

# ASSIGNMENT # 1 ADVANCED MACHINE LEARNING

## Assignment overview:

Using the mlp model architecture provided by the professor containing 1 hidden layer and the activation function Sigmoid change the architecture to 1 hidden layer and work with activation function tanh then change the architecture to 3 hidden layers and use the activation sigmoid then change it to 3 hidden layers with activation function tanh. Compare the training and testing results of each. After comparing the results of each of the 4 different architectures and functions, evaluate which worked best, which didn't, and why.

## Original Template: 1 Hidden Layer and Activation Function Sigmoid

```
1 import torch
2 import torch.nn as nn
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import torchvision.transforms as transforms
6
7 # Function to plot images
8 def imshow(img):
9     npimg = img.numpy()
10    npimg = np.transpose(npimg, (1, 2, 0))
11
12    # unnormalize
13    mean = np.array([0.5, 0.5, 0.5])
14    std = np.array([0.5, 0.5, 0.5])
15    npimg = std * npimg + mean
16    npimg = np.clip(npimg, 0, 1)
17
18    plt.imshow(npimg)
19    plt.show()
20
21 # Load the dataset
22 from torchvision.datasets import MNIST
23
24 train_dataset = MNIST('./mnist_data', train=True, download=True, transform=transforms.ToTensor())
25 test_dataset = MNIST('./mnist_data', train=False, download=True, transform=transforms.ToTensor())
26
27 # Define the batch size and number of iterations to calculate epochs
28 batch_size = 16
29 n_iters = 5000
30 num_epochs = n_iters / (len(train_dataset) / batch_size)
31 num_epochs = int(num_epochs)
32
33 # Load dataset to dataloader
34 train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
35
36
37 # Building model with hidden layer (MLP : Multi Layer Perceptron)
38 class FeedforwardModel(nn.Module):
39     def __init__(self, input_dim, hidden_dim, output_dim):
40         super(FeedforwardModel, self).__init__()
41         # Linear function
42         self.fc1 = nn.Linear(input_dim, hidden_dim)
43
44         # Non-linearity
45         self.sigmoid = nn.Sigmoid()
46
47         # Linear function (readout)
48         self.fc2 = nn.Linear(hidden_dim, output_dim)
49
50     def forward(self, x):
51         # Linear function # LINEAR
52         out = self.fc1(x)
53
54         # Non-linearity # NON-LINEAR
55         out = self.sigmoid(out)
56
57         # Linear function (readout) # LINEAR
58         out = self.fc2(out)
59         return out
60
61 # Instantiate model
62 input_dim = 28*28
63 hidden_dim = 100
64 output_dim = 10
65
66 model = FeedforwardModel(input_dim, hidden_dim, output_dim)
67
68
69 # Instantiate Criterions
70 # 1. Does 2 things at the same time
71 # 1. Computes softmax (logistic/softmax function)
72 # 2. Computes cross entropy
73 criterion = nn.CrossEntropyLoss()
74
75 # Instantiate optimizer
76 learning_rate = 0.01
77 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
78
79 # Load model weights
80 filepath = 'mn_model_weights_MNIST_2.pth'
81 model.load_state_dict(torch.load(filepath))
```

```

102 # Train model
103 iter = 0
104 for epoch in range(num_epochs):
105     for i, (images, labels) in enumerate(train_loader):
106         # Load images as tensors with gradient accumulation abilities
107         images = images.view(-1, 28*28).requires_grad_()
108         # Clear gradients w.r.t. parameters
109         optimizer.zero_grad()
110         # Forward pass to get output/logits
111         # 100 = 10
112         outputs = model(images)
113         # Calculate loss: softmax -> cross entropy loss
114         loss = criterion(outputs, labels)
115         # Getting gradients w.r.t. parameters
116         loss.backward()
117         # Updating parameters
118         optimizer.step()
119     iter += 1
120     if iter % 500 == 0:
121         # Calculate Accuracy
122         correct = 0
123         total = 0
124         # Iterate through test dataset
125         for images, labels in test_loader:
126             # Load images to torch tensors with gradient accumulation abilities
127             images = images.view(-1, 28*28).requires_grad_()
128             # Forward pass only to get logits/output
129             outputs = model(images)
130             # Get predictions from the maximum value
131             # 100 = 1
132             _, predicted = torch.max(outputs.data, 1)
133             # Total number of labels
134             total += labels.size(0)
135             # Total correct predictions
136             correct += (predicted == labels).sum()
137         accuracy = 100 * correct.item() / total
138         # Print time
139         print('Iteration: {}, Loss: {}, Accuracy: {}'.format(iter, loss.item(), accuracy))

