Homework 2 CSCE 790: Neural Networks and Their Applications

Part A

1

- A new learning procedure called back-propagation is presented for neural network models. It minimizes errors by propagating them backwards to adjust connection weights.
- The network has layers of input, output, and hidden units with weighted connections between them. Input vectors are fed forward to generate output vectors.
- Errors between actual and desired outputs are then propagated backwards. This adjusts the weights and minimizes the errors.
- The method allows the network to learn internal representations that capture important features of the task domain. The hidden units come to represent key features not explicit in the inputs/outputs.
- The procedure is applied to tasks like detecting input symmetry and storing family tree information. It shows how gradient descent can build useful representations.
- The simplest form is for layered feedforward networks with connections skipping layers but no backwards connections. Inputs are fed forward layer-by-layer to get outputs.
- A linear input function combines inputs before applying nonlinearity for simplicity. The aim is outputs close to the desired vectors.
- Errors are defined as the sum of squared differences between actual and desired outputs. Gradient descent minimizes error by changing weights proportionally.
- Partial derivatives of the error with respect to each weight are computed in a forward and backward pass. The backward pass propagates derivatives layer-by-layer.
- Weights are changed proportional to the accumulated derivatives over all cases, improving on simpler methods. An acceleration method further improves convergence.
- Symmetry detection requires intermediate units, as individual inputs alone don't indicate symmetry. The method finds an elegant two-unit solution.
- Family tree information storage shows how features not explicit in inputs/outputs are captured. The network generalizes correctly from limited training.
- Equivalence is shown between layered feedforward networks and recurrent networks run iteratively. The learning procedure extends to recurrent nets.
- Local minima are rarely a big issue, as extra connections provide paths around barriers. Poor minima mainly occur in minimally connected networks.

2

- The paper analyzes backpropagation convergence to explain phenomena observed in practice and give tricks to avoid undesirable behaviors. Many tricks rarely exposed in serious publications can help backpropagation work well.
- Stochastic learning is usually much faster than batch learning, especially on large redundant datasets, as it avoids wasteful recomputation. Stochastic learning also often finds better solutions as noise helps weights jump to deeper local minima.
- Input transformations like subtracting means and decorrelating help by improving conditioning and avoiding biases slowing learning. Normalizing variances balances learning rates and scaling down less useful inputs makes them less visible.
- Symmetric sigmoids often converge faster by producing more centered outputs. Adding a small linear term can help avoid flat spots.

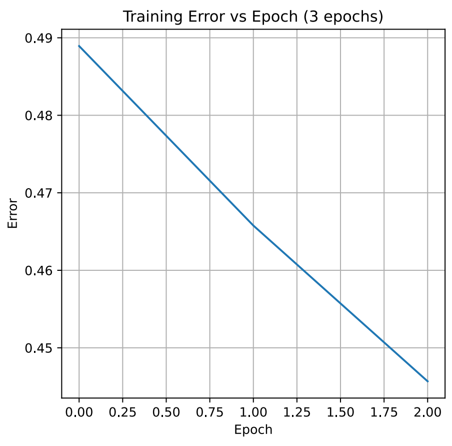
- Target values at sigmoid extremes can cause instability as outputs are driven asymptotically to minimize error. Moderate target values utilize nonlinearity without saturating units.
- Random weight initialization with proper standard deviations starts learning in the sigmoid's linear region before nonlinear parts. This captures easier linear mappings first.
- Learning rates should equalize convergence speeds, being proportional to fan-in and typically larger in lower layers. Adaptive techniques can help track changes and avoid oscillations.
- Momentum damps steps in high curvature directions, increasing the effective rate in low curvature directions. This speeds nonspherical surfaces.
- Conjugate gradient finds descent directions minimizing interference with previous steps through a line search. It works well for small precise problems.
- Quasi-Newton methods estimate the Hessian's inverse without matrix inversion using gradient information. But storage remains impractical for large networks.
- Gauss-Newton uses the Jacobian approximation to the Hessian, requiring means squared error. Levenberg-Marquardt adds regularization to avoid instability.
- Various techniques compute Hessian information without explicit calculation like backpropagating second derivatives. This enables some acceleration.
- Analysis shows problematic effects on the Hessian of non-centered inputs, inter-layer mismatches, and correlations. Tricks address these issues.
- Classical second-order methods have limited usefulness, but stochastic diagonal Levenberg-Marquardt helps by estimating individual learning rates.
- Computing Hessian eigenvectors without the full matrix can estimate optimal learning rates. The power method iteration or running averages work.

