GAT

File Edit View Insert Runtime Tools Help All changes saved

Comment Share a

Files

+ Code + Text

RAM Disk Editing

```
..
sample_data
  README.md
  anscombe.json
  california_housing_test.csv
  california_housing_train.csv
  cora.cites
  cora.content
  mnist_test.csv
  mnist_train_small.csv
617.pkl
717.pkl
Digraph.gv
Digraph.gv.pdf
```

Disk 55.18 GB available

```
import numpy as np
import scipy.sparse as sp
import torch
```

```
[2] def encode_onehot(labels):
        # The classes must be sorted before encoding to enable static class encoding.
        # In other words, make sure the first class always maps to index 0.
        classes = sorted(list(set(labels)))
        classes_dict = {c: np.identity(len(classes))[i, :] for i, c in enumerate(classes)}
        labels_onehot = np.array(list(map(classes_dict.get, labels)), dtype=np.int32)
        return labels_onehot
```

```
[3] def load_data(path="/content/sample_data/cora", dataset="cora"):
        """Load citation network dataset (cora only for now)"""
        print('Loading {} dataset...'.format(dataset))

        idx_features_labels = np.genfromtxt("{}.content".format(path, dataset), dtype=np.dtype(str))
        features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
        labels = encode_onehot(idx_features_labels[:, -1])

        # build graph
        idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
        idx_map = {j: i for i, j in enumerate(idx)}
        edges_unordered = np.genfromtxt("{}.cites".format(path, dataset), dtype=np.int32)
        edges = np.array(list(map(idx_map.get, edges_unordered.flatten())), dtype=np.int32).reshape(edges_unordered.shape)
        adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])), shape=(labels.shape[0], labels.shape[0]), dtype=np.float32)

        # build symmetric adjacency matrix
        adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)

        features = normalize_features(features)
        adj = normalize_adj(adj + sp.eye(adj.shape[0]))

        idx_train = range(140)
        idx_val = range(200, 500)
        idx_test = range(500, 1500)

        adj = torch.FloatTensor(np.array(adj.todense()))
        features = torch.FloatTensor(np.array(features.todense()))
        labels = torch.LongTensor(np.where(labels)[1])

        idx_train = torch.LongTensor(idx_train)
        idx_val = torch.LongTensor(idx_val)
        idx_test = torch.LongTensor(idx_test)

        return adj, features, labels, idx_train, idx_val, idx_test
```

```
[4] def normalize_adj(mx):
        """Row-normalize sparse matrix"""
        rowsum = np.array(mx.sum(1))
        r_inv_sqrt = np.power(rowsum, -0.5).flatten()
        r_inv_sqrt[np.isinf(r_inv_sqrt)] = 0.
        r_mat_inv_sqrt = sp.diags(r_inv_sqrt)
        return mx.dot(r_mat_inv_sqrt).transpose().dot(r_mat_inv_sqrt)
```

```
[5] def normalize_features(mx):
        """Row-normalize sparse matrix"""
        rowsum = np.array(mx.sum(1))
        r_inv = np.power(rowsum, -1).flatten()
        r_inv[np.isinf(r_inv)] = 0.
