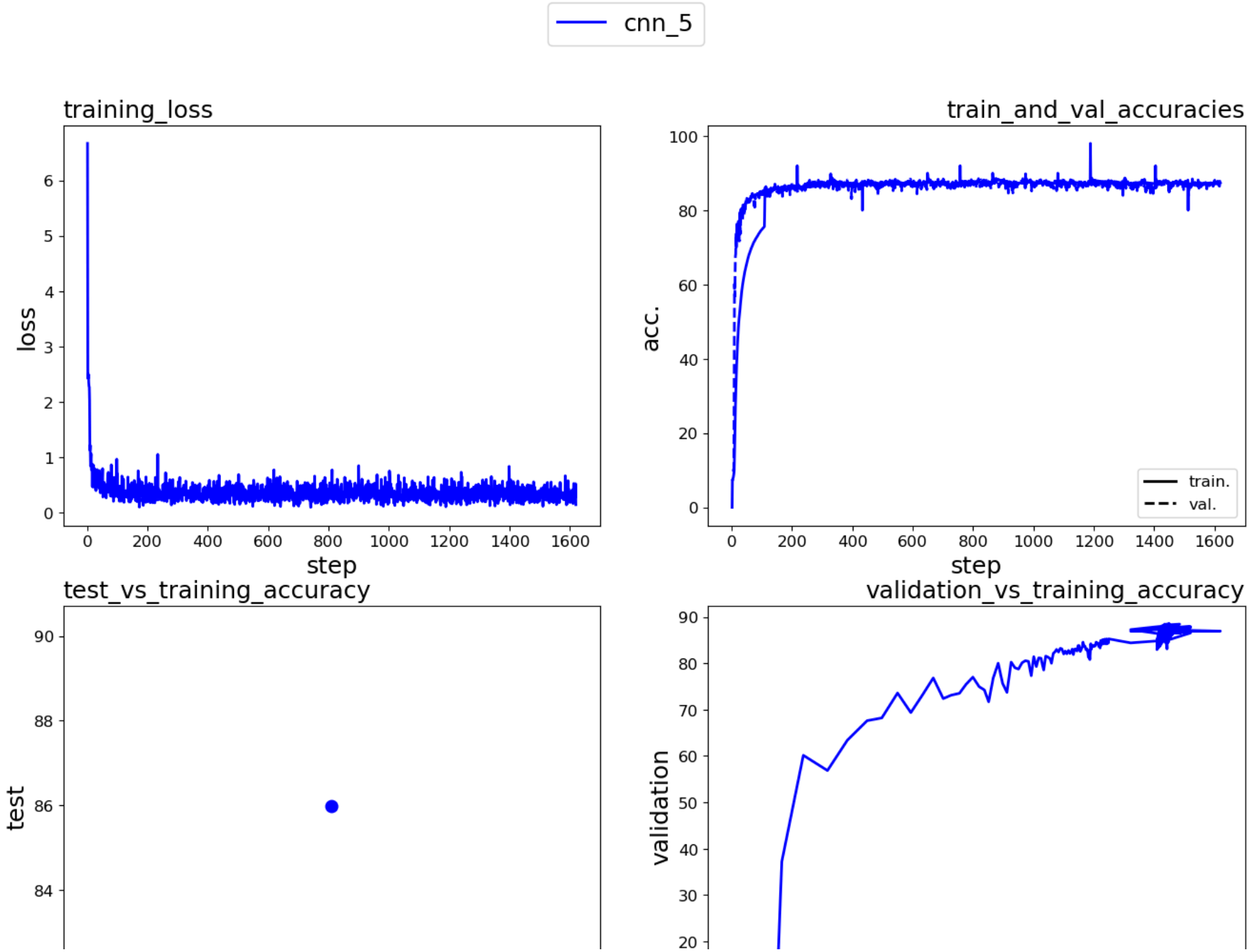
```
85.99
```

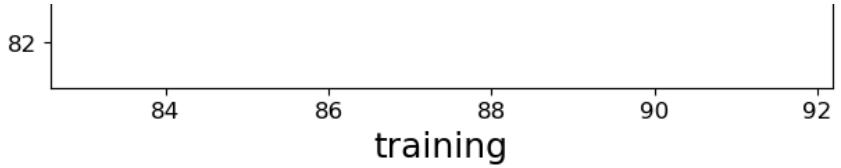
In [9]:
```
# Form the dictionary
architecture_name = "cnn_5"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

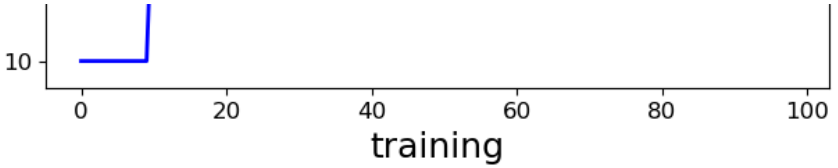
```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

```
Experiment results saved to part3_cnn_5.json
```

In [10]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```

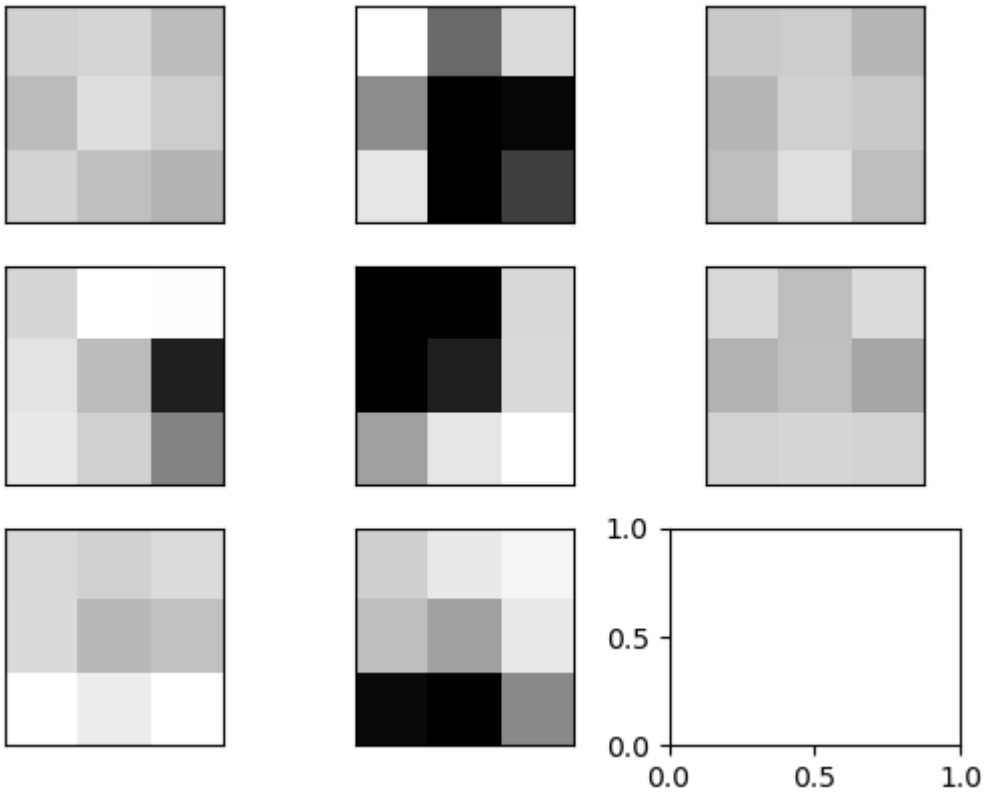

cnn_5
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [11]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```

```
/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1/Part3/utils.py:438: UserWarning: FigureCa
nvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



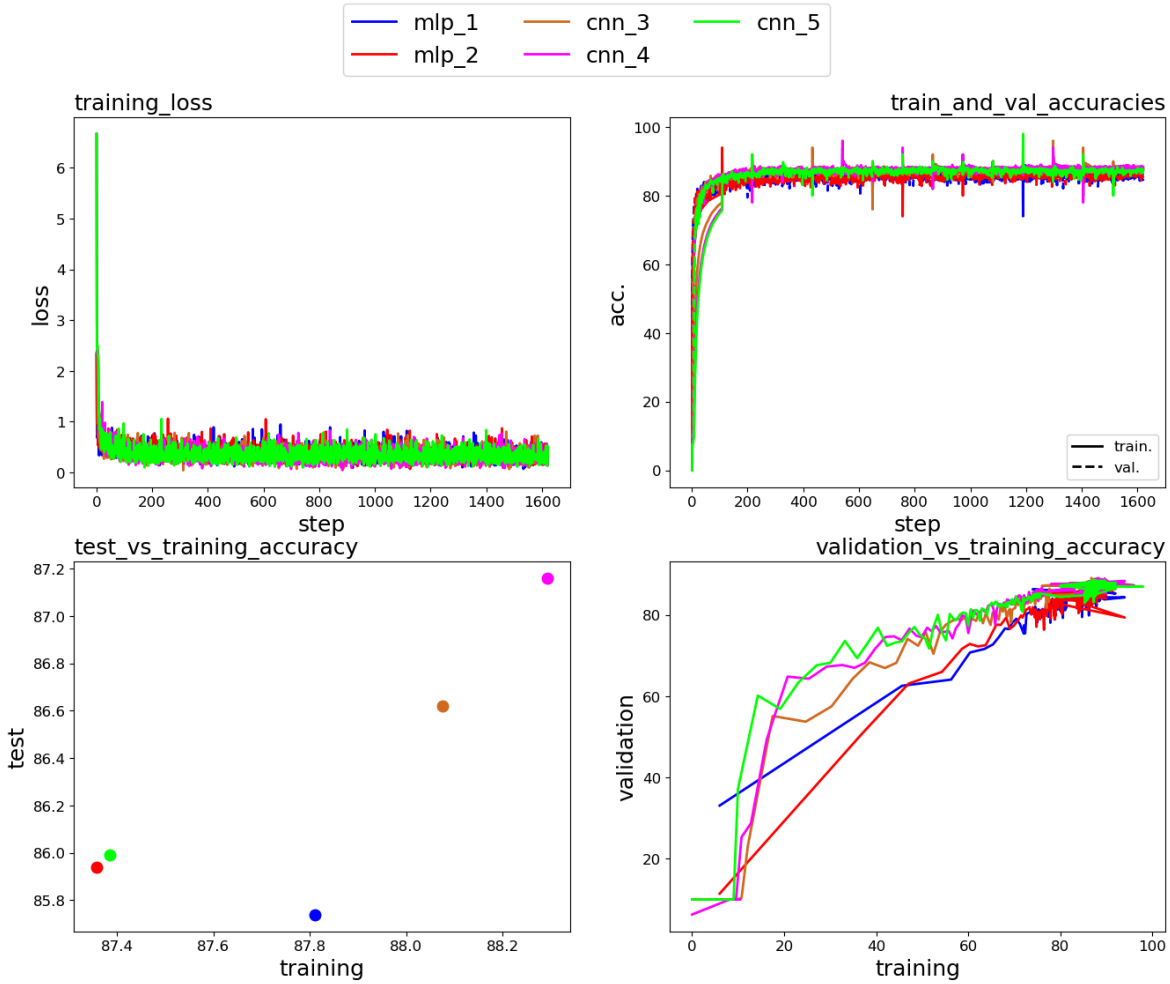
In [ ]:

In [8]:
```
import json
import utils as ut
```

In [9]:
```
architecture_name = ["mlp_1","mlp_2","cnn_3","cnn_4","cnn_5"]
```

In [27]:
```
file1 = "part3_mlp_1.json"
file2 = "part3_mlp_2.json"
file3 = "part3_cnn_3.json"
file4 = "part3_cnn_4.json"
file5 = "part3_cnn_5.json"