```

```

140 # Test model with a single image
141 img_1, label_1 = test_dataset[0]
142 outputs = model(img_1.view(-1, 28*28).requires_grad_())
143 _, predicted = torch.max(outputs.data, 1)
144 print('Prediction: {}, Labels: {}'.format(predicted, label_1))
145
146 # Test model with a data batch
147 for idx, data in enumerate(test_dataset):
148     images, labels = data
149     outputs = model(images.view(-1, 28*28).requires_grad_())
150     _, predicted = torch.max(outputs.data, 1)
151     print('Prediction: {}, Labels: {}'.format(predicted, labels))
152     if idx > batch_size:
153         break
154
155 # Save trained model weights
156 torch.save(model.state_dict(), filepath)

```

## Training Result:

```

Iteration: 500. Loss: 2.130690574645996.
Accuracy: 50.28
Iteration: 1000. Loss: 1.9725408554077148.
Accuracy: 52.52
Iteration: 1500. Loss: 1.6072311401367188.
Accuracy: 70.96
Iteration: 2000. Loss: 1.1860986948013306.
Accuracy: 74.64
Iteration: 2500. Loss: 1.1157934665679932.
Accuracy: 79.81
Iteration: 3000. Loss: 0.8797474503517151.
Accuracy: 81.97
Iteration: 3500. Loss: 0.76375412940979.
Accuracy: 83.98

```

## Testing Result:

```

Prediction: tensor([0]). Labels: 0
Prediction: tensor([4]). Labels: 4
Prediction: tensor([1]). Labels: 1
Prediction: tensor([9]). Labels: 4
Prediction: tensor([9]). Labels: 9
Prediction: tensor([5]). Labels: 5
Prediction: tensor([9]). Labels: 9
Prediction: tensor([0]). Labels: 0
Prediction: tensor([2]). Labels: 6
Prediction: tensor([9]). Labels: 9
Prediction: tensor([0]). Labels: 0
Prediction: tensor([1]). Labels: 1
Prediction: tensor([3]). Labels: 5
Prediction: tensor([9]). Labels: 9
Prediction: tensor([7]). Labels: 7

```

## 2: 1 Hidden Layer and Activation Function Tanh

```

1 import torch
2 import torch.nn as nn
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import torchvision.transforms as transforms
6
7 # Function to plot images
8 def imshow(img):
9     npimg = img.numpy()
10    npimg = np.transpose(npimg, (1, 2, 0))
11
12    # unnormalize
13    mean = np.array([0.5, 0.5, 0.5])
14    std = np.array([0.5, 0.5, 0.5])
15    npimg = std * npimg + mean
16    npimg = np.clip(npimg, 0, 1)
17
18    plt.imshow(npimg)
19    plt.show()
20
21 # Load the dataset
22 from torchvision.datasets import MNIST
23
24 train_dataset = MNIST('./mnist_data', train=True, download=True, transform=transforms.ToTensor())
25 test_dataset = MNIST('./mnist_data', train=False, download=True, transform=transforms.ToTensor())
26
27 # Define the batch size and number of iterations to calculate epochs
28 batch_size = 16
29 n_iters = 5000
30 num_epochs = n_iters / (len(train_dataset) / batch_size)
31 num_epochs = int(num_epochs)
32
33 # Load dataset to dataloader
34 train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
35 test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

```

```

37 # Building model with hidden layer (MLP: Multi Layer Perceptron)
38 class FeedforwardModelTanh(nn.Module):
39     def __init__(self, input_dim, hidden_dim, output_dim):
40         super(FeedforwardModelTanh, self).__init__()
41         self.fc1 = nn.Linear(input_dim, hidden_dim)
42         self.tanh = nn.Tanh()
43         self.fc2 = nn.Linear(hidden_dim, output_dim)
44
45     def forward(self, x):
46         out = self.fc1(x)
47         out = self.tanh(out)
48         out = self.fc2(out)
49         return out
50
51 # Instantiate model
52 input_dim = 28*28
53 hidden_dim = 100
54 output_dim = 10
55
56 model = FeedforwardModelTanh(input_dim, hidden_dim, output_dim)
57
58 # Instantiate CrossEntropyLoss
59 # It does 2 things at the same time.
60 # 1. Computes softmax (logistic/softmax function)
61 # 2. Computes cross entropy
62 criterion = nn.CrossEntropyLoss()
63
64 # Instantiate optimizer
65 learning_rate = 0.01
66 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
67
68 # Optional: load model weights (only if the model was previously trained)
69 # Make sure weights match the current architecture
70 # Comment this out if training from scratch
71 filepath = 'mn_model_weights_MNIST_1hidden_layer_tanh.pth'
72 model.load_state_dict(torch.load(filepath))
73

