C50875135

Homework 1 Report

Mohammad Hossein Abbasi

C50875135

Deep Learning (CPSC 8430)

Prof. Feng Luo

February 24

2024

GitHub Link

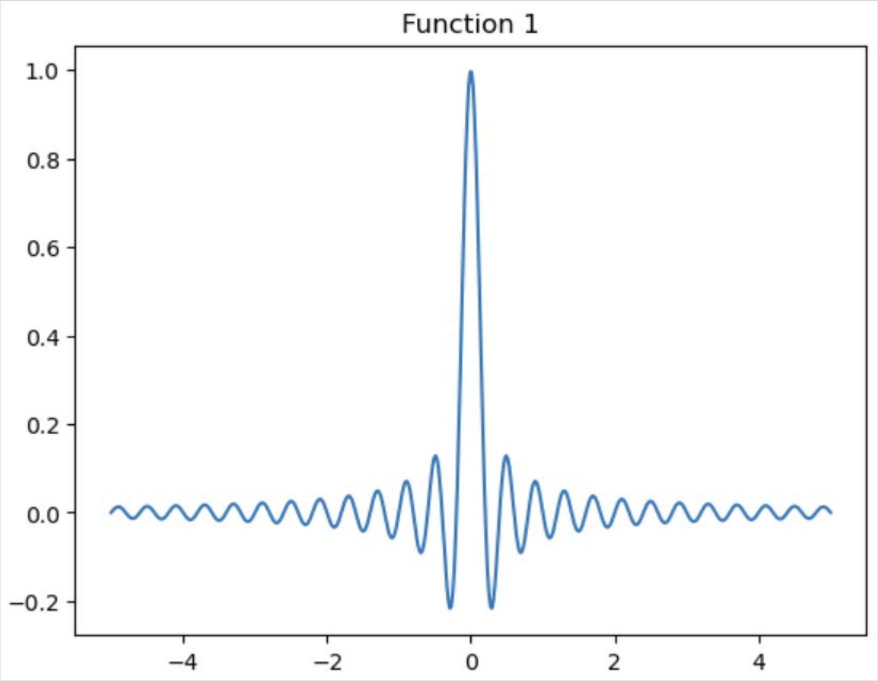
https://github.com/hosseinAB30/Deep-Learning-CPSC-8430

Outline

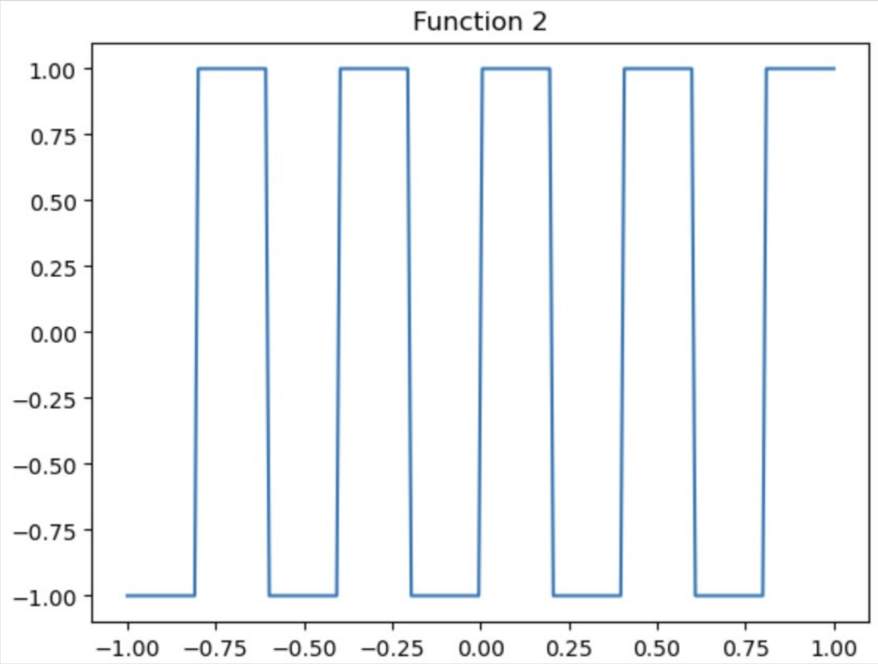
- Deep vs Shallow
  - Simulate a function
  - Train on actual task
- Optimization
  - Visualize the optimization process
  - Observe gradient norm during training
  - What happens when gradient is almost zero?
- Generalization
  - Can network fit random labels?
  - Number of parameters vs Generalization
  - Flatness vs Generalization

Simulate a function

The following functions are used



$$\frac{\sin(5\pi x)}{5\pi x}$$



$$\mathrm{sgn}(\sin(5\pi x))$$

Simulate a function

The following three models are trained to simulate the two functions:

```
class model1(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
                nn.Linear(1,5),
                nn.ReLU(),
                nn.Linear(5,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,5),
                nn.ReLU(),
                nn.Linear(5,1)
        )
        self.loss_fnc = nn.MSELoss()

    def forward(self, x):
        return self.layers(x)
```

Number of parameters: 571

```
class model2(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
                nn.Linear(1,10),
                nn.ReLU(),
                nn.Linear(10,18),
                nn.ReLU(),
                nn.Linear(18,15),
                nn.ReLU(),
                nn.Linear(15,4),
                nn.ReLU(),
                nn.Linear(4,1)
        )
        self.loss_fnc = nn.MSELoss()

    def forward(self, x):
        return self.layers(x)
```

Number of parameters: 572

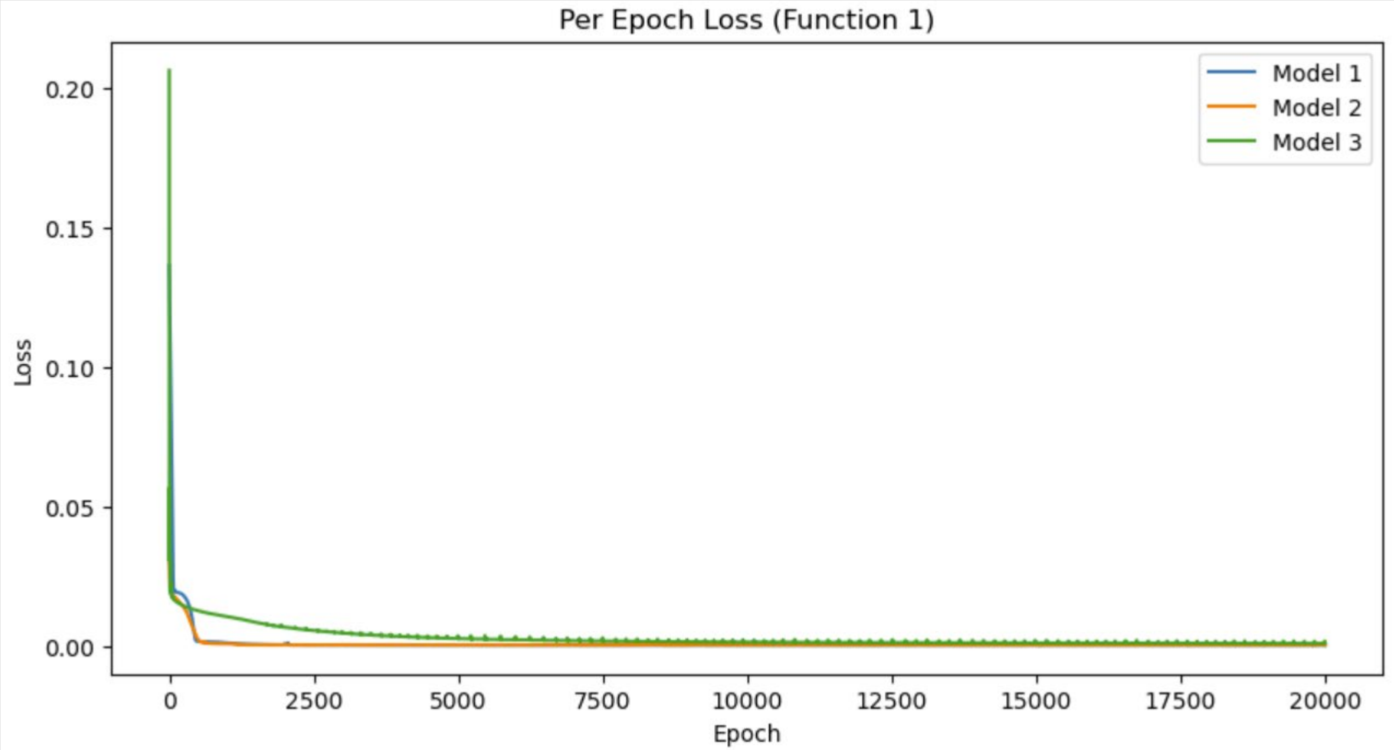
```
class model3(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
                nn.Linear(1,190),
                nn.ReLU(),
                nn.Linear(190,1)
        )
        self.loss_fnc = nn.MSELoss()

    def forward(self, x):
        return self.layers(x)
```