with open(file1) as user_file:
    data1 = json.load(user_file)

with open(file2) as user_file:
    data2 = json.load(user_file)

with open(file3) as user_file:
    data3 = json.load(user_file)

with open(file4) as user_file:
    data4 = json.load(user_file)

with open(file5) as user_file:
    data5 = json.load(user_file)
```

In [28]:
```
liste = [data1, data2, data3, data4, data5]

ut.part3Plots(liste)
```


mlp_1
mlp_2
cnn_3
cnn_4
cnn_5
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
training
validation_vs_training_accuracy
validation
training

In [ ]:

In [15]:
```
import torchvision
import torch
import torch.nn.functional as F
import numpy as np
import json
import utils as ut
```

In [16]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [17]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
train_subset = torch.utils.data.Subset(train_data, train_indices)
```

```
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [18]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_mlp, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_mlp.fc1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [19]:
```
class mlp_1(torch.nn.Module):
    def __init__(self, input_size):
        super(mlp_1, self).__init__()
        self.input_size = input_size
        # Adjust the hidden layer size to 32
        self.fc1 = torch.nn.Linear(input_size, 32)  # FC-32
        self.fc2 = torch.nn.Linear(32, 10)  # Prediction layer, size = num_classes

    def forward(self, x):
        x = x.view(-1, self.input_size)  # Flatten the input if needed
        hidden = self.fc1(x)  # Pass through the first fully connected layer
        sigmoid_out = torch.sigmoid(hidden)  # Apply ReLU activation
        output = self.fc2(sigmoid_out)  # Pass through the prediction layer
        return output

model_mlp = mlp_1(784)
```

In [20]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_mlp.parameters(), lr = 0.01, momentum = 0.0)
```

In [21]:
```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_mlp,train_generator,val_generator)
```

```
Epoch:  1/15    Loss: 2.3258    Train Acc: 8.0000   Val Acc:14.3833
Epoch:  1/15    Loss: 1.5949    Train Acc: 39.2727   Val Acc:52.9667
Epoch:  1/15    Loss: 1.2702    Train Acc: 49.0476   Val Acc:61.2333
Epoch:  1/15    Loss: 1.0405    Train Acc: 55.9355   Val Acc:67.4500
Epoch:  1/15    Loss: 0.9264    Train Acc: 59.7561   Val Acc:70.4500
Epoch:  1/15    Loss: 0.7780    Train Acc: 62.0784   Val Acc:72.9667
Epoch:  1/15    Loss: 0.8603    Train Acc: 63.8033   Val Acc:70.9167
Epoch:  1/15    Loss: 0.7659    Train Acc: 65.3239   Val Acc:73.9333
Epoch:  1/15    Loss: 0.8346    Train Acc: 66.3951   Val Acc:72.3167
Epoch:  1/15    Loss: 0.7458    Train Acc: 67.0989   Val Acc:75.1167
Epoch:  1/15    Loss: 0.6164    Train Acc: 67.8416   Val Acc:75.9833
Epoch:  1/15    Loss: 0.5817    Train Acc: 68.8468   Val Acc:77.2167
Epoch:  1/15    Loss: 0.5587    Train Acc: 69.9835   Val Acc:76.3833
Epoch:  1/15    Loss: 0.7440    Train Acc: 70.6718   Val Acc:77.3833
Epoch:  1/15    Loss: 0.6704    Train Acc: 70.8369   Val Acc:78.6333
Epoch:  1/15    Loss: 0.5029    Train Acc: 71.3245   Val Acc:79.1667
Epoch:  1/15    Loss: 0.4278    Train Acc: 71.7888   Val Acc:77.9167
Epoch:  1/15    Loss: 0.4917    Train Acc: 72.2807   Val Acc:78.4833
Epoch:  1/15    Loss: 0.4854    Train Acc: 72.7514   Val Acc:79.0500
Epoch:  1/15    Loss: 0.6747    Train Acc: 73.0995   Val Acc:79.8833
Epoch:  1/15    Loss: 0.4257    Train Acc: 73.4030   Val Acc:79.8500
Epoch:  1/15    Loss: 0.3937    Train Acc: 73.8483   Val Acc:78.3000
Epoch:  1/15    Loss: 0.3969    Train Acc: 74.0905   Val Acc:79.9333
Epoch:  1/15    Loss: 0.5298    Train Acc: 74.3203   Val Acc:79.6167
Epoch:  1/15    Loss: 0.4636    Train Acc: 74.6556   Val Acc:80.8167
Epoch:  1/15    Loss: 0.5209    Train Acc: 74.9402   Val Acc:81.1167
Epoch:  1/15    Loss: 0.4025    Train Acc: 75.3180   Val Acc:80.0000
Epoch:  1/15    Loss: 0.5016    Train Acc: 75.5572   Val Acc:80.7167
Epoch:  1/15    Loss: 0.5319    Train Acc: 75.7722   Val Acc:81.0833
Epoch:  1/15    Loss: 0.5200    Train Acc: 76.0069   Val Acc:81.6000
Epoch:  1/15    Loss: 0.6812    Train Acc: 76.1927   Val Acc:81.1000
Epoch:  1/15    Loss: 0.4873    Train Acc: 76.3730   Val Acc:80.6500
Epoch:  1/15    Loss: 0.4688    Train Acc: 76.5421   Val Acc:81.3500
Epoch:  1/15    Loss: 0.5815    Train Acc: 76.6767   Val Acc:79.4833
Epoch:  1/15    Loss: 0.3827    Train Acc: 76.8035   Val Acc:79.5167
Epoch:  1/15    Loss: 0.4257    Train Acc: 76.9060   Val Acc:82.3833
Epoch:  1/15    Loss: 0.4062    Train Acc: 77.0526   Val Acc:79.6500
Epoch:  1/15    Loss: 0.4554    Train Acc: 77.1590   Val Acc:78.9167
Epoch:  1/15    Loss: 0.3727    Train Acc: 77.2231   Val Acc:81.4833
Epoch:  1/15    Loss: 0.7064    Train Acc: 77.3555   Val Acc:82.0833
Epoch:  1/15    Loss: 0.4013    Train Acc: 77.4015   Val Acc:81.3500
```

```
Epoch: 15/15    Loss: 0.2709    Train Acc: 87.6418   Val Acc:84.8833
Epoch: 15/15    Loss: 0.2964    Train Acc: 87.6617   Val Acc:85.7167
Epoch: 15/15    Loss: 0.2544    Train Acc: 87.6566   Val Acc:85.4500
Epoch: 15/15    Loss: 0.1960    Train Acc: 87.6471   Val Acc:84.5167
Epoch: 15/15    Loss: 0.3060    Train Acc: 87.6268   Val Acc:84.6333
Epoch: 15/15    Loss: 0.5122    Train Acc: 87.6395   Val Acc:85.6833
Epoch: 15/15    Loss: 0.5801    Train Acc: 87.6391   Val Acc:84.9667
Epoch: 15/15    Loss: 0.2776    Train Acc: 87.6302   Val Acc:85.0833
Epoch: 15/15    Loss: 0.4847    Train Acc: 87.5983   Val Acc:84.7000
Epoch: 15/15    Loss: 0.4120    Train Acc: 87.6067   Val Acc:86.0000
Epoch: 15/15    Loss: 0.2025    Train Acc: 87.6087   Val Acc:85.6667
Epoch: 15/15    Loss: 0.1396    Train Acc: 87.6453   Val Acc:85.4500
Epoch: 15/15    Loss: 0.2905    Train Acc: 87.6387   Val Acc:85.4333
Epoch: 15/15    Loss: 0.3878    Train Acc: 87.6104   Val Acc:85.0833
Epoch: 15/15    Loss: 0.5430    Train Acc: 87.6123   Val Acc:84.7167
Epoch: 15/15    Loss: 0.4664    Train Acc: 87.5886   Val Acc:85.5833
Epoch: 15/15    Loss: 0.1386    Train Acc: 87.6101   Val Acc:85.5500
Epoch: 15/15    Loss: 0.3744    Train Acc: 87.5927   Val Acc:85.4833
Epoch: 15/15    Loss: 0.3139    Train Acc: 87.5852   Val Acc:85.4000
Epoch: 15/15    Loss: 0.4321    Train Acc: 87.5589   Val Acc:84.9833
Epoch: 15/15    Loss: 0.2140    Train Acc: 87.5742   Val Acc:85.1833
```

In [22]:
```
test_accuracy = validate_model(model_mlp,test_generator)
print(test_accuracy)
```