        r_mat_inv = sp.diags(r_inv)
        mx = r_mat_inv.dot(mx)
        return mx
```

```
[6] def accuracy(output, labels):
        preds = output.max(1)[1].type_as(labels)
        correct = preds.eq(labels).double()
        correct = correct.sum()
        return correct / len(labels)
```

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

```
[8] class GraphAttentionLayer(nn.Module):
        """
        Simple GAT layer, similar to https://arxiv.org/abs/1710.10903
        """
        def __init__(self, in_features, out_features, dropout, alpha, concat=True):
            super(GraphAttentionLayer, self).__init__()
            self.dropout = dropout
            self.in_features = in_features
            self.out_features = out_features
            self.alpha = alpha
            self.concat = concat

            self.W = nn.Parameter(torch.empty(size=(in_features, out_features)))
            nn.init.xavier_uniform_(self.W.data, gain=1.414)
            self.a = nn.Parameter(torch.empty(size=(2*out_features, 1)))
            nn.init.xavier_uniform_(self.a.data, gain=1.414)

            self.leakyrelu = nn.LeakyReLU(self.alpha)

        def forward(self, h, adj):
            Wh = torch.mm(h, self.W) # h.shape: (N, in_features), Wh.shape: (N, out_features)
            e = self._prepare_attentional_mechanism_input(Wh)

            zero_vec = -9e15*torch.ones_like(e)
            attention = torch.where(adj > 0, e, zero_vec)
            attention = F.softmax(attention, dim=1)
            attention = F.dropout(attention, self.dropout, training=self.training)
            h_prime = torch.matmul(attention, Wh)

            if self.concat:
```

```
            return F.elu(h_prime)
        else:
            return h_prime

    def _prepare_attentional_mechanism_input(self, Wh):
        # Wh.shape (N, out_feature)
        # self.a.shape (2 * out_feature, 1)
        # Wh1&2.shape (N, 1)
        # e.shape (N, N)
        Wh1 = torch.matmul(Wh, self.a[:self.out_features, :])
        Wh2 = torch.matmul(Wh, self.a[self.out_features:, :])
        # broadcast add
        e = Wh1 + Wh2.T
        return self.leakyrelu(e)

    def __repr__(self):
        return self.__class__.__name__ + ' (' + str(self.in_features) + ' -> ' + str(self.out_features) + ')'
```

```
[9] class SpecialSpmmFunction(torch.autograd.Function):
        """Special function for only sparse region backpropataion layer."""
        @staticmethod
        def forward(ctx, indices, values, shape, b):
            assert indices.requires_grad == False
            a = torch.sparse_coo_tensor(indices, values, shape)
            ctx.save_for_backward(a, b)
            ctx.N = shape[0]
            return torch.matmul(a, b)

        @staticmethod
        def backward(ctx, grad_output):
            a, b = ctx.saved_tensors
            grad_values = grad_b = None
            if ctx.needs_input_grad[1]:
                grad_a_dense = grad_output.matmul(b.t())
                edge_idx = a._indices()[0, :] * ctx.N + a._indices()[1, :]
                grad_values = grad_a_dense.view(-1)[edge_idx]
            if ctx.needs_input_grad[3]:
                grad_b = a.t().matmul(grad_output)
            return None, grad_values, None, grad_b
```

```
[10] class SpecialSpmm(nn.Module):
        def forward(self, indices, values, shape, b):
            return SpecialSpmmFunction.apply(indices, values, shape, b)
```

```
[11] class SpGraphAttentionLayer(nn.Module):
        """
        Sparse version GAT layer, similar to https://arxiv.org/abs/1710.10903
        """

        def __init__(self, in_features, out_features, dropout, alpha, concat=True):
            super(SpGraphAttentionLayer, self).__init__()
            self.in_features = in_features
            self.out_features = out_features
            self.alpha = alpha
            self.concat = concat

            self.W = nn.Parameter(torch.zeros(size=(in_features, out_features)))
            nn.init.xavier_normal_(self.W.data, gain=1.414)

            self.a = nn.Parameter(torch.zeros(size=(1, 2*out_features)))
            nn.init.xavier_normal_(self.a.data, gain=1.414)

            self.dropout = nn.Dropout(dropout)
            self.leakyrelu = nn.LeakyReLU(self.alpha)
            self.special_spmm = SpecialSpmm()

        def forward(self, input, adj):
            dv = 'cuda' if input.is_cuda else 'cpu'

            N = input.size()[0]
            edge = adj.nonzero().t()

            h = torch.mm(input, self.W)
            # h: N x out
            assert not torch.isnan(h).any()

            # Self-attention on the nodes - Shared attention mechanism
            edge_h = torch.cat((h[edge[0, :], :], h[edge[1, :], :]), dim=1).t()
            # edge: 2*D x E

            edge_e = torch.exp(-self.leakyrelu(self.a.mm(edge_h).squeeze()))
            assert not torch.isnan(edge_e).any()
            # edge_e: E

            e_rowsum = self.special_spmm(edge, edge_e, torch.Size([N, N]), torch.ones(size=(N,1), device=dv))
            # e_rowsum: N x 1

            edge_e = self.dropout(edge_e)
            # edge_e: E

            h_prime = self.special_spmm(edge, edge_e, torch.Size([N, N]), h)
            assert not torch.isnan(h_prime).any()
            # h_prime: N x out

            h_prime = h_prime.div(e_rowsum)
            # h_prime: N x out
            assert not torch.isnan(h_prime).any()

            if self.concat:
                # if this layer is not last layer,
                return F.elu(h_prime)
            else:
                # if this layer is last layer,
                return h_prime

        def __repr__(self):
            return self.__class__.__name__ + ' (' + str(self.in_features) + ' -> ' + str(self.out_features) + ')'
```

```
[12] import torch
     import torch.nn as nn
     import torch.nn.functional as F
```

```
[13] class GAT(nn.Module):
        def __init__(self, nfeat, nhid, nclass, dropout, alpha, nheads):
            """Dense version of GAT."""
