8.23-9.5 周报

赵晓辉

2021 年 9 月 5 日

1 写在前面

这个双周主要完成了 cs231n 课程的 Lecture 1 至 Lecture 6，主要内容包括距离函数、KNN、SVM、损失函数及优化、BP 算法、CNN 架构、非线性激活函数以及神经网络的参数优化等，并完成 cs231n assignment1。课程概要笔记及 assignment 将在https://github.com/zxh991103/cs231NOTE持续跟踪。此外使用 torch，初步学习和实现了 GAT 算法。

2 Lec 1-6 课程概要

2.1 距离函数

L_1 Distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L_2 Distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

2.2 KNN

计算测试样本与所有训练集样本之间的距离值，并根据 K 值投票选举出最相似的标签。

2.3 SVM

计算能够划分训练集样本且距离最大的超平面。

$$w \cdot x + b = 0$$

2.4 损失函数

损失函数评估模型预测值与模型真实值之间的差异性，我们要将其最小化。对于给定的训练集 $(x_i, y_i)_{i=1}^{N}$，我们有损失函数：

$$L = \frac{1}{N}\sum_i L_i(f(x_i, W), y_i)$$

对于 Multi-SVM，我们有损失函数，即 hinge loss：

$$s = f(x_i, W)$$

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

对于 softmax loss:

$$L_i = -log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

2.5 正则化

根据奥卡姆剃刀原则，模型越简单越符合实际，所以我们将正则惩罚项加在损失函数上。

$$L = \frac{1}{N}\sum_i L_i(f(x_i, W), y_i) + R(W)$$

L1

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

L2

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Elastic

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

2.6 BP 算法

链式法则：

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

故在计算损失函数对于参数的梯度值时，我们应当将本地梯度值与上游回传梯度值相乘。

此时，我们也可以发现 Relu 函数，即 max gate 中只有前向传播计算中的正值能影响下游。

A vectorized example：

$$f(x,W) = ||W\ x||^2 = \sum_{i=1}^{n}(W\ x)_i^2$$

$$q = W \cdot x$$

$$\nabla_W f = 2q \cdot x^T$$

2.7 NN

假设我们将神经网络的计算图表示为：

$$f = Softmax(W_2 Relu(W_1 x))$$

神经元 A 拥有 W_1，能够具有识别出来 100 种特征的功能，比如识别出马的左脸或者右脸、车头或者车位。而神经元 B 拥有 W_2, 其功能就在于将马的左脸或右脸合并为马的特征，将车头或车尾合并成车的特征，从而进行识别。

2.8 CNN

卷积层：

假设有 32*32*3 的图片，卷积核 w 5*5*3 ，以及偏置 b，卷积后我们获得 28*28*1 的矩阵，其中 1 时卷积核的数量。卷积公式为（每位相乘再求和）：

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty}\sum_{n_2=-\infty}^{\infty} f[x,y] * g[x-n_1, y-n_2]$$

步长（stride）：

假设我们有 7*7 的输入，3*3 的卷积核，2 的步长，最后的输出为 3*3。此时 outputsize = $\frac{(N-F)}{stride} + 1$

填充（Pad）：

图像四周补充 0，来防止在深层卷积时张量过小。此时，outputsize = $\frac{(N-F+2P)}{stride} + 1$。

Example:

input volume 32*32*3,10 5*5 filters (include 3 depth), stride 1 ,pad 2 ,wo have 760 parameters (10 *(5 * 5 *3 +1 bias)=760)

pooling layer: 相当于下采样。

maxpooling:

一般，每一个池化 filter 具有和步长相同的大小以避免 overlap.

例如，

$$\begin{matrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{matrix}$$

我们使用 2*2 的 filter 和 2 的 stride,maxpooling 后变为：

$$\begin{matrix} 6 & 8 \\ 3 & 4 \end{matrix}$$

2.9 激活函数

若全部线性连接则等同于一个线性连接，所以网络中需要非线性的激活函数变换。

sigmod

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

tanh

$$tanh(x)$$

Relu

$$max(0, x)$$

LeakyRelu

$$max(0.1x, x)$$

Maxout

$$\max\left(w_1^T x + b_1, w_2^T x + b_2\right)$$

Elu

$$\begin{cases} x & x \geq 0 \\ \alpha\left(e^x - 1\right) & x < 0 \end{cases}$$

SELU

$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha\left(e^x - 1\right) & \text{otherwise} \end{cases}$$

softmax 的问题：

- x 过大或过小，本地梯度接近 0，使得与上游梯度乘积也接近于 0，更新缓慢。
- x 的值恒正或恒负，本地梯度总是大于 0 的，造成 w 的移动时锯齿状的，接近最优点放缓。

relu 的问题：

- 若 $w \cdot x + b$ 总是负的，则本地梯度为 0，造成参数不更新。

2.10 数据处理

```
# ZERO-CENTER
X -= np.mean(X,axis = 0)
# normalize
X /= np.std(X,axis = 0)
```