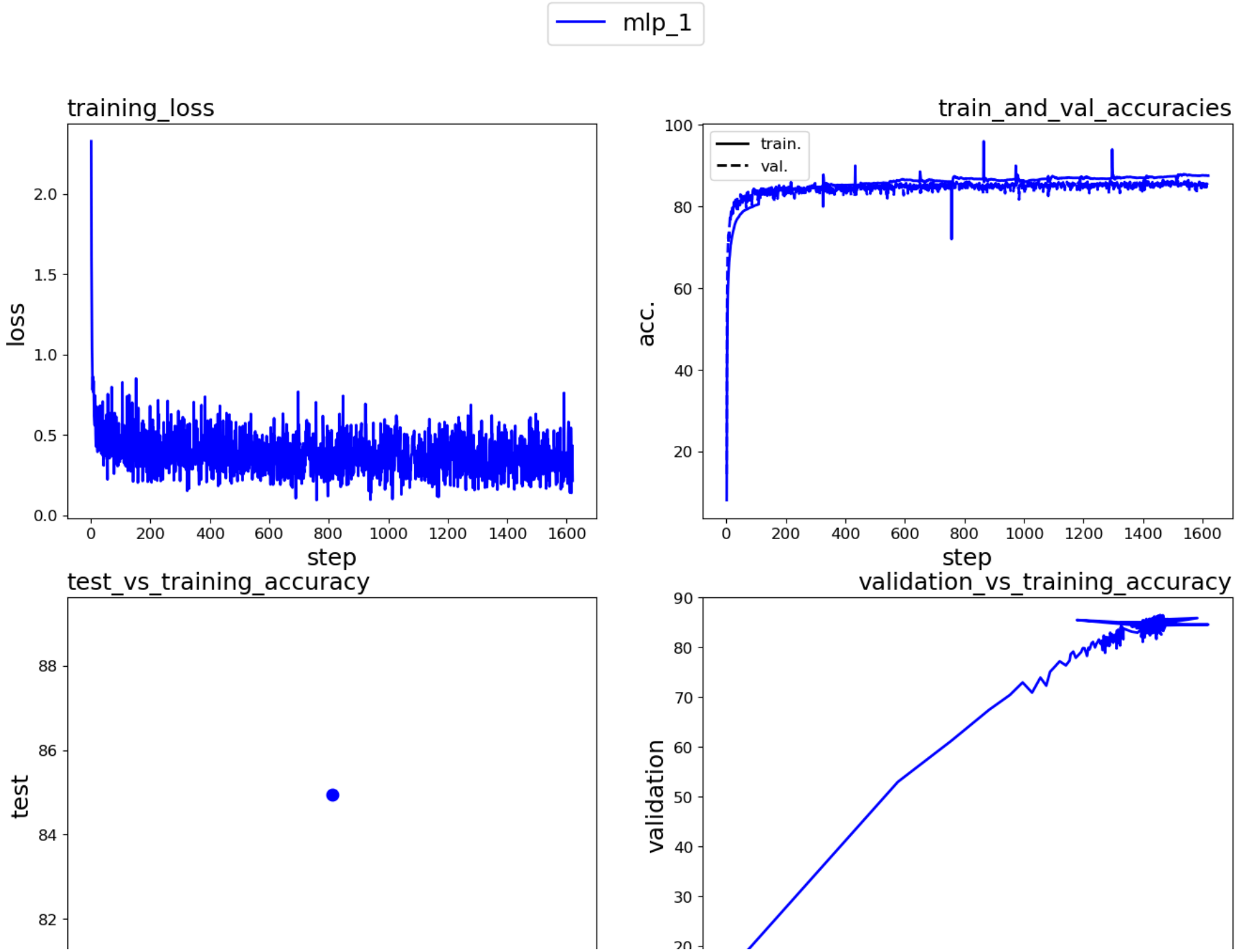
```
84.95
```

In [23]:
```
# Form the dictionary
architecture_name = "mlp_1"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

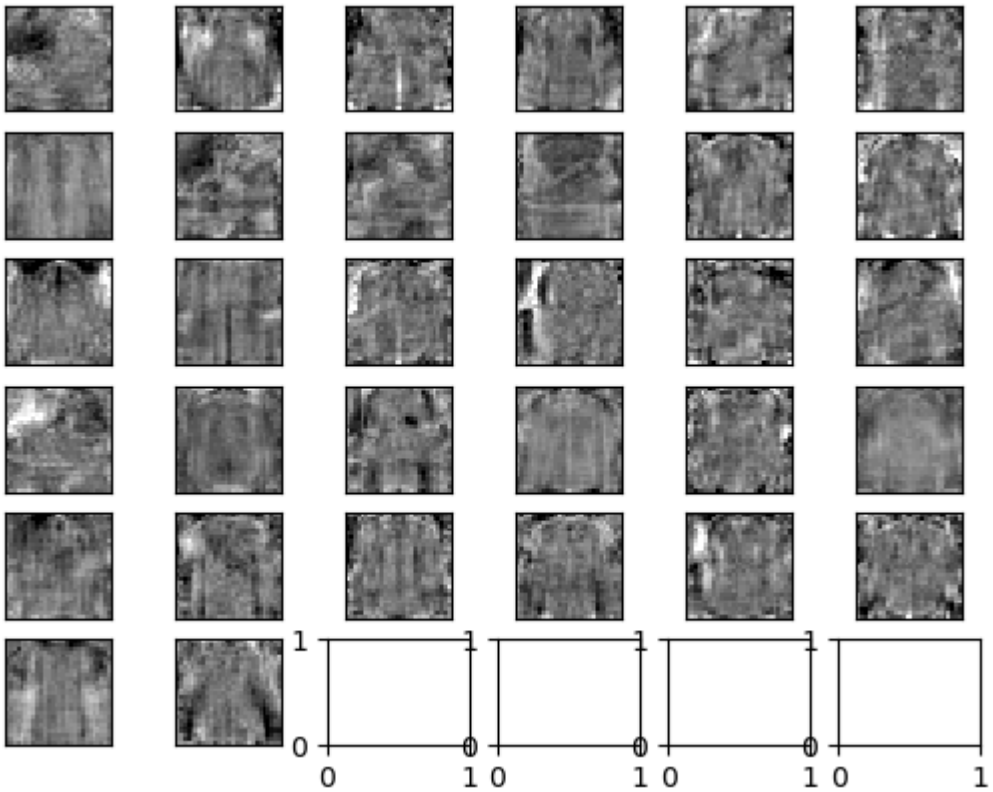
```
Experiment results saved to part3_mlp_1.json
```

In [24]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```


mlp_1
training_loss
loss
step
train_and_val_accuracies
train.
val.
acc.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation

In [25]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```

```
/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1/Part4/utils.py:438: UserWarning: FigureCa
nvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



In [ ]:

In [1]:
```
import torchvision
import torch
import torch.nn.functional as F
import numpy as np
import json
import utils as ut
```

In [2]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [3]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
train_subset = torch.utils.data.Subset(train_data, train_indices)
```

```
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [4]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_mlp, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_mlp.fc1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [5]:
```
class mlp_2(torch.nn.Module):
    def __init__(self, input_size):
        super(mlp_2, self).__init__()
        self.input_size = input_size
        self.fc1 = torch.nn.Linear(input_size, 32)  # FC-32
        self.fc2 = torch.nn.Linear(32, 64)
        self.fc3 = torch.nn.Linear(64, 10)  # Prediction layer, number of classes = 10

    def forward(self, x):
        x = x.view(-1, self.input_size)  # Flatten the input if needed
        hidden = self.fc1(x)  # Pass through the first fully connected layer
        sigmoid_out = torch.sigmoid(hidden)  # Apply Sigmoid activation
        hidden = self.fc2(sigmoid_out)  # Pass through the second fully connected layer
        sigmoid_out = torch.sigmoid(hidden)  # Apply Sigmoid activation again
        output = self.fc3(sigmoid_out)  # Pass through the prediction layer
        return output

# Instantiate the model with the input size for a flattened 28x28 image
model_mlp = mlp_2(784)
```

In [6]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_mlp.parameters(), lr = 0.01, momentum = 0.0)
```

In [7]:
```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_mlp,train_generator,val_generator)
```

```
Epoch:  1/15    Loss: 2.3449    Train Acc: 8.0000   Val Acc:10.0000
Epoch:  1/15    Loss: 2.1449    Train Acc: 18.1818   Val Acc:28.5167
Epoch:  1/15    Loss: 1.7426    Train Acc: 23.4286   Val Acc:32.6333
Epoch:  1/15    Loss: 1.4097    Train Acc: 28.2581   Val Acc:59.5000
Epoch:  1/15    Loss: 1.2266    Train Acc: 35.6098   Val Acc:54.7833
Epoch:  1/15    Loss: 1.1134    Train Acc: 40.3137   Val Acc:54.6833
Epoch:  1/15    Loss: 0.9325    Train Acc: 43.6721   Val Acc:59.1333
Epoch:  1/15    Loss: 0.9454    Train Acc: 46.4225   Val Acc:65.0000
Epoch:  1/15    Loss: 0.9147    Train Acc: 48.7654   Val Acc:67.3167
Epoch:  1/15    Loss: 0.8176    Train Acc: 50.2198   Val Acc:69.4500
Epoch:  1/15    Loss: 0.3967    Train Acc: 52.4158   Val Acc:68.3167
Epoch:  1/15    Loss: 0.8152    Train Acc: 53.8559   Val Acc:71.8833
Epoch:  1/15    Loss: 0.5926    Train Acc: 55.5207   Val Acc:72.8167
Epoch:  1/15    Loss: 0.5270    Train Acc: 56.8092   Val Acc:72.5833
Epoch:  1/15    Loss: 0.7945    Train Acc: 57.9007   Val Acc:74.4000
Epoch:  1/15    Loss: 0.6158    Train Acc: 58.9272   Val Acc:74.3167
Epoch:  1/15    Loss: 0.5999    Train Acc: 59.7764   Val Acc:74.1500
Epoch:  1/15    Loss: 0.6454    Train Acc: 60.5965   Val Acc:75.2000
Epoch:  1/15    Loss: 0.6836    Train Acc: 61.4807   Val Acc:75.4333
Epoch:  1/15    Loss: 0.6765    Train Acc: 62.1466   Val Acc:75.4667
Epoch:  1/15    Loss: 0.4212    Train Acc: 62.8557   Val Acc:75.3833
Epoch:  1/15    Loss: 0.4632    Train Acc: 63.6114   Val Acc:77.1000
Epoch:  1/15    Loss: 0.5439    Train Acc: 64.0905   Val Acc:76.0333
Epoch:  1/15    Loss: 0.6254    Train Acc: 64.5368   Val Acc:76.7667
Epoch:  1/15    Loss: 0.6216    Train Acc: 65.1203   Val Acc:77.7667
Epoch:  1/15    Loss: 0.7007    Train Acc: 65.5936   Val Acc:76.2000
Epoch:  1/15    Loss: 0.6816    Train Acc: 66.0996   Val Acc:76.2500
Epoch:  1/15    Loss: 0.7302    Train Acc: 66.4945   Val Acc:77.2667
Epoch:  1/15    Loss: 0.5065    Train Acc: 67.0178   Val Acc:78.0500
Epoch:  1/15    Loss: 0.7141    Train Acc: 67.4914   Val Acc:76.3167
Epoch:  1/15    Loss: 0.4849    Train Acc: 67.8339   Val Acc:78.8500
Epoch:  1/15    Loss: 0.6327    Train Acc: 68.0193   Val Acc:73.8167
Epoch:  1/15    Loss: 0.5330    Train Acc: 68.3863   Val Acc:81.0500
Epoch:  1/15    Loss: 0.6972    Train Acc: 68.6647   Val Acc:79.5833
Epoch:  1/15    Loss: 0.3398    Train Acc: 69.0029   Val Acc:79.8167
Epoch:  1/15    Loss: 0.4780    Train Acc: 69.3105   Val Acc:81.7167
Epoch:  1/15    Loss: 0.4331    Train Acc: 69.6122   Val Acc:80.9333
Epoch:  1/15    Loss: 0.3482    Train Acc: 69.9515   Val Acc:80.4833
Epoch:  1/15    Loss: 0.5702    Train Acc: 70.2362   Val Acc:78.9167
Epoch:  1/15    Loss: 0.6775    Train Acc: 70.4757   Val Acc:77.9500
Epoch:  1/15    Loss: 0.6450    Train Acc: 70.6683   Val Acc:81.4333
```