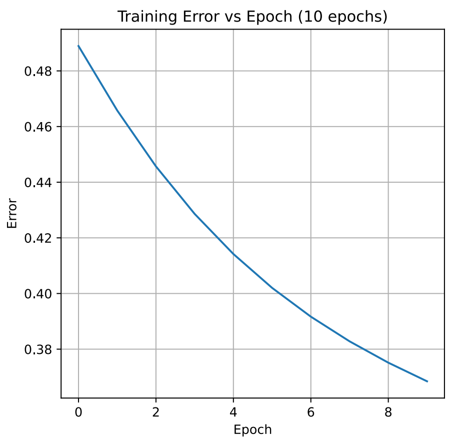
Part B

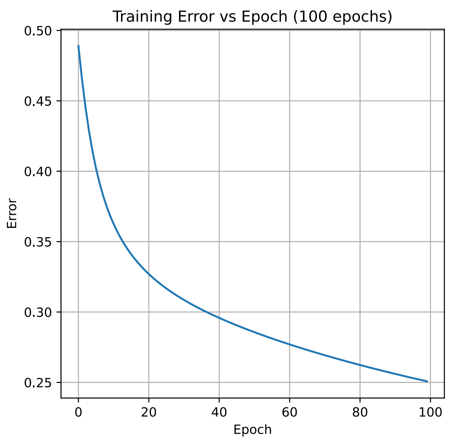
1

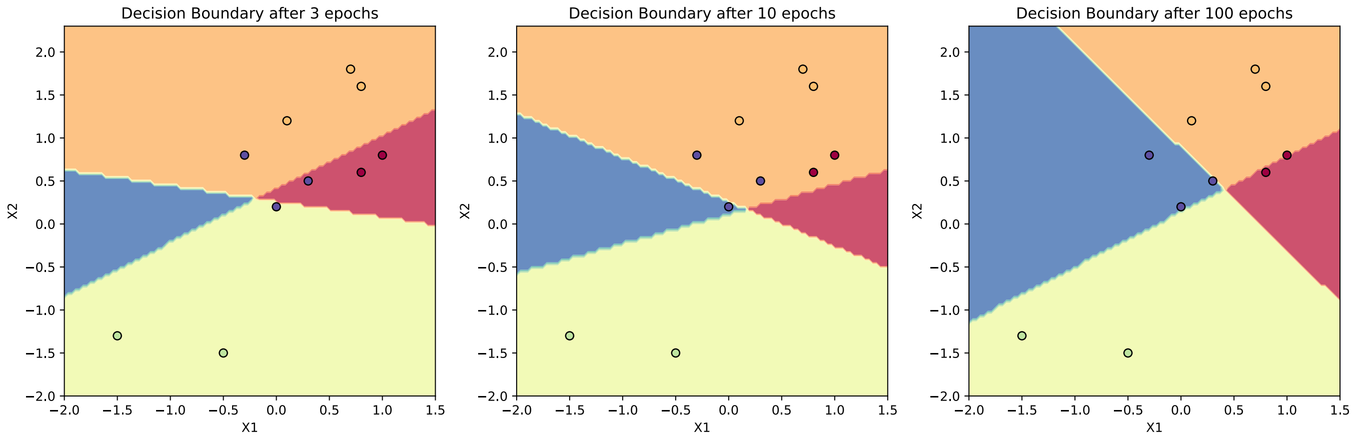
The code for this problem can be found on [nn_classification.ipynb]

The results are shown in the figures below:







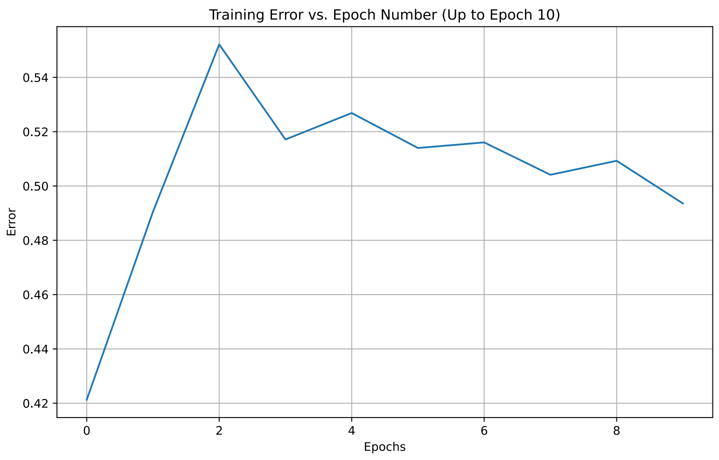


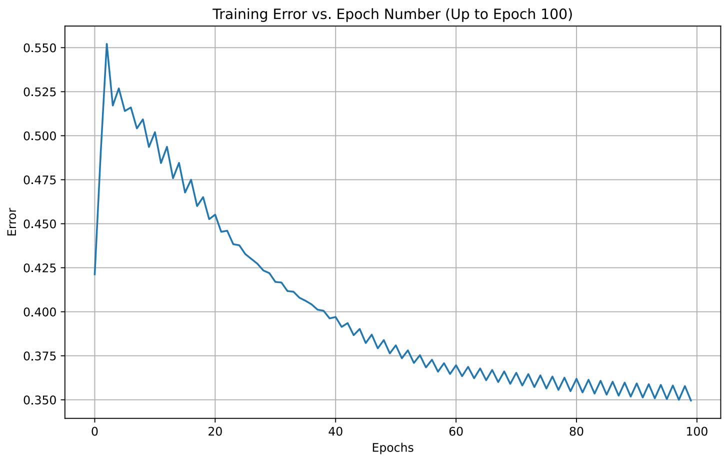
2

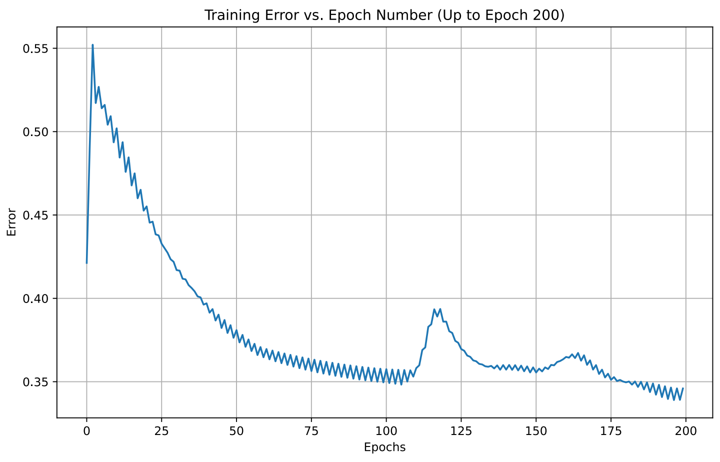
The code for this problem can be found on [approximate.ipynb]

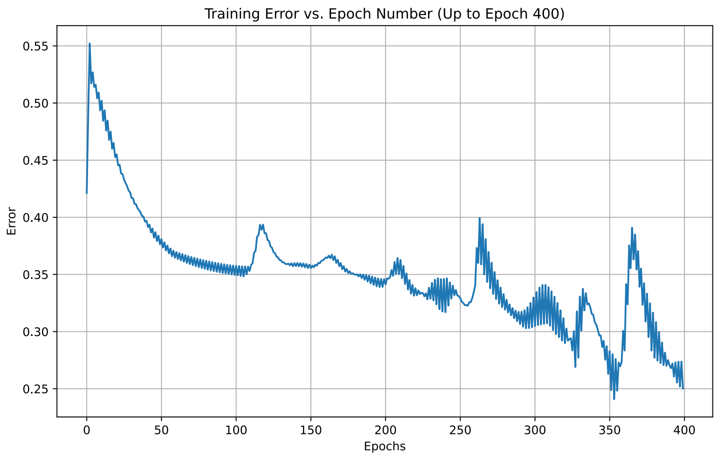
The results are shown in the figures below:

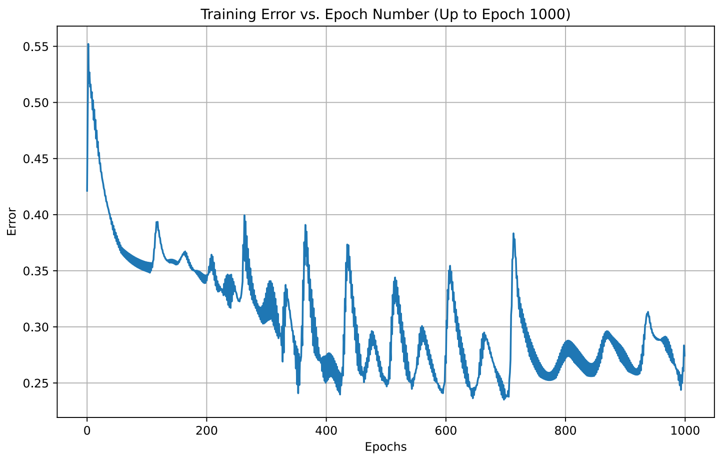
- **Training Error vs. Epoch**



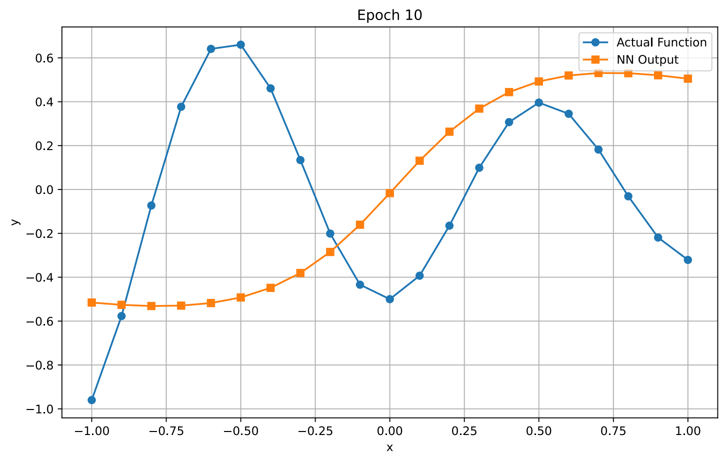



Training Error vs. Epoch Number (Up to Epoch 200)
Error
Epochs
0.55
0.50
0.45
0.40
0.35
0
25
50
75
100
125
150
175
200