Listing 1: normalization

2.11 参数初始化

Naive: 为参数初始化小随机数。但是随着网络深度的增加，本地梯度与上游梯度相乘之后接近零，学习十分缓慢。

Xavier：

```
W = np.random.randn(dim_in,dim_out)/np.sqrt(dim_in)
```

Listing 2: Xavier

原因：

we want Var(y) = Var(x_i) , and we have

$$y = \sum_{i=1}^{Din} x_i w_i$$

and we assume that every x has same var. so we have

$$var(y) = Din \times var(x) \times var(w_i)$$

and obviously initial w_i $N(0, 1)$, we make $\frac{w_i}{\sqrt{Din}}$ to achieve the var is $\frac{1}{Din}$

Kaiming/MSRA:

```
    W = np.random.randn(dim_in,dim_out)*np.sqrt(2/dim_in)
```

Listing 3: MSRA

2.12 Batch Normalization

we have the input x like N×D

$$\widehat{x}^{(k)}=\frac{x^{(k)}-\mathrm{E}\left[x^{(k)}\right]}{\sqrt{\operatorname{Var}\left[x^{(k)}\right]}}$$

so that we have:

$$\mu_j=\frac{1}{N}\sum_{i=1}^{N}x_{i,j}$$

$$\sigma_j^2=\frac{1}{N}\sum_{i=1}^{N}(x_{i,j}-\mu_j)^2$$

$$\hat{x}_{i,j}=\frac{x_{i,j}-\mu_j}{\sqrt{\sigma_j^2+\varepsilon}}$$

the net is supposed to learn $\gamma \in R^D$ and $\beta \in R^D$

$$y_{i,j}=\gamma_j\hat{x}_{i,j}+\beta_j$$

BN 层经常用于全连接或卷积层后。

2.13 Norm For Conv.

batch norm , 对于每一个 batch 中的每个 channel 取平均得到 (1× 1 × C)

layer norm , 对于所有的 batch 我们取一个平均的图片 (H × W× C)

instance norm , 对于所有的 batch 我们将取一个平均的单通道图片 (H × W ×1)

group norm , 对于所有 batch，我们将 channel 分为 k 个组，在每个组山上取平均值得到 (H × W × k)

3 Assignment1

3.1 KNN

双循环实现距离矩阵:

```
num_test = X.shape[0]
num_train = self.X_train.shape[0]
dists = np.zeros((num_test, num_train))
for i in range(num_test):
    for j in range(num_train):
        t1 = X[i]
```

```
            t2 = self.X_train[j]
            t = t1 - t2
            t = t*t
            t = t.sum()
            t = t**0.5
            dists[i][j] = t
```

Listing 4: KNN 双循环实现距离矩阵

单循环实现距离矩阵:

```
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        dists[i,:] = np.sum((X[i,:]-self.X_train)**2,axis = 1)
```

Listing 5: KNN 单循环实现距离矩阵

向量操作实现距离矩阵（矩阵下的完全平方公式）:

```
    M = np.dot(X, self.X_train.T)
    nrow=M.shape[0]
    ncol=M.shape[1]
    te = np.diag(np.dot(X,X.T))
    tr = np.diag(np.dot(self.X_train,self.X_train.T))
    te= np.reshape(np.repeat(te,ncol),M.shape)
    tr = np.reshape(np.repeat(tr, nrow), M.T.shape)
    sq=-2 * M +te+tr.T
    dists = np.sqrt(sq)
```

Listing 6: KNN 向量操作实现距离矩阵

KNN 预测:

```
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    labelnum = len(np.unique(self.y_train))
    for i in range(num_test):
        # A list of length k storing the labels of the k nearest neighbors
    to
        # the ith test point.
        closest_y = []
        closest_y = np.argsort(dists[i,:])[:k]
        t = np.zeros(labelnum,dtype=int)
        for j in closest_y:
            t[self.y_train[j]]+=1
        y_pred[i] = t.argmax()
```

Listing 7: KNN 预测

3.2 SVM

计算 SVM loss、dW(naive):

```
dW = np.zeros(W.shape)  # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
    scores = X[i].dot(W)
    correct_class_score = scores[y[i]]
    for j in range(num_classes):
        if j == y[i]:
            continue
        margin = scores[j] - correct_class_score + 1  # note delta = 1
        if margin > 0:
            loss += margin
loss /= num_train

# Add regularization to the loss.
loss += reg * np.sum(W * W)

for i in range(num_train):
    scores = X[i].dot(W)
    correct_class_score = scores[y[i]]
    for j in range(num_classes):
        if j == y[i]:
            continue
        margin = scores[j] - correct_class_score + 1  # note delta = 1
        if margin > 0:
            dW[:,j] += X[i]
            dW[:,y[i]] -= X[i]
dW /= num_train
dW += reg * W * 2
```

Listing 8: SVM(naive)

计算 SVM loss、dW(vectorized):

```
loss = 0.0
dW = np.zeros(W.shape)  # initialize the gradient as zero
N = X.shape[0]
scores = X.dot(W)
score_yi = scores[range(N),y].reshape(-1,1)
t = scores - score_yi + 1
t[range(N),y] = 0
condition = (t>0).astype(int)
t = condition*t
t = t.sum() / N
loss = t + 2 * reg * np.sum(W * W)
```

```

condition[range(N), y] = - np.sum(condition, axis = 1)
dW += np.dot(X.T,condition)/N + 2 * reg * W
```

Listing 9: SVM(vectorized)

训练线性分类器（batch）：

```
num_train, dim = X.shape
num_classes = (
    np.max(y) + 1
)  # assume y takes values 0...K-1 where K is number of classes
if self.W is None:
    # lazily initialize W
    self.W = 0.001 * np.random.randn(dim, num_classes)

# Run stochastic gradient descent to optimize W
loss_history = []
for it in range(num_iters):
    X_batch = None
    y_batch = None
    indices = np.random.choice(num_train,batch_size)
    X_batch = X[indices]
    y_batch = y[indices]

    # evaluate loss and gradient
    loss, grad = self.loss(X_batch, y_batch, reg)
    loss_history.append(loss)

    # perform parameter update
    self.W -= learning_rate*grad
```

Listing 10: 训练线性分类器

SVM 预测：

```
y_pred = np.argmax(np.dot(X,self.W),axis = 1)
```

Listing 11: SVM 预测

SVM grid search：

```
for i in learning_rates:
for j in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train, y_train, learning_rate=i, reg=j,
                  num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    y_val_pred = svm.predict(X_val)
    y_train_acc = np.mean(y_train == y_train_pred)
    y_val_acc = np.mean(y_val == y_val_pred)
    results[(i,j)] = (y_train_acc,y_val_acc)
    if y_val_acc > best_val:
```

```
            best_val = y_val_acc
            best_svm = svm
```

Listing 12: SVM grid search

3.3 softmax

Softmax loss、dW(Naive):

```
loss = 0.0
dW = np.zeros_like(W)

N , D = X.shape
C = W.shape[1]

for i in range(N):
    f = np.dot(X[i],W)
    f -= np.max(f) # avoid overflow
    loss = loss + np.log(np.sum(np.exp(f))) - f[y[i]]
    dW[:, y[i]] -= X[i]
    s = np.exp(f).sum()
    for j in range(C):
        dW[:, j] += np.exp(f[j]) / s * X[i]
loss = loss / N +  reg * np.sum(W * W)
dW = dW / N + 2*reg * W
```

Listing 13: Softmax(Naive)

Softmax loss、dW(vectorized):

```
N , D = X.shape
C = W.shape[1]
score = np.dot(X,W)
t = np.max(score,axis = 1).reshape(N,1)
score -= t
s = np.exp(score).sum(axis = 1)
loss = - score[range(N),y].sum() + np.log(s).sum()
counts = np.exp(score) / s.reshape(N, 1)
counts[range(N),y] -= 1
dW = np.dot(X.T,counts)

loss = loss/N + reg * np.sum(W * W)

dW = dW/N + reg *W
```

Listing 14: Softmax(vectorized)

$$Loss = -\log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}) = -s_{y_i} + \log\sum_j e^{s_j}$$

$$\frac{dL}{dW} = -\frac{ds_{y_i}}{dW} + \frac{1}{\sum_j e^{s_j}} \cdot (\sum_j (e^{s_j}\frac{dS_j}{dW}))$$

softmax grid search:

```
from cs231n.classifiers.linear_classifier import Softmax

for lr in learning_rates:
for rs in regularization_strengths:
    softmax = Softmax()
    softmax.train(X_train, y_train, learning_rate = lr, reg=rs,
 num_iters = 1500,
                    verbose = True)

    y_pred_train = softmax.predict(X_train)
    acc_train = np.mean(y_pred_train == y_train)

    y_pred_val = softmax.predict(X_val)
    acc_val = np.mean(y_pred_val == y_val)
    results[(lr, rs)] = (acc_train, acc_val)

    if acc_val > best_val:
        best_val = acc_val
        best_softmax = softmax
```