```
Epoch: 15/15    Loss: 0.3283    Train Acc: 87.6257   Val Acc:87.1000
Epoch: 15/15    Loss: 0.3679    Train Acc: 87.6095   Val Acc:87.3333
Epoch: 15/15    Loss: 0.3683    Train Acc: 87.6162   Val Acc:86.7500
Epoch: 15/15    Loss: 0.4308    Train Acc: 87.5871   Val Acc:87.3500
Epoch: 15/15    Loss: 0.2566    Train Acc: 87.5851   Val Acc:87.3667
Epoch: 15/15    Loss: 0.1970    Train Acc: 87.5896   Val Acc:87.6500
Epoch: 15/15    Loss: 0.2779    Train Acc: 87.6004   Val Acc:87.4667
Epoch: 15/15    Loss: 0.2196    Train Acc: 87.6217   Val Acc:87.3333
Epoch: 15/15    Loss: 0.2719    Train Acc: 87.6257   Val Acc:87.3167
Epoch: 15/15    Loss: 0.4000    Train Acc: 87.6400   Val Acc:86.9833
Epoch: 15/15    Loss: 0.2815    Train Acc: 87.6725   Val Acc:86.3833
Epoch: 15/15    Loss: 0.3525    Train Acc: 87.6718   Val Acc:86.3333
Epoch: 15/15    Loss: 0.3959    Train Acc: 87.6468   Val Acc:86.3167
Epoch: 15/15    Loss: 0.4962    Train Acc: 87.6084   Val Acc:87.2500
Epoch: 15/15    Loss: 0.2202    Train Acc: 87.6004   Val Acc:87.6167
Epoch: 15/15    Loss: 0.3240    Train Acc: 87.6239   Val Acc:87.2500
Epoch: 15/15    Loss: 0.3925    Train Acc: 87.6450   Val Acc:87.1667
Epoch: 15/15    Loss: 0.3079    Train Acc: 87.6369   Val Acc:87.5833
Epoch: 15/15    Loss: 0.2291    Train Acc: 87.6594   Val Acc:87.5167
Epoch: 15/15    Loss: 0.3784    Train Acc: 87.6588   Val Acc:86.6667
Epoch: 15/15    Loss: 0.2875    Train Acc: 87.6433   Val Acc:86.6333
```

In [8]:
```
test_accuracy = validate_model(model_mlp,test_generator)
print(test_accuracy)
```

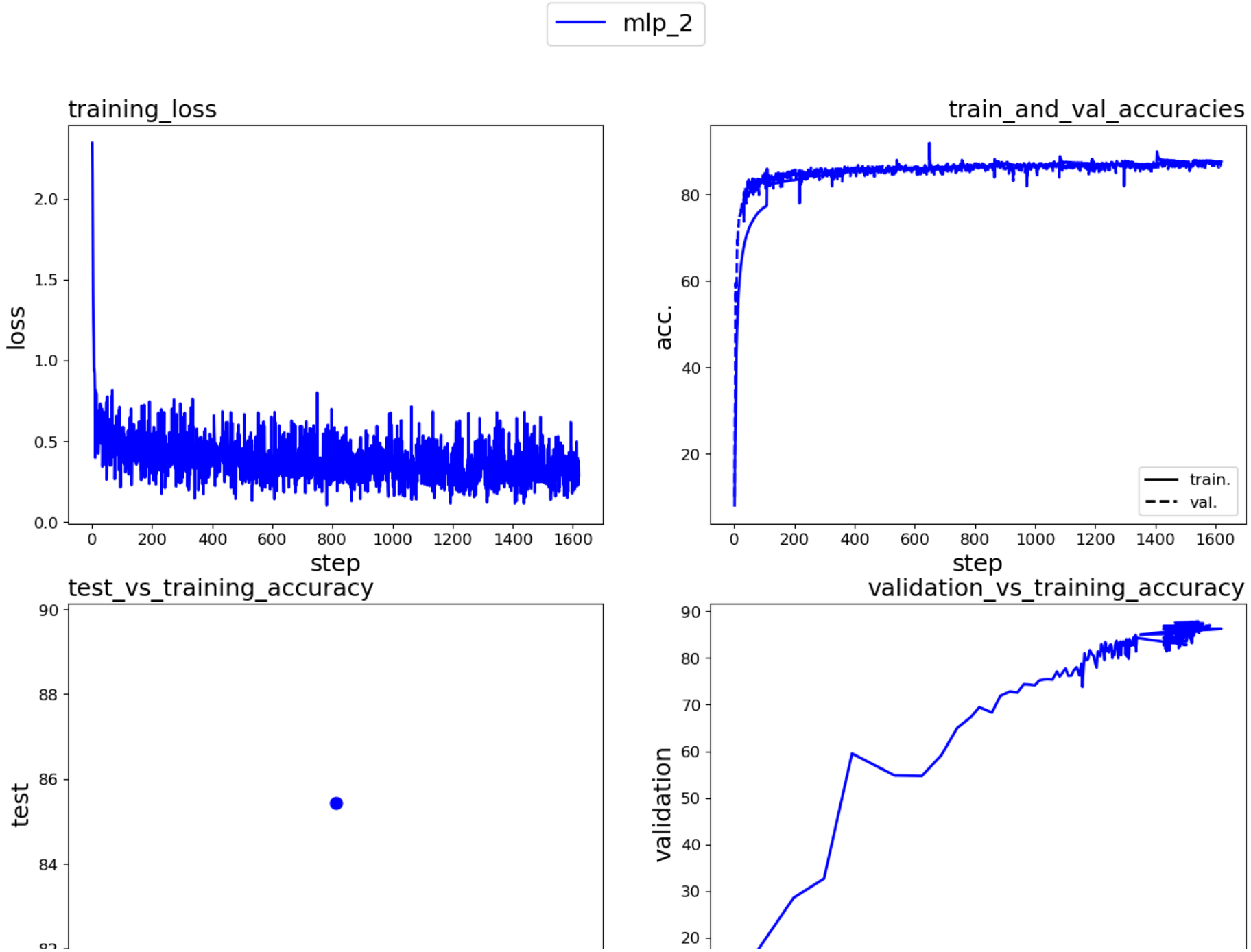
```
85.44
```

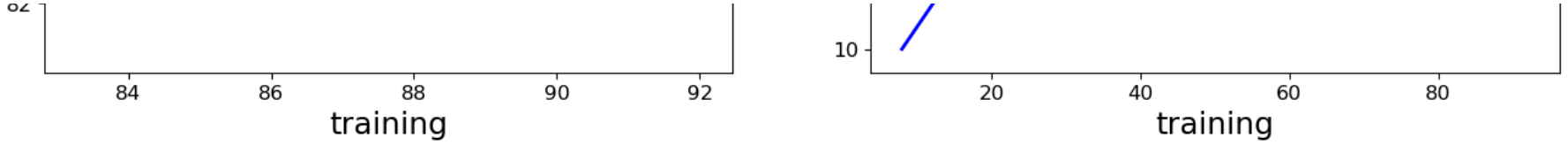
In [9]:
```
# Form the dictionary
architecture_name = "mlp_2"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

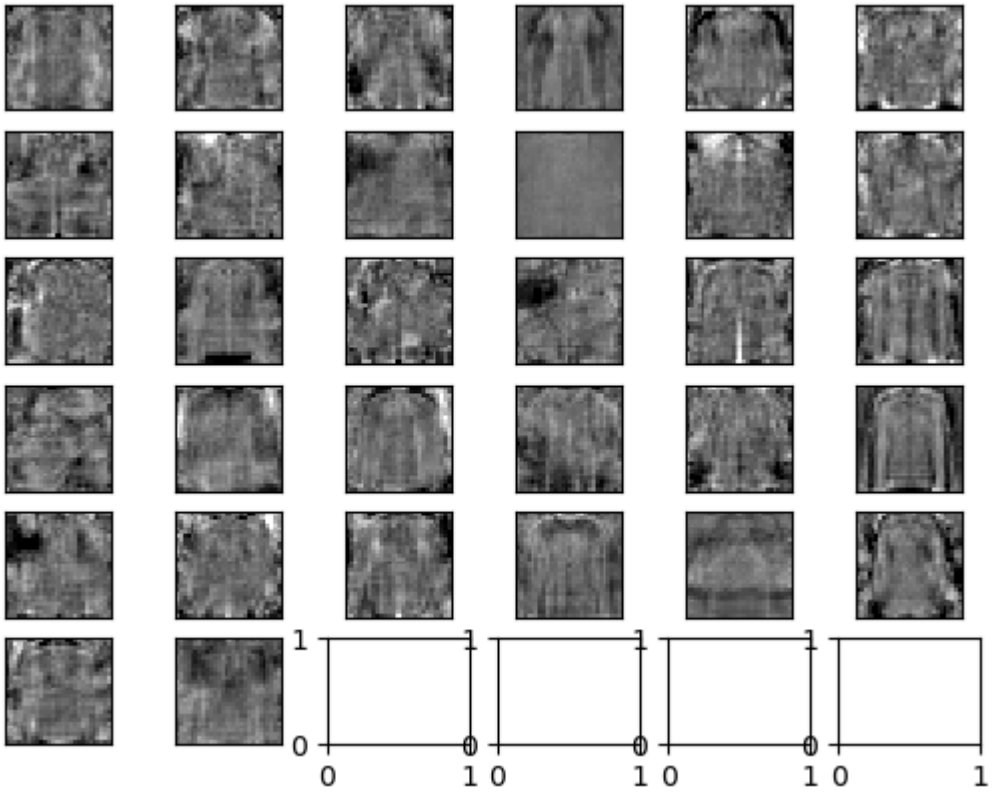
```
Experiment results saved to part3_mlp_2.json
```

In [10]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```


mlp_2
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation



In [12]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```



In [ ]:
```

```

In [13]:
```
import torchvision
import torch
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import json
import utils as ut
```

In [14]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [15]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
```

```
train_subset = torch.utils.data.Subset(train_data, train_indices)
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [16]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, train_acc, val_acc = [], [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_cnn, val_generator)
                val_acc.append(val_accuracy)

                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}".format(epo

    first_layer_weights = model_cnn.conv1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss
```

```
def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [17]:
```
class cnn_3(nn.Module):
    def __init__(self, num_classes=10):
        super(cnn_3, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=5, padding=2)
        self.conv3 = nn.Conv2d(8, 16, kernel_size=7, padding=3)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc = nn.Linear(16 * 7 * 7, num_classes)

    def forward(self, x):
        x = torch.sigmoid(self.conv1(x))  # Apply Sigmoid after first convolution
        x = torch.sigmoid(self.conv2(x))  # Apply Sigmoid after second convolution
        x = self.pool(x)  # Apply max pooling
        x = torch.sigmoid(self.conv3(x))  # Apply Sigmoid after third convolution
        x = self.pool(x)  # Apply max pooling
        x = x.view(-1, 16 * 7 * 7)  # Flatten the output
        x = self.fc(x)  # Pass through the prediction layer
        return x


model_cnn = cnn_3(784)
```

In [18]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_cnn.parameters(), lr = 0.01, momentum = 0.0)
```

In [19]:
```
first_layer_weights,train_acc,val_acc,train_loss = train_model(model_cnn,train_generator,val_generator)
```

```
Epoch:  1/15    Loss: 6.8773    Train Acc: 0.0000   Val Acc:10.0000
Epoch:  1/15    Loss: 3.2265    Train Acc: 11.2727   Val Acc:10.0000
Epoch:  1/15    Loss: 2.9161    Train Acc: 10.4762   Val Acc:10.0000
Epoch:  1/15    Loss: 2.4459    Train Acc: 10.2581   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3514    Train Acc: 10.5366   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3756    Train Acc: 10.2745   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3039    Train Acc: 10.6230   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3161    Train Acc: 10.5070   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3395    Train Acc: 10.3704   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3565    Train Acc: 10.3956   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2807    Train Acc: 10.6139   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3231    Train Acc: 10.7207   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3078    Train Acc: 10.8264   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3365    Train Acc: 10.8092   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3502    Train Acc: 11.0071   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2648    Train Acc: 11.1391   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3766    Train Acc: 10.8571   Val Acc:10.0000
Epoch:  1/15    Loss: 2.4099    Train Acc: 10.7719   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3214    Train Acc: 10.6630   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3698    Train Acc: 10.6806   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2947    Train Acc: 10.5871   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3085    Train Acc: 10.5308   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3365    Train Acc: 10.5068   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3615    Train Acc: 10.4242   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3222    Train Acc: 10.3900   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3988    Train Acc: 10.4143   Val Acc:10.1333
Epoch:  1/15    Loss: 2.1226    Train Acc: 10.4981   Val Acc:28.1833
Epoch:  1/15    Loss: 1.5513    Train Acc: 11.5424   Val Acc:48.2667
Epoch:  1/15    Loss: 1.1976    Train Acc: 12.9822   Val Acc:60.8167
Epoch:  1/15    Loss: 0.9280    Train Acc: 14.9278   Val Acc:67.5500
Epoch:  1/15    Loss: 0.7278    Train Acc: 16.6844   Val Acc:68.7000
Epoch:  1/15    Loss: 0.8318    Train Acc: 18.3601   Val Acc:72.5500
Epoch:  1/15    Loss: 0.7562    Train Acc: 19.9502   Val Acc:70.9333
Epoch:  1/15    Loss: 0.7386    Train Acc: 21.4985   Val Acc:71.7500
Epoch:  1/15    Loss: 0.7149    Train Acc: 22.9384   Val Acc:70.2667
Epoch:  1/15    Loss: 0.7419    Train Acc: 24.3818   Val Acc:72.5500
Epoch:  1/15    Loss: 0.7581    Train Acc: 25.7507   Val Acc:74.0000
Epoch:  1/15    Loss: 0.4310    Train Acc: 27.1267   Val Acc:70.8667
Epoch:  1/15    Loss: 0.5188    Train Acc: 28.5039   Val Acc:74.1833
Epoch:  1/15    Loss: 1.2128    Train Acc: 29.5499   Val Acc:71.8000
Epoch:  1/15    Loss: 0.5641    Train Acc: 30.6534   Val Acc:75.6833
```