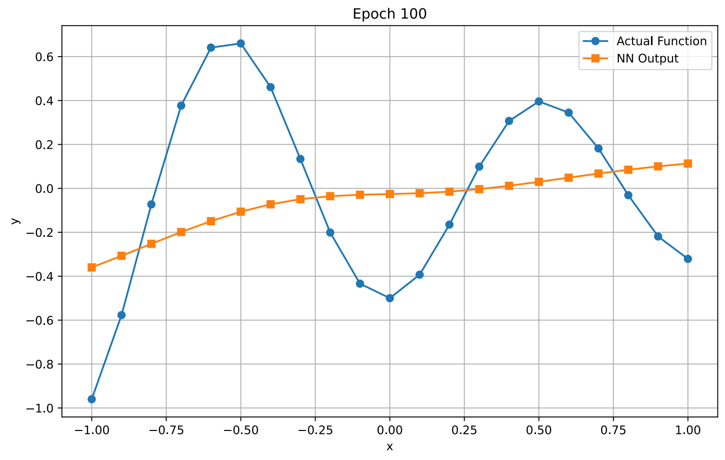

Training Error vs. Epoch Number (Up to Epoch 400)
Error
Epochs
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0
50
100
150
200
250
300
350
400


Training Error vs. Epoch Number (Up to Epoch 1000)
Error
Epochs
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0
200
400
600
800
1000

