dmlc / dgl Public

dgl / examples / pytorch / rgcn / link.py

frozenbugs and Ubuntu examples (#5323) ✕ 10 months ago

352 lines (305 loc) · 11.1 KB

Code Blame Raw

```python
import dgl
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import tqdm
from dgl.data.knowledge_graph import FB15k237Dataset
from dgl.dataloading import GraphDataLoader
from dgl.nn.pytorch import RelGraphConv


# for building training/testing graphs
def get_subset_g(g, mask, num_rels, bidirected=False):
    src, dst = g.edges()
    sub_src = src[mask]
    sub_dst = dst[mask]
    sub_rel = g.edata["etype"][mask]

    if bidirected:
        sub_src, sub_dst = torch.cat([sub_src, sub_dst]), torch.cat(
            [sub_dst, sub_src]
        )
        sub_rel = torch.cat([sub_rel, sub_rel + num_rels])

    sub_g = dgl.graph((sub_src, sub_dst), num_nodes=g.num_nodes())
    sub_g.edata[dgl.ETYPE] = sub_rel
    return sub_g


class GlobalUniform:
    def __init__(self, g, sample_size):
        self.sample_size = sample_size
        self.eids = np.arange(g.num_edges())

    def sample(self):
        return torch.from_numpy(np.random.choice(self.eids, self.sample_size))


class NegativeSampler:
    def __init__(self, k=10):  # negative sampling rate = 10
        self.k = k

    def sample(self, pos_samples, num_nodes):
        batch_size = len(pos_samples)
        neg_batch_size = batch_size * self.k
        neg_samples = np.tile(pos_samples, (self.k, 1))

        values = np.random.randint(num_nodes, size=neg_batch_size)
        choices = np.random.uniform(size=neg_batch_size)
        subj = choices > 0.5
        obj = choices <= 0.5
        neg_samples[subj, 0] = values[subj]
        neg_samples[obj, 2] = values[obj]
        samples = np.concatenate((pos_samples, neg_samples))

        # binary labels indicating positive and negative samples
        labels = np.zeros(batch_size * (self.k + 1), dtype=np.float32)
        labels[:batch_size] = 1

        return torch.from_numpy(samples), torch.from_numpy(labels)


class SubgraphIterator:
    def __init__(self, g, num_rels, sample_size=30000, num_epochs=6000):
        self.g = g
        self.num_rels = num_rels
        self.sample_size = sample_size
        self.num_epochs = num_epochs
        self.pos_sampler = GlobalUniform(g, sample_size)
        self.neg_sampler = NegativeSampler()

    def __len__(self):
        return self.num_epochs

    def __getitem__(self, i):
        eids = self.pos_sampler.sample()
        src, dst = self.g.find_edges(eids)
        src, dst = src.numpy(), dst.numpy()
        rel = self.g.edata[dgl.ETYPE][eids].numpy()

        # relabel nodes to have consecutive node IDs
        uniq_v, edges = np.unique((src, dst), return_inverse=True)
        num_nodes = len(uniq_v)
        # edges is the concatenation of src, dst with relabeled ID
        src, dst = np.reshape(edges, (2, -1))
        relabeled_data = np.stack((src, rel, dst)).transpose()

        samples, labels = self.neg_sampler.sample(relabeled_data, num_nodes)

        # use only half of the positive edges
        chosen_ids = np.random.choice(
            np.arange(self.sample_size),
            size=int(self.sample_size / 2),
            replace=False,
        )
        src = src[chosen_ids]
        dst = dst[chosen_ids]
        rel = rel[chosen_ids]
        src, dst = np.concatenate((src, dst)), np.concatenate((dst, src))
        rel = np.concatenate((rel, rel + self.num_rels))
        sub_g = dgl.graph((src, dst), num_nodes=num_nodes)
        sub_g.edata[dgl.ETYPE] = torch.from_numpy(rel)
        sub_g.edata["norm"] = dgl.norm_by_dst(sub_g).unsqueeze(-1)
        uniq_v = torch.from_numpy(uniq_v).view(-1).long()

        return sub_g, uniq_v, samples, labels


class RGCN(nn.Module):
    def __init__(self, num_nodes, h_dim, num_rels):
        super().__init__()
        # two-layer RGCN
        self.emb = nn.Embedding(num_nodes, h_dim)
        self.conv1 = RelGraphConv(
            h_dim,
            h_dim,
            num_rels,
            regularizer="bdd",
            num_bases=100,
            self_loop=True,
        )
        self.conv2 = RelGraphConv(
            h_dim,
            h_dim,
            num_rels,
            regularizer="bdd",
            num_bases=100,
            self_loop=True,
        )
        self.dropout = nn.Dropout(0.2)

    def forward(self, g, nids):
        x = self.emb(nids)
        h = F.relu(self.conv1(g, x, g.edata[dgl.ETYPE], g.edata["norm"]))
        h = self.dropout(h)
        h = self.conv2(g, h, g.edata[dgl.ETYPE], g.edata["norm"])
        return self.dropout(h)


class LinkPredict(nn.Module):
    def __init__(self, num_nodes, num_rels, h_dim=500, reg_param=0.01):
        super().__init__()
        self.rgcn = RGCN(num_nodes, h_dim, num_rels * 2)
        self.reg_param = reg_param
        self.w_relation = nn.Parameter(torch.Tensor(num_rels, h_dim))
        nn.init.xavier_uniform_(
            self.w_relation, gain=nn.init.calculate_gain("relu")
        )

    def calc_score(self, embedding, triplets):
```

```
    def calc_score(self, embedding, triplets):
        s = embedding[triplets[:, 0]]
        r = self.w_relation[triplets[:, 1]]
        o = embedding[triplets[:, 2]]
        score = torch.sum(s * r * o, dim=1)
        return score

    def forward(self, g, nids):
        return self.rgcn(g, nids)

    def regularization_loss(self, embedding):
        return torch.mean(embedding.pow(2)) + torch.mean(self.w_relation.pow(2))

    def get_loss(self, embed, triplets, labels):
        # each row in the triplets is a 3-tuple of (source, relation, destination)
        score = self.calc_score(embed, triplets)
        predict_loss = F.binary_cross_entropy_with_logits(score, labels)
        reg_loss = self.regularization_loss(embed)
        return predict_loss + self.reg_param * reg_loss


def filter(
    triplets_to_filter, target_s, target_r, target_o, num_nodes, filter_o=True
):
    """Get candidate heads or tails to score"""
    target_s, target_r, target_o = int(target_s), int(target_r), int(target_o)
    # Add the ground truth node first
    if filter_o:
        candidate_nodes = [target_o]
    else:
        candidate_nodes = [target_s]
    for e in range(num_nodes):
        triplet = (
            (target_s, target_r, e) if filter_o else (e, target_r, target_o)
        )
        # Do not consider a node if it leads to a real triplet
        if triplet not in triplets_to_filter:
            candidate_nodes.append(e)
    return torch.LongTensor(candidate_nodes)


def perturb_and_get_filtered_rank(
    emb, w, s, r, o, test_size, triplets_to_filter, filter_o=True
):
    """Perturb subject or object in the triplets"""
    num_nodes = emb.shape[0]
    ranks = []
    for idx in tqdm.tqdm(range(test_size), desc="Evaluate"):
        target_s = s[idx]
        target_r = r[idx]
        target_o = o[idx]
        candidate_nodes = filter(
            triplets_to_filter,
            target_s,
            target_r,
            target_o,
            num_nodes,
            filter_o=filter_o,
        )
        if filter_o:
            emb_s = emb[target_s]
            emb_o = emb[candidate_nodes]
        else:
            emb_s = emb[candidate_nodes]
            emb_o = emb[target_o]
        target_idx = 0
        emb_r = w[target_r]
        emb_triplet = emb_s * emb_r * emb_o
        scores = torch.sigmoid(torch.sum(emb_triplet, dim=1))

        _, indices = torch.sort(scores, descending=True)
        rank = int((indices == target_idx).nonzero())
        ranks.append(rank)
    return torch.LongTensor(ranks)


def calc_mrr(
    emb, w, test_mask, triplets_to_filter, batch_size=100, filter=True
):
    with torch.no_grad():
        test_triplets = triplets_to_filter[test_mask]
        s, r, o = test_triplets[:, 0], test_triplets[:, 1], test_triplets[:, 2]
        test_size = len(s)
        triplets_to_filter = {
            tuple(triplet) for triplet in triplets_to_filter.tolist()
        }
        ranks_s = perturb_and_get_filtered_rank(
            emb, w, s, r, o, test_size, triplets_to_filter, filter_o=False
        )
        ranks_o = perturb_and_get_filtered_rank(
            emb, w, s, r, o, test_size, triplets_to_filter
        )
        ranks = torch.cat([ranks_s, ranks_o])
        ranks += 1  # change to 1-indexed
        mrr = torch.mean(1.0 / ranks.float()).item()
    return mrr


def train(
    dataloader,
    test_g,
    test_nids,
    test_mask,
    triplets,
    device,
    model_state_file,
    model,
):
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
    best_mrr = 0
    for epoch, batch_data in enumerate(dataloader):  # single graph batch
```

```
        model.train()
        g, train_nids, edges, labels = batch_data
        g = g.to(device)
        train_nids = train_nids.to(device)
        edges = edges.to(device)
        labels = labels.to(device)

        embed = model(g, train_nids)
        loss = model.get_loss(embed, edges, labels)
        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(
            model.parameters(), max_norm=1.0
        )  # clip gradients
        optimizer.step()
        print(
            "Epoch {:04d} | Loss {:.4f} | Best MRR {:.4f}".format(
                epoch, loss.item(), best_mrr
            )
        )
        if (epoch + 1) % 500 == 0:
            # perform validation on CPU because full graph is too large
            model = model.cpu()
            model.eval()
            embed = model(test_g, test_nids)
            mrr = calc_mrr(
                embed, model.w_relation, test_mask, triplets, batch_size=500
            )
            # save best model
            if best_mrr < mrr:
                best_mrr = mrr
                torch.save(
                    {"state_dict": model.state_dict(), "epoch": epoch},
                    model_state_file,
                )
            model = model.to(device)


if __name__ == "__main__":
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Training with DGL built-in RGCN module")

    # load and preprocess dataset
    data = FB15k237Dataset(reverse=False)
    g = data[0]
    num_nodes = g.num_nodes()
    num_rels = data.num_rels
    train_g = get_subset_g(g, g.edata["train_mask"], num_rels)
    test_g = get_subset_g(g, g.edata["train_mask"], num_rels, bidirected=True)
    test_g.edata["norm"] = dgl.norm_by_dst(test_g).unsqueeze(-1)
    test_nids = torch.arange(0, num_nodes)
    test_mask = g.edata["test_mask"]
    subg_iter = SubgraphIterator(train_g, num_rels)  # uniform edge sampling
    dataloader = GraphDataLoader(
        subg_iter, batch_size=1, collate_fn=lambda x: x[0]
```

```python
    )

    # Prepare data for metric computation
    src, dst = g.edges()
    triplets = torch.stack([src, g.edata["etype"], dst], dim=1)

    # create RGCN model
    model = LinkPredict(num_nodes, num_rels).to(device)

    # train
    model_state_file = "model_state.pth"
    train(
        dataloader,
        test_g,
        test_nids,
        test_mask,
        triplets,
        device,
        model_state_file,
        model,
    )

    # testing
    print("Testing...")
    checkpoint = torch.load(model_state_file)
    model = model.cpu()  # test on CPU
    model.eval()
    model.load_state_dict(checkpoint["state_dict"])
    embed = model(test_g, test_nids)
    best_mrr = calc_mrr(
        embed, model.w_relation, test_mask, triplets, batch_size=500
    )
    print(
        "Best MRR {:.4f} achieved using the epoch {:04d}".format(
            best_mrr, checkpoint["epoch"]
        )
    )
```