```
Epoch: 15/15    Loss: 0.1874    Train Acc: 92.2641   Val Acc:89.5000
Epoch: 15/15    Loss: 0.3414    Train Acc: 92.2565   Val Acc:89.2500
Epoch: 15/15    Loss: 0.1719    Train Acc: 92.2379   Val Acc:89.5833
Epoch: 15/15    Loss: 0.1345    Train Acc: 92.2442   Val Acc:88.9167
Epoch: 15/15    Loss: 0.1461    Train Acc: 92.2503   Val Acc:89.1333
Epoch: 15/15    Loss: 0.2483    Train Acc: 92.2410   Val Acc:90.0333
Epoch: 15/15    Loss: 0.2124    Train Acc: 92.2449   Val Acc:89.1333
Epoch: 15/15    Loss: 0.5292    Train Acc: 92.2232   Val Acc:89.3833
Epoch: 15/15    Loss: 0.2119    Train Acc: 92.2166   Val Acc:89.0000
Epoch: 15/15    Loss: 0.2476    Train Acc: 92.1956   Val Acc:89.4833
Epoch: 15/15    Loss: 0.1154    Train Acc: 92.2019   Val Acc:88.7167
Epoch: 15/15    Loss: 0.3085    Train Acc: 92.2120   Val Acc:88.5333
Epoch: 15/15    Loss: 0.1309    Train Acc: 92.2079   Val Acc:89.0167
Epoch: 15/15    Loss: 0.1463    Train Acc: 92.2238   Val Acc:89.3167
Epoch: 15/15    Loss: 0.2704    Train Acc: 92.2057   Val Acc:88.9333
Epoch: 15/15    Loss: 0.3814    Train Acc: 92.2214   Val Acc:89.0667
Epoch: 15/15    Loss: 0.1674    Train Acc: 92.2114   Val Acc:88.7833
Epoch: 15/15    Loss: 0.4452    Train Acc: 92.1806   Val Acc:89.2000
Epoch: 15/15    Loss: 0.1852    Train Acc: 92.1618   Val Acc:88.5333
Epoch: 15/15    Loss: 0.2488    Train Acc: 92.1546   Val Acc:88.6167
Epoch: 15/15    Loss: 0.2319    Train Acc: 92.1382   Val Acc:88.7667
```

In [20]:
```
test_accuracy = validate_model(model_cnn,test_generator)
print(test_accuracy)
```

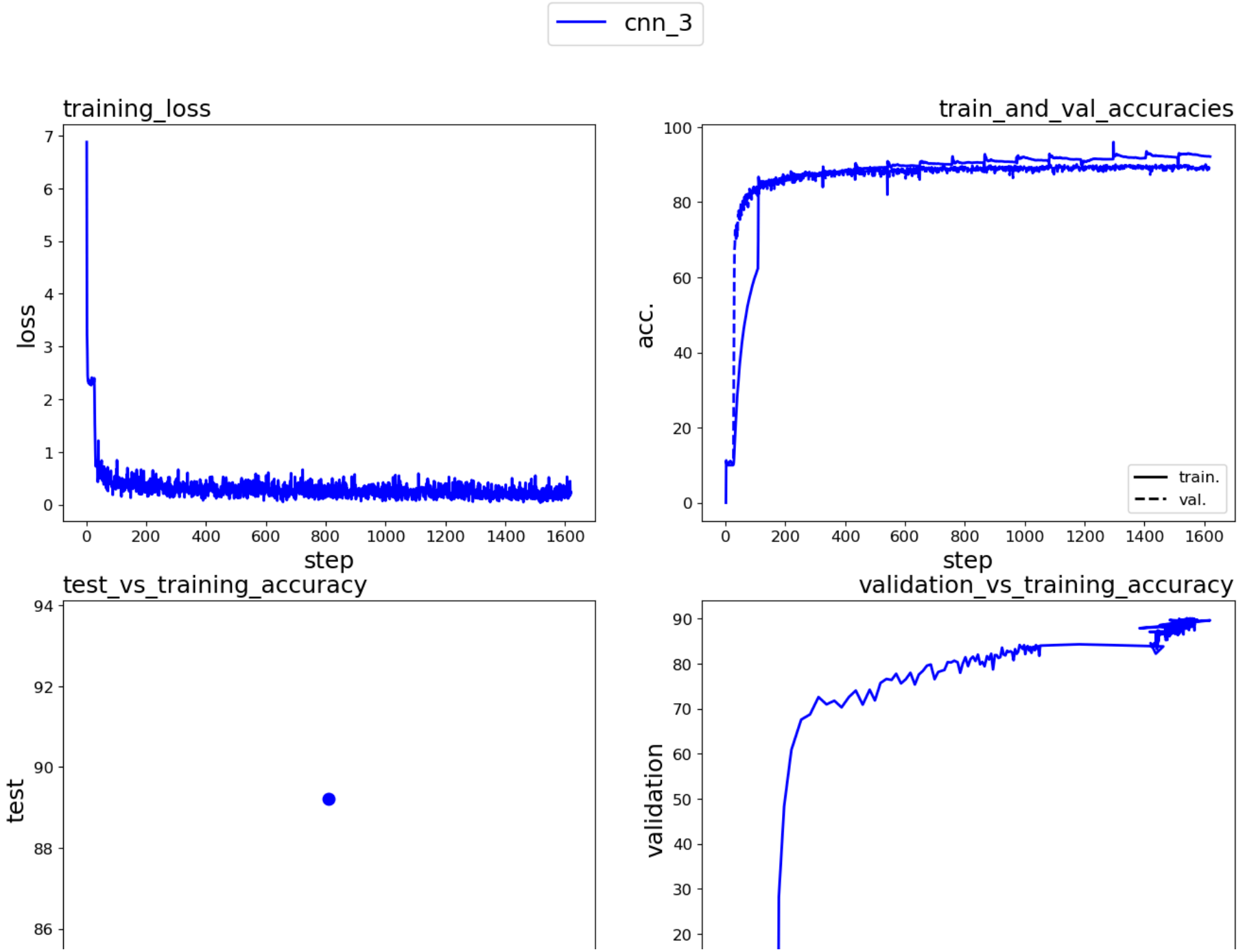
```
89.22
```

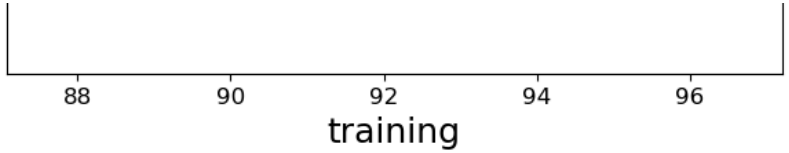
In [21]:
```
# Form the dictionary
architecture_name = "cnn_3"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

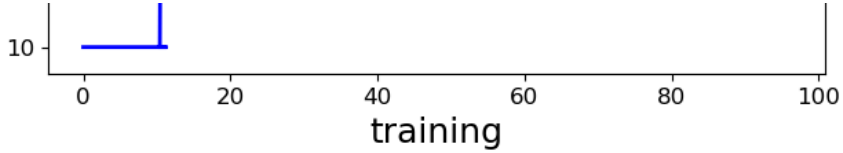
```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

```
Experiment results saved to part3_cnn_3.json
```

In [22]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```

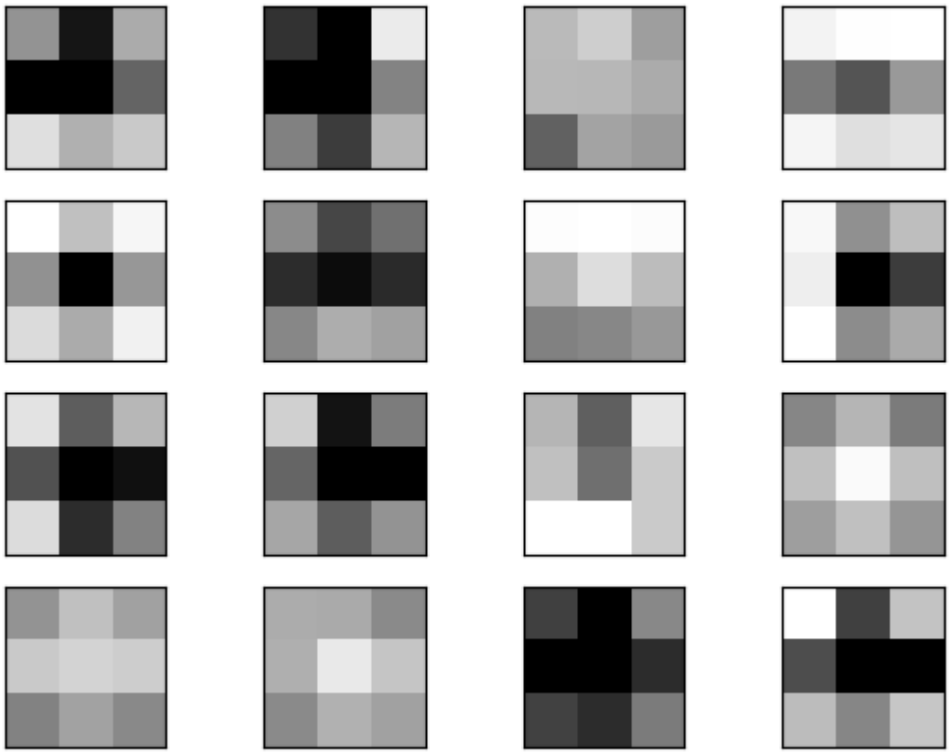

cnn_3
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [23]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```

```
/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1/Part4/utils.py:438: UserWarning: FigureCa
nvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



In [ ]:
```
```

In [8]:
```
import torchvision
import torch
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import json
import utils as ut
```

In [9]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [10]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
```

```
train_subset = torch.utils.data.Subset(train_data, train_indices)
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [11]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss = []
    grad_magnitudes = []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            grad_magnitude = model.conv1.weight.grad.norm().item()
            grad_magnitudes.append(grad_magnitude)  # Store the magnitude

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                #train_accuracy = 100 * correct / total
                #train_acc.append(train_accuracy)
                #val_accuracy = validate_model(model_cnn, val_generator)
                #val_acc.append(val_accuracy)

                #print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}    Val Acc:{4:2.4f}".format(ep
```