```

```

74 # Train model
75 iter = 0
76 for epoch in range(num_epochs):
77     for i, (images, labels) in enumerate(train_loader):
78         # Load images as tensors with gradient accumulation abilities
79         images = images.view(-1, 28*28).requires_grad_()
80
81         # Clear gradients w.r.t. parameters
82         optimizer.zero_grad()
83
84         # Forward pass to get output/ logits
85         outputs = model(images)
86
87         # Calculate loss: softmax -> cross entropy loss
88         loss = criterion(outputs, labels)
89
90         # Getting gradients w.r.t. parameters
91         loss.backward()
92
93         # Updating parameters
94         optimizer.step()
95
96         iter += 1
97
98     if iter % 500 == 0:
99         # Calculate Accuracy
100         correct = 0
101         total = 0
102         # Iterate through test dataset
103         for images, labels in test_loader:
104             # Load images to torch tensors with gradient accumulation abilities
105             images = images.view(-1, 28*28).requires_grad_()
106
107             # Forward pass only to get logits/output
108             outputs = model(images)
109
110             # Get predictions from the maximum value
111             _, predicted = torch.max(outputs.data, 1)
112
113             # Total number of labels
114             total += labels.size(0)
115
116             # Total correct predictions
117             correct += (predicted == labels).sum()
118
119         accuracy = 100 * correct.item() / total
120
121     # Print loss
122     print('Iteration: {}, Loss: {}, Accuracy: {}'.format(iter, loss.item(), accuracy))

```

```

124 # Test model with a single image
125 img_1, label_1 = test_dataset[0]
126 outputs = model(img_1.view(1, 28*28).requires_grad_())
127 _, predicted = torch.max(outputs.data, 1)
128 print('Prediction: {}, labels: {}'.format(predicted, label_1))
129
130 # Test model with a data batch
131 for idx, data in enumerate(test_dataset):
132     images, labels = data
133     outputs = model(images.view(-1, 28*28).requires_grad_())
134     _, predicted = torch.max(outputs.data, 1)
135     print('Prediction: {}, labels: {}'.format(predicted, labels))
136     if idx > batch_size:
137         break
138
139 # Save trained model weight
140 torch.save(model.state_dict(), filepath)
141

```

## Training Results:

```

Iteration: 500. Loss: 0.6740624308586121.
Accuracy: 82.03
Iteration: 1000. Loss: 0.6168676018714905.
Accuracy: 86.44
Iteration: 1500. Loss: 0.3922017216682434.
Accuracy: 87.74
Iteration: 2000. Loss: 0.33130553364753723.
Accuracy: 89.27
Iteration: 2500. Loss: 0.12082458287477493.
Accuracy: 89.9
Iteration: 3000. Loss: 0.5180799961090088.
Accuracy: 90.33
Iteration: 3500. Loss: 0.6953763961791992.
Accuracy: 90.62

```

Testing Results:

```

Prediction: tensor([0]). Labels: 0
Prediction: tensor([4]). Labels: 4
Prediction: tensor([1]). Labels: 1
Prediction: tensor([4]). Labels: 4
Prediction: tensor([9]). Labels: 9
Prediction: tensor([6]). Labels: 5
Prediction: tensor([9]). Labels: 9
Prediction: tensor([0]). Labels: 0
Prediction: tensor([6]). Labels: 6
Prediction: tensor([9]). Labels: 9
Prediction: tensor([0]). Labels: 0
Prediction: tensor([1]). Labels: 1
Prediction: tensor([5]). Labels: 5
Prediction: tensor([9]). Labels: 9
Prediction: tensor([7]). Labels: 7

