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.

Orignal Template: 1 Hidden Layer and Activation Function Sigmoid

```python
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
import torchvision.transforms as transforms

# Function to plot images
def imshow(img):
    npimg = img.numpy()
    npimg = np.transpose(npimg, (1, 2, 0))

    # unnormalize
    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
from torchvision.datasets import MNIST

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
```

```python

# Building model with hidden layer (MLP : Multi Layer Perceptron)
class FeedforwardModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardModel, self).__init__()
        # Linear function
        self.fc1 = nn.Linear(input_dim, hidden_dim)

        # Non-linearity
        self.sigmoid = nn.Sigmoid()

        # Linear function (readout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Linear function  # LINEAR
        out = self.fc1(x)

        # Non-linearity  # NON-LINEAR
        out = self.sigmoid(out)

        # Linear function (readout)  # LINEAR
        out = self.fc2(out)
        return out

# Instantiate model
input_dim = 28*28
hidden_dim = 100
output_dim = 10

model = FeedforwardModel(input_dim, hidden_dim, output_dim)

```

```python
# Instantiate CrossEntropyLoss
# It does 2 things at the same time.
# 1. Computes softmax (logistic/softmax function)
# 2. Computes cross entropy
criterion = nn.CrossEntropyLoss()

# Instantiate optimizer
learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# load model weights
filepath = 'fnn_model_weights_MNIST_2.pth'
model.load_state_dict(torch.load(filepath))
```

```python
# Train model
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as tensors with gradient accumulation abilities
        images = images.view(-1, 28*28).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        # 100 x 10
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to Torch Tensors with gradient accumulation abilities
                images = images.view(-1, 28*28).requires_grad_()

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                # 100 x 1
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))
```

```python
# Test model with a single image
img_1, label_1 = test_dataset[0]
outputs = model(img_1.view(-1, 28*28).requires_grad_())
_, predicted = torch.max(outputs.data, 1)
print('Prediction: {}. Labels: {}'.format(predicted, label_1))

# Test model with a data batch
for idx, data in enumerate(test_dataset):
    images, labels = data
    outputs = model(images.view(-1, 28*28).requires_grad_())
    _, predicted = torch.max(outputs.data, 1)
    print('Prediction: {}. Labels: {}'.format(predicted, labels))
    if idx > batch_size:
        break

# Save trained model weights
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([6]). 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

```python
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
import torchvision.transforms as transforms

# Function to plot images
def imshow(img):
    npimg = img.numpy()
    npimg = np.transpose(npimg, (1, 2, 0))

    # unnormalize
    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
from torchvision.datasets import MNIST

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)
```

```python
# Building model with hidden layer (MLP : Multi Layer Perceptron)
class FeedforwardModelTanh(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardModelTanh, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.tanh = nn.Tanh()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out = self.fc1(x)
        out = self.tanh(out)
        out = self.fc2(out)
        return out

# Instantiate model
input_dim = 28*28
hidden_dim = 100
output_dim = 10

model = FeedforwardModelTanh(input_dim, hidden_dim, output_dim)

# Instantiate CrossEntropyLoss
# It does 2 things at the same time.
# 1. Computes softmax (logistic/softmax function)
# 2. Computes cross entropy
criterion = nn.CrossEntropyLoss()

# Instantiate optimizer
learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Optional: load model weights (only if the model was previously trained)
# Make sure weights match the current architecture
# Comment this out if training from scratch
filepath = 'fnn_model_weights_MNIST_1hidden_layer_tanh.pth'
model.load_state_dict(torch.load(filepath))
```

```python
# Train model
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Load images as tensors with gradient accumulation abilities
        images = images.view(-1, 28*28).requires_grad_()

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        loss = criterion(outputs, labels)

        # Getting gradients w.r.t. parameters
        loss.backward()

        # Updating parameters
        optimizer.step()

        iter += 1

        if iter % 500 == 0:
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                # Load images to Torch Tensors with gradient accumulation abilities
                images = images.view(-1, 28*28).requires_grad_()

                # Forward pass only to get logits/output
                outputs = model(images)

                # Get predictions from the maximum value
                _, predicted = torch.max(outputs.data, 1)

                # Total number of labels
                total += labels.size(0)

                # Total correct predictions
                correct += (predicted == labels).sum()

            accuracy = 100 * correct.item() / total

            # Print Loss
            print('Iteration: {}. Loss: {}. Accuracy: {}'.format(iter, loss.item(), accuracy))
```

```python
# Test model with a single image
img_1, label_1 = test_dataset[0]
outputs = model(img_1.view(-1, 28*28).requires_grad_())
_, predicted = torch.max(outputs.data, 1)
print('Prediction: {}. Labels: {}'.format(predicted, label_1))