Number of parameters: 571

In all models the following are used:
- Adam optimizer
- Learning rate 1e-3
- Loss function mean square error

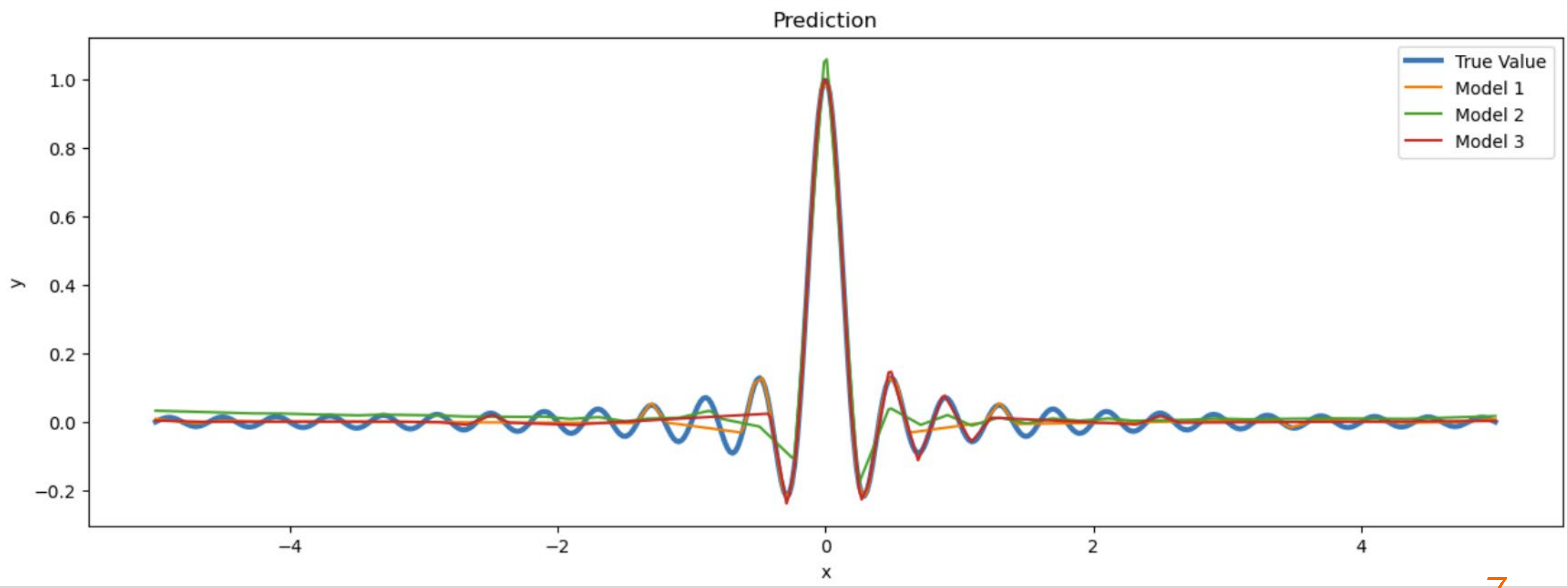
Simulate a function

The results of simulating $\frac{\sin(5\pi x)}{5\pi x}$



Simulate a function

The results of simulating $\frac{\sin(5\pi x)}{5\pi x}$



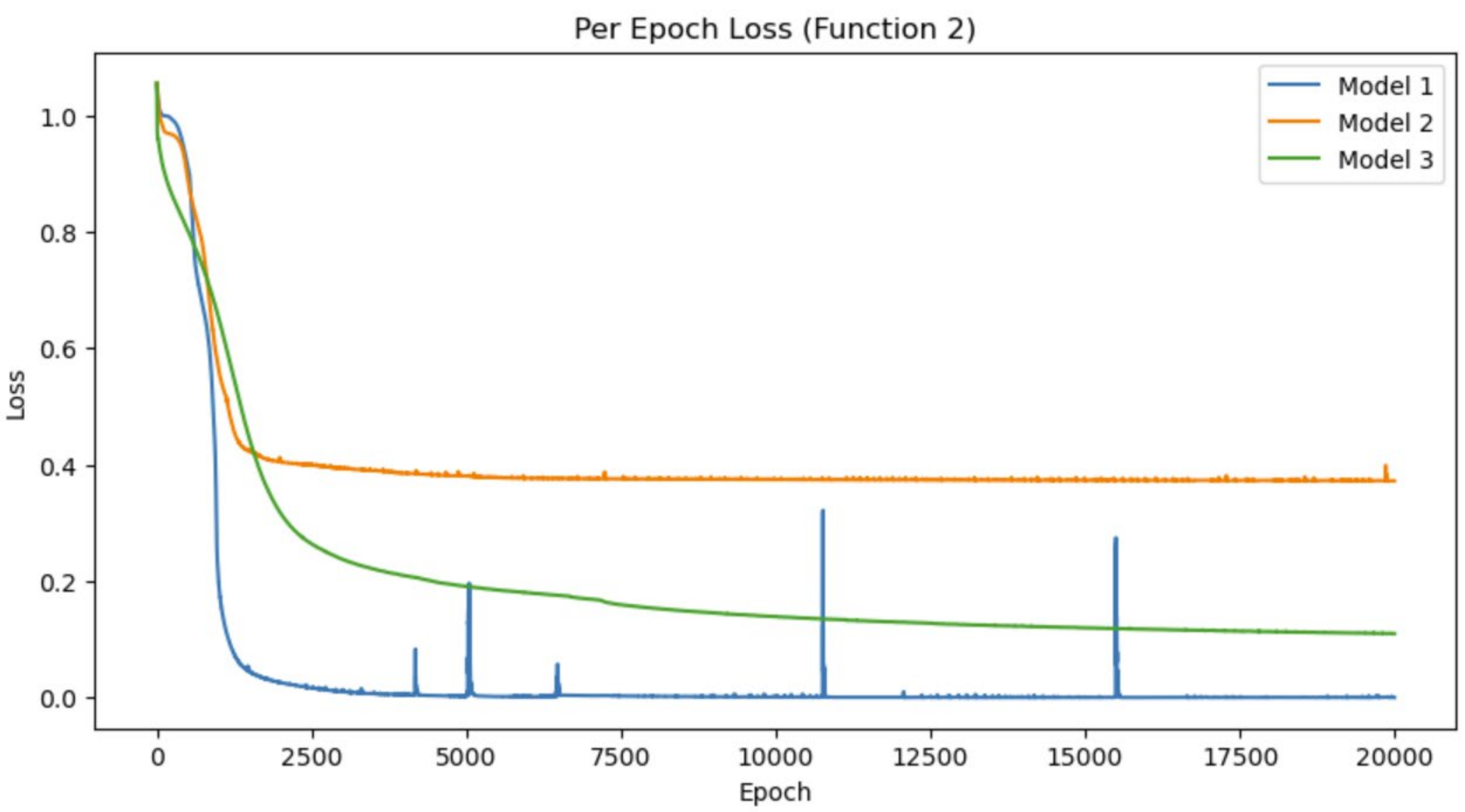
Simulate a function

The results of simulating $\frac{\sin(5\pi x)}{5\pi x}$

Comments: as seen, models 1 and 2 converge faster due to having more layers. Their performance is close, but models 1 and 3 seem to have better results in this case.

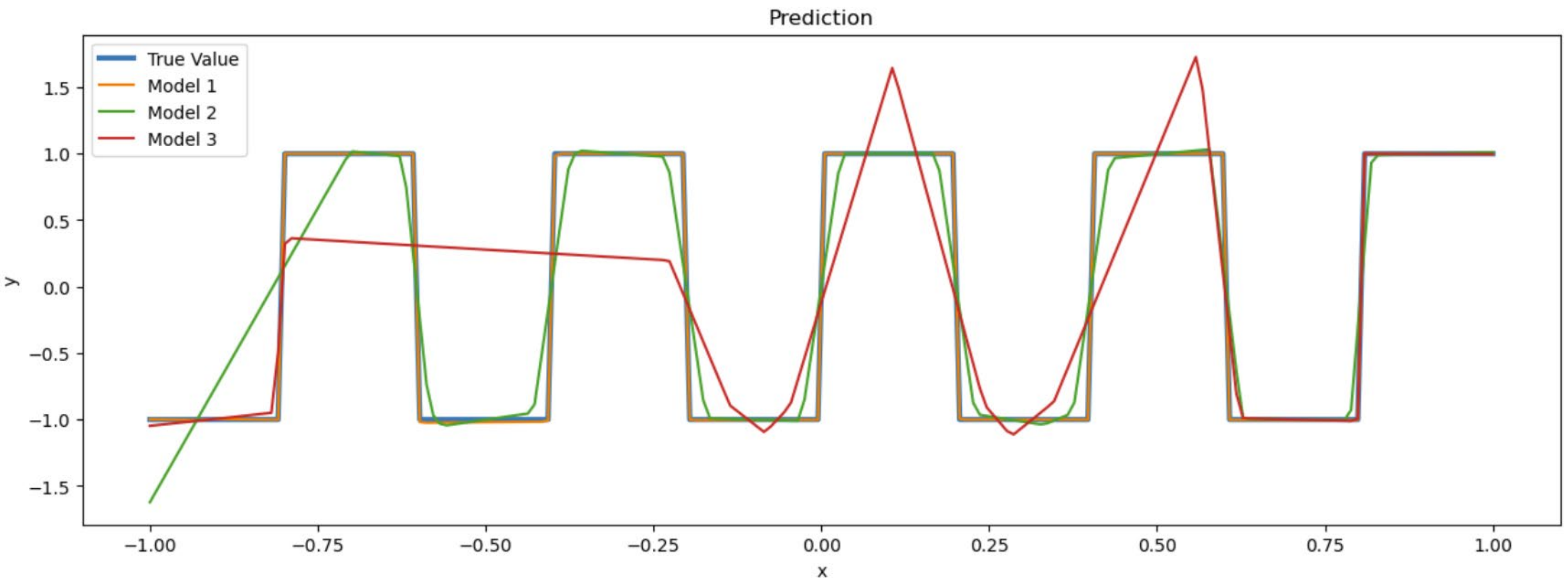
Simulate a function

The results of simulating $\mathrm{sgn}(\sin(5\pi x))$



Simulate a function

The results of simulating $\mathrm{sgn}(\sin(5\pi x))$



Simulate a function

The results of simulating $\mathrm{sgn}(\sin(5\pi x))$

Comments: again, models 1 and 2 converge faster. In terms of performance, model 1 performs best followed by model 2 and 3. Hence, the deepest model performs best as opposed to the shallowest model.

Train on actual task

- I trained two networks on both CIFAR10 and MNIST datasets.
- The two networks for CIFAR10 and the two networks for MNIST are different (four networks in total).

- Networks used for CIFAR10:

```python
class my_cnn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3,32,kernel_size=(3,3),stride=1,padding=1)
        self.act1 = nn.ReLU()
        self.drop1 = nn.Dropout(0.3)

        self.conv2 = nn.Conv2d(32,32,kernel_size=(3,3),stride=1,padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2))

        self.flat = nn.Flatten()

        self.fc3 = nn.Linear(32*16*16,512)
        self.act3 = nn.ReLU()
        self.drop3 = nn.Dropout(0.5)

        self.fc4 = nn.Linear(512,10)

        self.cross_ent = nn.CrossEntropyLoss()
    def forward(self, x):
        x = self.act1(self.conv1(x))
        x = self.drop1(x)             # input  3x32x32, output 32x32x32

        x = self.act2(self.conv2(x)) # input 32x32x32, output 32x32x32
        x = self.pool2(x)             # input 32x32x32, output 32x16x16

        x = self.flat(x)              # input 32x16x16, output 8192

        x = self.act3(self.fc3(x))
        # x = self.drop3(x)             # input 8192,     output 512

        x = self.fc4(x)               # input 512,      output 10
        return x
```

```python
class my_cnn1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3,96,kernel_size=(3,3),stride=1,padding=1)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2))
        self.drop1 = nn.Dropout(0.5)

        self.conv2 = nn.Conv2d(96,80,kernel_size=(3,3),stride=1,padding=1)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2))
        self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(80,96,kernel_size=(3,3),stride=1,padding=1)
        self.act3 = nn.ReLU()

        self.conv4 = nn.Conv2d(96,64,kernel_size=(3,3),stride=1,padding=1)
        self.act4 = nn.ReLU()

        self.flat = nn.Flatten()

        self.fc5 = nn.Linear(64*8*8,512)
        self.act5 = nn.ReLU()

        self.fc6 = nn.Linear(512,10)

        self.cross_ent = nn.CrossEntropyLoss()
    def forward(self, x):
        x = self.act1(self.conv1(x)) # input  3x32x32, output 96x32x32
        x = self.pool1(x)             # input  3x32x32, output 96x16x16
        # x = self.drop1(x)             # input 96x16x16, output 96x16x16

        x = self.act2(self.conv2(x)) # input 96x16x16, output 80x16x16
        x = self.pool2(x)             # input 80x16x16, output 80x8x8
        # x = self.drop2(x)             # input 80x8x8,   output 80x8x8

        x = self.act3(self.conv3(x)) # input 80x8x8,   output 96x8x8

        x = self.act4(self.conv4(x)) # input 96x8x8,   output 64x8x8

        x = self.flat(x)              # input 64x8x8,   output 4096

        x = self.act5(self.fc5(x))   # input 4096,     output 512

        x = self.fc6(x)               # input 512,      output 10
        return x
```

C50875135

Train on actual task

Networks used for MNIST:

For both CIFAR10 and MNIST the following are used:
- Adam optimizer
- Learning rate 1e-4
- Cross entropy loss

For CIFAR10:
- Epochs = 70

For MNIST:
- Epochs = 10

```python
class my_cnn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1,10,3)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2))

        self.conv2 = nn.Conv2d(10,16,3)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2))

        self.flat = nn.Flatten()

        self.fc3 = nn.Linear(16*5*5,512)
        self.act3 = nn.ReLU()

        self.fc4 = nn.Linear(512,256)
        self.act4 = nn.ReLU()

        self.fc5 = nn.Linear(256,125)
        self.act5 = nn.ReLU()

        self.fc6 = nn.Linear(125,10)

        self.cross_ent = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.act1(self.conv1(x))
        x = self.pool1(x)

        x = self.act2(self.conv2(x))
        x = self.pool2(x)

        x = self.flat(x)

        x = self.act3(self.fc3(x))
        x = self.act4(self.fc4(x))
        x = self.act5(self.fc5(x))
        x = self.fc6(x)
        return x
```

```python
class my_cnn1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1,32,5)
        self.act1 = nn.ReLU()

        self.conv2 = nn.Conv2d(32,32,5)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2))
        self.drop2 = nn.Dropout(0.5)

        self.conv3 = nn.Conv2d(32,64,5)
        self.act3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=(2,2))
        self.drop3 = nn.Dropout(0.5)

        self.flat = nn.Flatten()

        self.fc4 = nn.Linear(576,256)
        self.act4 = nn.ReLU()
        self.drop4 = nn.Dropout(0.5)

        self.fc5 = nn.Linear(256,10)

        self.cross_ent = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.act1(self.conv1(x))

        x = self.act2(self.conv2(x))
        x = self.pool2(x)
        x = self.drop2(x)

        x = self.act3(self.conv3(x))
        x = self.pool3(x)
        x = self.drop3(x)

        x = self.flat(x)

        x = self.act4(self.fc4(x))
        x = self.drop4(x)

        x = self.fc5(x)
        return x
```

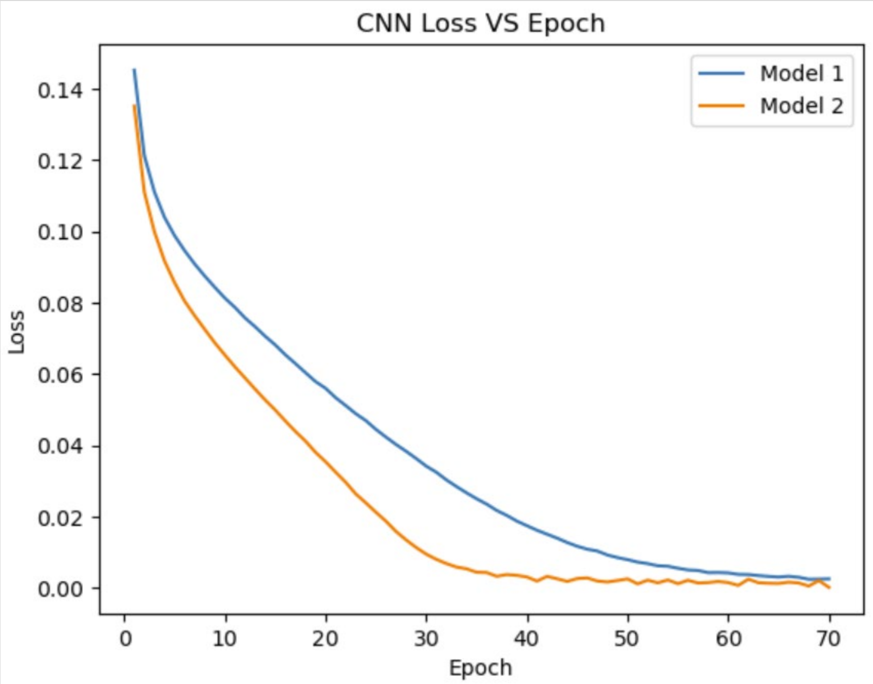
C50875135

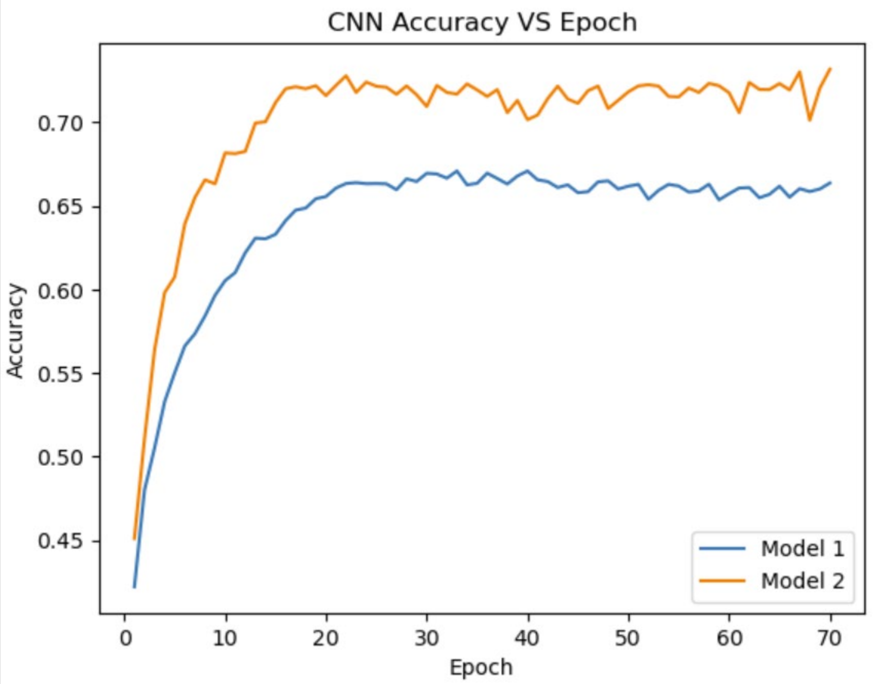
Train on actual task

CIFAR10 results

Comments: model 2 has more layers and more parameters. It converges faster and yields higher accuracy when compared with model 1 results.

- Model 1 accuracy: 66.37%
- Model 2 accuracy: 73.18%



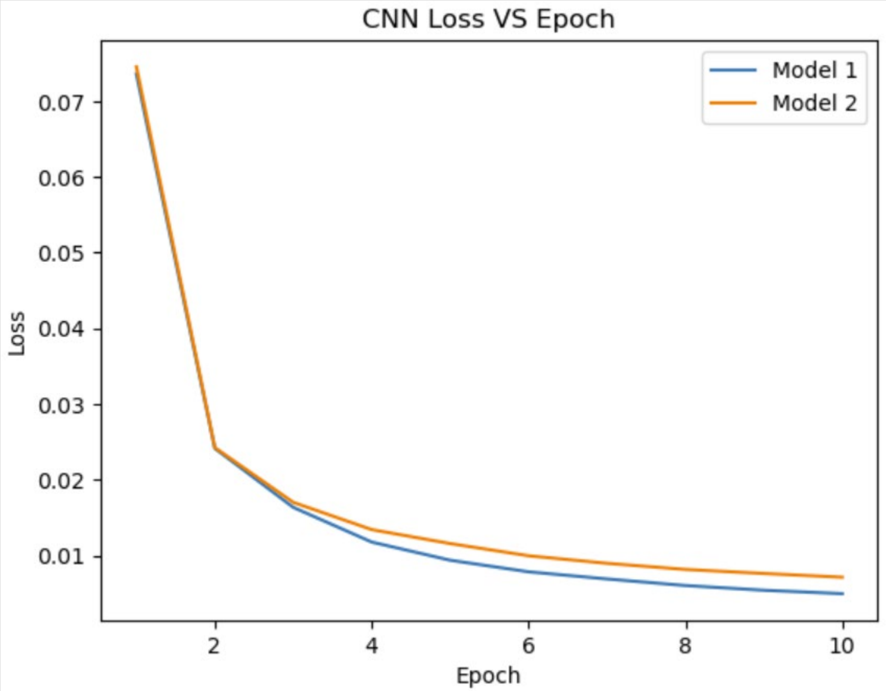


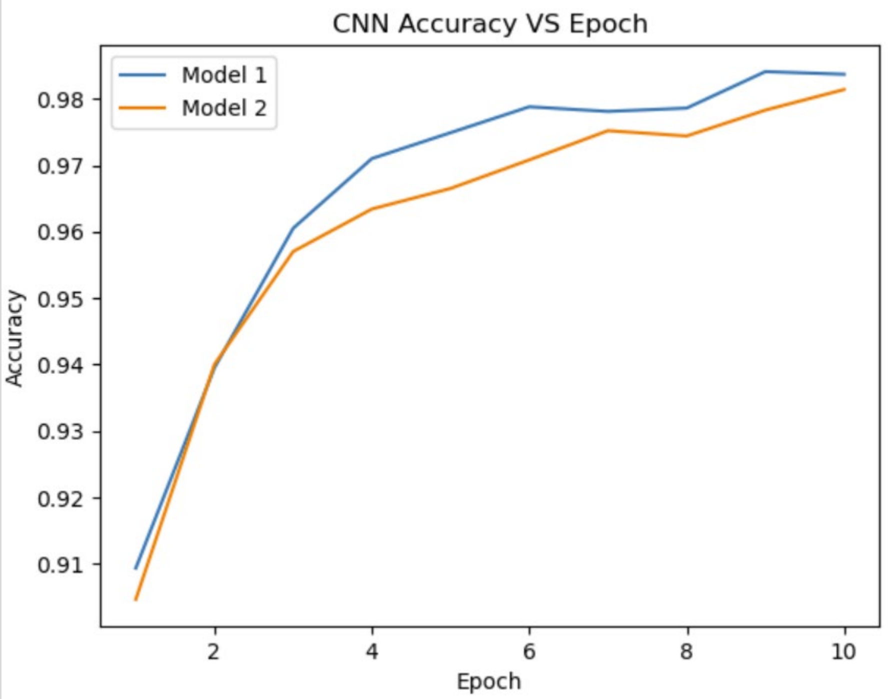
Train on actual task

MNIST results

Comments: model 1 has one more layer; thus, it converges faster and yields higher accuracy when compared with model 2.

- Model 1 accuracy: 98.37%
- Model 2 accuracy: 98.14%





Visualize the optimization process

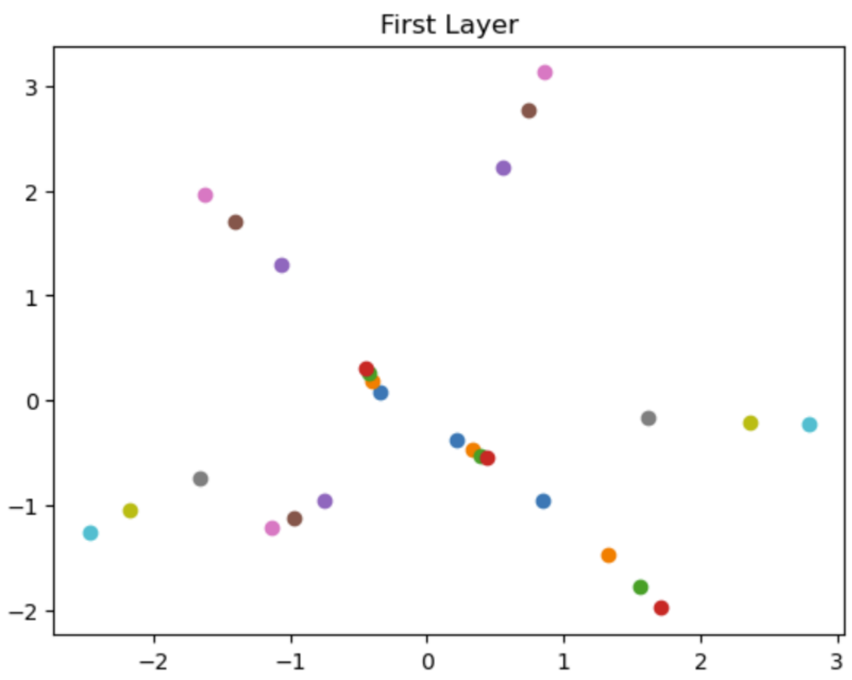
- MNIST dataset is used for this task
- Model parameters are collected every three epochs (total epochs = 30)
- Adam optimizer with the learning rate of 1e-4 is employed
- Dimension reduction is performed with the help of principal component analysis (PCA) approach. PCA is implemented in the code utilizing *sklearn*
- The model is trained 8 times and the results for the first layer and the whole model are plotted

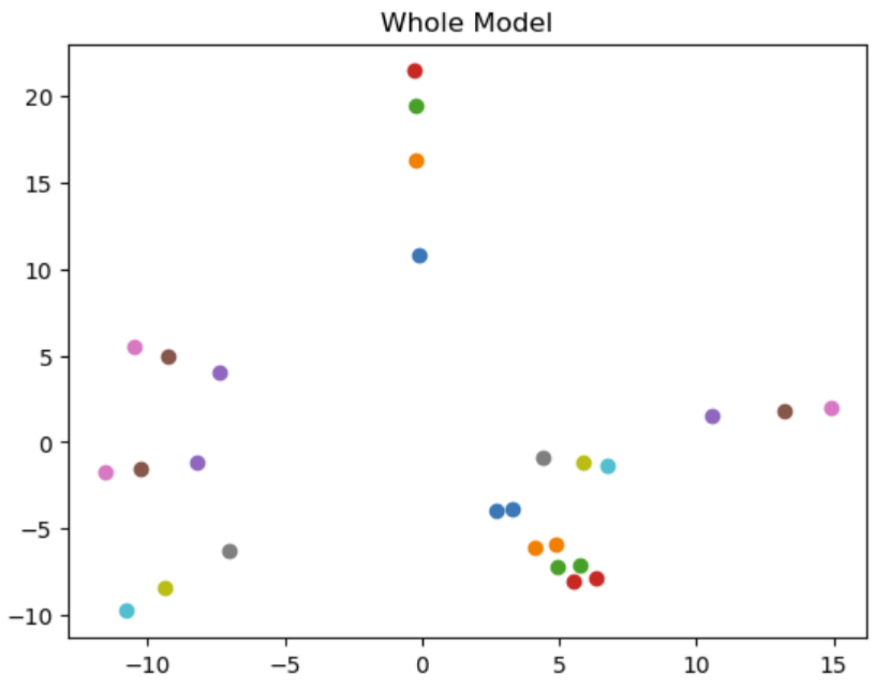
```
class my_cnn(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784,256)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(256,128)
        self.act2 = nn.ReLU()
        self.fc3 = nn.Linear(128,10)

        self.cross_ent = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.act1(self.fc1(x))
        x = self.act2(self.fc2(x))
        x = self.fc3(x)
        return x
```

The underlying DNN

Visualize the optimization process

Results

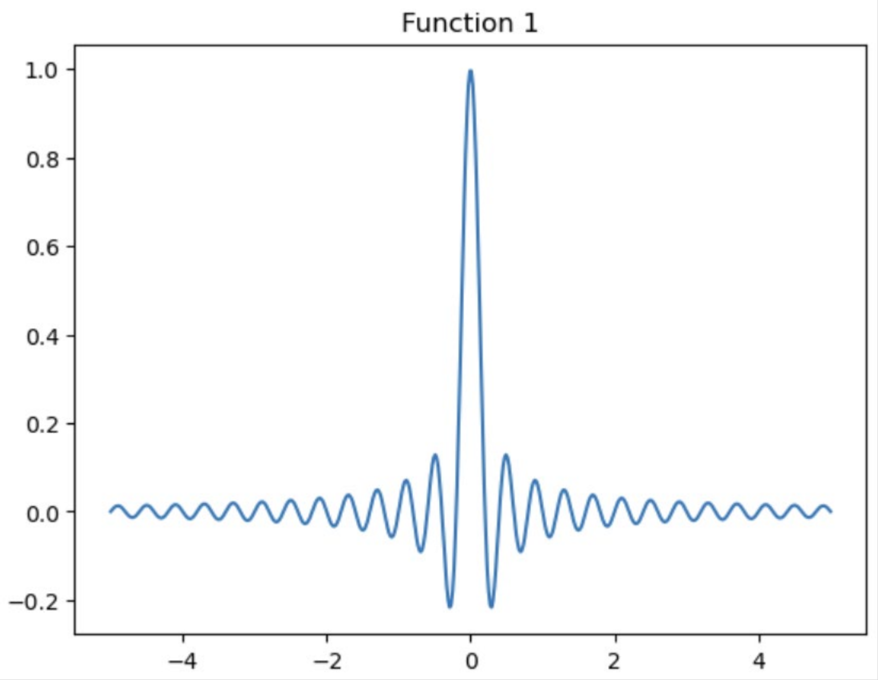




Observe gradient norm during training

The following function and fully connected network are considered

$$\frac{\sin(5\pi x)}{5\pi x}$$

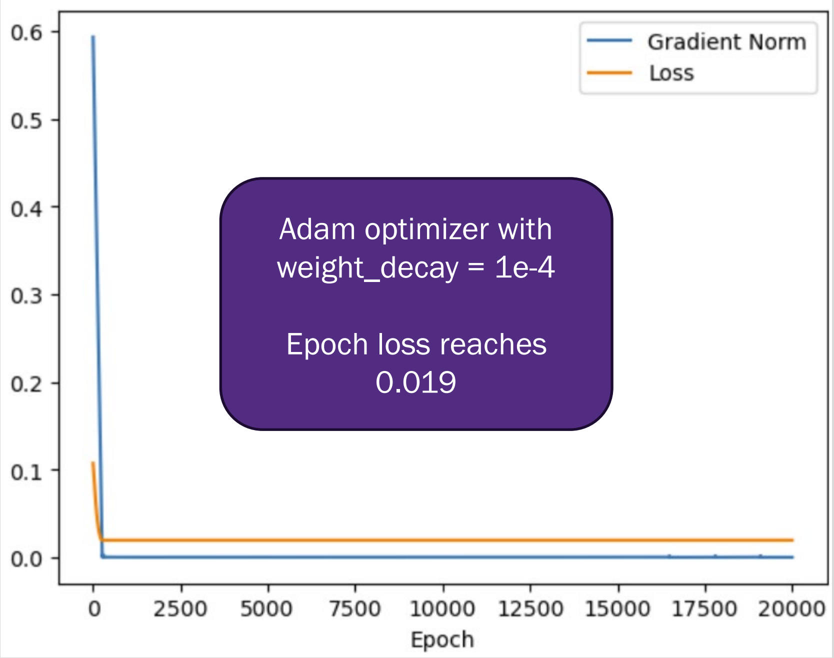


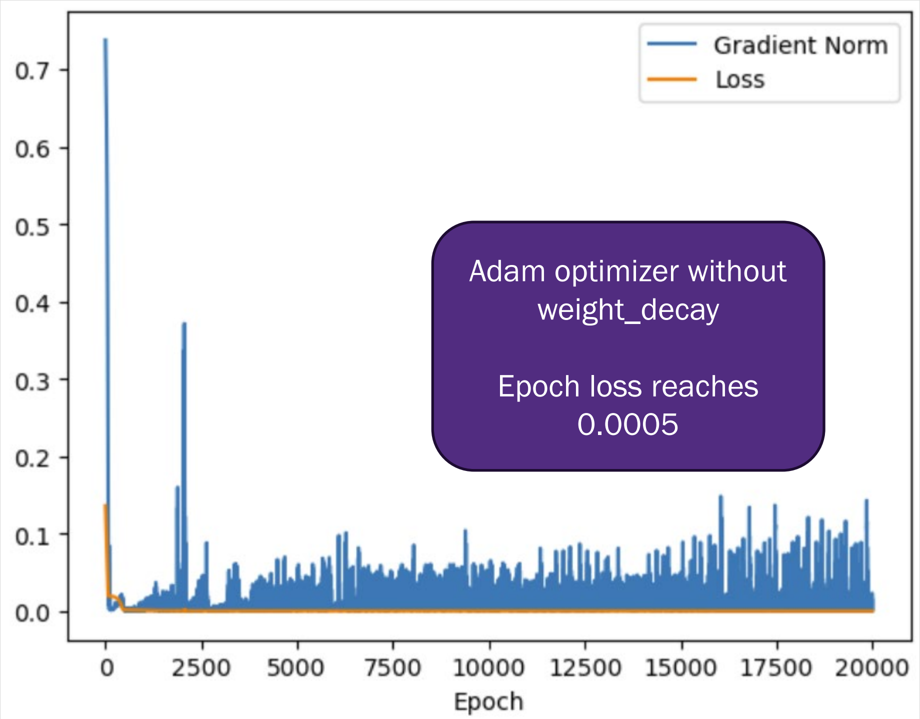

```python
class model1(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
                nn.Linear(1,5),
                nn.ReLU(),
                nn.Linear(5,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,5),
                nn.ReLU(),
                nn.Linear(5,1)
        )
        self.loss_fnc = nn.MSELoss()

    def forward(self, x):
        return self.layers(x)
```

Observe gradient norm during training

The results are obtained with the following settings:

- Adam optimizer with learning rate of 1e-3
- Mean square error loss
- Number of epochs = 20,000

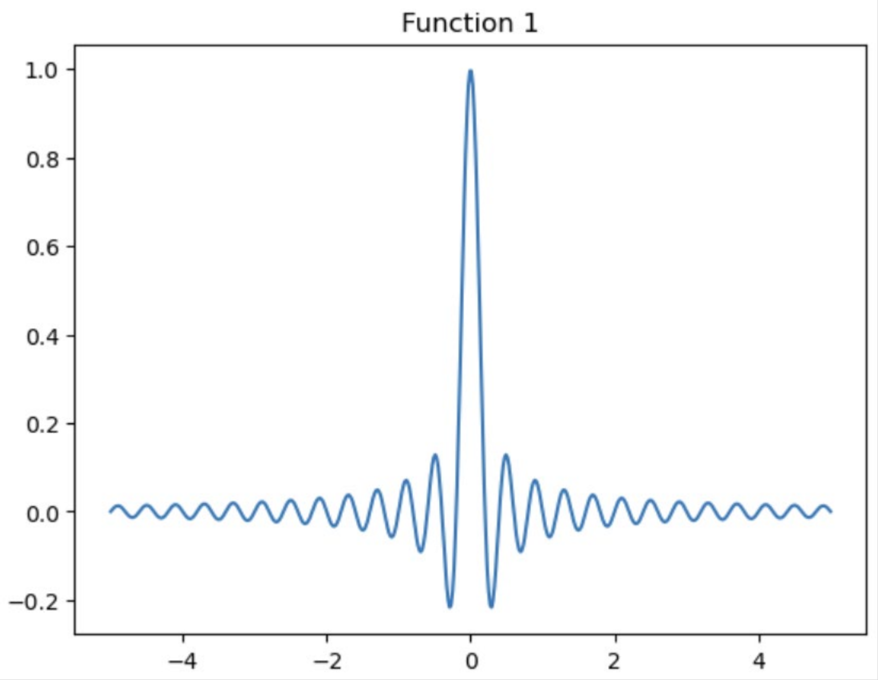




What happens when gradient is almost zero?

The following function and fully connected network are considered

$$\frac{\sin(5\pi x)}{5\pi x}$$




```python
class model1(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
                nn.Linear(1,5),
                nn.ReLU(),
                nn.Linear(5,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,10),
                nn.ReLU(),
                nn.Linear(10,5),
                nn.ReLU(),
                nn.Linear(5,1)
        )
        self.loss_fnc = nn.MSELoss()

    def forward(self, x):
        return self.layers(x)
```

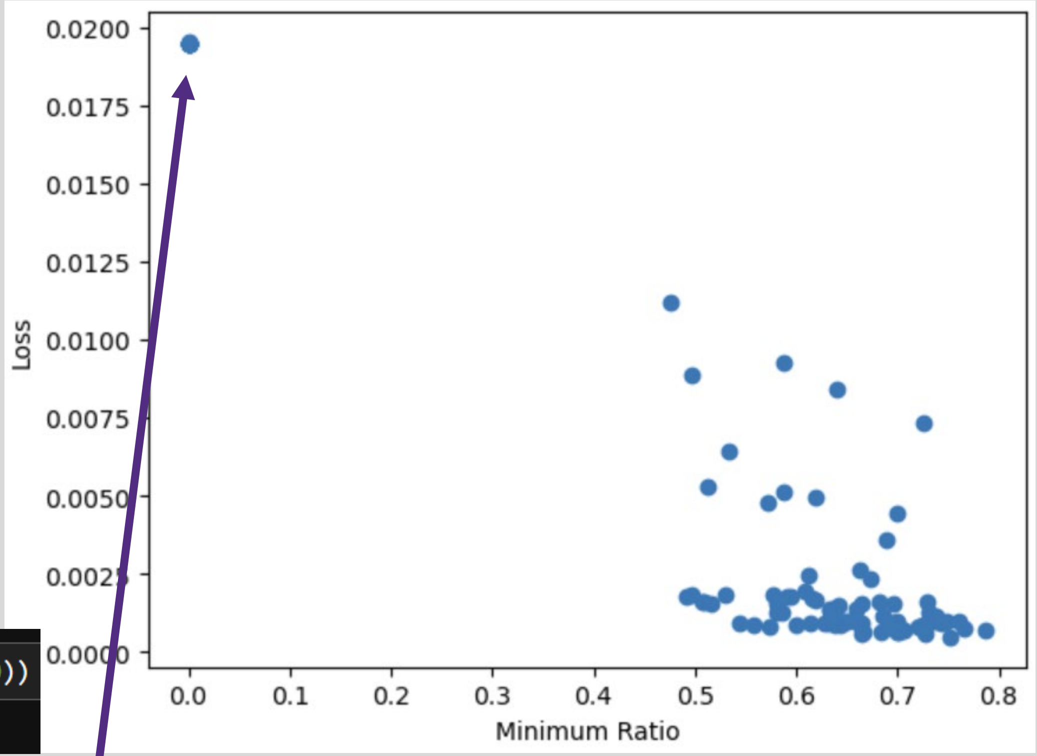
What happens when gradient is almost zero?

- The training is repeated 100 times with 1000 epochs each time. After each training, loss, gradient norm and minimum ratio are stored.
- The following steps, out of 100 steps, reach gradient of zero:

```
print(', '.join(str(i) for i, v in enumerate(grad_norms) if v == 0))
4, 30, 40, 42, 46, 49, 50, 53, 64, 73, 92, 95
```

- When gradient is zero, minimum ratio is zero too:

```
print(', '.join(str(min_ratios[i].item()) for i, v in enumerate(grad_norms) if v == 0))
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
```



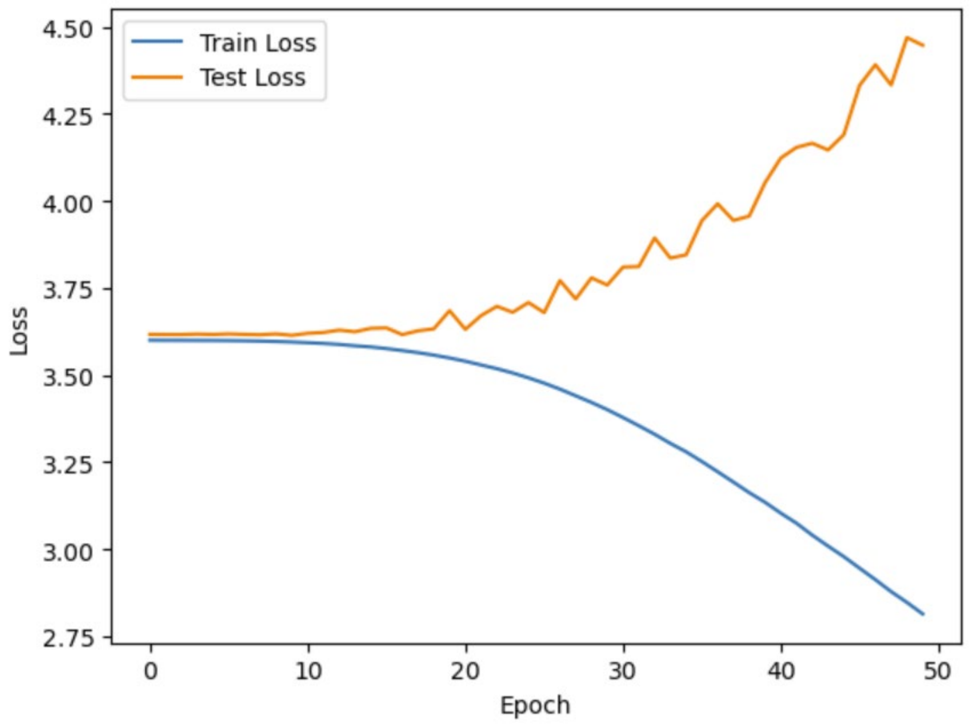
- And loss is equal to 0.019474269822239876

Can network fit random labels?

- MNIST is selected
- The following network is used
  - Adam optimizer is used with learning rate of 1e-4
  - Cross entropy loss is used
  - Number of epochs = 50

```python
class my_cnn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1,10,3)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2))

        self.conv2 = nn.Conv2d(10,16,3)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2))

        self.flat = nn.Flatten()

        self.fc3 = nn.Linear(16*5*5,512)
        self.act3 = nn.ReLU()

        self.fc4 = nn.Linear(512,256)
        self.act4 = nn.ReLU()

        self.fc5 = nn.Linear(256,125)
        self.act5 = nn.ReLU()

        self.fc6 = nn.Linear(125,10)

        self.cross_ent = nn.CrossEntropyLoss()
    def forward(self, x):
        x = self.act1(self.conv1(x))
        x = self.pool1(x)

        x = self.act2(self.conv2(x))
        x = self.pool2(x)

        x = self.flat(x)

        x = self.act3(self.fc3(x))
        x = self.act4(self.fc4(x))
        x = self.act5(self.fc5(x))
        x = self.fc6(x)
        return x
```

The network cannot fit random labels



Number of parameters vs Generalization

- MNIST is used for this task.
- Each network has three fully-connected layers where the number of hidden neurons differ
- The following setting is identical in all ten networks:
  - Cross entropy loss
  - Adam optimizer with learning rate of 1e-4
  - Training with 10 epochs

```python
class m1(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784,4)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(4,8)
        self.act2 = nn.ReLU()
        self.fc3 = nn.Linear(8,10)
        self.cross_ent = nn.CrossEntropyLoss()
    def forward(self, x):
        x = self.act1(self.fc1(x))
        x = self.act2(self.fc2(x))
        x = self.fc3(x)
        return x
```

Networks 1 and 10

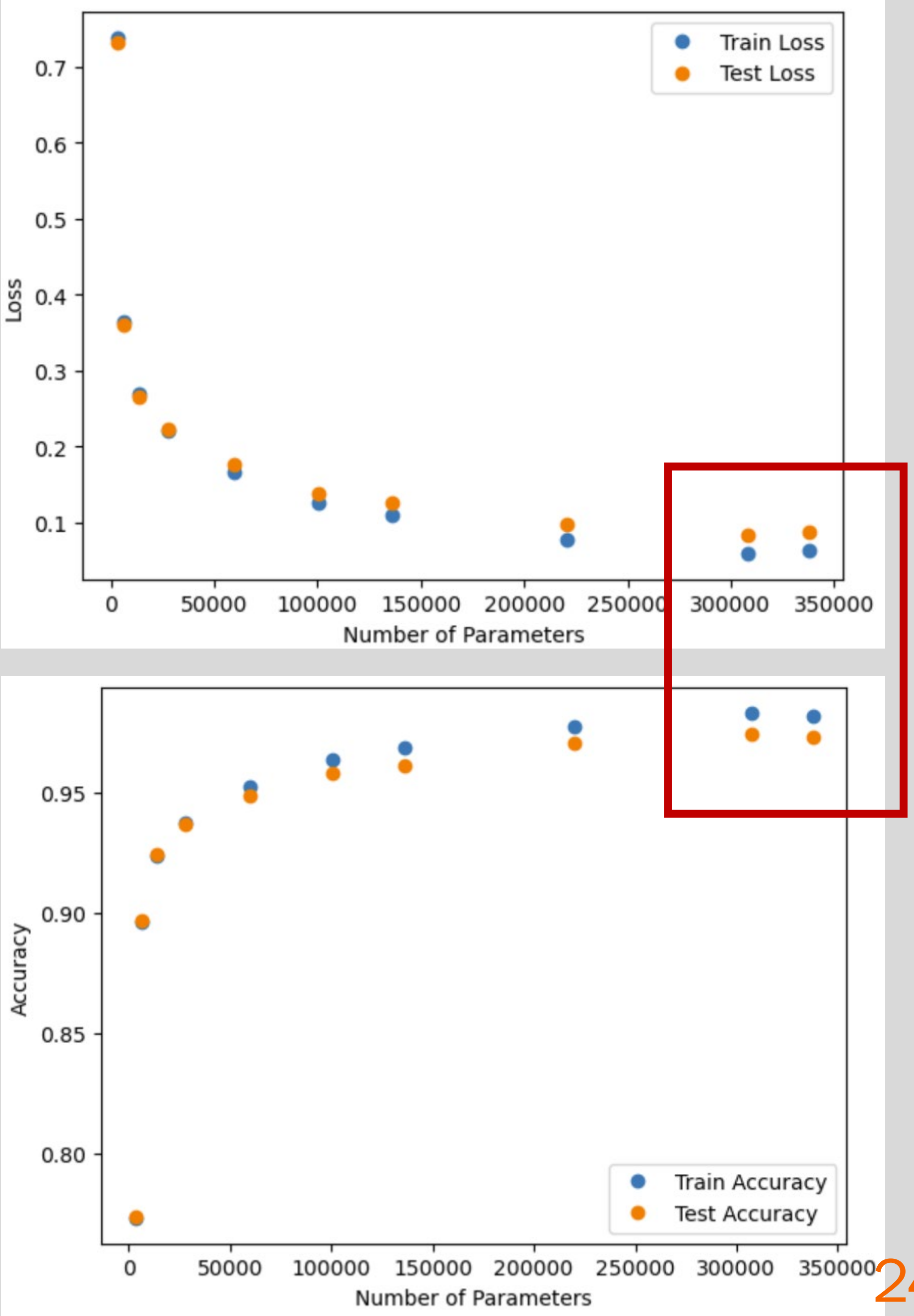
```python
class m10(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784,256)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(256,512)
        self.act2 = nn.ReLU()
        self.fc3 = nn.Linear(512,10)
        self.cross_ent = nn.CrossEntropyLoss()
    def forward(self, x):
        x = self.act1(self.fc1(x))
        x = self.act2(self.fc2(x))
        x = self.fc3(x)
        return x
```

CS0875135

Number of parameters vs Generalization

The results

Comments: as the number of parameters increases the test accuracy start to fall short of train accuracy because the network begins overfitting. In the last two highlighted instances, it is observed that the results are plateauing. Further increase of the networks parameters may or may not improve the results.



Flatness vs Generalization (part 1)

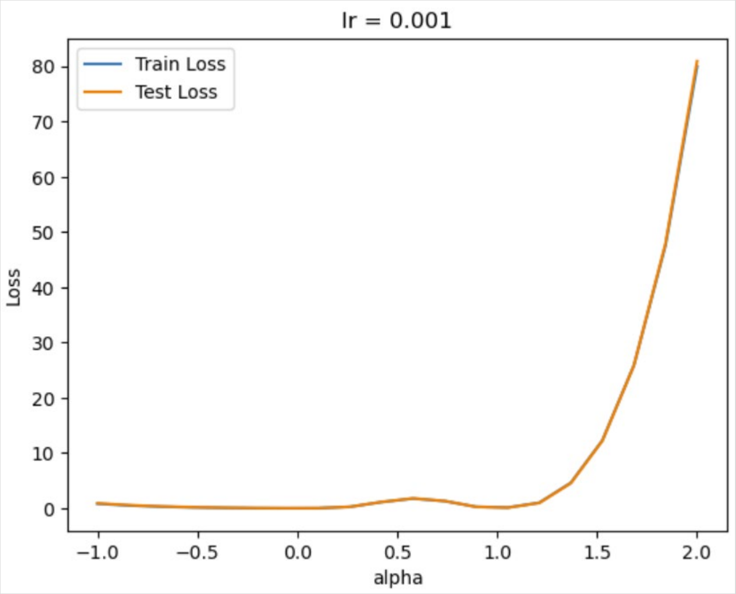
- MNIST is used
- Two different datasets are created with batch sizes of 64 and 1024, respectively.
- Adam optimizer is employed with two different learning rates of 1e-2 and 1e-3.
- Cross entropy loss is utilized
- Number of epochs = 10

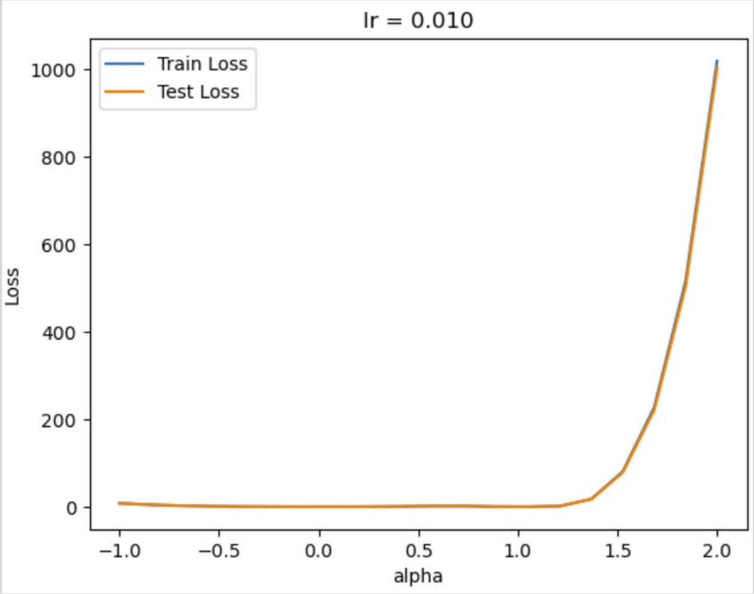
```python
class my_cnn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1,10,3)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2))

        self.conv2 = nn.Conv2d(10,16,3)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2))

        self.flat = nn.Flatten()

        self.fc3 = nn.Linear(16*5*5,32)
        self.act3 = nn.ReLU()

        self.fc4 = nn.Linear(32,10)

        self.cross_ent = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.act1(self.conv1(x))
        x = self.pool1(x)

        x = self.act2(self.conv2(x))
        x = self.pool2(x)

        x = self.flat(x)

        x = self.act3(self.fc3(x))
        x = self.fc4(x)
        return x
```

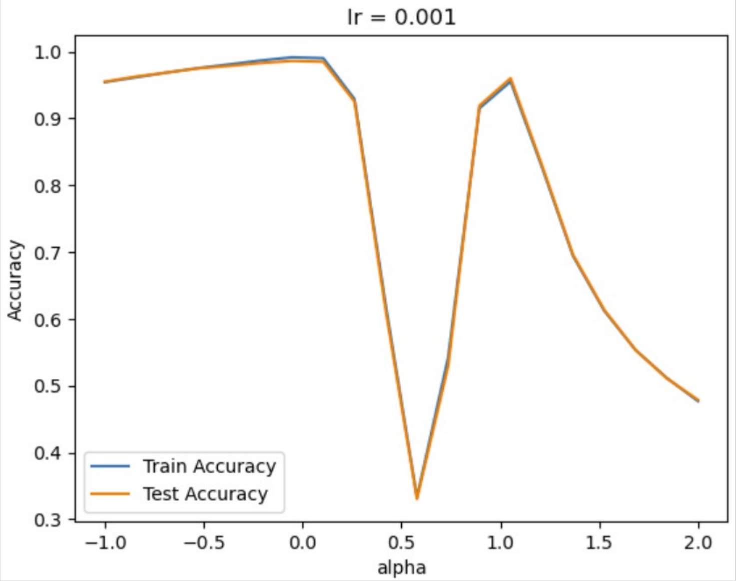
Flatness vs Generalization (part 1)

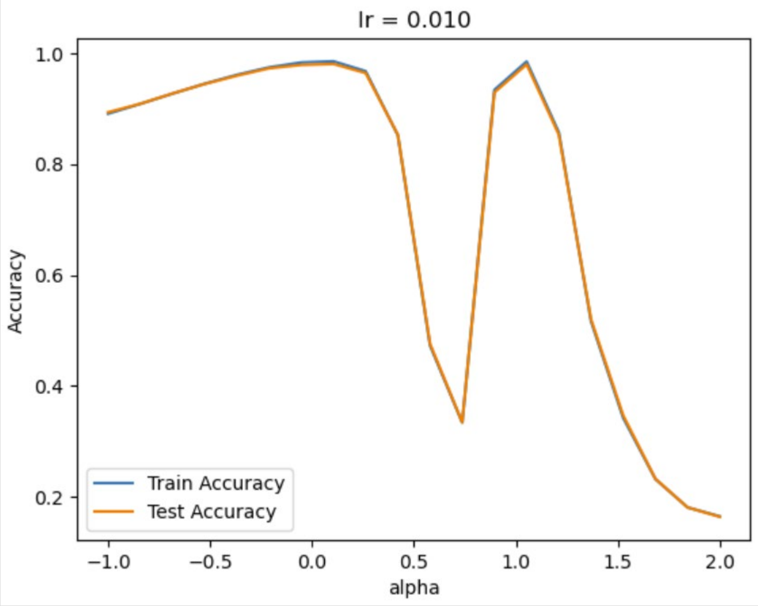
The results

Comments: in both cases of learning rates, the loss exhibits increasing trend, whereas the accuracy fluctuates and drops by the increase in loss. The loss elevation and the accuracy drop are more pronounced when lr=0.01. In both cases, the maximum accuracy is reached when $0 \leq \alpha \leq 0.5$. Increasing α beyond 1.5 continuously deteriorates the results. Additionally, when accuracy is close to 1, overfitting is visible where test accuracy slightly becomes less than train accuracy.









Flatness vs Generalization (part 2)

- MNIST is used
- Four different batch sizes are utilized to generate different training approaches
    - The batch sizes are [32, 64, 128, 256, 512]
- Adam optimizer with learning rate of 1e-4 is employed
- Cross entropy loss is used

```python
class my_cnn(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1,10,3)
        self.act1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=(2,2))

        self.conv2 = nn.Conv2d(10,16,3)
        self.act2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=(2,2))

        self.flat = nn.Flatten()

        self.fc3 = nn.Linear(16*5*5,32)
        self.act3 = nn.ReLU()

        self.fc4 = nn.Linear(32,10)

        self.cross_ent = nn.CrossEntropyLoss()

    def forward(self, x):
        x = self.act1(self.conv1(x))
        x = self.pool1(x)

        x = self.act2(self.conv2(x))
        x = self.pool2(x)

        x = self.flat(x)

        x = self.act3(self.fc3(x))
        x = self.fc4(x)
        return x
```

Flatness vs Generalization (part 2)

The results

Comments: larger batch size negatively impacts model's accuracy, i.e., a model with smaller batch size can learn faster. In contrast, large batch size generally reduces sensitivity.