```

### 3: 3 Hidden Layers with Activation Function Sigmoid

```

1  import torch
2  import torch.nn as nn
3  import matplotlib.pyplot as plt
4  import numpy as np
5  import torchvision.transforms as transforms
6  from torchvision.datasets import MNIST
7
8  # Function to plot images
9  def imshow(img):
10     npimg = img.numpy()
11     npimg = np.transpose(npimg, (1, 2, 0))
12     mean = np.array([0.5, 0.5, 0.5])
13     std = np.array([0.5, 0.5, 0.5])
14     npimg = std * npimg + mean
15     npimg = np.clip(npimg, 0, 1)
16     plt.imshow(npimg)
17     plt.show()
18
19 # Load the dataset
20 train_dataset = MNIST('./mnist_data', train=True, download=True, transform=transforms.ToTensor())
21 test_dataset = MNIST('./mnist_data', train=False, download=True, transform=transforms.ToTensor())
22
23 # Define the batch size and number of iterations to calculate epochs
24 batch_size = 16
25 n_iters = 5000
26 num_epochs = n_iters / (len(train_dataset) / batch_size)
27 num_epochs = int(num_epochs)
28
29 # Load dataset to dataloader
30 train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
31 test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
32

```

```

33 # Building model with 3 hidden layers (MLP : Multi Layer Perceptron) using Sigmoid activation
34 class FeedforwardModelSigmoid(nn.Module):
35     def __init__(self, input_dim, hidden_dim, output_dim):
36         super(FeedforwardModelSigmoid, self).__init__()
37         self.fc1 = nn.Linear(input_dim, hidden_dim)
38         self.sigmoid1 = nn.Sigmoid()
39         self.fc2 = nn.Linear(hidden_dim, hidden_dim)
40         self.sigmoid2 = nn.Sigmoid()
41         self.fc3 = nn.Linear(hidden_dim, hidden_dim)
42         self.sigmoid3 = nn.Sigmoid()
43         self.fc4 = nn.Linear(hidden_dim, output_dim)
44
45     def forward(self, x):
46         out = self.fc1(x)
47         out = self.sigmoid1(out)
48         out = self.fc2(out)
49         out = self.sigmoid2(out)
50         out = self.fc3(out)
51         out = self.sigmoid3(out)
52         out = self.fc4(out)
53         return out
54
55 # Instantiate model
56 input_dim = 28*28
57 hidden_dim = 100
58 output_dim = 10
59 model = FeedforwardModelSigmoid(input_dim, hidden_dim, output_dim)
60
61 # Instantiate CrossEntropyLoss
62 criterion = nn.CrossEntropyLoss()
63
64 # Instantiate optimizer
65 learning_rate = 0.01
66 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
67
68 # Load model weights
69 filepath = 'fnn_model_weights_MNIST_3hidden_sigmoid.pth'
70 model.load_state_dict(torch.load(filepath))

```

```

# Train model
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.view(-1, 28*28).requires_grad_()
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        iter += 1

        if iter % 500 == 0:
            correct = 0
            total = 0
            for images, labels in test_loader:
                images = images.view(-1, 28*28).requires_grad_()
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))

```

```

97 # Test model with a single image
98 img_1, label_1 = test_dataset[0]
99 outputs = model(img_1.view(-1, 28*28).requires_grad_())
100 _, predicted = torch.max(outputs.data, 1)
101 print('Prediction: {}. Labels: {}'.format(predicted, label_1))
102
103 # Test model with a 64 batch
104 for idx, data in enumerate(test_dataset):
105     images, labels = data
106     outputs = model(images.view(-1, 28*28).requires_grad_())
107     _, predicted = torch.max(outputs.data, 1)
108     print('Prediction: {}, Labels: {}'.format(predicted, labels))
109     if idx > batch_size:
110         break
111
112 # Save trained model weights
113 torch.save(model.state_dict(), filepath)

```

## Training Results:

```

Iteration: 500. Loss: 2.3478033542633057.
Accuracy: 10.1
Iteration: 1000. Loss: 2.3078713417053223.
Accuracy: 9.74
Iteration: 1500. Loss: 2.2825870513916016.
Accuracy: 9.74
Iteration: 2000. Loss: 2.3121373653411865.
Accuracy: 10.28
Iteration: 2500. Loss: 2.302565097808838.
Accuracy: 11.35
Iteration: 3000. Loss: 2.2815637588500977.
Accuracy: 10.28
Iteration: 3500. Loss: 2.2987074851989746.
Accuracy: 10.28

```

## Testing Results:

```

Prediction: tensor([6]). Labels: 0
Prediction: tensor([6]). Labels: 4
Prediction: tensor([6]). Labels: 1
Prediction: tensor([6]). Labels: 4
Prediction: tensor([6]). Labels: 9
Prediction: tensor([6]). Labels: 5
Prediction: tensor([6]). Labels: 9
Prediction: tensor([6]). Labels: 0
Prediction: tensor([6]). Labels: 6
Prediction: tensor([6]). Labels: 9
Prediction: tensor([6]). Labels: 0
Prediction: tensor([6]). Labels: 1
Prediction: tensor([6]). Labels: 5
Prediction: tensor([6]). Labels: 9
Prediction: tensor([6]). Labels: 7