```
                print("train loss: {0:2.4f}    grad: {1:2.4f}".format(loss,grad_magnitude))
    #first_layer_weights = model_cnn.conv1.weight.data.cpu().numpy()
    return train_loss,grad_magnitudes


def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [12]:

```
class cnn_4(nn.Module):
    def __init__(self, num_classes=10):
        super(cnn_4, self).__init__()
        # Convolutional and Pooling layers
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.conv3 = nn.Conv2d(8, 16, kernel_size=5, padding=2)  # Conv-5x5x16
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool-2x2
        self.conv4 = nn.Conv2d(16, 16, kernel_size=5, padding=2)  # Conv-5x5x16
        self.fc = nn.Linear(16 * 7 * 7, num_classes)  # Prediction layer

    def forward(self, x):
        x = torch.sigmoid(self.conv1(x))  # Conv-3x3x16 followed by Sigmoid
        x = torch.sigmoid(self.conv2(x))  # Conv-3x3x8 followed by Sigmoid
        x = torch.sigmoid(self.conv3(x))  # Conv-5x5x16 followed by Sigmoid
        x = self.pool(x)  # MaxPool-2x2
        x = torch.sigmoid(self.conv4(x))  # Conv-5x5x16 followed by Sigmoid
        x = self.pool(x)  # MaxPool-2x2 again
        x = x.view(-1, 16 * 7 * 7)  # Flatten the output for the fully connected layer
        x = self.fc(x)  # Prediction layer
        return x
```

```
# Instantiate the model
model_cnn = cnn_4(num_classes=10)
```

In [13]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_cnn.parameters(), lr = 0.01, momentum = 0.0)
```

In [14]:
```
train_loss,grad = train_model(model_cnn,train_generator,val_generator)
```

```
Epoch:  1/15    Loss: 2.3556    Train Acc: 6.0000   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3004    Train Acc: 13.0909   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3050    Train Acc: 12.0000   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3065    Train Acc: 11.0968   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2992    Train Acc: 10.3415   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3003    Train Acc: 10.2745   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3110    Train Acc: 10.5574   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3085    Train Acc: 10.4507   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3052    Train Acc: 10.3951   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3049    Train Acc: 10.1758   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3125    Train Acc: 10.1584   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3158    Train Acc: 10.2703   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3115    Train Acc: 10.2479   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3028    Train Acc: 10.0611   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2949    Train Acc: 10.2128   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2967    Train Acc: 10.2252   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2949    Train Acc: 10.1739   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3051    Train Acc: 10.2105   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3076    Train Acc: 10.1326   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2971    Train Acc: 10.0942   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2939    Train Acc: 10.1891   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3046    Train Acc: 10.1043   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3021    Train Acc: 10.0633   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3073    Train Acc: 10.0519   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3006    Train Acc: 10.0830   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2939    Train Acc: 10.0876   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3060    Train Acc: 10.0613   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3026    Train Acc: 10.0221   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2988    Train Acc: 10.0427   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3029    Train Acc: 10.0275   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2986    Train Acc: 10.0266   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3070    Train Acc: 9.9743   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3044    Train Acc: 9.9128   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3009    Train Acc: 9.9154   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3003    Train Acc: 9.9179   Val Acc:10.0000
Epoch:  1/15    Loss: 2.2989    Train Acc: 9.9031   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3035    Train Acc: 9.8781   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3040    Train Acc: 9.8976   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3071    Train Acc: 9.9370   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3120    Train Acc: 9.8721   Val Acc:10.0000
Epoch:  1/15    Loss: 2.3003    Train Acc: 9.8454   Val Acc:10.0000
```

```
Epoch: 15/15    Loss: 2.2925    Train Acc: 10.0804   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3151    Train Acc: 10.0613   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3011    Train Acc: 10.0539   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3064    Train Acc: 10.0488   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3071    Train Acc: 10.0351   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3031    Train Acc: 10.0239   Val Acc:10.0000
Epoch: 15/15    Loss: 2.2999    Train Acc: 10.0430   Val Acc:10.0000
Epoch: 15/15    Loss: 2.2969    Train Acc: 10.0531   Val Acc:10.0000
Epoch: 15/15    Loss: 2.2985    Train Acc: 10.0189   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3077    Train Acc: 10.0187   Val Acc:10.0000
Epoch: 15/15    Loss: 2.2894    Train Acc: 10.0227   Val Acc:10.0000
Epoch: 15/15    Loss: 2.2999    Train Acc: 10.0163   Val Acc:10.0000
Epoch: 15/15    Loss: 2.2924    Train Acc: 10.0061   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3102    Train Acc: 9.9940   Val Acc:10.0000
Epoch: 15/15    Loss: 2.2988    Train Acc: 9.9782   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3007    Train Acc: 9.9843   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3154    Train Acc: 9.9806   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3067    Train Acc: 9.9846   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3045    Train Acc: 10.0152   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3135    Train Acc: 10.0075   Val Acc:10.0000
Epoch: 15/15    Loss: 2.3103    Train Acc: 10.0131   Val Acc:10.0000
```

In [15]:
```
test_accuracy = validate_model(model_cnn,test_generator)
print(test_accuracy)
```

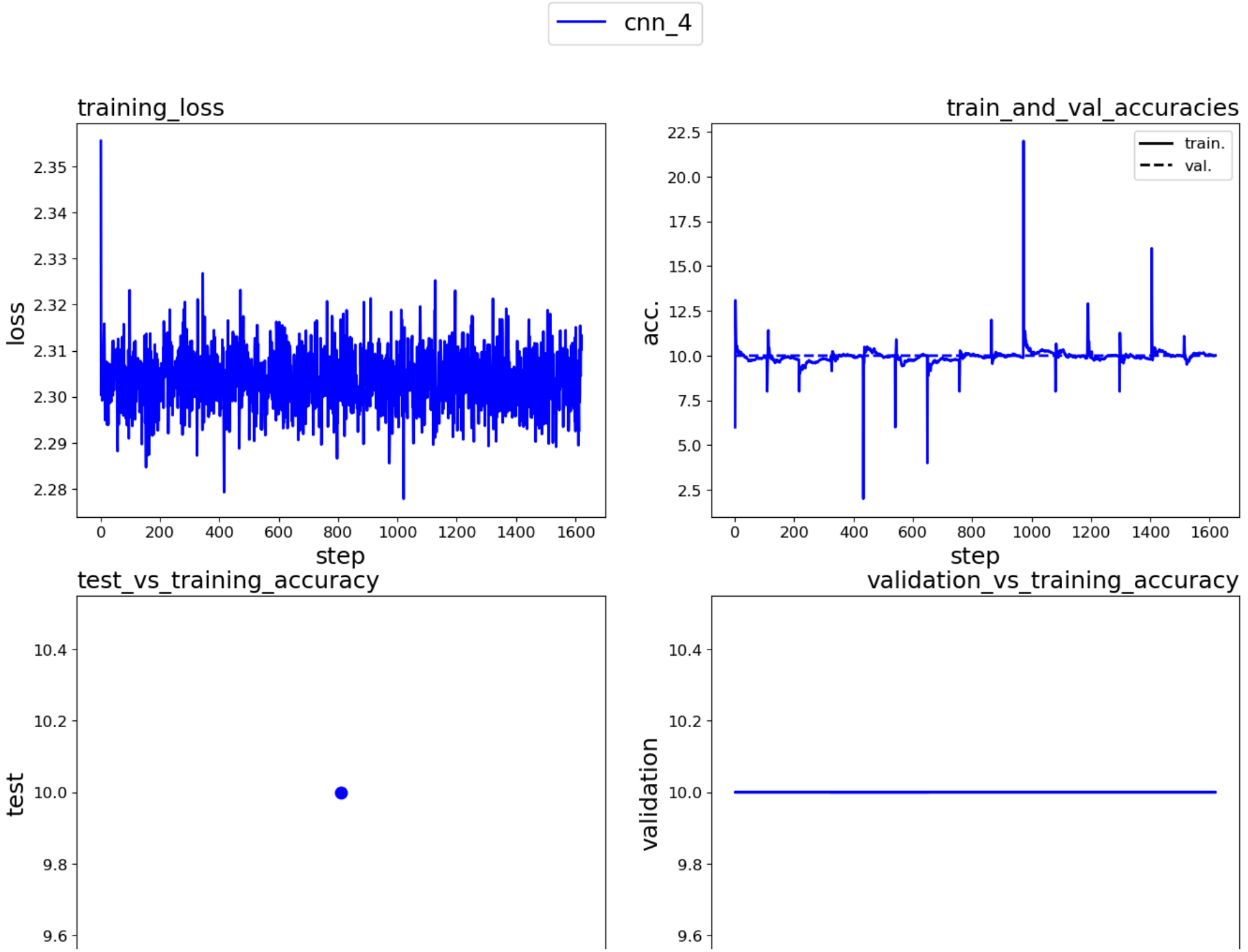
```
10.0
```

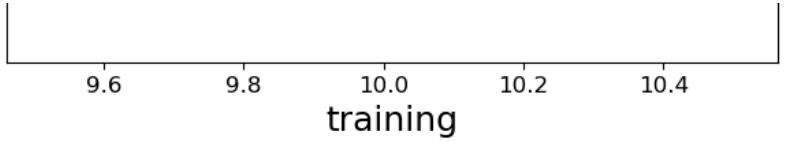
In [16]:
```
# Form the dictionary
architecture_name = "cnn_4"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist()  # Convert NumPy array to list for JSON compatibility
}

# Save the dictionary to a file
filename = f"part3_{architecture_name}.json"
```

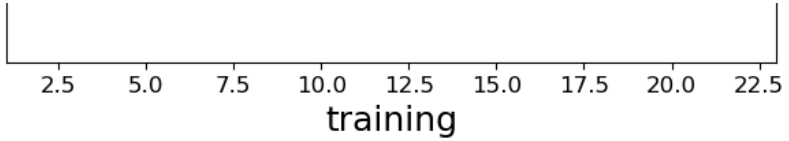
```
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

```
Experiment results saved to part3_cnn_4.json
```

In [17]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```

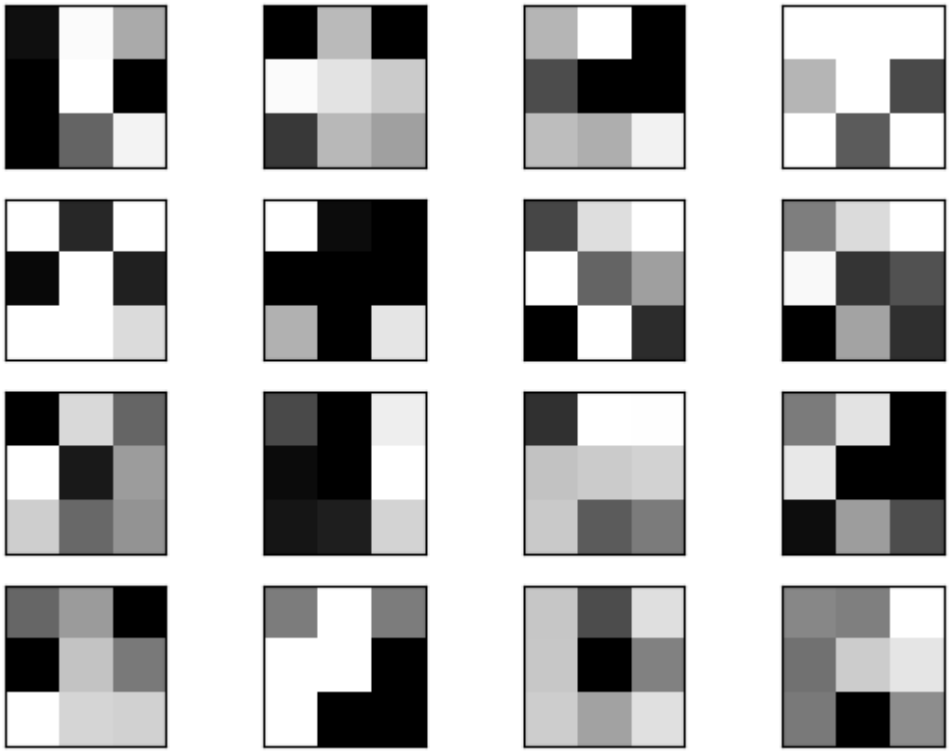

cnn_4
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [18]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```

```
/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1/Part4/utils.py:438: UserWarning: FigureCa
nvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



In [ ]:
```

```

In [22]:
```
import torchvision
import torch
import torch.nn.functional as F
import torch.nn as nn
import numpy as np
import json
import utils as ut
```

In [23]:
```
# training set
train_data = torchvision.datasets.FashionMNIST('./data', train = True, download = True,
transform = torchvision.transforms.ToTensor())
# test set
test_data = torchvision.datasets.FashionMNIST('./data', train = False,
transform = torchvision.transforms.ToTensor())
```

In [24]:
```
# Determine the number of samples per class for the split
samples_per_class = 6000
num_classes = 10
val_samples_per_class = int(samples_per_class * 0.1)  # 10% for validation