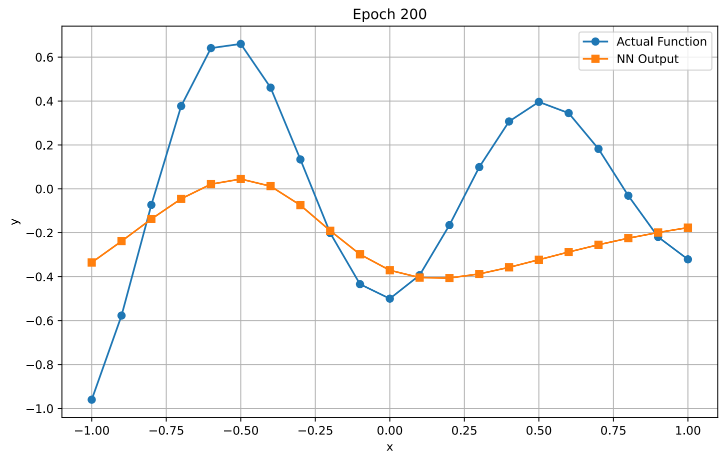
- **NN Output**



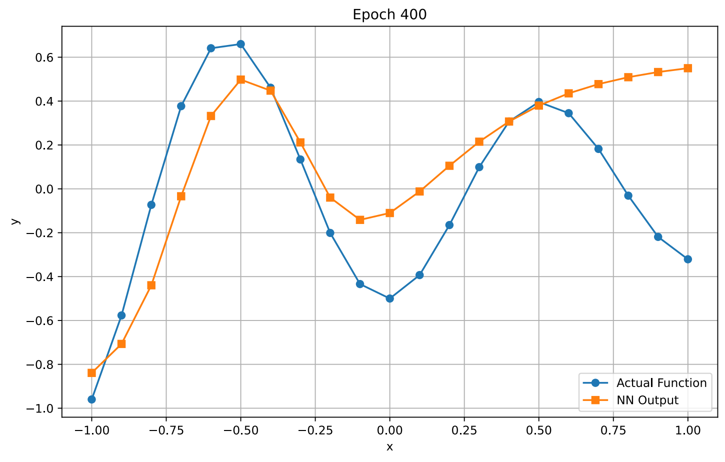
Epoch 10
Actual Function
NN Output
x
y



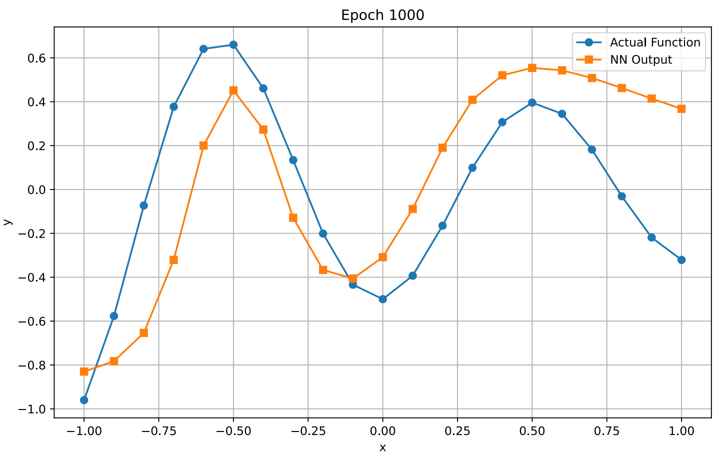
Epoch 100
Actual Function
NN Output
x
y



Epoch 200
Actual Function
NN Output
x
y



Epoch 400
Actual Function
NN Output
x
y

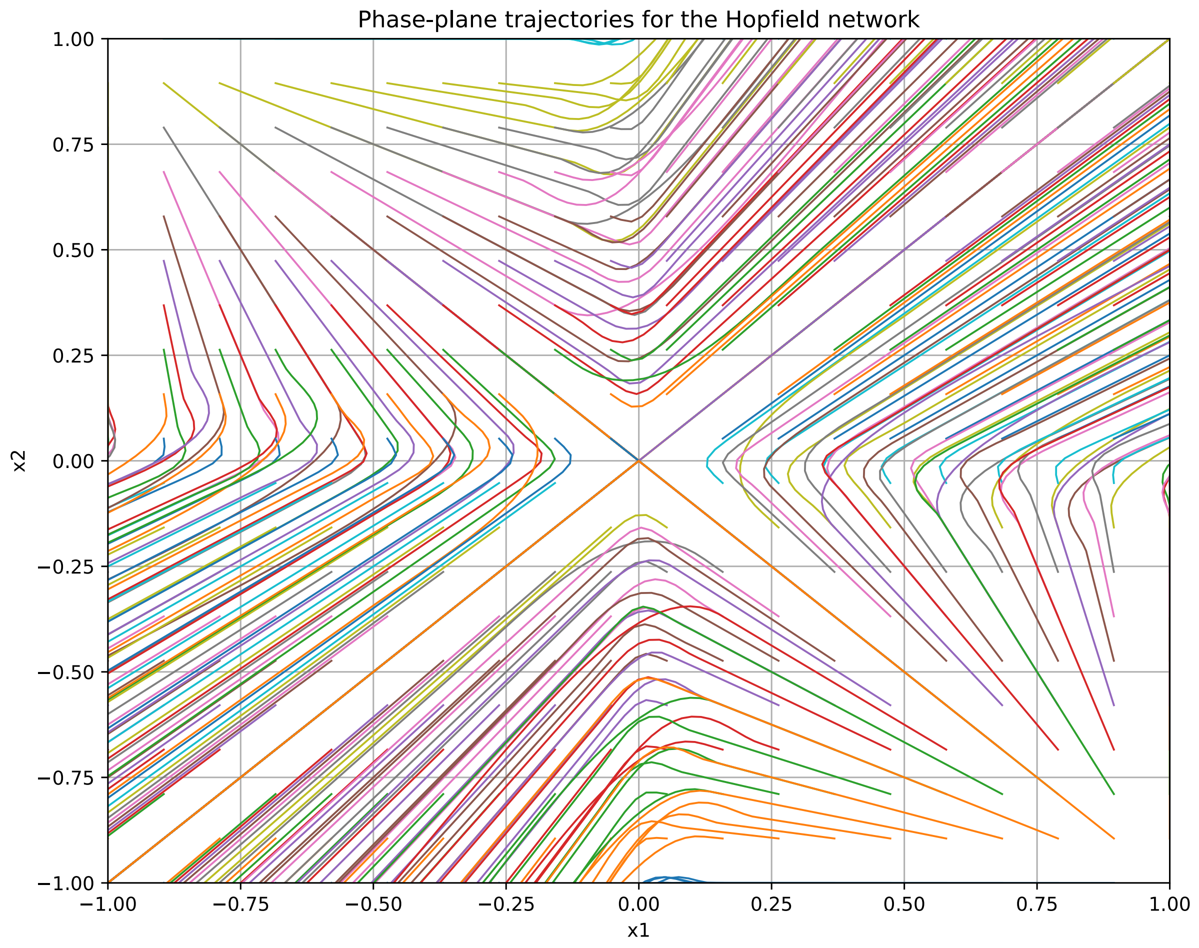


Epoch 1000
Actual Function
NN Output
y
x
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

3

The code for this problem can be found on [hopfield.ipynb]

The results are shown in the figures below:



4

The code for this problem can be found on [transformer.ipynb]

Description:

1. Define Basic Building Blocks

1.1 Multi-Head Attention:

- **Purpose**: To compute attention scores between each pair of positions in a sequence. This allows the model to focus on different aspects of the input sequence.
- **Components**:
  - Multiple attention heads that process the input in parallel and capture different relationships.
  - Linear transformation layers that transform the input.
- **How it works**:
  1. Initialize the module with input parameters and define the linear transformation layers.
  2. Calculate attention scores.
  3. Reshape the input tensor to create multiple heads.

4. Combine the attention outputs from all heads.

1.2 Position-wise Feed-Forward Networks:

- **Purpose**: To transform the output of the attention layers.
- **Components**:
  - Two linear transformation layers.
  - A ReLU activation function.
- **How it works**:
  1. Initialize the class with the transformation layers and the activation function.
  2. During the forward pass, apply the transformations and activation function sequentially.

1.3 Positional Encoding:

- **Purpose**: To provide the model with information about the position of tokens in the sequence since the Transformer does not have any inherent notion of order.
- **Components**:
  - Sine and cosine functions used to generate position-specific values.
- **How it works**:
  1. Initialize the class and create a tensor to store positional encoding values.
  2. Calculate sine and cosine values for different positions.
  3. During the forward pass, add the positional encoding values to the input tensor.