```

## 4: 3 Hidden Layers with Activation Function Tanh

```

import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
import torchvision.transforms as transforms
from torchvision.datasets import MNIST

# Function to plot images
def imshow(img):
    npimg = img.numpy()
    npimg = np.transpose(npimg, (1, 2, 0))
    mean = np.array([0.5, 0.5, 0.5])
    std = np.array([0.5, 0.5, 0.5])
    npimg = std * npimg + mean
    npimg = np.clip(npimg, 0, 1)
    plt.imshow(npimg)
    plt.show()

# Load the dataset
train_dataset = MNIST('~/.mnist_data', train=True, download=True, transform=transforms.ToTensor())
test_dataset = MNIST('~/.mnist_data', train=False, download=True, transform=transforms.ToTensor())

# Define the batch size and number of iterations to calculate epochs
batch_size = 16
n_iters = 5000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)

# Load dataset to dataloader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

```

```

34 # Building model with 3 hidden layers (MLP - Multi Layer Perceptron) using tanh activation
35 class FeedforwardModelTanh(nn.Module):
36     def __init__(self, input_dim, hidden_dim, output_dim):
37         super(FeedforwardModelTanh, self).__init__()
38         self.fc1 = nn.Linear(input_dim, hidden_dim)
39         self.tanh1 = nn.Tanh()
40         self.fc2 = nn.Linear(hidden_dim, hidden_dim)
41         self.tanh2 = nn.Tanh()
42         self.fc3 = nn.Linear(hidden_dim, hidden_dim)
43         self.tanh3 = nn.Tanh()
44         self.fc4 = nn.Linear(hidden_dim, output_dim)
45
46     def forward(self, x):
47         out = self.fc1(x)
48         out = self.tanh1(out)
49         out = self.fc2(out)
50         out = self.tanh2(out)
51         out = self.fc3(out)
52         out = self.tanh3(out)
53         out = self.fc4(out)
54         return out
55
56 # Instantiate model
57 input_dim = 28*28
58 hidden_dim = 100
59 output_dim = 10
60 model = FeedforwardModelTanh(input_dim, hidden_dim, output_dim)
61
62 # Instantiate CrossEntropyLoss
63 criterion = nn.CrossEntropyLoss()
64
65 # Instantiate optimizer
66 learning_rate = 0.01
67 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
68
69 # Load model weights
70 filepath = 'fun_model_weights_MNIST_3hidden_tanh.pth'
71 model.load_state_dict(torch.load(filepath))

```

```

72 # Train model
73 iter = 0
74 for epoch in range(num_epochs):
75     for i, (images, labels) in enumerate(train_loader):
76         images = images.view(-1, 28*28).requires_grad_()
77         optimizer.zero_grad()
78         outputs = model(images)
79         loss = criterion(outputs, labels)
80         loss.backward()
81         optimizer.step()
82         iter += 1
83
84     if iter % 500 == 0:
85         correct = 0
86         total = 0
87         for images, labels in test_loader:
88             images = images.view(-1, 28*28).requires_grad_()
89             outputs = model(images)
90             _, predicted = torch.max(outputs.data, 1)
91             total += labels.size(0)
92             correct += (predicted == labels).sum()
93
94         accuracy = 100 * correct.item() / total
95         print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))
96
97 # Test model with a single image
98 img_1, label_1 = test_dataset[0]
99 outputs = model(img_1.view(-1, 28*28).requires_grad_())
100 _, predicted = torch.max(outputs.data, 1)
101 print('Prediction: {}. Labels: {}'.format(predicted, label_1))

```

```

97 # Test model with a single image
98 img_1, label_1 = test_dataset[0]
99 outputs = model(img_1.view(-1, 28*28).requires_grad_())
100 _, predicted = torch.max(outputs.data, 1)
101 print('Prediction: {}, Labels: {}'.format(predicted, label_1))
102
103 # Test model with a data batch
104 for idx, data in enumerate(test_dataset):
105     images, labels = data
106     outputs = model(images.view(-1, 28*28).requires_grad_())
107     _, predicted = torch.max(outputs.data, 1)
108     print('Prediction: {}, Labels: {}'.format(predicted, labels))
109     if idx > batch_size:
110         break
111
112 # Save trained model weights
113 torch.save(model.state_dict(), filepath)
114

