ML algorithms retrospective: Part 2

1 Logistic Regression

Logistic regression is a supervised machine learning algorithm used for classification tasks where the goal is to predict the probability that an instance belongs to a given class or not. Logistic regression is a statistical algorithm which analyze the relationship between two data factors.

Suppose we have a dataset consists of students' study hours and their exam outcomes (pass or fail):

Study Hours	Pass (1) / Fail (0)
1	0
2	0
3	0
4	1
5	1
6	1
7	1

Table 1: Study Hours vs. Exam Outcomes

1.1 Logistic Regression Model

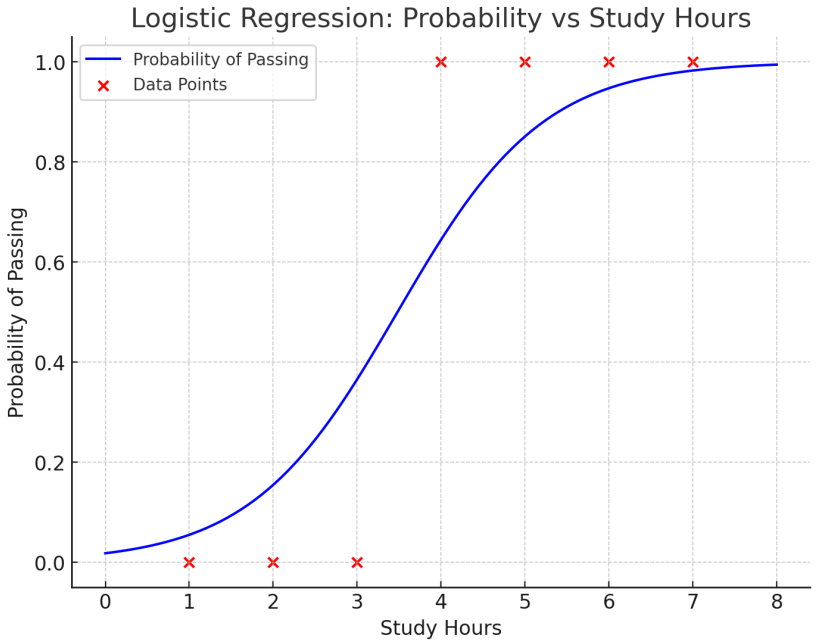
Logistic regression predicts the probability of a student passing ($Y = 1$) given their study hours (X) using the following equation:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(b_0 + b_1 X)}}$$

where:

- $P(Y = 1|X)$: Probability of passing.
- b_0, b_1: Model parameters (intercept and coefficient).
- X: Study hours.

The model determines the parameters b_0 and b_1 by maximizing the likelihood of observing the given data.



The blue curve represents the predicted probability of passing as a function of study hours. The red dots represent the original data points, showing whether the student passed (1) or failed (0). This sigmoid curve demonstrates how logistic regression transitions smoothly between predictions for fail (probability near 0) and pass (probability near 1).

1.2 Making Predictions

The model predicts probabilities for each student based on their study hours. For instance:

- If $P(Y = 1|X) > 0.5$, the student is predicted to pass.
- If $P(Y = 1|X) \leq 0.5$, the student is predicted to fail.

1.3 Implementation

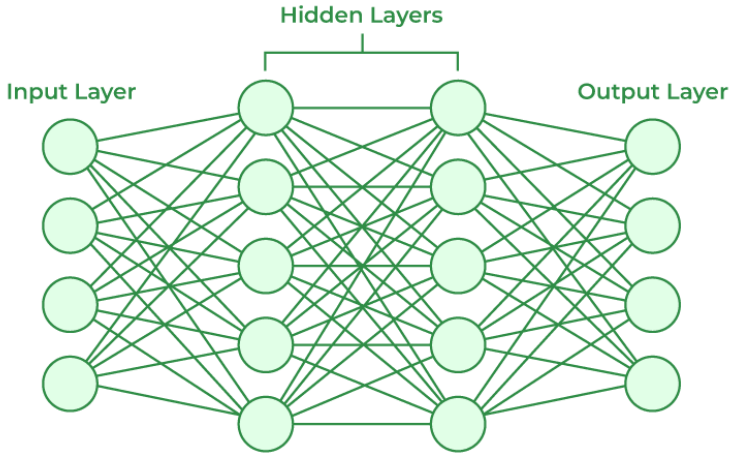
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

# Data
X = np.array([[1], [2], [3], [4], [5], [6], [7]])  # Study
    hours
y = np.array([0, 0, 0, 1, 1, 1, 1])  # Pass (1) / Fail (0)
```

```
8
9  # Create the logistic regression model
10 model = LogisticRegression()
11 model.fit(X, y)
12
13 # Predict probabilities
14 study_hours = np.array([[1], [3], [5], [7]])
15 predicted_probabilities = model.predict_proba(study_hours)
       [:, 1]
16 predicted_classes = model.predict(study_hours)
17
18 # Print results
19 for hours, prob, pred in zip(study_hours.flatten(),
       predicted_probabilities, predicted_classes):
20     print(f"Study Hours: {hours}, Predicted Probability of
           Passing: {prob:.2f}, Predicted Class: {pred}")
21
22 # Plotting the sigmoid curve
23 x_range = np.linspace(0, 8, 100).reshape(-1, 1)
24 probs = model.predict_proba(x_range)[:, 1]
25 plt.figure(figsize=(8, 6))
26 plt.plot(x_range, probs, label="Probability of Passing")
27 plt.scatter(X, y, color="red", label="Data points")
28 plt.xlabel("Study Hours")
29 plt.ylabel("Probability of Passing")
30 plt.title("Logistic Regression: Probability of Passing vs.
       Study Hours")
31 plt.legend()
32 plt.show()
```

2 ANN

Artificial Neural Networks contain artificial neurons which are called units . These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset. Commonly, Artificial Neural Network has an input layer, an output layer as well as hidden layers. The input layer receives data from the outside world which the neural network needs to analyze or learn about. Then this data passes through one or multiple hidden layers that transform the input into data that is valuable for the output layer. Finally, the output layer provides an output in the form of a response of the Artificial Neural Networks to input data provided.



Forward Propagation

Forward propagation calculates the output of the neural network by propagating the input through its layers.

1. **Input to Hidden Layer**:

$$z_1^{(1)} = w_{11}x_1 + w_{12}x_2 + b_1, \quad z_2^{(1)} = w_{21}x_1 + w_{22}x_2 + b_2$$

where $z_1^{(1)}$ and $z_2^{(1)}$ are the weighted sums for the hidden layer, and b_1, b_2 are the biases.

2. **Activation in Hidden Layer**:

$$h_1 = \sigma(z_1^{(1)}), \quad h_2 = \sigma(z_2^{(1)})$$

where $\sigma(x)$ is the activation function (e.g., ReLU or sigmoid).

3. **Hidden to Output Layer**:

$$z^{(2)} = w_{31}h_1 + w_{32}h_2 + b_3$$

4. **Output Activation**:

$$y_{\text{pred}} = \sigma(z^{(2)})$$

Backward Propagation

Backward propagation computes the gradients of the loss function with respect to the weights and biases for optimization. 1. **Loss Function (Mean Squared Error)**:

$$L = \frac{1}{2}(y_{\text{pred}} - y_{\text{true}})^2$$

2. **Gradient at Output Layer**:

$$\delta^{(2)} = (y_{\text{pred}} - y_{\text{true}}) \cdot \sigma'(z^{(2)})$$

3. **Gradients for Weights (Hidden to Output)**:

$$\frac{\partial L}{\partial w_{31}} = \delta^{(2)} \cdot h_1, \quad \frac{\partial L}{\partial w_{32}} = \delta^{(2)} \cdot h_2$$

4. **Gradient for Bias at Output**:

$$\frac{\partial L}{\partial b_3} = \delta^{(2)}$$

5. **Gradient at Hidden Layer**:

$$\delta_j^{(1)} = \delta^{(2)} \cdot w_{3j} \cdot \sigma'(z_j^{(1)}) \quad \text{for each hidden neuron } j.$$

6. **Gradients for Weights (Input to Hidden)**:

$$\frac{\partial L}{\partial w_{ij}} = \delta_j^{(1)} \cdot x_i$$

7. **Gradient for Bias at Hidden**:

$$\frac{\partial L}{\partial b_j} = \delta_j^{(1)}$$

A homework from CMU:

Consider the neural network architecture shown above for a 2-class (0, 1) classification problem. The values for weights and biases are shown in the figure. We define:

$a_1 = w_{11}x_1 + b_{11}$

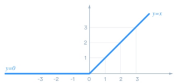
$a_2 = w_{12}x_1 + b_{12}$

$a_3 = w_{21}z_1 + w_{22}z_2 + b_{21}$

$z_1 = \text{relu}(a_1)$

$z_2 = \text{relu}(a_2)$

$z_3 = \sigma(a_3),\ \sigma(x) = \frac{1}{1+e^{-x}}$



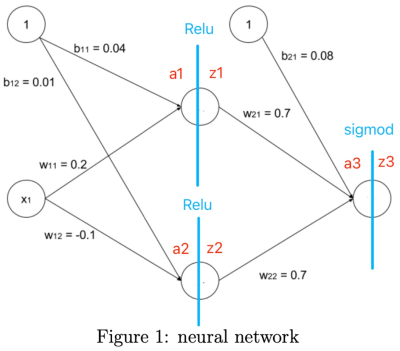


Figure 1: neural network

Assuming this is a well trained model

Input:

$x_1 = 0.3$

What is $\hat{y}(x_1 = 0.3)$

$$\text{classification} = \begin{cases} 1 & \text{if } \hat{y} \geq 0.5 \\ 0 & \text{if } \hat{y} < 0.5 \end{cases}$$

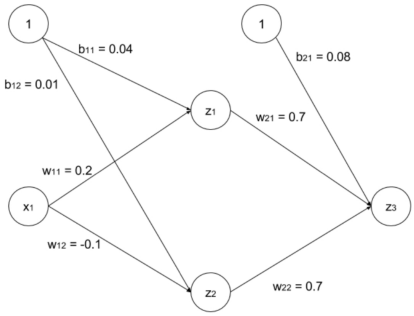


Figure 1: neural network

Forward:

$$a_1 = w_{11}x_1 + b_{11} = 0.2 \times 0.3 + 0.04 = 0.06 + 0.04 = 0.1$$

$$a_2 = w_{12}x_1 + b_{12} = -0.1 \times 0.3 + 0.01 = -0.03 + 0.01 = -0.02$$

$$z_1 = \text{ReLU}(a_1) = \max(0, a_1) = \max(0, 0.1) = 0.1$$

$$z_2 = \text{ReLU}(a_2) = \max(0, a_2) = \max(0, -0.02) = 0$$

$$a_3 = w_{21}z_1 + w_{22}z_2 + b_{21} = 0.7 \times 0.1 + 0.7 \times 0 + 0.08 = 0.07 + 0 + 0.08 = 0.15$$

$$z_3 = \sigma(a_3) = \frac{1}{1+e^{-a_3}} = \frac{1}{1+e^{-0.15}}$$