2. Building Encoder and Decoder Layers

2.1 Encoder Layer:

- **Components**:
  - Multi-Head Attention layer.
  - Position-wise Feed-Forward layer.
  - Two Layer Normalization layers.
- **How it works**:
  1. Initialize the class with its components.
  2. During the forward pass, apply self-attention, then add the attention output to the input tensor and normalize it.
  3. Compute the position-wise feed-forward output, combine it with the normalized self-attention output, and normalize again.

2.2 Decoder Layer:

- **Components**:
  - Two Multi-Head Attention layers (for masked self-attention and cross-attention).
  - Position-wise Feed-Forward layer.
  - Three Layer Normalization layers.
- **How it works**:
  1. Initialize the class with its components.
  2. During the forward pass:
     - Calculate the masked self-attention output, add it to the input, apply dropout, and normalize.
     - Compute cross-attention between decoder and encoder outputs, normalize, and combine with masked self-attention.
     - Calculate position-wise feed-forward output, combine with previous outputs, apply dropout, and normalize.

3. Build the Complete Transformer Model

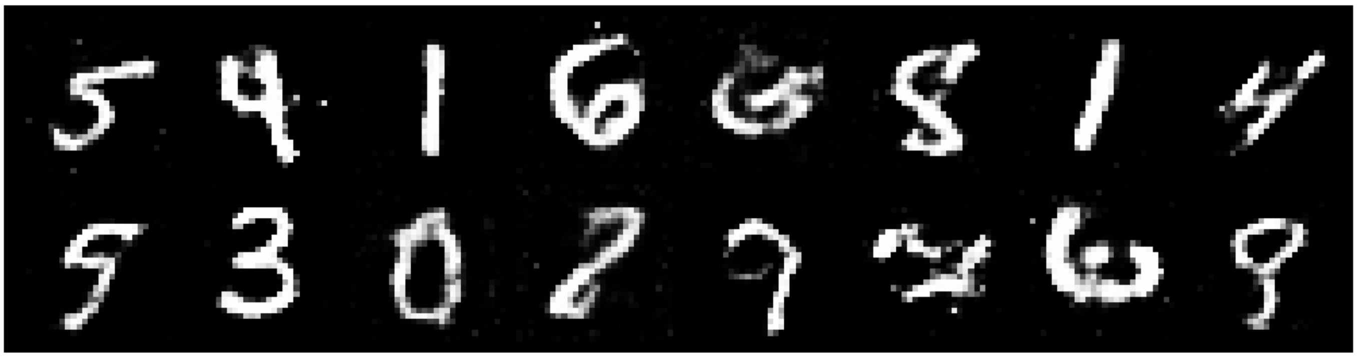
3.1 Transformer Model:

- **Components**:
  - Embedding layers for source and target sequences.
  - Positional Encoding module.
  - Stacked Encoder and Decoder layers.
  - Linear layer for projecting decoder output.
- **How it works**:
  1. Initialize the class and its components.
  2. Define the `generate_mask` method to create masks for source and target sequences.
  3. During the forward pass:
     - Generate masks for source and target sequences.
     - Compute embeddings and apply positional encoding and dropout.
     - Process the source sequence through the encoder layers.
     - Process the target sequence through the decoder layers, using encoder outputs and masks.
     - Apply the linear projection layer to the decoder output to obtain the final logits.

The Transformer model processes input sequences and produces output sequences by combining the functionalities of its components, ensuring attention is paid to relevant parts of the input and capturing complex relationships between input and output.

5

a) The code for the vanilla GAN can be found on vanilla_GAN.ipynb



Vanilla GAN output.

Description:

1. Data Loading and Preprocessing:

- **Purpose**: Load the MNIST dataset, preprocess it, and create a DataLoader for batching.
- **Components**:
  - `mnist_data()`: function to load and preprocess the MNIST dataset.
  - `data_loader`: DataLoader to load batches of data.
- **How it works**:
  1. Use the `mnist_data` function to load the dataset with normalization.
  2. Create a DataLoader with a batch size of 100 and shuffling enabled.

2. Discriminator Network:

- **Purpose**: Distinguish between real and fake (generated) images.
- **Components**:
    - Three hidden layers with dropout and LeakyReLU activation.
    - Output layer with sigmoid activation.
- **How it works**:
    1. Initialize the layers in the `__init__` method.
    2. During the forward pass, process the input through each hidden layer sequentially and then through the output layer.

3. Generator Network:

- **Purpose**: Generate fake images from random noise.
- **Components**:
    - Three hidden layers with LeakyReLU activation.
    - Output layer with Tanh activation.
- **How it works**:
    1. Initialize the layers in the `__init__` method.
    2. During the forward pass, process the noise through each hidden layer sequentially and then through the output layer.

4. Optimization:

- **Purpose**: Define optimizers and loss functions for training.
- **Components**:
    - Adam optimizer for both discriminator and generator.
    - Binary Cross Entropy (BCE) loss.
- **How it works**:
    1. Initialize the optimizers with the respective model parameters and a learning rate of 0.0002.
    2. Use BCE loss to compare the model's output against real (1) or fake (0) labels.