# Prepare indices for splitting
indices = np.arange(len(train_data))
targets = np.array(train_data.targets)

train_indices = []
val_indices = []

for i in range(num_classes):
    class_indices = indices[targets == i]
    np.random.shuffle(class_indices)  # Shuffle indices to ensure random split
    val_indices.extend(class_indices[:val_samples_per_class])
    train_indices.extend(class_indices[val_samples_per_class:])

# Convert to PyTorch tensors
train_indices = torch.tensor(train_indices)
val_indices = torch.tensor(val_indices)

# Create subset datasets
```

```
train_subset = torch.utils.data.Subset(train_data, train_indices)
val_subset = torch.utils.data.Subset(train_data, val_indices)

# Create DataLoaders
train_generator = torch.utils.data.DataLoader(train_subset, batch_size=50, shuffle=True)
val_generator = torch.utils.data.DataLoader(val_subset, batch_size=50, shuffle=False)
test_generator = torch.utils.data.DataLoader(test_data, batch_size = 50, shuffle = False)
```

In [25]:

```
def train_model(model, train_loader, val_loader, epochs=15, learning_rate=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    train_loss, grad_magnitudes = [] ,[]
    train_acc, val_acc = [], []

    for epoch in range(1,epochs+1,1):
        model.train()  # Set model to training mode
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the gradients
            outputs = model(inputs)  # Forward pass
            loss = F.cross_entropy(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass

            # Compute the gradient magnitude for the first layer
            grad_magnitude = model.conv1.weight.grad.norm().item()
            grad_magnitudes.append(grad_magnitude)  # Store the magnitude

            optimizer.step()  # Update weights

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if i % 10 == 0:  # Record every 10 steps
                train_loss.append(loss.item())
                train_accuracy = 100 * correct / total
                train_acc.append(train_accuracy)
                val_accuracy = validate_model(model_cnn, val_generator)
                val_acc.append(val_accuracy)
```

```
                print("Epoch: {0:2d}/{1:2d}    Loss: {2:2.4f}    Train Acc: {3:2.4f}   Val Acc:{4:2.4f}   Grad:{5:2

    first_layer_weights = model_cnn.conv1.weight.data.cpu().numpy()
    return first_layer_weights,train_acc,val_acc,train_loss,grad_magnitudes


def validate_model(model, loader):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total
```

In [26]:

```
class cnn_5(nn.Module):
    def __init__(self, num_classes=10):
        super(cnn_5, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.conv3 = nn.Conv2d(16, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.conv4 = nn.Conv2d(8, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.pool1 = nn.MaxPool2d(2, 2)  # MaxPool-2x2 after the fourth conv layer
        self.conv5 = nn.Conv2d(16, 16, kernel_size=3, padding=1)  # Conv-3x3x16
        self.conv6 = nn.Conv2d(16, 8, kernel_size=3, padding=1)  # Conv-3x3x8
        self.pool2 = nn.MaxPool2d(2, 2)  # MaxPool-2x2 after the sixth conv layer

        # Prediction layer
        self.fc = nn.Linear(8 * 7 * 7, num_classes)  # The flattened size is 8*7*7 due to two pooling layers

    def forward(self, x):
        x = torch.sigmoid(self.conv1(x))  # Conv-3x3x8 + Sigmoid
        x = torch.sigmoid(self.conv2(x))  # Conv-3x3x16 + Sigmoid
        x = torch.sigmoid(self.conv3(x))  # Conv-3x3x8 + Sigmoid
```

```
        x = torch.sigmoid(self.conv4(x))  # Conv-3x3x16 + Sigmoid
        x = self.pool1(x)  # MaxPool-2x2
        x = torch.sigmoid(self.conv5(x))  # Conv-3x3x16 + Sigmoid
        x = torch.sigmoid(self.conv6(x))  # Conv-3x3x8 + Sigmoid
        x = self.pool2(x)  # MaxPool-2x2
        x = x.view(-1, 8 * 7 * 7)  # Flatten the tensor
        x = self.fc(x)  # Prediction layer
        return x

# The input parameter 784 in the instantiation is not necessary for this model definition
model_cnn = cnn_5(num_classes=10)
```

In [27]:
```
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_cnn.parameters(), lr = 0.01, momentum = 0.0)
```

In [28]:
```
first_layer_weights,train_acc,val_acc,train_loss,grad_magnitudes = train_model(model_cnn,train_generator,val_genera
```

```
Epoch:  1/15    Loss: 2.3130    Train Acc: 12.0000   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3144    Train Acc: 7.8182   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3075    Train Acc: 8.9524   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3075    Train Acc: 9.6774   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2989    Train Acc: 10.0000   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3047    Train Acc: 10.0000   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3104    Train Acc: 9.9344   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3078    Train Acc: 9.7746   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3040    Train Acc: 9.8519   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3032    Train Acc: 10.0879   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3064    Train Acc: 10.0198   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3003    Train Acc: 10.3243   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2971    Train Acc: 10.2149   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3104    Train Acc: 10.1679   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2966    Train Acc: 10.0851   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3067    Train Acc: 9.9603   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3009    Train Acc: 10.0497   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2976    Train Acc: 10.1053   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3087    Train Acc: 10.1878   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3054    Train Acc: 10.1885   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3012    Train Acc: 10.1990   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3123    Train Acc: 10.1896   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3130    Train Acc: 10.1357   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2957    Train Acc: 10.1818   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2986    Train Acc: 10.1162   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3098    Train Acc: 10.0159   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2979    Train Acc: 10.0766   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3096    Train Acc: 10.0000   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2947    Train Acc: 9.9929   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3016    Train Acc: 10.0550   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2878    Train Acc: 10.0465   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3047    Train Acc: 10.0322   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2983    Train Acc: 10.0125   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3036    Train Acc: 9.9879   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3088    Train Acc: 10.0000   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3072    Train Acc: 9.9715   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3118    Train Acc: 9.9501   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3053    Train Acc: 9.9515   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3057    Train Acc: 9.9843   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.2993    Train Acc: 9.9693   Val Acc:10.0000   Grad:0.0000
Epoch:  1/15    Loss: 2.3073    Train Acc: 9.9751   Val Acc:10.0000   Grad:0.0000
```

```
Epoch: 15/15    Loss: 2.3070    Train Acc: 9.8645   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3048    Train Acc: 9.8956   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3024    Train Acc: 9.8967   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3043    Train Acc: 9.8979   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3024    Train Acc: 9.9078   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3054    Train Acc: 9.9088   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3054    Train Acc: 9.9055   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3063    Train Acc: 9.9129   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3042    Train Acc: 9.9096   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3042    Train Acc: 9.9126   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.2974    Train Acc: 9.9176   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3046    Train Acc: 9.9144   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.2990    Train Acc: 9.9273   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3070    Train Acc: 9.9161   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3091    Train Acc: 9.8932   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.2996    Train Acc: 9.9040   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.2992    Train Acc: 9.9049   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3174    Train Acc: 9.9328   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.2967    Train Acc: 9.9220   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3055    Train Acc: 9.9265   Val Acc:10.0000   Grad:0.0000
Epoch: 15/15    Loss: 2.3067    Train Acc: 9.9253   Val Acc:10.0000   Grad:0.0000
```

In [29]:
```
test_accuracy = validate_model(model_cnn,test_generator)
print(test_accuracy)
```

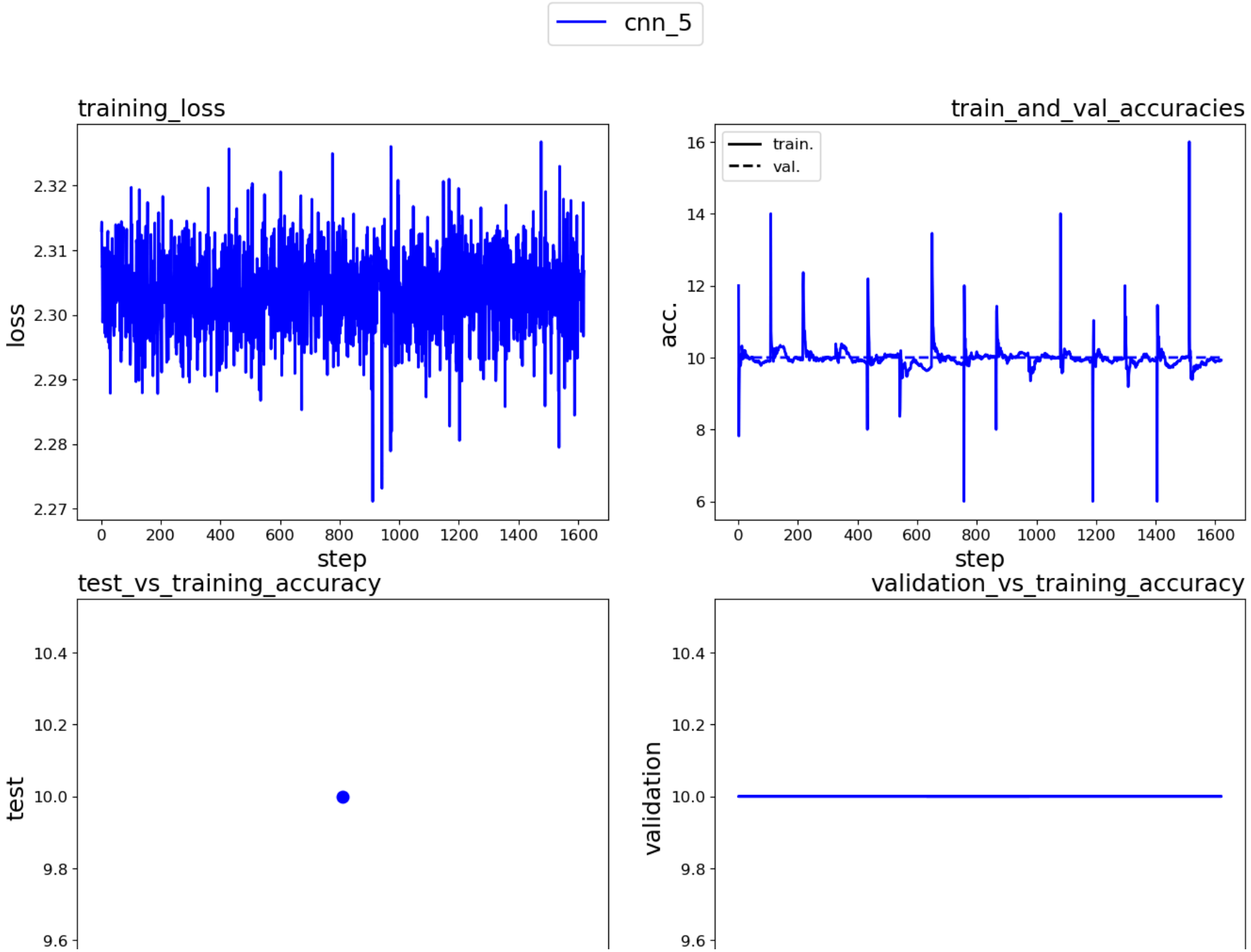
```
10.0
```

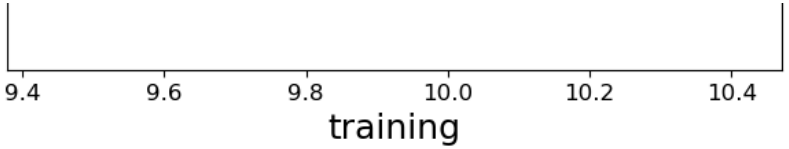
In [31]:
```
# Form the dictionary
architecture_name = "cnn_5"