$$z_3 = \frac{1}{1+e^{-0.15}} \approx \frac{1}{1+0.8607} \approx \frac{1}{1.8607} \approx 0.5374$$

$$\hat{y}(x_1 = 0.3) \approx 0.5374$$

$$\text{classification} = 1 \text{ (since } 0.5374 \geq 0.5)$$

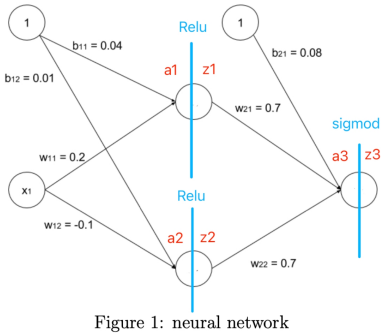


Figure 1: neural network

Backward:

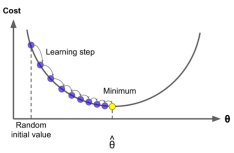
Assuming this is not a well trained model
More data comes in, learning rate = 0.01
Use back propagation to update bias b21 and b12
Define loss function as $\text{Loss} = y - \hat{y} = y - z_3$

Input: $x_1 = 0.3$

Output: Updated parameters w21, b21

$$b_{21}^{\text{new}} = b_{21} - \eta \frac{\partial \text{Loss}}{\partial b_{21}}$$

$$w_{21}^{\text{new}} = w_{21} - \eta \frac{\partial \text{Loss}}{\partial w_{21}}$$



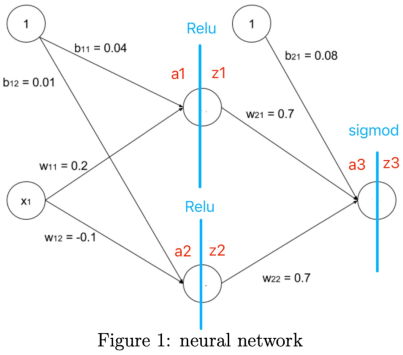


Figure 1: neural network

$$b_{21}^{\text{new}} = b_{21} - \eta \frac{\partial \text{Loss}}{\partial b_{21}} \qquad \eta = 0.01$$

$$\frac{\partial L}{\partial b_{21}} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a_3}\frac{\partial a_3}{\partial b_{21}} \quad \text{chain rule}$$

$$\text{Loss} = y - \hat{y} = y - z_3$$

$$z_3 = \sigma(a_3),\ \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \text{Loss}}{\partial z_3} = -1 \qquad \frac{\partial z_3}{\partial a_3} = z_3(1 - z_3) \qquad \frac{\partial a_3}{\partial b_{21}} = 1$$

$$\frac{\partial \text{Loss}}{\partial b_{21}} = -z_3(1 - z_3)$$

$$\frac{\partial \text{Loss}}{\partial b_{21}} = -0.5374 \times 0.4626 \approx -0.2485$$

$$b_{21}^{\text{new}} = b_{21} - 0.01 \times (-0.2485) = b_{21} + 0.01 \times 0.2485 = b_{21} + 0.002485$$

$$b_{21}^{\text{new}} = 0.08 + 0.002485 = 0.082485$$

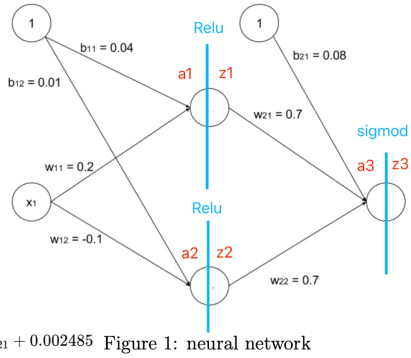


Figure 1: neural network

$$w_{21}^{\text{new}} = w_{21} - \eta \frac{\partial \text{Loss}}{\partial w_{21}}$$

$$\frac{\partial \text{Loss}}{\partial w_{21}} = \frac{\partial \text{Loss}}{\partial a_3} \cdot \frac{\partial a_3}{\partial w_{21}} = -0.2485 \times 0.1 = -0.02485$$

$$w_{21}^{\text{new}} = w_{21} - 0.01 \times (-0.02485) = w_{21} + 0.01 \times 0.02485 = w_{21} + 0.0002485$$

$$w_{21}^{\text{new}} = 0.7 + 0.0002485 \approx 0.70025$$

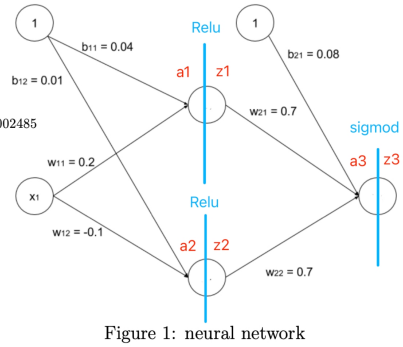
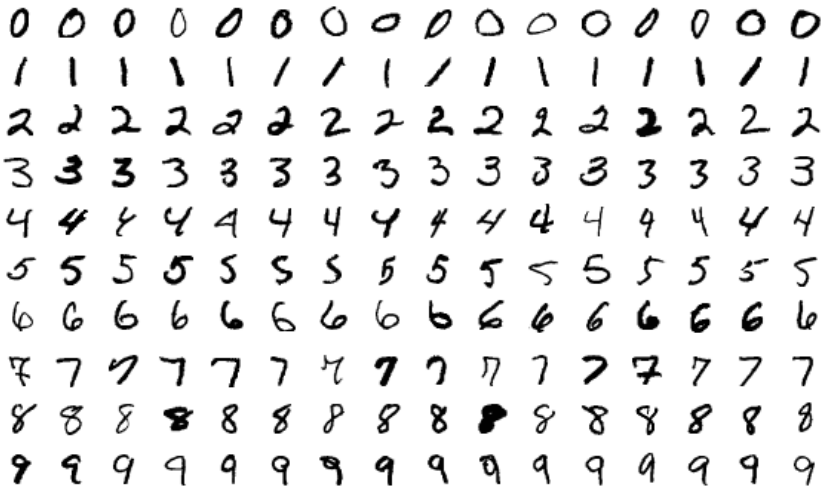


Figure 1: neural network

https://www.kaggle.com/c/digit-recognizer.

This example demonstrates how an Artificial Neural Network (ANN) can be used to classify handwritten digits from the popular MNIST dataset. Each image is a 28x28 grayscale image, flattened into a vector of 784 features. The goal is to classify each image into one of 10 classes (digits 0-9).



The ANN model used in this example consists of:

- **Input Layer**: 784 neurons (one for each pixel in the image).
- **Hidden Layers**: Two hidden layers with 128 and 64 neurons, respectively, using the ReLU activation function.
- **Output Layer**: 10 neurons (one for each digit), using the softmax activation function to output probabilities for each class.

2.1 Key Points in the Example

- **Data Preprocessing**:

- Normalize pixel values (scale between 0 and 1).
- Convert class labels to one-hot encoding for multi-class classification.

- **Model Design**:
  - Use a `Flatten` layer to convert 2D image input into a 1D vector.
  - Fully connected `Dense` layers with ReLU activation add non-linearity.
  - The output layer uses softmax activation to output probabilities for the 10 classes.

- **Training and Evaluation**:
  - The model is trained using the Adam optimizer and categorical crossentropy loss.
  - Test accuracy is used to measure model performance.

2.2 Implementation

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize the data
X_train = X_train / 255.0
X_test = X_test / 255.0

# Convert labels to one-hot encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Define the ANN model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten the 28x28
        image into a 1D array of 784 features
    Dense(128, activation='relu'),  # First hidden layer
    Dense(64, activation='relu'),   # Second hidden layer
    Dense(10, activation='softmax') # Output layer with
        softmax for multi-class classification
])

# Compile the model
model.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=['accuracy'])
```

```
28
29 # Train the model
30 model.fit(X_train, y_train, epochs=10, batch_size=32,
       validation_split=0.2)
31
32 # Evaluate the model on the test set
33 test_loss, test_accuracy = model.evaluate(X_test, y_test)
34 print(f"Test Accuracy: {test_accuracy:.2f}")
35
36 # Make predictions
37 predictions = model.predict(X_test[:5])
38 print("Predictions:", tf.argmax(predictions, axis=1).numpy()
       )
```

3 CNN

Example: The goal is to classify 28x28 grayscale images of handwritten digits into one of 10 classes (0 to 9).

3.1 Architecture

1. **Input Layer:** Accepts a 28x28 grayscale image.
2. **Convolutional Layers:** A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

   The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images.

   - First layer applies 32 filters of size 3x3 with ReLU activation.
   - Second layer applies 64 filters of size 3x3 with ReLU activation.
3. **Pooling Layers:** Each convolutional layer is followed by a 2x2 MaxPooling layer to reduce spatial dimensions.
4. **Flatten Layer:** Converts the 2D feature maps into a 1D vector.
5. **Dense Layers:**
   - A fully connected layer with 64 neurons and ReLU activation.
   - Output layer with 10 neurons (one for each class) and softmax activation.

3.2 Implementation

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load MNIST data
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Preprocess the data
X_train = X_train.reshape((X_train.shape[0], 28, 28, 1)).
    astype('float32') / 255
X_test = X_test.reshape((X_test.shape[0], 28, 28, 1)).astype
    ('float32') / 255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# Build the CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape
        =(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=5, batch_size=64,
    validation_split=0.2)

# Evaluate the model
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_acc:.2f}")
```

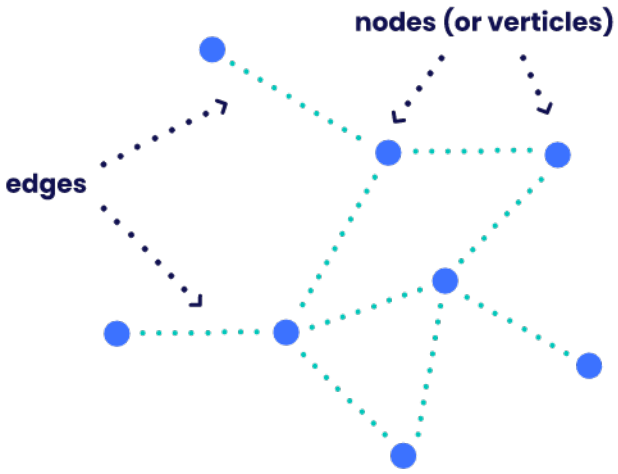
- **Data Preprocessing:** Images are normalized to the range [0, 1], reshaped to include the channel dimension (28x28x1), and labels are one-hot encoded.
- **Model Architecture:** The CNN consists of convolutional, pooling, flatten, and dense layers to extract features and classify the images.
- **Training:** The model is trained for 5 epochs using the Adam optimizer.

- **Evaluation:** The accuracy on the test set is printed to measure the model's performance.

4 GNN

Short for graph neural network, a GNN is a system of machine learning software that analyzes data that is presented to it in the form of a graph. GNNs use deep learning to reach conclusions based on two chief parts of the input graphs: their nodes and their edges.

The nodes, which are the vertices on the graph represent input data points, and the edges – the lines between nodes – represent the connection between data points.



By analyzing nodes and edges, graph neural networks can interpret data that is presented to it as a graph, such as the diagram above, and then carry out problem-solving and predictions using the interpretation.

Such decision-making can apply to many industry sectors and online systems. For example, by spotting patterns between edges and nodes that, for instance, represent connections between apparently disparate addresses, GNNs can not only generate predictions but also find anomalies in those points of data. This helps to spot suspicious online activity, or other vital outliers in datasets.

For GNN, we use the Cora citation dataset, where:

- **Nodes**: Represent papers.
- **Edges**: Represent citations between papers.
- **Features**: Word embeddings of the paper abstracts.

- **Task**: Predict the category (topic) of each paper based on the graph structure and node features.

The GNN model consists of:

1. **Input Layer**: Node features and adjacency matrix.
2. **GCN Layers**: Two Graph Convolutional Network (GCN) layers:
   - The first GCN layer maps input features to a 16-dimensional embedding space.
   - The second GCN layer outputs the probabilities for each class.
3. **Activation and Dropout**: ReLU activation and dropout for regularization.
4. **Output**: Log-softmax applied to the node representations for classification.

4.1 Implementation

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.datasets import Planetoid

# Load a sample dataset (Cora)
dataset = Planetoid(root='data/Cora', name='Cora')

# Define the GNN model
class GCN(torch.nn.Module):
    def __init__(self):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(dataset.num_node_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)

# Load the data and model
data = dataset[0]  # Single graph in Cora
model = GCN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01,
    weight_decay=5e-4)

```

```
# Training loop
model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.
        train_mask])
    loss.backward()
    optimizer.step()

# Evaluate the model
model.eval()
pred = model(data).argmax(dim=1)
correct = (pred[data.test_mask] == data.y[data.test_mask]).
    sum()
acc = int(correct) / int(data.test_mask.sum())
print(f'Test Accuracy: {acc:.4f}')
```

4.2 Explanation of the Code

- **Dataset**: The Planetoid dataset (Cora) contains node features, edge indices, and labels for each node.
- **Model Architecture**:
  - `GCNConv`: Graph Convolutional layers aggregate features from neighboring nodes.
  - Two layers: The first extracts 16-dimensional embeddings, and the second classifies nodes into categories.
- **Training**: The model is trained for 200 epochs using negative log-likelihood loss.
- **Evaluation**: The model's accuracy is tested on nodes reserved for evaluation.

The trained GNN predicts the topic of each paper with high accuracy, leveraging the graph structure and node features. This demonstrates the power of GNNs for graph-based machine learning tasks.