5. Training:

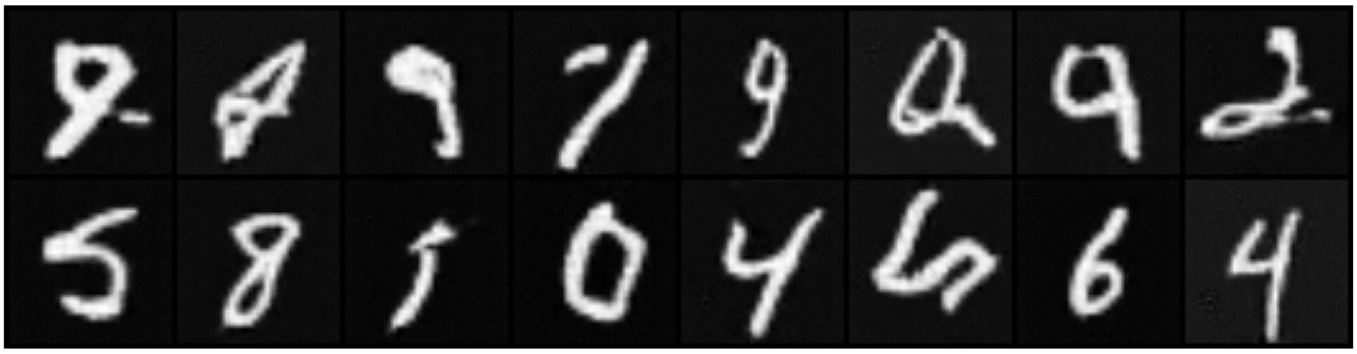
- **Purpose**: Train both the discriminator and generator.
- **Components**:
    - `train_discriminator`: Function to train the discriminator with real and fake data.
    - `train_generator`: Function to train the generator to produce fake data that the discriminator thinks are real.
- **How it works**:
    1. For each epoch and batch, get real data and generate fake data.
    2. Train the discriminator with both real and fake data.
    3. Train the generator with the generated fake data.
    4. Log errors, display progress, and save models at specified intervals.

6. Visualization and Logging:

- **Purpose**: Monitor the training process and visualize generated images.
- **Components**:
    - `Logger` class from the `utils` module (not provided in the code).
- **How it works**:

1. Initialize the logger.
2. During training, after specific intervals, clear the display.
3. Display generated images and log the training status.
4. Save model checkpoints.

b) The code for the DC GAN can be found on [DC_GAN.ipynb]



DC-GAN output.

Description:

1. Data Loading and Preprocessing:

- **Purpose**: Load the MNIST dataset, preprocess it by resizing and normalization, and create a DataLoader for batching.
- **Components**:
  - `mnist_data()`: function to load and preprocess the MNIST dataset.
  - `data_loader`: DataLoader to load batches of data.
- **How it works**:
  1. Use the `mnist_data` function to load the dataset with resizing and normalization.
  2. Create a DataLoader with a batch size of 100 and shuffling enabled.

2. Discriminator Network:

- **Purpose**: Distinguish between real and fake (generated) images.
- **Components**:
  - Four convolutional layers with batch normalization and LeakyReLU activation.
  - Fully connected output layer with sigmoid activation.
- **How it works**:
  1. Initialize the convolutional and output layers in the `__init__` method.
  2. During the forward pass, process the input through each convolutional layer sequentially, flatten the result, and then process through the output layer.

3. Generator Network:

- **Purpose**: Generate fake images from random noise.
- **Components**:
  - One fully connected layer to reshape the noise.
  - Four transposed convolutional layers with batch normalization and ReLU activation (except the last one).
  - Tanh activation at the output.
- **How it works**:
  1. Initialize the layers in the `__init__` method.

2. During the forward pass, project and reshape the noise, process it through each transposed convolutional layer sequentially, and finally apply Tanh activation.

4. Optimization:

- **Purpose**: Define optimizers and loss functions for training.
- **Components**:
    - Adam optimizer for both discriminator and generator.
    - Binary Cross Entropy (BCE) loss.
- **How it works**:
    1. Initialize the optimizers with the respective model parameters, learning rate of 0.0002, and specific betas.
    2. Use BCE loss to compare the model's output against real (1) or fake (0) labels.

5. Training:

- **Purpose**: Train both the discriminator and generator.
- **Components**:
    - `train_discriminator`: Function to train the discriminator with real and fake data.
    - `train_generator`: Function to train the generator to produce fake data that the discriminator thinks are real.
- **How it works**:
    1. For each epoch and batch, get real data and generate fake data.
    2. Train the discriminator with both real and fake data.
    3. Train the generator with the generated fake data.
    4. Log errors, display progress, and save models at specified intervals.

6. Visualization and Logging:

- **Purpose**: Monitor the training process and visualize generated images.
- **Components**:
    - `Logger` class from the `utils` module (not provided in the code).
- **How it works**:
    1. Initialize the logger.
    2. During training, after specific intervals, clear the display.
    3. Display generated images and log the training status.
    4. Save model checkpoints.

Part C

1

1,

we test if the function satisfies the superposition principle

$f(ax_1 + bx_2) = af(x_1) + bf(x_2)$

a) $f(x) = |x|$

Test for $x_1 = 1$ $x_2 = -1$ $a = 2$ $b = 3$

$f(2\times1 + 3\times(-1)) = |2-3| = 1$

$2f(1) + 3f(-1) = 2 + 3 = 5$ $\qquad 1 \neq 5 \rightarrow$ not linear

b) $f(x) = x^2 + 2x + 2$

Test for $x_1 = 1$, $x_2 = -1$, $a = 2$, $b = 3$

$f(2\times1 + 3\times(-1)) = f(-1) = 1 - 2 + 2 = 1$

$2f(1) + 3f(-1) = 2(1+2+2) + 3(1-2+2) = 13$ $\qquad 1 \neq 13 \rightarrow$ not linear

c) $f(x) = \frac{1}{x}$

Test for $x_1 = 2$, $x_2 = 3$, $a = 1$, $b = 1$

$f(2+3) = \frac{1}{5}$

$f(2) + f(3) = \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$ $\qquad \frac{1}{5} \neq \frac{5}{6} \rightarrow$ not linear

2

a) $f(x) = x^2 + 1$

$f'(x) = 2x$

$f''(x) = 2$ $f''(x)$ is always positive so $f(x)$ is convex

b) $f(x) = ax + b$

$f'(x) = a$

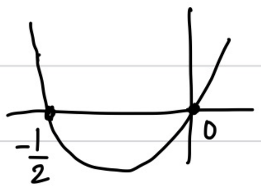
$f''(x) = 0$ $f(x)$ is both convex and concave

c) $f(x) = x^4 + x^3$

$f'(x) = 4x^3 + 3x^2$

$f''(x) = 12x^2 + 6x$

$12x^2 + 6x = 0 \begin{cases} x = 0 \\ x = -\frac{1}{2} \end{cases}$



for $x < -\frac{1}{2}$, $f''(x)$ is positive $\rightarrow$ $f(x)$ is convex

for $-\frac{1}{2} < x < 0$, $f''(x)$ is negative $\rightarrow$ $f(x)$ is concave

for $x > 0$, $f''(x) > 0$ $\rightarrow$ $f(x)$ is convex

3

3

1 $f(x) = |x|$

The function is not differentiable at $x=0$ so $x=0$ is a critical point.

2 $f(x) = x^2+2x+2$

$f'(x) = 2x+2$ The critical point is $x=-1$

3 $f(x) = \frac{1}{x}$

$f'(x) = \frac{-1}{x^2}$ undefined at $x=0$ but 0 is not in the domain so there is no critical point.

4) $f(x) = x^2+1$

$f'(x) = 2x$ $x=0$ is the critical point

5) $f(x) = ax+b$

$f'(x) = a$ no critical points

6) $f(x) = x^4+x^3$

$f'(x) = 4x^3+3x^2$ $x=0$ and $x=-\frac{3}{4}$ are critical points.

4

1. Let x be the input vector.

2. Let W_1 be the weight matrix connecting the input layer to the hidden layer, and b_1 be the bias vector for the hidden layer.
3. Let z_1 be the weighted input to the hidden layer, and h be the output of the hidden layer.
4. Let W_2 be the weight matrix connecting the hidden layer to the output layer, and b_2 be the bias vector for the output layer.
5. Let z_2 be the weighted input to the output layer, and y be the output of the network.
6. Let t be the target output.
7. The loss function L is defined as the mean squared error between the predicted output y and the target output t:
   $L = \frac{1}{2}(y - t)^2$

The forward propagation:

1. $z_1 = W_1 x + b_1$
2. $h = \tanh(z_1)$
3. $z_2 = W_2 h + b_2$
4. $y = \tanh(z_2)$

The backpropagation algorthm:

1. **Compute the Gradient of the Loss with Respect to the Output:**
   $\frac{\partial L}{\partial y} = y - t$
2. **Gradient of the Loss with Respect to z_2:**
   The derivative of the tanh function is:
   $\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z)$
   So,
   $\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial y} \times \frac{\partial y}{\partial z_2} = (y - t) \times (1 - y^2)$
3. **Gradient of the Loss with Respect to W_2 and b_2:**
   $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \times h$
   $\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2}$
4. **Gradient of the Loss with Respect to h:**
   $\frac{\partial L}{\partial h} = \frac{\partial L}{\partial z_2} \times W_2$
5. **Gradient of the Loss with Respect to z_1:**
   $\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h} \times \frac{\partial h}{\partial z_1} = \frac{\partial L}{\partial h} \times (1 - h^2)$
6. **Gradient of the Loss with Respect to W_1 and b_1:**
   $\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} \times x$
   $\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_1}$

The weight update rules:

1. $W_1 = W_1 - \eta \times \frac{\partial L}{\partial W_1}$
2. $b_1 = b_1 - \eta \times \frac{\partial L}{\partial b_1}$
3. $W_2 = W_2 - \eta \times \frac{\partial L}{\partial W_2}$
4. $b_2 = b_2 - \eta \times \frac{\partial L}{\partial b_2}$

Where $$\eta$ is the learning rate.