experiment_results = {
    'name': architecture_name,
    'loss_curve': train_loss,
    'train_acc_curve': train_acc,
    'val_acc_curve': val_acc,
    'test_acc': test_accuracy,
    'weights': first_layer_weights.tolist(),  # Convert NumPy array to list for JSON compatibility
    'grad_magnitudes':grad_magnitudes
}

# Save the dictionary to a file
```

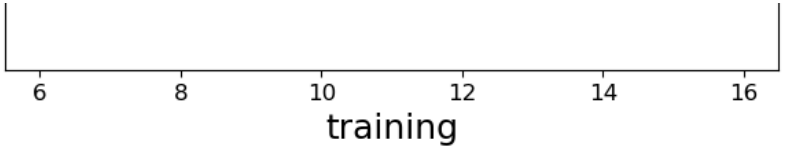
```
filename = f"part4_{architecture_name}.json"
with open(filename, 'w') as file:
    json.dump(experiment_results, file)

print(f"Experiment results saved to {filename}")
```

```
Experiment results saved to part4_cnn_5.json
```

In [32]:
```
liste = [experiment_results]
ut.part3Plots(liste)
```

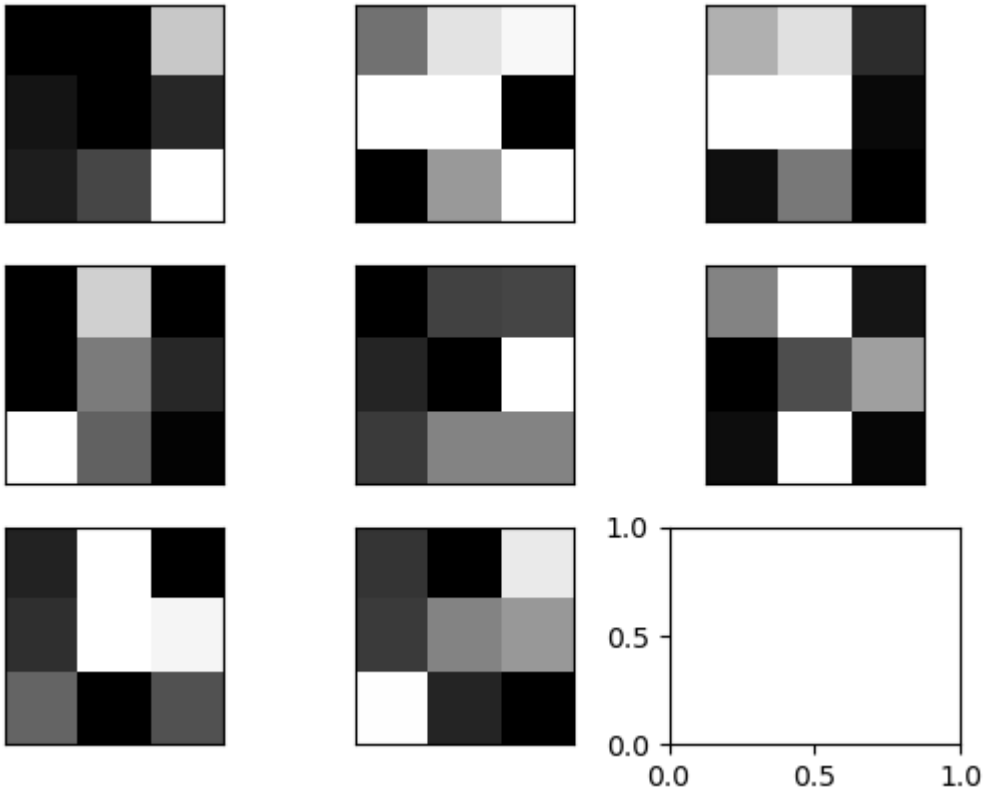

cnn_5
training_loss
loss
step
train_and_val_accuracies
train.
val.
acc.
step
test_vs_training_accuracy
test
validation_vs_training_accuracy
validation





In [33]:
```
ut.visualizeWeights(first_layer_weights,'/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1
```

```
/Users/mustafaerdikararmaz/Documents/GitHub/Computational_Intelligence/HW1/Part4/utils.py:438: UserWarning: FigureCa
nvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



In [ ]:

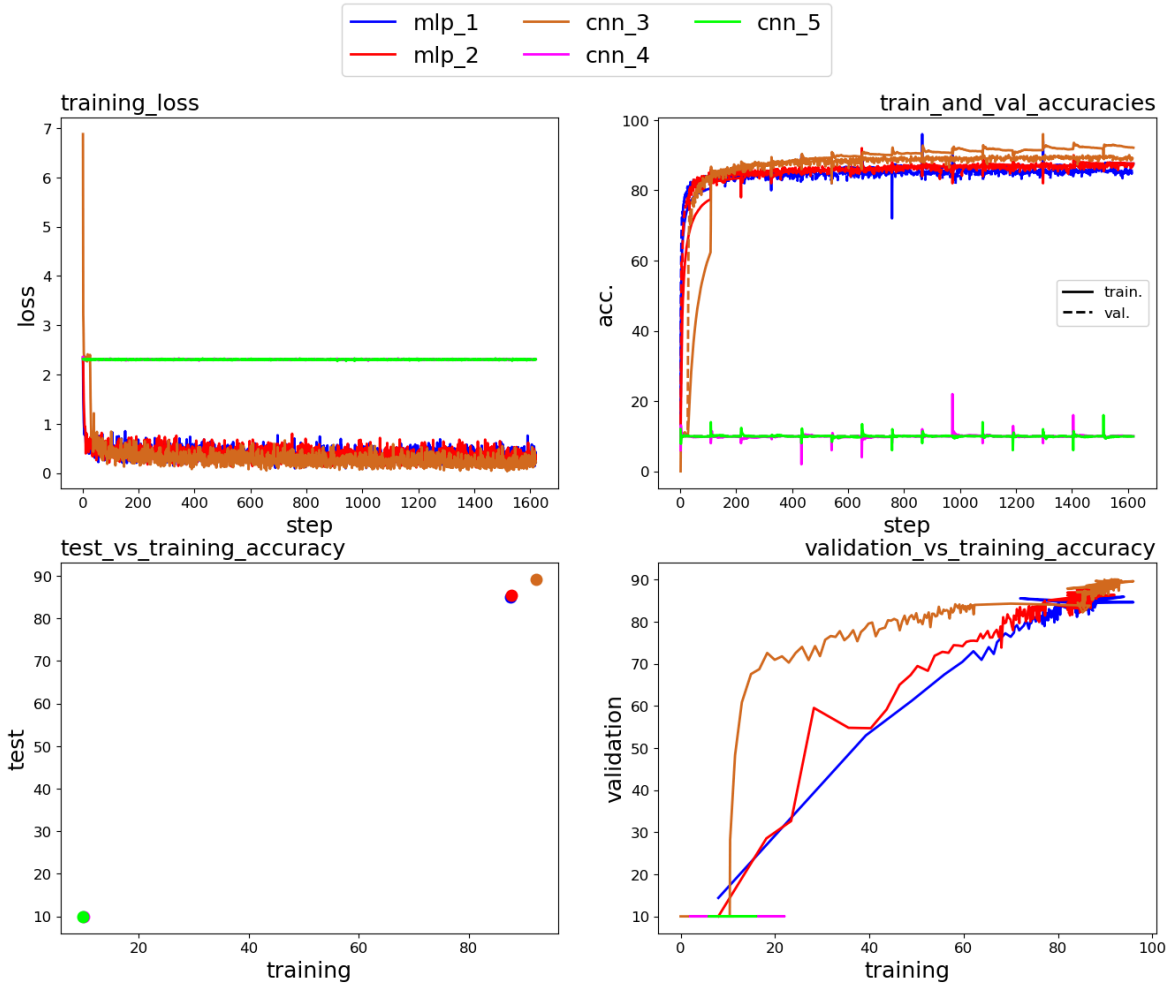
In [1]:
```
import json
import utils as ut
```

```
/opt/anaconda3/lib/python3.11/site-packages/torchvision/io/image.py:13: UserWarning: Failed to load image Python extension: 'dlopen(/opt/anaconda3/lib/python3.11/site-packages/torchvision/image.so, 0x0006): Symbol not found: __ZN3c1017RegisterOperatorsD1Ev
  Referenced from: <CFED5F8E-EC3F-36FD-AAA3-2C6C7F8D3DD9> /opt/anaconda3/lib/python3.11/site-packages/torchvision/image.so
  Expected in:     <EDCC189C-6D3F-3948-87DF-FA6A06787893> /opt/anaconda3/lib/python3.11/site-packages/torch/lib/libtorch_cpu.dylib'If you don't plan on using image functionality from `torchvision.io`, you can ignore this warning. Otherwise, there might be something wrong with your environment. Did you have `libjpeg` or `libpng` installed before building `torchvision` from source?
  warn(
```

In [2]:
```
architecture_name = ["mlp_1","mlp_2","cnn_3","cnn_4","cnn_5"]
```

In [3]:
```
file1 = "part4_mlp_1.json"
file2 = "part4_mlp_2.json"
file3 = "part4_cnn_3.json"
file4 = "part4_cnn_4.json"
file5 = "part4_cnn_5.json"

with open(file1) as user_file:
    data1 = json.load(user_file)

with open(file2) as user_file:
    data2 = json.load(user_file)

with open(file3) as user_file:
    data3 = json.load(user_file)

with open(file4) as user_file:
    data4 = json.load(user_file)

with open(file5) as user_file:
    data5 = json.load(user_file)
```

In [4]:
```
liste = [data1, data2, data3, data4, data5]

ut.part3Plots(liste)
```


mlp_1
mlp_2
cnn_3
cnn_4
cnn_5
training_loss
loss
step
train_and_val_accuracies
acc.
train.
val.
step
test_vs_training_accuracy
test
training
validation_vs_training_accuracy
validation
training

In [ ]: