Graph Neural Networks: Challenge

Handout: 24.03.2025 08:00
Due: 02.04.2025 23:59

Spring 2025
Hands-On Deep Learning

Challenge Open in IDE

Escape the Maze

In this challenge, you will solve a very practical task. How to find your way in a maze! First, let's look at the concrete task.

We generate mazes on a grid of a fixed size and create two different versions of the same dataset. First, we can represent mazes as images. The input for the task are two points (pixels) in the maze that are marked with the same color. We have to find the (unique) path between these two points and mark them in the output. Instead of representing a maze as an image, we can also use graphs for this task. See the graph representation of a maze, the according image to the right of it and the ground truth for the image.

And for a bigger maze:

The task: for each node you have an input feature indicating wheter the node is one of the two endpoints. Our goal is to determine for each node if it is on the path between the two marked endpoints.

Setup

Now that you had a look at the dataset and the mazes we can think about how we want to design a GNN that can solve this task.

First, we have the setup for the training loop that you should now be very familiar with. We have the model, number of epochs and learning rate that we can adjust. For this task, we do not want to temper with the batch size, so just leave it at 1 (otherwise some functionalities might break down the road).

```
def eval_model(model, dataset, mode=None):
    model.eval()
    acc = 0
    tot_nodes = 0
    tot_graphs = 0
    perf = 0
    gpred = []
    gsol = []

    for step, batch in enumerate(dataset):
        n = len(batch.x)/batch.num_graphs
        with torch.no_grad():
            batch = batch.to(device)
            pred = model(batch, int(n))

        if mode == "small":
            if n > 24:
                break
        elif mode == "medium":
            if n > 84:
                break
        elif mode == "large":
            if n > 1e4:
                break

        y_pred = torch.argmax(pred,dim=1)
        tot_nodes += len(batch.x)
        tot_graphs += batch.num_graphs

        graph_acc = torch.sum(y_pred == batch.y).item()

        acc += graph_acc
        for p in y_pred:
            gpred.append(int(p.item()))
        for p in batch.y:
            gsol.append(int(p.item()))
        if graph_acc == n:
            perf += 1

    gpred = torch.tensor(gpred)
    gsol = torch.tensor(gsol)
    f1score = f1_score(gpred, gsol)

    return f"node accuracy: {acc/tot_nodes:.3f} | node f1 score: {f1score:.3f} | graph accuracy: {perf/tot_graphs:.3f}"

def train_model(model, train_dataset_gen, epochs=20, lr=0.0004):
    dataset = train_dataset_gen(n_samples=200)

    criterion = torch.nn.NLLLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    val_split = 0.2
    train_size = int(val_split*len(dataset))
    train_loader = DataLoader(dataset[:train_size], batch_size=1, shuffle=True)
    val_set = DataLoader(dataset[train_size:], batch_size = 1)

    model.train()

    worst_loss = +1
    best_model = None


    for epoch in range(epochs):
        running_loss = 0.0
        for i, data in enumerate(train_loader):
            optimizer.zero_grad()
            data = data.to(device)

            # could change additional parameters here
            pred = model(data, data.num_nodes)

            loss = criterion(pred, data.y.to(torch.long))

            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()

            running_loss += loss.item()
        ss = eval_model(model, val_set)

        graph_val = float((ss.split(" "))[-1])
        print(f'Epoch: {epoch + 1} loss: {running_loss / len(train_loader.dataset):.5f} \t {ss}')
        comp = (-graph_val, running_loss)
        if worst_loss == +1 or comp < worst_loss:
            worst_loss = comp
            best_model = deepcopy(model)
            print('store new best model', comp)

        running_loss = 0.0
    return best_model
```

We call this specific GNN the MazeGNN and furthermore we want to use our own custom convolution (the MazeConv). The skeleton provided here is very basic and you can change and customize it the way you want it to be. You might want to consider to include some of the tricks we have seen in the GNNs previously, such as including MLPs etc.

```
class MazeConv(MessagePassing):
    def __init__(self):
        super(MazeConv, self).__init__(aggr='add')
        #TODO
        #initialize custom message passing, store the MLP's for the convolution

    def forward(self, x, edge_index):
        #TODO
        #define own computation, call the round with
        #self.propagate(edge_index, x=x)
        x = x + self.propagate(edge_index, x = x)
        return x

    def message(self, x_j, x_i):
        #TODO
        #define the custom message that gnns exchange, x_i is own state, x_j is neighboring state
        return x_j


class MazeGNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.dropout = 0.2
        hidden_dim = 8

        # we recommend using a larger dimension during the graph computation
        # therefore the encoder and decoder map in-/output to the used dimension
        self.encoder = self.get_mlp(2,8,hidden_dim)
        self.decoder = self.get_mlp(hidden_dim,32,2, last_relu = False)

        self.conv = MazeConv() # TODO you might want to add additional stuff here such as MLPs


    # get the graph and number of nodes in the graph
    def forward(self, data, num_nodes):
        #things to consider:
        # how many convolutions do we need to execute?
        # what kind of custom convolution could help

        x, edge_index = data.x, data.edge_index
        input = x

        x = self.encoder(x)

        #----------------------
        # Here you should specify the graph computations
        for i in range(2):
            x = self.conv(x, edge_index)

        #----------------------
        x = self.decoder(x)

        #output is logits of belonging to two classes
        return F.log_softmax(x, dim=1)

    # helper function - generates an MLP w. relu activation with 3 layers
    def get_mlp(self, input_dim, hidden_dim, output_dim, last_relu = True):
        modules = [torch.nn.Linear(input_dim, int(hidden_dim)), torch.nn.ReLU(), torch.nn.Dropout(self.dropout), torch.nn.Linear(int(hidden_dim), output_dim)]
        if last_relu:
            modules.append(torch.nn.ReLU())
        return torch.nn.Sequential(*modules)
```

Problem specification

For the challenge you are *only* allowed to train on the 4×4 graphs in `train_dataset`. Your goal is to make a GNN which generalises to larger graphs without being trained on them. The test dataset contains 200 graphs of sizes 4×4, 8×8, 16×16 32×32 each.

Things you might want to try in order to improve the GNN:

- Add an MLP in the update step of the graph convolution (similar to the tasks we have seen in the notebook)
- Add an MLP in the message step of the graph convolution and concatenate x_i x_j (as seen in the notebook)
- Increase the number of convolutional layers
- Increase the number of training samples (a bit more than 200 might train better)

This should already increase performance by quite a bit on the small graphs. For getting better performance on the larger graphs you might want to consider the following:

- Make the number of convolutional layers dependent on the number of nodes in the graph
- Before each graph convolution: concatenate the input and the x vector and put it through an MLP, this will help the GNN to remember the original input

Scoring System

Points are awarded based on the model's final score, according to the thresholds below:

Model accuracy (running)	CodeExpert score (submitting)	Points Earned
< 0.1	< 8.3%	0
≥ 0.1	≥ 8.3%	1
≥ 0.2	≥ 25.0%	2
≥ 0.3	≥ 41.7%	3
≥ 0.4	≥ 58.3%	4
≥ 0.6	≥ 75.0%	5
≥ 0.8	≥ 91.7%	6

The model accuracy is your models performance on the test set, see logs after running a job. The CodeExpert score is the percentage displayed in CodeExpert when submitting the job.

Task

Your goal is to complete the `init_model` and `train_model` functions. Your functions are imported and called in for evaluation. **Function signature must remain unchanged.** Your code will be called by the following function:

```
def run():
  set_seed(42)

  # Get datasets for training and testing
  train_generator, test_dataset = get_data()

  # Initialize the model using student's init_model function
  model = init_model()

  # Train the model using student's train_model function
  model = train_model(model, train_generator)

  # Evaluate the model on the test set
  model.eval()  # Set the model to evaluation mode
  score = evaluate_model(model, test_dataset)

  return score
```

The training and testing datasets are provided to you. The model is set to evaluation mode for evaluation. You need to move models and data between RAM and the GPU.

Rules

You are not allowed to use samples in the test dataset for training, or to use a model pre-trained on the test dataset.

You cannot set a new seed.

Your code must run in 15 minutes.

Needs to be a learned solution that is using a GNN.

Using a shortest path finder is not within the rules of the challenge. To specify this: we should be able to give your approach to many different algorithmic graph learning problems and get similar results.

Tutorial

Tutorial on how to use CodeExpert.

Toblerone

The top three solutions will receive a Toblerone. Good luck! 🚀