# Test model with a data batch
for idx, data in enumerate(test_dataset):
    images, labels = data
    outputs = model(images.view(-1, 28*28).requires_grad_())
    _, predicted = torch.max(outputs.data, 1)
    print('Prediction: {}. Labels: {}'.format(predicted, labels))
    if idx > batch_size:
        break

# Save trained model weights
torch.save(model.state_dict(), filepath)
```

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

```python
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)
```

```python
# Building model with 3 hidden layers (MLP : Multi Layer Perceptron) using Sigmoid activation
class FeedforwardModelSigmoid(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardModelSigmoid, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.sigmoid1 = nn.Sigmoid()
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.sigmoid2 = nn.Sigmoid()
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)
        self.sigmoid3 = nn.Sigmoid()
        self.fc4 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out = self.fc1(x)
        out = self.sigmoid1(out)
        out = self.fc2(out)
        out = self.sigmoid2(out)
        out = self.fc3(out)
        out = self.sigmoid3(out)
        out = self.fc4(out)
        return out

# Instantiate model
input_dim = 28*28
hidden_dim = 100
output_dim = 10
model = FeedforwardModelSigmoid(input_dim, hidden_dim, output_dim)

# Instantiate CrossEntropyLoss
criterion = nn.CrossEntropyLoss()

# Instantiate optimizer
learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Load model weights
filepath = 'fnn_model_weights_MNIST_3hidden_sigmoid.pth'
model.load_state_dict(torch.load(filepath))
```

```python
# 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))
```

```python
# Test model with a single image
img_1, label_1 = test_dataset[0]
outputs = model(img_1.view(-1, 28*28).requires_grad_())
_, predicted = torch.max(outputs.data, 1)
print('Prediction: {}. Labels: {}'.format(predicted, label_1))

# Test model with a data batch
for idx, data in enumerate(test_dataset):
    images, labels = data
    outputs = model(images.view(-1, 28*28).requires_grad_())
    _, predicted = torch.max(outputs.data, 1)
    print('Prediction: {}. Labels: {}'.format(predicted, labels))
    if idx > batch_size:
        break

# Save trained model weights
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

```python
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)
```

```python
# Building model with 3 hidden layers (MLP : Multi Layer Perceptron) using Tanh activation
class FeedforwardModelTanh(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(FeedforwardModelTanh, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.tanh1 = nn.Tanh()
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.tanh2 = nn.Tanh()
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)
        self.tanh3 = nn.Tanh()
        self.fc4 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out = self.fc1(x)
        out = self.tanh1(out)
        out = self.fc2(out)
        out = self.tanh2(out)
        out = self.fc3(out)
        out = self.tanh3(out)
        out = self.fc4(out)
        return out

# Instantiate model
input_dim = 28*28
hidden_dim = 100
output_dim = 10
model = FeedforwardModelTanh(input_dim, hidden_dim, output_dim)

# Instantiate CrossEntropyLoss
criterion = nn.CrossEntropyLoss()

# Instantiate optimizer
learning_rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Load model weights
filepath = 'fnn_model_weights_MNIST_3hidden_tanh.pth'
model.load_state_dict(torch.load(filepath))
```

```python
# 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))

# Test model with a single image
img_1, label_1 = test_dataset[0]
outputs = model(img_1.view(-1, 28*28).requires_grad_())
_, predicted = torch.max(outputs.data, 1)
print('Prediction: {}. Labels: {}'.format(predicted, label_1))
```

```python
# Test model with a single image
img_1, label_1 = test_dataset[0]
outputs = model(img_1.view(-1, 28*28).requires_grad_())
_, predicted = torch.max(outputs.data, 1)
print('Prediction: {}. Labels: {}'.format(predicted, label_1))

# Test model with a data batch
for idx, data in enumerate(test_dataset):
    images, labels = data
    outputs = model(images.view(-1, 28*28).requires_grad_())
    _, predicted = torch.max(outputs.data, 1)
    print('Prediction: {}. Labels: {}'.format(predicted, labels))
    if idx > batch_size:
        break

# Save trained model weights
torch.save(model.state_dict(), filepath)
```

Training Results:

```
Iteration: 500. Loss: 1.3279650211334229. Accuracy: 64.9
Iteration: 1000. Loss: 0.8662739396095276. Accuracy: 80.41
Iteration: 1500. Loss: 0.4800269901752472. 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.