Applied Machine Learning

Homework 7

Anand Kamble
Department of Scientific Computing
Florida State University

Introduction

In this assignment, we are training a regression neural network (NN) to predict pixel values of an image given its coordinates. We utilized the square loss function for optimization and experiment with different numbers of hidden layers and neurons. The image used was 'horse033b.png' with dimensions 128×93, resulting in 11,904 pixels. The task is to train the network using Stochastic Gradient Descent (SGD) and display the results for each configuration.

Environment Setup

The environment setup was as follows:

```
conda create -n "homework7" python=3.11 &&\
conda activate homework7 &&\
pip install numpy torch torchvision matplotlib tqdm
```

The code for this homework is available on GitHub:
`https://github.com/anand-kamble/FSU-assignments/blob/main/Machine%20Learning/HW07/main.py`

Implementation

We start by loading the grayscale image 'horse033b.png' (size 128×93) and converting it into 'numpy' array. We create input coordinates (x, y) using a coordinate grid, and both the input coordinates and pixel intensities are standardized to improve the training process. The inputs are standardized to have zero mean and unit variance, while the pixel intensities are centered by subtracting the mean.

```
# Load the image
image = Image.open('horse033b.png').convert('L')  # Convert to grayscale
image = np.array(image)

# Get image dimensions (height x width)
height, width = image.shape  # 93 x 128

# Create coordinate grid
x_coords = np.arange(width)
y_coords = np.arange(height)
xx, yy = np.meshgrid(x_coords, y_coords)
```

Listing 1: Image Loading and Coordinate Grid

Then we flatten and standardize the coordinates and pixel intensities. We also ensure that the inputs have zero mean and unit variance, while the targets (pixel values) are centered by subtracting their mean.

```
# Flatten and stack input coordinates
inputs = np.stack([xx.flatten(), yy.flatten()], axis=1)
targets = image.flatten().astype(np.float32)

# Standardize inputs and center targets
inputs_mean = inputs.mean(axis=0)
inputs_std = inputs.std(axis=0)
```

```
inputs_standardized = (inputs - inputs_mean) / inputs_std

targets_mean = targets.mean()
targets_standardized = targets - targets_mean
```

Listing 2: Input and Target Standardization

Then we convert inputs and targets to PyTorch tensors to be used in the neural network training:

```
# Convert to PyTorch tensors
inputs_tensor = torch.from_numpy(inputs_standardized).float()
targets_tensor = torch.from_numpy(targets_standardized).float().unsqueeze(1)
```

Listing 3: Conversion to PyTorch Tensors

Then We create a 'DataLoader' for batching and shuffling the data, which will enable efficient mini-batch training with a batch size of 64:

```
# Create dataset and DataLoader
dataset = TensorDataset(inputs_tensor, targets_tensor)
data_loader = DataLoader(dataset, batch_size=64, shuffle=True)
```

Listing 4: Creating DataLoader

Then we initialize the weights of the linear layers using the Xavier uniform distribution, while the biases are initialized to zero. This technique helps improve the convergence of the network during training. The following code performs the initialization:

```
# Initialize the network
net = Net_d()

# Xavier initialization for weights and zero initialization for biases
def initialize_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            nn.init.constant_(m.bias, 0)

net.apply(initialize_weights)
```

Listing 5: Network and Weight Initialization

For faster computation, we use GPU acceleration if a CUDA-compatible device is available.

```
# Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net.to(device)
```

Listing 6: Device Selection for Training

We use the Mean Squared Error (MSE) loss function since the task is a regression problem where the goal is to predict pixel intensities. Stochastic Gradient Descent (SGD) is used as the optimizer, with an initial learning rate of 0.0001. A learning rate scheduler is applied to reduce the learning rate by half every 100 epochs to help the model converge smoothly.

```
# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(net.parameters(), lr=0.0001)

# Learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)
```

Listing 7: Loss Function and Optimizer Setup

This setup ensures that the training process starts with appropriate initial values and adjusts the learning rate as the training progresses, allowing the model to converge efficiently over the 300 epochs.

Results

I am using a learning rate of 0.0001 for results below. I also tried 0.003 and 0.001 values for learning rate, but I got the best results using 0.0001.

A.

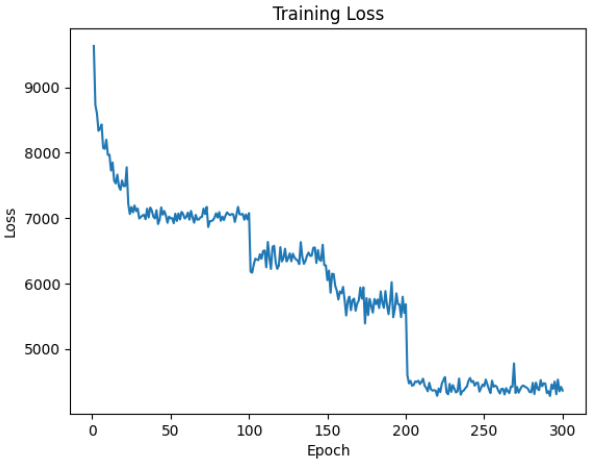
Our first model uses single hidden layer with 128 neurons and ReLU activation. The training loop iterated over 300 epochs with a batch size of 64. The code for Model A is as follows:

```
class Net_a(nn.Module):
    def __init__(self):
        super(Net_a, self).__init__()
        self.fc1 = nn.Linear(2, 128)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

Listing 8: Model A: Single Hidden Layer

The figure below shows the training loss over epochs and the reconstructed image for Model A.



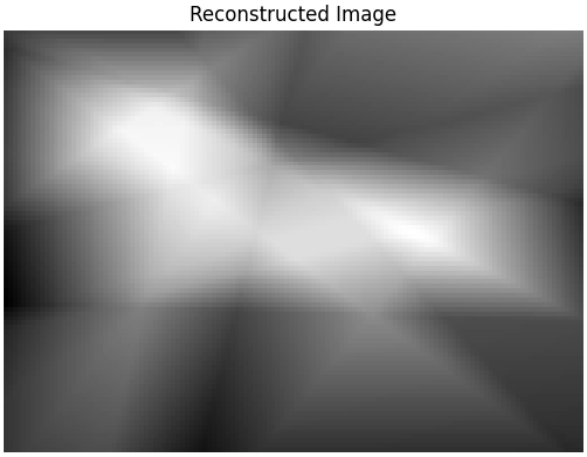


Figure 1: Reconstructed Image for Model A

B.

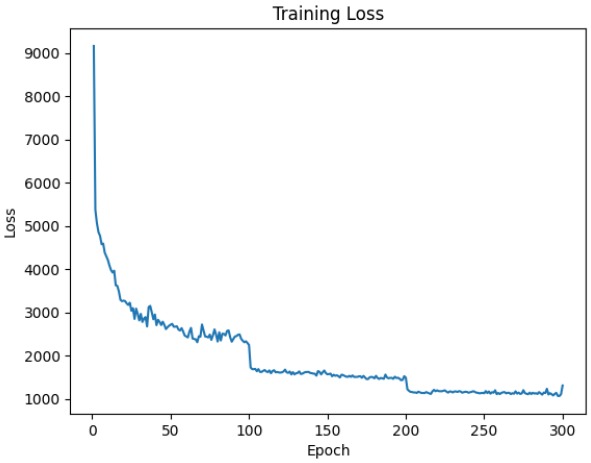
Our second model has two hidden layers: the first with 32 neurons, and the second with 128 neurons, each followed by ReLU. The code for Model B is shown below:

```
class Net_b(nn.Module):
    def __init__(self):
        super(Net_b, self).__init__()
        self.fc1 = nn.Linear(2, 32)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(32, 128)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return x
```

Listing 9: Model B: Two Hidden Layers

The training loss and reconstructed image are presented below.



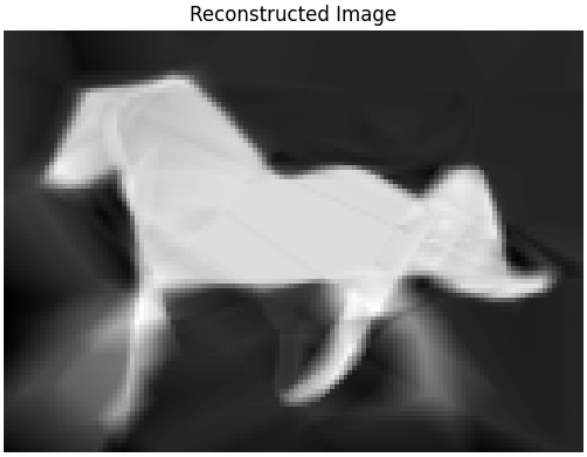


Figure 2: Reconstructed Image for Model B

C.

Our third model has three hidden layers with 32, 64, and 128 neurons respectively. The code for this model is provided below:

```
class Net_c(nn.Module):
    def __init__(self):
        super(Net_c, self).__init__()
        self.fc1 = nn.Linear(2, 32)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(32, 64)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(64, 128)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        x = self.relu3(x)
        x = self.fc4(x)
        return x
```

Listing 10: Model C: Three Hidden Layers

The results for Model C are as follows:

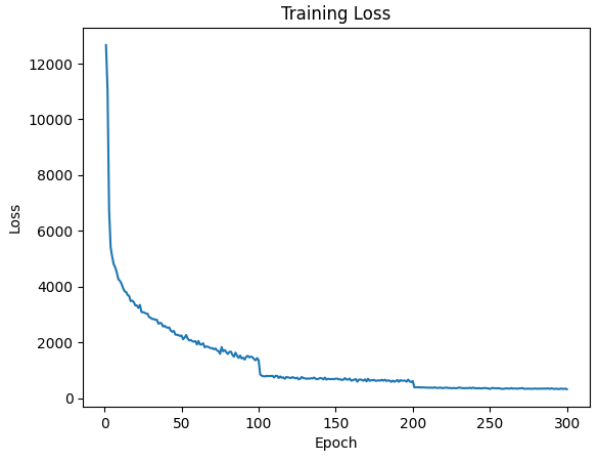


Figure 3: Training Loss vs Epoch for Model C

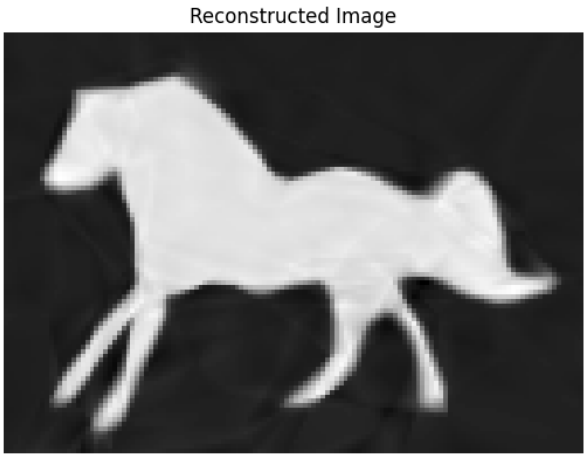


Figure 4: Reconstructed Image for Model C

D.

Our final model has four hidden layers, with neuron counts of 32, 64, 128, and 128 respectively. Below is the code for Model D:

```
class Net_d(nn.Module):
    def __init__(self):
        super(Net_d, self).__init__()
        self.fc1 = nn.Linear(2, 32)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(32, 64)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(64, 128)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(128, 128)
        self.relu4 = nn.ReLU()
        self.fc5 = nn.Linear(128, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        x = self.relu3(x)
        x = self.fc4(x)
        x = self.relu4(x)
        x = self.fc5(x)
        return x
```

Listing 11: Model D: Four Hidden Layers

The following figures show the training loss and the reconstructed image for Model D.

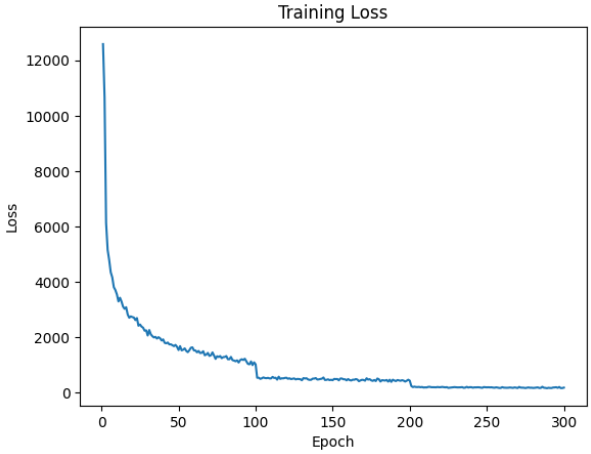


Figure 5: Training Loss vs Epoch for Model D

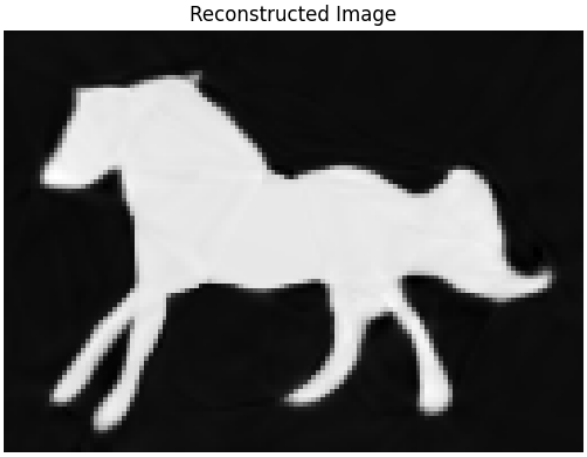


Figure 6: Reconstructed Image for Model D