```

```
            super(GAT, self).__init__()
            self.dropout = dropout

            self.attentions = [GraphAttentionLayer(nfeat, nhid, dropout=dropout, alpha=alpha, concat=True) for _ in range(nheads)]
            for i, attention in enumerate(self.attentions):
                self.add_module('attention_{}'.format(i), attention)

            self.out_att = GraphAttentionLayer(nhid * nheads, nclass, dropout=dropout, alpha=alpha, concat=False)

        def forward(self, x, adj):
            x = F.dropout(x, self.dropout, training=self.training)
            x = torch.cat([att(x, adj) for att in self.attentions], dim=1)
            x = F.dropout(x, self.dropout, training=self.training)
            x = F.elu(self.out_att(x, adj))
            return F.log_softmax(x, dim=1)
```

```
[14] class SpGAT(nn.Module):
        def __init__(self, nfeat, nhid, nclass, dropout, alpha, nheads):
            """Sparse version of GAT."""
            super(SpGAT, self).__init__()
            self.dropout = dropout

            self.attentions = [SpGraphAttentionLayer(nfeat,
                                                     nhid,
                                                     dropout=dropout,
                                                     alpha=alpha,
                                                     concat=True) for _ in range(nheads)]
            for i, attention in enumerate(self.attentions):
                self.add_module('attention_{}'.format(i), attention)

            self.out_att = SpGraphAttentionLayer(nhid * nheads,
                                                 nclass,
                                                 dropout=dropout,
                                                 alpha=alpha,
                                                 concat=False)

        def forward(self, x, adj):
            x = F.dropout(x, self.dropout, training=self.training)
            x = torch.cat([att(x, adj) for att in self.attentions], dim=1)
            x = F.dropout(x, self.dropout, training=self.training)
            x = F.elu(self.out_att(x, adj))
            return F.log_softmax(x, dim=1)
```

```
[15] from __future__ import division
     from __future__ import print_function

     import os
     import glob
     import time
     import random
     import argparse
     import numpy as np
     import torch
     import torch.nn as nn
     import torch.nn.functional as F
     import torch.optim as optim
     from torch.autograd import Variable
```

```
[16] parser = argparse.ArgumentParser()
```

```
[17] parser.add_argument('--no-cuda', action='store_true', default=False, help='Disables CUDA training.')
     parser.add_argument('--fastmode', action='store_true', default=False, help='Validate during training pass.')
     parser.add_argument('--sparse', action='store_true', default=False, help='GAT with sparse version or not.')
     parser.add_argument('--seed', type=int, default=72, help='Random seed.')
     parser.add_argument('--epochs', type=int, default=10000, help='Number of epochs to train.')
     parser.add_argument('--lr', type=float, default=0.005, help='Initial learning rate.')
     parser.add_argument('--weight_decay', type=float, default=5e-4, help='Weight decay (L2 loss on parameters).')
     parser.add_argument('--hidden', type=int, default=8, help='Number of hidden units.')
     parser.add_argument('--nb_heads', type=int, default=8, help='Number of head attentions.')
     parser.add_argument('--dropout', type=float, default=0.6, help='Dropout rate (1 - keep probability).')
     parser.add_argument('--alpha', type=float, default=0.2, help='Alpha for the leaky_relu.')
     parser.add_argument('--patience', type=int, default=100, help='Patience')
     parser.add_argument('-f')

     _StoreAction(option_strings=['-f'], dest='f', nargs=None, const=None, default=None, type=None, choices=None, help=None, metavar=None)
```

```
[18] args = parser.parse_args()
     args.cuda = not args.no_cuda and torch.cuda.is_available()
```

```
[19] random.seed(args.seed)
     np.random.seed(args.seed)
     torch.manual_seed(args.seed)
     if args.cuda:
         torch.cuda.manual_seed(args.seed)
```

```
[20] adj, features, labels, idx_train, idx_val, idx_test = load_data()

     Loading cora dataset...
```

```
[21] if args.sparse:
         model = SpGAT(nfeat=features.shape[1],
                       nhid=args.hidden,
                       nclass=int(labels.max()) + 1,
                       dropout=args.dropout,
                       nheads=args.nb_heads,
                       alpha=args.alpha)
     else:
         model = GAT(nfeat=features.shape[1],
                     nhid=args.hidden,
                     nclass=int(labels.max()) + 1,
                     dropout=args.dropout,
                     nheads=args.nb_heads,
                     alpha=args.alpha)
     optimizer = optim.Adam(model.parameters(),
                            lr=args.lr,
                            weight_decay=args.weight_decay)
```

```
[22] if args.cuda:
         model.cuda()
         features = features.cuda()
         adj = adj.cuda()
         labels = labels.cuda()
```

```
        idx_train = idx_train.cuda()
        idx_val = idx_val.cuda()
        idx_test = idx_test.cuda()

    features, adj, labels = Variable(features), Variable(adj), Variable(labels)
```

```
[23] def train(epoch):
        t = time.time()
        model.train()
        optimizer.zero_grad()
        output = model(features, adj)
        loss_train = F.nll_loss(output[idx_train], labels[idx_train])
        acc_train = accuracy(output[idx_train], labels[idx_train])
        loss_train.backward()
        optimizer.step()

        if not args.fastmode:
            # Evaluate validation set performance separately,
            # deactivates dropout during validation run.
            model.eval()
            output = model(features, adj)

        loss_val = F.nll_loss(output[idx_val], labels[idx_val])
        acc_val = accuracy(output[idx_val], labels[idx_val])
        print('Epoch: {:04d}'.format(epoch+1),
              'loss_train: {:.4f}'.format(loss_train.data.item()),
              'acc_train: {:.4f}'.format(acc_train.data.item()),
              'loss_val: {:.4f}'.format(loss_val.data.item()),
              'acc_val: {:.4f}'.format(acc_val.data.item()),
              'time: {:.4f}s'.format(time.time() - t))

        return loss_val.data.item()
```

```
[24] def compute_test():
        model.eval()
        output = model(features, adj)
        loss_test = F.nll_loss(output[idx_test], labels[idx_test])
