ML4G-PA2_GROUP6

January 22, 2024

1 Machine Learning for Graphs - Tutorial B: The Graph Convolutional Network

Fill in your names and group number here:

NAME STUDENT A : Alexandru Dochian (2776000)

NAME STUDENT B : Matteo De Rizzo (2749303)

GROUP NUMBER : 6

Implementing a machine learning experiment with graph data is an important skill that you will learn as part of this course. This hands-on tutorial will help you develop this skill, as well as help you familiarize yourself with many of the steps and techniques that you will likely need to use for your final project.

Representation learning is the task of learning sensible representations for your samples given some downstream task. On graphs, representation learning is commonly used to learn vector representations of the nodes. These node representations are often called *embeddings vectors* or just *embeddings*. Graph Neural Networks (GNN) are ideal for learning node embeddings, since the identity of a node is a function of its neighbourhood (up to depth d) and since GNNs learn internal node representations by applying an aggregation operator on exactly this neighbourhood. Different models with various choices of aggregation operator have been introduced over the past couple of years, with the *convolutional* and *attention* operators being the more popular choices.

For this tutorial, you are asked to implement the original *Graph Convolutional Network* (GCN) and to replicate some of the classification experiments from the paper that introduced it [1]. To help you on your way, we have already prepared this Python Notebook.

You are asked to team up with another student and to work together on this tutorial. Please register your team by creating a new group and by adding both members.

1.1 [1] Kipf, T. N., & Welling, M. Semi-supervised Classification With Graph Convolutional Networks (2017).

1.2 NumPy and PyTorch

In this course we will make use of the NumPy package for working with vector data, and the PyTorch machine learning package. Both of these are probably already installed in your environment as part of the first tutorial (Numpy as a dependency of PyTorch) but if this is not the case then running the following cell will install these packages for you.

Run the cell below to install the NumPy and PyTorch packages in your Python environment

[ ]:
```
%pip install numpy torch
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: numpy in /home/samanu/.local/lib/python3.10/site-
packages (1.25.0)
Requirement already satisfied: torch in /home/samanu/.local/lib/python3.10/site-
packages (2.1.2)
Requirement already satisfied: filelock in /usr/lib/python3/dist-packages (from
torch) (3.6.0)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (12.1.0.106)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (12.1.105)
Requirement already satisfied: networkx in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (2.6.3)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (11.4.5.107)
Requirement already satisfied: fsspec in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (2023.6.0)
Requirement already satisfied: sympy in /home/samanu/.local/lib/python3.10/site-
packages (from torch) (1.12)
Requirement already satisfied: jinja2 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (3.1.2)
Requirement already satisfied: triton==2.1.0 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (2.1.0)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (11.0.2.54)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-nccl-cu12==2.18.1 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (2.18.1)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (10.3.2.106)
Requirement already satisfied: nvidia-cudnn-cu12==8.9.2.26 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (8.9.2.26)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (12.1.3.1)
Requirement already satisfied: typing-extensions in
/home/samanu/.local/lib/python3.10/site-packages (from torch) (4.4.0)
Requirement already satisfied: nvidia-nvjitlink-cu12 in
/home/samanu/.local/lib/python3.10/site-packages (from nvidia-cusolver-
```

```
cu12==11.4.5.107->torch) (12.3.101)
Requirement already satisfied: MarkupSafe>=2.0 in
/home/samanu/.local/lib/python3.10/site-packages (from jinja2->torch) (2.1.1)
Requirement already satisfied: mpmath>=0.19 in
/home/samanu/.local/lib/python3.10/site-packages (from sympy->torch) (1.3.0)
Note: you may need to restart the kernel to use updated packages.
```

Run the cells below to import the necessay packages and to set a manual seed

[ ]:
```python
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

seed = 42  # for reproducability
torch.manual_seed(seed)
np.random.seed(seed)

# If you're using GPU (cuda), set the seed for it as well
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cuda
```

1.3 Data Preparation

In the previous tutorial we used the *RDFlib* package to import the dataset. That the dataset was encoded using an open standard made this possible. This is not always the case, however: it is very common to come accross graph datasets that use an arbitrary encoding. In the case of the *Cora* dataset loaded below, the graph has been stored in two parts: the first a set of integer-encoded edges $[i, j]$, with i and j the indices of the nodes, and the second as a set of n-hot encoded node representations. Being a citation graph, the edges convey who cites who, whereas the node vectors e_i represent a sparse bag-of-words with vocabulary Ω for which holds that $e_i[j] = 1$ if word $\Omega[j]$ occurs in the document and 0 otherwise.

To import the Cora dataset we first process the raw files using NumPy and cast the generated arrays to the correct datatypes. Next, we generate a node-to-integer map and reindex the edges to ensure that their node identifiers match those of the nodes.

Run the following cells to import and process the data

[ ]:
```python
path = './data/'

data = np.genfromtxt(path + "cora.content", dtype = str)
edges = np.genfromtxt(path + "cora.cites", dtype = int)
```

[ ]:
```
# these have the same order
features = data[:, 1:-1].astype(int)
labels = data[:, -1]
nodes = data[:, 0].astype(int)

n2i = {n:i for i,n in enumerate(nodes)}
edges_reindexed = np.array([[n2i[source], n2i[target]] for source, target in␣
 ↪edges])

num_nodes = len(nodes)
num_edges = len(edges)
num_features = len(features[0])
```

[ ]:
```
# inspect the data
print(f"Number of nodes: {num_nodes}")
print(f"Number of edges: {num_edges}")
print(f"Number of features: {num_features}\n")

for i in range(5):
    print(f"Node ID: {nodes[i]}")
    print(f"Node features: {features[i]}")
    print(f"Node label: {labels[i]}\n")

print(f"Edges: \n{edges_reindexed[:5]}")
```

```
Number of nodes: 2708
Number of edges: 5429
Number of features: 1433

Node ID: 31336
Node features: [0 0 0 … 0 0 0]
Node label: Neural_Networks

Node ID: 1061127
Node features: [0 0 0 … 0 0 0]
Node label: Rule_Learning

Node ID: 1106406
Node features: [0 0 0 … 0 0 0]
Node label: Reinforcement_Learning

Node ID: 13195
Node features: [0 0 0 … 0 0 0]
Node label: Reinforcement_Learning

Node ID: 37879
Node features: [0 0 0 … 0 0 0]
```

```
Node label: Probabilistic_Methods

Edges:
[[ 163  402]
 [ 163  659]
 [ 163 1696]
 [ 163 2295]
 [ 163 1274]]
```

1.4 Task 1: Vectorizing the graph

Since graph neural networks aggregate the information from the neighbourhoods of nodes, they need to know which nodes are adjacent to which other nodes. Because the information from those neighbours must also be aggregated from *their* neighbourhoods, these models thus need a relatively large amount of information about the structure of a graph. This information comes in the form of an *adjacency matrix* A, such that $A[i, j] = 1$ if there exists a link between nodes i and j, and 0 otherwise.

Of course, the adjacency matrix only tells the model which nodes to aggregate. To also know *what* to aggregate, we need another matrix which uniquely identifies each node. This matrix is often called the *node feature matrix* X. If our nodes comes with one or more attributes, or *features*, then we can fill up this matrix with the corresponding values. This is commonly done with *multimodal learning*. More often, however, it is easier to just ignore the node features (if any), and to let X equal the identity matrix I such that $X[i, j] = 1$ iff $i = j$ and 0 otherwise.

Finally, since the downstream task is *node classification*, we need a vector representation, the *target vector* y, for the class labels that are used to compute the loss and accuracy scores. Since we need to calculate the gradients during this step, we need a numerical encoding for the labels.

1.4.1 Task 1a: Creating a feature matrix

Write a procedure to generate a node feature matrix that maps each node to its respective feature vector. The result should be a *sparse* float tensor `X`, such that `X[i]` refers to the feature vector of node `i`. Since the Cora dataset comes with integer-encoded node features (the bag-of-words) there is no need to generate an indentity matrix. Remember that the whole set of features is stored in variable `features`.

[ ]:
```python
import torch

X = torch.FloatTensor(features)

# making X sparse
nonzero_indices = torch.nonzero(X)
values = X[nonzero_indices[:, 0], nonzero_indices[:, 1]]
X = torch.sparse.FloatTensor(nonzero_indices.t(), values, X.size())

print("Sparse X:")
print(X.to_dense()[:10, :10])
```

```
X = X.to(device)
```

```
Sparse X:
tensor([[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., 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.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 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., 0.]])

/tmp/ipykernel_123824/1308467794.py:8: UserWarning:
torch.sparse.SparseTensor(indices, values, shape, *, device=) is deprecated.
Please use torch.sparse_coo_tensor(indices, values, shape, dtype=, device=).
(Triggered internally at ../torch/csrc/utils/tensor_new.cpp:605.)
  X = torch.sparse.FloatTensor(nonzero_indices.t(), values, X.size())
```

Run the following code to check your feature matrix

1.4.2 Task 1b: Create an adjacency matrix

Write a procedure to generate the adjacency matrix for the Cora graph. The result should be a *sparse* float tensor `A`, such that `A[i,j]` equals `1` if there exists an edge between nodes `i` and `j`, and `0` otherwise. Be aware that the GCN requires all nodes to have a reflexive edge (loops) which ensures that the nodes remember their previous state when updating.

Run the following code to check your adjacency matrix

[ ]:
```
def create_adjacency_matrix(edges_np: np.array):
    edges_tensor = torch.tensor(edges_np)

    num_nodes = torch.max(edges_tensor).item() + 1

    # creating reflexive edges
    reflexive_edges = torch.arange(0, num_nodes).unsqueeze(1).repeat(1, 2)

    # adding the reflexive edges to the rest
    edges_tensor_with_reflexive = torch.cat([edges_tensor, reflexive_edges],␣
 ↪dim=0)

    indices = edges_tensor_with_reflexive.t()
    values = torch.ones(edges_tensor_with_reflexive.size(0), dtype=torch.long)

    return torch.sparse.FloatTensor(indices, values, torch.Size([num_nodes,␣
 ↪num_nodes]))
```

```
## test case
test_data = np.array([[0, 1], [1, 2]])

expected_dense_matrix = np.array([
    [1, 1, 0],
    [0, 1, 1],
    [0, 0, 1],
])
assert (create_adjacency_matrix(test_data).to_dense().numpy() ==␣
 ↪expected_dense_matrix).all()
```

[ ]:
```
A = create_adjacency_matrix(edges_reindexed)
print("A.shape\n", A.shape)
print("\nA.to_dense()\n", A.to_dense())
# Check your adjacency matrix by using the sum as proxy
print(f"The number of connections, {int(A.sum())}, must equal the number of␣
 ↪edges, {num_edges},"
      f" plus the number of nodes, {num_nodes}")
assert int(A.sum()) == num_edges + num_nodes

A = A.to(device)
```

```
A.shape
 torch.Size([2708, 2708])

A.to_dense()
 tensor([[1, 0, 0,  …, 0, 0, 0],
        [0, 1, 0,  …, 0, 0, 0],
        [0, 0, 1,  …, 0, 0, 0],
        …,
        [0, 0, 0,  …, 1, 0, 0],
        [0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 0, 1]])
The number of connections, 8137, must equal the number of edges, 5429, plus the
number of nodes, 2708
```

1.4.3 Task 1c: Create the target vector

Write a procedure to generate the target vector with integer-encoded class labels. The result should be a `long` vector `y_true`, such that `y_true[i]` holds the target label of node `i`. Note that, with PyTorch, different loss functions require differently formatted target vectors.

[ ]:
```
class_to_index = {label: idx for idx, label in enumerate(np.unique(labels))}
y_true = [class_to_index[label] for label in labels]
num_labels = len(class_to_index.keys())
```

Run the following code to check your target vector

```
[ ]: print(f'number of unique labels: {num_labels}\n')

     print(f'y: {y_true[:10]}')
```

```
number of unique labels: 7

y: [2, 5, 4, 4, 3, 3, 6, 2, 2, 6]
```

1.5 Task 2: Partition the dataset

To properly perform our experiments we first need to partition our data into a *train* and *test* split. These splits are used to train and test our model, respectively, and must be disjoint to avoid information leakage. Ideally, we would als create a *validation* split to use for model selection and/or hyperparameter optimization, but we dispense with that for now.

Create a procedure to create a train and test split with a ratio of 4 to 1. The result should be two vectors, `train_idx` and `test_idx`, that contain indices that point to the actual data (a *mask*) that are randomly drawn from the set of all indices.

```
[ ]: train_to_full_dataset_ratio = 3 / 4

     torch.manual_seed(seed)
     shuffled_indices = torch.randperm(num_nodes).numpy()

     num_train = int(train_to_full_dataset_ratio * num_nodes)
     num_test = num_nodes - num_train
     assert num_train + num_test == num_nodes

     # use mask
     train_idx = shuffled_indices[:num_train]
     test_idx = shuffled_indices[num_train:]

     # test disjoint datasets
     assert len(np.intersect1d(train_idx, test_idx)) == 0, "Train and test sets are␣
      ↪not disjoint!"
```

Run the following code to check your partitions

```
[ ]: print(f"number of training samples: {num_train}")
     print(f"number of testing samples: {num_test}")

     print(f"\ntrain indices:\n{train_idx[:5]}")
     print(f"\ntest indices:\n{test_idx[:5]}")
```

```
number of training samples: 2031
number of testing samples: 677

train indices:
[1594  519  528 1642  606]
```

```
test indices:
[1558  307  465  857 1580]
```

1.6 The Graph Convolutional

The *Graph Convolutional Network* (GCN) is arguably the first major breakthrough in GNN development. Developed in 2017, the GCN introduces the idea of the *spectral graph convolution*, which, analogues to its visual counterpart, aggregates the information surrounding an object. In the case of *Convolutional Neural Networks* (CNN), these objects are pixels, whereas with the GCN these are nodes. This comparison becomes evident when you consider images as regular (grid-shaped) graphs with pixel as nodes.

The GCN is defined as a network with one or more *Graph Convolution* layers. Each of these layers applies the convolution operator to its input, and is defined as

$$H^{l+1} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{l}W^{l})$$

where $\tilde{A}$ is the adjacency matrix with reflexive edges, $\tilde{D}$ the degree matrix derived from $\tilde{A}$, H^l the internal node representations of layer l, W^l the weight matrix of layer l, and σ a nonlinearity like $ReLU$. Note that the initial node representation matrix $H^0 = X$.

In the experiments that we are reproducing the GCN is used for the task of node classification. For this purpose, the GCN is given two graph convolution layers, but with the nonlinearity of the last layer replaced by a softmax function:

$$y = softmax(\hat{A}\ \sigma(\hat{A}XW^0)\ W^1)$$

with $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$

1.6.1 Task 3a: Implement the Graph Convolution

Implement the graph convolution layer as a subclass of PyTorch `nn.Module`. Concretely, you must implement the `__init__` and `forward` functions. Ensure that the computation supports sparse tensors, and that the input and output dimensions can be set on initialisation.

[ ]:
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class GraphConvolutionLayer(nn.Module):
    """
    A single Graph Convolution Layer
    """

    def __init__(self, in_features, out_features, adjacency_matrix):
        super(GraphConvolutionLayer, self).__init__()
        self.weight = nn.Parameter(torch.FloatTensor(in_features, out_features))
```

```
        torch.manual_seed(seed)
        nn.init.xavier_uniform_(self.weight)  # Using Xavier/Glorot␣
↪initialization
        self.register_buffer('adjacency_matrix', adjacency_matrix.float().
↪to(device))

    def forward(self, X):
        degree_matrix = torch.diag(torch.sparse.sum(self.adjacency_matrix,␣
↪dim=1).to_dense()).float().to(device)
        degree_matrix_inverse = torch.sqrt(torch.inverse(degree_matrix)).
↪to(device)
        normalized_adjacency = degree_matrix_inverse @ self.adjacency_matrix @␣
↪degree_matrix_inverse

        return normalized_adjacency @ X @ self.weight.to(device)
```

Run the following cell to initialize and test your implementation

[ ]:
```
conv = GraphConvolutionLayer(num_features, num_features, A)
output = conv(X)
print(output)
```

```
tensor([[-0.0503, -0.0973, -0.0206,  …,  0.0846, -0.0349, -0.1280],
        [-0.2820, -0.0378,  0.1060,  …,  0.1424,  0.0420, -0.0302],
        [-0.1249,  0.0794,  0.0165,  …, -0.0176, -0.0475, -0.1004],
        …,
        [ 0.1391,  0.0575,  0.1543,  …, -0.0943,  0.0026, -0.1369],
        [-0.0500,  0.1061,  0.0262,  …,  0.0724, -0.0114, -0.1300],
        [-0.0101, -0.1099, -0.0559,  …,  0.1023, -0.1390,  0.2049]],
       device='cuda:0', grad_fn=<MmBackward0>)
```

1.6.2 Task 3b: Implement the Graph Convolutional Model

Implement the GCN as specified in the paper [1]. Concretely, implement a two-layer GCN with a ReLU activation function and dropout after the first layer, and with a softmax layer after the second.

[ ]:
```
class GCN(nn.Module):
    def __init__(self, in_features, hidden_features, output_classes,␣
↪adjacency_matrix, dropout_prob=0.5):
        super().__init__()
        self.layer_1 = GraphConvolutionLayer(in_features, hidden_features,␣
↪adjacency_matrix)
        self.layer_2 = GraphConvolutionLayer(hidden_features, output_classes,␣
↪adjacency_matrix)
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, X) -> torch.Tensor:
```

```
        layer_1_activations = F.relu(self.dropout(self.layer_1(X)))
        output = F.log_softmax(self.layer_2(layer_1_activations), dim=1)
        return output.to(device)
```

Run the following cell to initialize and test your implementation

[ ]:
```
model = GCN(num_features, 10000, num_labels, A).to(device)
y_pred = model(X)
print(y_pred[0:10])
```

```
tensor([[-2.0318, -1.8900, -2.0016, -2.0271, -1.9593, -1.7542, -1.9879],
        [-2.0674, -1.8944, -1.9483, -1.9754, -1.9565, -1.8809, -1.9102],
        [-1.9990, -2.0754, -1.8633, -1.9979, -1.9430, -1.8202, -1.9450],
        [-1.9439, -1.8578, -2.1384, -1.8479, -2.0470, -1.9104, -1.9087],
        [-1.9488, -2.0067, -1.9578, -1.8417, -2.0636, -1.8956, -1.9229],
        [-1.9867, -1.8985, -1.9323, -1.9272, -2.0152, -1.9887, -1.8805],
        [-2.0702, -1.8889, -1.9786, -2.0278, -1.9403, -1.8065, -1.9323],
        [-1.8707, -1.9810, -1.8666, -2.0683, -1.9829, -1.9567, -1.9105],
        [-1.9334, -1.9024, -1.9995, -2.0428, -1.9988, -1.8324, -1.9273],
        [-2.0950, -1.8680, -1.9431, -2.0150, -1.8539, -1.9961, -1.8745]],
       device='cuda:0', grad_fn=<SliceBackward0>)
```

1.7 Training and testing

In normal circumstances the GCN updates its internal representation for all nodes in the graph after each pass. In other words, the GCN operates on the entire graph at once, rather than on just the training, test, or validation set. Since these sets are disjoint, it necessarily means that only part of the class labels are available each time. This is called *semi-supervised learning*. Because the model sees the entire graph each pass, it still outputs predictions for all the nodes. However, by just calculating the loss and accuracy on a specific split, we ensure that only the error on the nodes in that split is backpropagated.

1.7.1 Task 4: Implementing evaluation metrics

Write a procedure to calculate the loss *and* a procedure to calculate the accuracy. Assume that we have a tensor with true labels, `y_true`, and a tensor with predicted labels, `y_pred`.

[ ]:
```
def compute_accuracy(y_true: list[int], y_pred: torch.Tensor) -> float:
    y_true = torch.tensor(y_true).to(device)
    predicted_labels = torch.argmax(y_pred, dim=1)
    correct_predictions = (predicted_labels == y_true).sum().item()
    accuracy = correct_predictions / len(y_true)

    return accuracy

def compute_loss(y_true: list[int], y_pred: torch.Tensor) -> torch.Tensor:
    y_true_tensor = torch.tensor(y_true, dtype=torch.long).to(device)
    loss_function = torch.nn.NLLLoss()
```

```
    return loss_function(y_pred, y_true_tensor)
```

Run the following cell to test your code:

[ ]:
```
y_pred_labels = torch.argmax(y_pred, axis = 1)
print(f'Predicted labels: {y_pred_labels[:10]}')

acc = compute_accuracy(y_true, y_pred)
print(f'Accuracy: {acc:.3f}')

loss = compute_loss(y_true, y_pred)
print(f'Loss: {loss:.3f}')
```

```
Predicted labels: tensor([5, 5, 5, 3, 3, 6, 5, 2, 5, 4], device='cuda:0')
Accuracy: 0.136
Loss: 1.945
```

1.7.2 Task 5a: Implement the training loop

Write a procedure to train the model. Specifically, create a loop that passes the entire graph through the model every epoch, while computing the loss and accuracy on just the training set. Use the Adam optimizer and the negative log likelihood loss.

[ ]:
```
# set hyperparameters
learning_rate = 0.01
num_epoch = 100
hidden_features = num_features * 10

model = GCN(num_features, hidden_features, num_labels, A).to(device)
# set optimizer
optimizer = torch.optim.Adam(
    model.parameters(),
    lr = learning_rate
)

loss_train_history = []
loss_test_history = []
acc_train_history = []
acc_test_history = []

for epoch in range(1, num_epoch+1):
    # allow model parameters to be learned
    model.train()

    y_pred = model(X)

    # we will compute the loss only with respect to train data
    y_true_train = np.array(y_true)[train_idx].tolist()
```

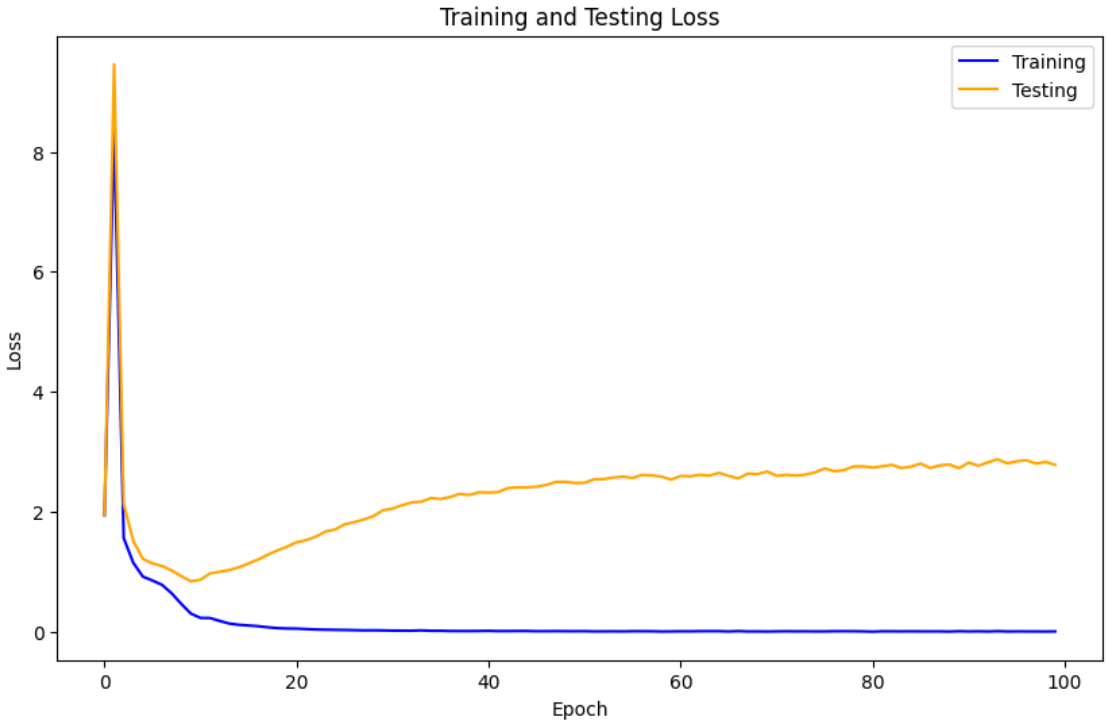
```
    y_pred_train = y_pred[train_idx]
    loss_train = compute_loss(y_true_train, y_pred_train)
    acc_train = compute_accuracy(y_true_train, y_pred_train)
    loss_train_history.append(loss_train.item())
    acc_train_history.append(acc_train)


    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss_train.backward()
    optimizer.step()

    loss_train = float(loss_train)  # release memory of computation graph

    ## I'm adding metrics for test dataset here so we see model performance
    ## Working with test data will have no effect on training
    with torch.no_grad():
        y_true_test = np.array(y_true)[test_idx].tolist()
        y_pred_test = y_pred[test_idx]

        loss_test = compute_loss(y_true_test, y_pred_test)
        acc_test = compute_accuracy(y_true_test, y_pred_test)
        loss_test_history.append(loss_test.item())
        acc_test_history.append(acc_test)

        # just making sure
        loss_test = float(loss_test)

    if epoch % 5 == 0:
        print(f'Epoch {epoch:3d} - ', end='')
        print()
        print(f'Train loss: {loss_train:0.4f}\tTrain acc: {acc_train:0.4f}')
        print(f'Test loss: {loss_test:0.4f}\tTest acc: {acc_test:0.4f}')
        print()
```

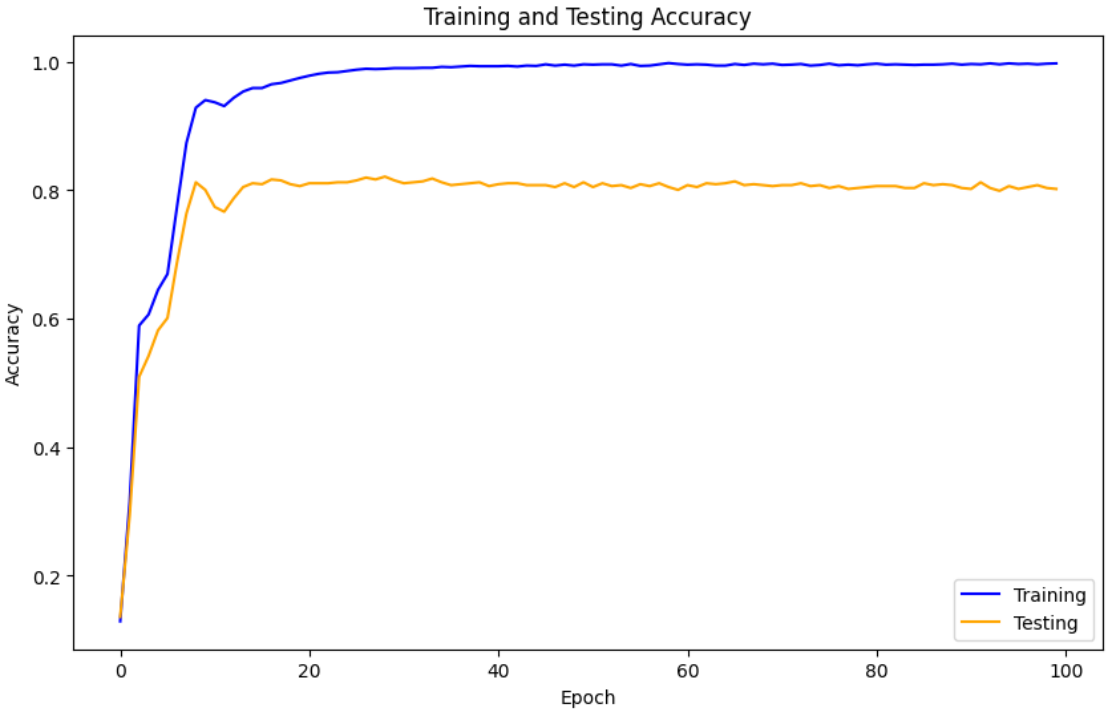
```
Epoch   5 -
Train loss: 0.9217      Train acc: 0.6450
Test loss: 1.2178       Test acc: 0.5820

Epoch  10 -
Train loss: 0.3063      Train acc: 0.9404
Test loss: 0.8453       Test acc: 0.8006

Epoch  15 -
Train loss: 0.1199      Train acc: 0.9591
Test loss: 1.0774       Test acc: 0.8109
```

```
Epoch  20 -
Train loss: 0.0580      Train acc: 0.9749
Test loss: 1.4202       Test acc: 0.8065

Epoch  25 -
Train loss: 0.0357      Train acc: 0.9857
Test loss: 1.7066       Test acc: 0.8124

Epoch  30 -
Train loss: 0.0260      Train acc: 0.9902
Test loss: 2.0271       Test acc: 0.8154

Epoch  35 -
Train loss: 0.0199      Train acc: 0.9921
Test loss: 2.2321       Test acc: 0.8124

Epoch  40 -
Train loss: 0.0167      Train acc: 0.9931
Test loss: 2.3279       Test acc: 0.8065

Epoch  45 -
Train loss: 0.0167      Train acc: 0.9936
Test loss: 2.4097       Test acc: 0.8080

Epoch  50 -
Train loss: 0.0121      Train acc: 0.9961
Test loss: 2.4819       Test acc: 0.8124

Epoch  55 -
Train loss: 0.0096      Train acc: 0.9966
Test loss: 2.5880       Test acc: 0.8035

Epoch  60 -
Train loss: 0.0091      Train acc: 0.9966
Test loss: 2.5390       Test acc: 0.8006

Epoch  65 -
Train loss: 0.0136      Train acc: 0.9941
Test loss: 2.6516       Test acc: 0.8109

Epoch  70 -
Train loss: 0.0076      Train acc: 0.9970
Test loss: 2.6733       Test acc: 0.8065

Epoch  75 -
Train loss: 0.0095      Train acc: 0.9951
Test loss: 2.6589       Test acc: 0.8080
```

```
Epoch  80 -
Train loss: 0.0097      Train acc: 0.9961
Test loss: 2.7586       Test acc: 0.8050

Epoch  85 -
Train loss: 0.0111      Train acc: 0.9951
Test loss: 2.7550       Test acc: 0.8035

Epoch  90 -
Train loss: 0.0132      Train acc: 0.9956
Test loss: 2.7275       Test acc: 0.8035

Epoch  95 -
Train loss: 0.0084      Train acc: 0.9975
Test loss: 2.8091       Test acc: 0.8065

Epoch 100 -
Train loss: 0.0095      Train acc: 0.9975
Test loss: 2.7841       Test acc: 0.8021
```

[ ]:
```python
import matplotlib.pyplot as plt

def plot_history(train_history, test_history, ylabel, title):
    plt.figure(figsize=(10, 6))
    plt.plot(train_history, label='Training', color='blue')
    plt.plot(test_history, label='Testing', color='orange')
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel(ylabel)
    plt.legend()
    plt.show()

plot_history(loss_train_history, loss_test_history, 'Loss', 'Training and␣
 ↪Testing Loss')
plot_history(acc_train_history, acc_test_history, 'Accuracy', 'Training and␣
 ↪Testing Accuracy')
```


Training and Testing Loss
Training
Testing
Loss
Epoch
0
2
4
6
8
0
20
40
60
80
100


Training and Testing Accuracy
Training
Testing
Accuracy
Epoch
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100

1.7.3 Task 5b: Implement the test procedure

Write a procedure to test the now-trained model. Ensure that the weights of your model are frozen during testing, and that the loss and accuracy scores are calculated on just the test set.

[ ]:
```
# freeze model parameters for evaluation
model.eval()

y_pred = model(X)

# we will compute the loss only with respect to train data
y_true_test = np.array(y_true)[test_idx].tolist()
y_pred_test = y_pred[test_idx]

loss = compute_loss(y_true_test, y_pred_test)
acc = compute_accuracy(y_true_test, y_pred_test)

loss = float(loss)  # release memory of computation graph

print(f'test loss: {loss:0.4f}\ntest acc: {acc:0.4f}')
```

```
test loss: 2.8162
test acc: 0.8080
```