```

## Training Results:

```

Iteration: 500. Loss: 1.327965021134229. Accuracy: 64.9
Iteration: 1000. Loss: 0.866273939695276. Accuracy: 80.41
Iteration: 1500. Loss: 0.4800269991752472. Accuracy: 85.29
Iteration: 2000. Loss: 0.32519271969795227. Accuracy: 88.12
Iteration: 2500. Loss: 0.40641695261001587. Accuracy: 89.4
Iteration: 3000. Loss: 0.7688746452331543. Accuracy: 89.98
Iteration: 3500. Loss: 0.3215646743774414. Accuracy: 90.31

```

## Testing Results:

```

Prediction: tensor([0]). Labels: 0
Prediction: tensor([4]). Labels: 4
Prediction: tensor([1]). Labels: 1
Prediction: tensor([4]). Labels: 4
Prediction: tensor([9]). Labels: 9
Prediction: tensor([6]). Labels: 5
Prediction: tensor([9]). Labels: 9
Prediction: tensor([0]). Labels: 0
Prediction: tensor([6]). Labels: 6
Prediction: tensor([9]). Labels: 9
Prediction: tensor([0]). Labels: 0
Prediction: tensor([1]). Labels: 1
Prediction: tensor([5]). Labels: 5
Prediction: tensor([9]). Labels: 9
Prediction: tensor([7]). Labels: 7

```

## 1 Hidden layer + Sigmoid, initial, mid, final

	loss	accuracy	predictions
500	2.13	50.28	4/20 results wrong
2000	1.18	74.64	
3500	0.76	83.98	

## 1 Hidden layer + Tanh, initial, mid, final

	loss	accuracy	predictions
500	0.67	82.03%	1/20 results wrong
2000	0.33	89.27%	
3500	0.69	90.62%	

## 3 Hidden layers + Sigmoid, initial, mid, final

	loss	accuracy	predictions
500	2.34	10.10%	1/20 results correct
2000	2.31	10.28%	
3500	2.29	10.28%	

## 3 Hidden layers + Tanh, initial, mid, final

	loss	accuracy	predictions
500	1.32	64.9%	1/20 results wrong
2000	0.32	88.12%	
3500	0.32	90.31%	

## Comparison

### 1. Activation Functions:

- **Tanh** consistently outperforms **Sigmoid** across all configurations. The loss values are lower, and the accuracy is higher for the Tanh configurations at all stages.
- The initial performance of both **1 Hidden Layer + Tanh** and **3 Hidden Layers + Tanh** is significantly better than their Sigmoid counterparts.

### 2. Model Complexity:

- The addition of **3 hidden layers** does not improve the model's performance when using the Sigmoid activation function. In fact, it results in significantly poor performance, maintaining an accuracy around 10%, indicating severe underfitting.
- The **3 Hidden Layers + Tanh** configuration shows better results compared to **3 Hidden Layers + Sigmoid**, indicating that the deeper architecture can benefit from the Tanh activation.

### 3. Loss and Accuracy Trends:

- For both **1 Hidden Layer** configurations, accuracy improves significantly with each iteration, especially for the Tanh model. The **1 Hidden Layer + Tanh** achieves a final accuracy of **90.62%**, compared to **83.98%** for the **1 Hidden Layer + Sigmoid**.
- The **3 Hidden Layers + Tanh** model also achieves high accuracy (90.31%) but starts at a significantly higher initial loss compared to its single-layer counterpart.

### 4. Prediction Performance:

- **1 Hidden Layer + Tanh** has the best predictive accuracy (1 wrong out of 20) at the initial stage, while **3 Hidden Layers + Sigmoid** performs poorly with only 1 correct prediction out of 20.
- This indicates that the complexity added by the additional layers is detrimental when using Sigmoid but beneficial with Tanh.

## Conclusion

- **Tanh** is the better activation function for this task, leading to higher accuracy and lower loss in both single-layer and multi-layer configurations.
- The performance of a model does not necessarily improve with more hidden layers when using **Sigmoid**; it is crucial to choose both the activation function and the architecture wisely.
- The **1 Hidden Layer + Tanh** configuration provides the best balance of performance, accuracy, and computational efficiency for this dataset. In contrast, **3 Hidden Layers + Sigmoid** appears to be ineffective and does not capture the complexities of the MNIST dataset well.