        acc_test = accuracy(output[idx_test], labels[idx_test])
        print("Test set results:",
              "loss= {:.4f}".format(loss_test.data.item()),
              "accuracy= {:.4f}".format(acc_test.data.item()))
```

```
[25] # Train model
    t_total = time.time()
    loss_values = []
    bad_counter = 0
    best = args.epochs + 1
    best_epoch = 0
    for epoch in range(args.epochs):
        loss_values.append(train(epoch))

        torch.save(model.state_dict(), '{}.pkl'.format(epoch))
        if loss_values[-1] < best:
            best = loss_values[-1]
            best_epoch = epoch
            bad_counter = 0
        else:
            bad_counter += 1

        if bad_counter == args.patience:
            break

        files = glob.glob('*.pkl')
        for file in files:
            epoch_nb = int(file.split('.')[0])
            if epoch_nb < best_epoch:
                os.remove(file)

        files = glob.glob('*.pkl')
        for file in files:
            epoch_nb = int(file.split('.')[0])
            if epoch_nb > best_epoch:
                os.remove(file)
```

```
Epoch: 0661 loss_train: 0.5724 acc_train: 0.8143 loss_val: 0.6579 acc_val: 0.8100 time: 0.0737s
Epoch: 0662 loss_train: 0.5605 acc_train: 0.8357 loss_val: 0.6581 acc_val: 0.8100 time: 0.0737s
Epoch: 0663 loss_train: 0.5759 acc_train: 0.8429 loss_val: 0.6582 acc_val: 0.8100 time: 0.0737s
Epoch: 0664 loss_train: 0.6254 acc_train: 0.8214 loss_val: 0.6583 acc_val: 0.8100 time: 0.0737s
Epoch: 0665 loss_train: 0.5461 acc_train: 0.8571 loss_val: 0.6583 acc_val: 0.8100 time: 0.0738s
Epoch: 0666 loss_train: 0.5912 acc_train: 0.8357 loss_val: 0.6580 acc_val: 0.8100 time: 0.0738s
Epoch: 0667 loss_train: 0.5772 acc_train: 0.8357 loss_val: 0.6579 acc_val: 0.8100 time: 0.0738s
Epoch: 0668 loss_train: 0.5881 acc_train: 0.8143 loss_val: 0.6576 acc_val: 0.8100 time: 0.0737s
Epoch: 0669 loss_train: 0.5910 acc_train: 0.8143 loss_val: 0.6572 acc_val: 0.8100 time: 0.0736s
Epoch: 0670 loss_train: 0.5294 acc_train: 0.8571 loss_val: 0.6568 acc_val: 0.8200 time: 0.0736s
Epoch: 0671 loss_train: 0.6458 acc_train: 0.7643 loss_val: 0.6563 acc_val: 0.8200 time: 0.0737s
Epoch: 0672 loss_train: 0.5741 acc_train: 0.8571 loss_val: 0.6563 acc_val: 0.8200 time: 0.0737s
Epoch: 0673 loss_train: 0.5568 acc_train: 0.8286 loss_val: 0.6566 acc_val: 0.8200 time: 0.0737s
Epoch: 0674 loss_train: 0.6350 acc_train: 0.7929 loss_val: 0.6567 acc_val: 0.8200 time: 0.0738s
Epoch: 0675 loss_train: 0.5922 acc_train: 0.8429 loss_val: 0.6569 acc_val: 0.8233 time: 0.0738s
Epoch: 0676 loss_train: 0.5214 acc_train: 0.8500 loss_val: 0.6571 acc_val: 0.8233 time: 0.0738s
Epoch: 0677 loss_train: 0.5714 acc_train: 0.8643 loss_val: 0.6572 acc_val: 0.8233 time: 0.0737s
Epoch: 0678 loss_train: 0.6063 acc_train: 0.8000 loss_val: 0.6574 acc_val: 0.8233 time: 0.0737s
Epoch: 0679 loss_train: 0.6239 acc_train: 0.8071 loss_val: 0.6577 acc_val: 0.8267 time: 0.0737s
Epoch: 0680 loss_train: 0.5992 acc_train: 0.8500 loss_val: 0.6580 acc_val: 0.8267 time: 0.0737s
Epoch: 0681 loss_train: 0.6340 acc_train: 0.7929 loss_val: 0.6583 acc_val: 0.8267 time: 0.0737s
Epoch: 0682 loss_train: 0.5761 acc_train: 0.8500 loss_val: 0.6586 acc_val: 0.8267 time: 0.0737s
Epoch: 0683 loss_train: 0.5393 acc_train: 0.8429 loss_val: 0.6589 acc_val: 0.8300 time: 0.0738s
Epoch: 0684 loss_train: 0.6477 acc_train: 0.8286 loss_val: 0.6590 acc_val: 0.8300 time: 0.0737s
Epoch: 0685 loss_train: 0.6285 acc_train: 0.8571 loss_val: 0.6592 acc_val: 0.8267 time: 0.0737s
Epoch: 0686 loss_train: 0.6385 acc_train: 0.8429 loss_val: 0.6589 acc_val: 0.8267 time: 0.0737s
Epoch: 0687 loss_train: 0.6081 acc_train: 0.8000 loss_val: 0.6586 acc_val: 0.8300 time: 0.0737s
Epoch: 0688 loss_train: 0.6802 acc_train: 0.7857 loss_val: 0.6580 acc_val: 0.8300 time: 0.0738s
Epoch: 0689 loss_train: 0.5380 acc_train: 0.8786 loss_val: 0.6573 acc_val: 0.8300 time: 0.0737s
Epoch: 0690 loss_train: 0.5949 acc_train: 0.8214 loss_val: 0.6566 acc_val: 0.8300 time: 0.0737s
Epoch: 0691 loss_train: 0.5502 acc_train: 0.8429 loss_val: 0.6560 acc_val: 0.8300 time: 0.0738s
Epoch: 0692 loss_train: 0.6266 acc_train: 0.8214 loss_val: 0.6555 acc_val: 0.8300 time: 0.0740s
Epoch: 0693 loss_train: 0.6444 acc_train: 0.8429 loss_val: 0.6553 acc_val: 0.8300 time: 0.0737s
Epoch: 0694 loss_train: 0.4926 acc_train: 0.8571 loss_val: 0.6547 acc_val: 0.8300 time: 0.0736s
Epoch: 0695 loss_train: 0.5167 acc_train: 0.8357 loss_val: 0.6543 acc_val: 0.8267 time: 0.0736s
Epoch: 0696 loss_train: 0.6318 acc_train: 0.8357 loss_val: 0.6541 acc_val: 0.8233 time: 0.0736s
Epoch: 0697 loss_train: 0.6173 acc_train: 0.8500 loss_val: 0.6539 acc_val: 0.8200 time: 0.0738s
Epoch: 0698 loss_train: 0.7201 acc_train: 0.7571 loss_val: 0.6537 acc_val: 0.8200 time: 0.0738s
Epoch: 0699 loss_train: 0.6611 acc_train: 0.8071 loss_val: 0.6535 acc_val: 0.8200 time: 0.0738s
Epoch: 0700 loss_train: 0.6430 acc_train: 0.8143 loss_val: 0.6532 acc_val: 0.8167 time: 0.0738s
Epoch: 0701 loss_train: 0.7380 acc_train: 0.7714 loss_val: 0.6528 acc_val: 0.8167 time: 0.0737s
Epoch: 0702 loss_train: 0.6187 acc_train: 0.7929 loss_val: 0.6523 acc_val: 0.8167 time: 0.0737s
Epoch: 0703 loss_train: 0.6422 acc_train: 0.7857 loss_val: 0.6522 acc_val: 0.8167 time: 0.0737s
Epoch: 0704 loss_train: 0.5664 acc_train: 0.8286 loss_val: 0.6521 acc_val: 0.8167 time: 0.0738s
Epoch: 0705 loss_train: 0.5104 acc_train: 0.8714 loss_val: 0.6522 acc_val: 0.8167 time: 0.0736s
Epoch: 0706 loss_train: 0.5663 acc_train: 0.8571 loss_val: 0.6520 acc_val: 0.8167 time: 0.0737s
```

```
Epoch: 0707 loss_train: 0.6045 acc_train: 0.7929 loss_val: 0.6519 acc_val: 0.8167 time: 0.0737s
Epoch: 0708 loss_train: 0.6252 acc_train: 0.8143 loss_val: 0.6519 acc_val: 0.8167 time: 0.0737s
Epoch: 0709 loss_train: 0.5904 acc_train: 0.8143 loss_val: 0.6519 acc_val: 0.8167 time: 0.0737s
Epoch: 0710 loss_train: 0.6438 acc_train: 0.8000 loss_val: 0.6522 acc_val: 0.8133 time: 0.0737s
Epoch: 0711 loss_train: 0.5788 acc_train: 0.8429 loss_val: 0.6523 acc_val: 0.8167 time: 0.0738s
Epoch: 0712 loss_train: 0.6385 acc_train: 0.8214 loss_val: 0.6523 acc_val: 0.8167 time: 0.0738s
Epoch: 0713 loss_train: 0.6275 acc_train: 0.8214 loss_val: 0.6525 acc_val: 0.8200 time: 0.0738s
Epoch: 0714 loss_train: 0.6190 acc_train: 0.8357 loss_val: 0.6529 acc_val: 0.8200 time: 0.0736s
Epoch: 0715 loss_train: 0.5708 acc_train: 0.8429 loss_val: 0.6534 acc_val: 0.8200 time: 0.0738s
Epoch: 0716 loss_train: 0.6111 acc_train: 0.7857 loss_val: 0.6538 acc_val: 0.8200 time: 0.0740s
Epoch: 0717 loss_train: 0.5201 acc_train: 0.8500 loss_val: 0.6542 acc_val: 0.8167 time: 0.0742s
Epoch: 0718 loss_train: 0.7117 acc_train: 0.7500 loss_val: 0.6547 acc_val: 0.8167 time: 0.0738s
```

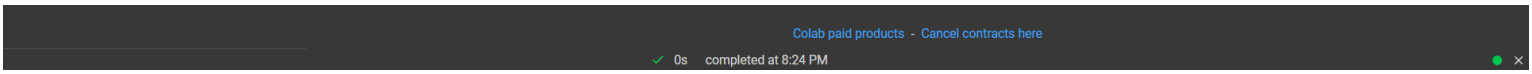
[26]
```
print("Optimization Finished!")
print("Total time elapsed: {:.4f}s".format(time.time() - t_total))

# Restore best model
print('Loading {}th epoch'.format(best_epoch))
model.load_state_dict(torch.load('{}.pkl'.format(best_epoch)))

# Testing
compute_test()
```

```
Optimization Finished!
Total time elapsed: 59.5648s
Loading 617th epoch
Test set results: loss= 0.6626 accuracy= 0.8410
```

[27]
```
from graphviz import Digraph

import torch
# import models

def make_dot(var, params):
    """ Produces Graphviz representation of PyTorch autograd graph

    Blue nodes are the Variables that require grad, orange are Tensors
    saved for backward in torch.autograd.Function

    Args:
        var: output Variable
        params: dict of (name, Variable) to add names to node that
            require grad (TODO: make optional)
    """
    param_map = {id(v): k for k, v in params.items()}
    print(param_map)

    node_attr = dict(style='filled',
                     shape='box',
                     align='left',
                     fontsize='12',
                     ranksep='0.1',
                     height='0.2')
    dot = Digraph(node_attr=node_attr, graph_attr=dict(size="12,12"))
    seen = set()

    def size_to_str(size):
        return '('+(', '.join(['%d'% v for v in size])+')'

    def add_nodes(var):
        if var not in seen:
            if torch.is_tensor(var):
                dot.node(str(id(var)), size_to_str(var.size()), fillcolor='orange')
            elif hasattr(var, 'variable'):
                u = var.variable
                node_name = '%s\n %s' % (param_map.get(id(u)), size_to_str(u.size()))
                dot.node(str(id(var)), node_name, fillcolor='lightblue')
            else:
                dot.node(str(id(var)), str(type(var).__name__))
            seen.add(var)
            if hasattr(var, 'next_functions'):
                for u in var.next_functions:
                    if u[0] is not None:
                        dot.edge(str(id(u[0])), str(id(var)))
                        add_nodes(u[0])
            if hasattr(var, 'saved_tensors'):
                for t in var.saved_tensors:
                    dot.edge(str(id(t)), str(id(var)))
                    add_nodes(t)
    add_nodes(var.grad_fn)
    return dot
```

[28]
```
inputs = torch.randn(100, 50).cuda()
adj = torch.randn(100, 100).cuda()
model = SpGAT(50, 8, 7, 0.5, 0.01, 3)
model = model.cuda()
y = model(inputs, adj)

g = make_dot(y, model.state_dict())
g.view()
```

```
{140600975158944: 'attention_0.W', 140600975158624: 'attention_0.a', 140600975158784: 'attention_1.W', 140600975159024: 'attention_1.a', 140600975157424: 'attention_2.W
'Digraph.gv.pdf'
```


Colab paid products - Cancel contracts here
0s completed at 8:24 PM