Listing 15: softmax grid search

3.4 FC Net

FC 单层向后传播:

```
    N = x.shape[0]
    D = x[0].reshape(-1,1).shape[0]
    M = b.shape[0]
#     print(N,D,M)
    out = np.zeros((N,M))
    for i in range(N):
        t = x[i].reshape(-1,1)
#         print(w.shape,t.shape)
        t = np.dot(w.T,t) + b.reshape(-1,1)
        out[i:,] = t.T
```

Listing 16: FC 单层前向传播

FC 单层向后传播:

```
    N = x.shape[0]
    t = x.shape
    D = x[0].reshape(-1,1).shape[0]
    M = b.shape[0]

    x = x.reshape(N,D)
    dx = np.dot(dout,w.T).reshape(t)
    dw = np.dot(x.T,dout)
```

```
    db = dout.sum(axis=0)
```

Listing 17: FC 单层向后传播

Relu 向前传播:

```
    condition = (x>0).astype(int)
    out = condition * x
```

Listing 18: Relu 向前传播

Relu 向后传播:

```
    dx = (x > 0 ).astype(int) * dout
```

Listing 19: Relu 向后传播

SVM loss、dx

```
    N,C = x.shape
    scores = x
    score_yi = scores[range(N),y].reshape(-1,1)

    t = scores - score_yi + 1
    t[range(N),y] = 0
    condition = (t>0).astype(int)
    t = condition*t
    t = t.sum() / N
    loss = t

    condition[range(N), y] = - np.sum(condition, axis = 1)
#     print(condition.shape)
    dx = condition/N
```

Listing 20: SVM loss、dx

Softmax loss、dx

```
    N,C = x.shape
    score = x - np.max(x,axis = 1).reshape(N,1)
    s = np.exp(score).sum(axis = 1).reshape(N,1)
    score_yi = score[range(N),y].reshape(N,1)
    loss = (-score_yi + np.log(s)).sum() /N


    expscore = np.exp(score)

    dx = (expscore / s).reshape(N,C)

    dx[range(N),y] -=1

    dx /= N
```

Listing 21: Softmax loss、dx

Two Layer Net Loss:

```
    scores = None
    N = X.shape[0]
    D = X.reshape(N,-1).shape[1]
    C = self.C

    X_new = X.reshape(N,D)

    # layer 1
    cp = X.reshape(N,D)
    cp = np.dot(cp,self.params['W1'])
    cp = cp + self.params['b1']
    h = cp

    # relu
    condition1 = (cp>0).astype(int)
    cp = condition1 * cp
    h1 = cp
    # layer 2
    cp = np.dot(cp,self.params['W2'])
    cp = cp + self.params['b2']
    # softmax
    scores = cp



    if y is None:
        return scores

    loss, grads = 0, {}

    scoresmax = np.max(scores,axis = 1).reshape(N,1)
    scores = scores - scoresmax
    expscores = np.exp(scores)
    t = expscores.sum(axis =1).reshape(N,1)
    expscores = expscores / t
    scores_yi = expscores[range(0,N),y]
    loss = -np.log(scores_yi).sum() /N+ 0.5*self.reg * (self.params['W1'] * self.params['W1']).sum() + 0.5*self.reg * (self.params['W2'] *self.params['W2']).sum()

#          print(loss)

    '''gradient'''
    # Loss
    dscore = (expscores).reshape(N,C)
    dscore[range(N),y] -= 1
    dscore /= N

    # f2
```

```python

    dw2 = np.dot(h1.T , dscore)
    db2 = np.sum(dscore,axis = 0)
    dh1 = np.dot(dscore,self.params['W2'].T)

    # relu

    dh = (h>0).astype(int)
    dh = dh * dh1

    # f1

    dw1 = np.dot(X_new.T,dh)
    db1 = np.sum(dh,axis = 0)

#          dw1 = dw1.reshape(self.params['W1'].shape)
#          dw2 = dw2.reshape(self.params['W2'].shape)
    dw1 += self.reg * self.params['W1']
    dw2 += self.reg * self.params['W2']

    grads['W1']=dw1
    grads['b1']=db1
    grads['W2']=dw2
    grads['b2']=db2
```

Listing 22: Two Layer Net Loss

sgd

```python
    w -= config['learning_rate'] * dw
```

Listing 23: sgd

FC Net grid search:

```python
    input_size = 32 * 32 * 3
    num_classes = 10
    best_acc =-1
    for bs in [200, 400]:
        for lr in [1e-3, 1e-4, 1e-5]:
            for hidden_size in [50, 100, 200]:
                net = TwoLayerNet(input_size, hidden_size, num_classes)
                solver = Solver(net, data,
                    num_train_samples=100,
                    lr_decay=0.9,
                    num_epochs=20,
                    print_every=50000,
                    batch_size = bs,
                    optim_config={
                        'learning_rate': lr,
                    }
                   )
```

```python
                solver.train()

                # Predict on the validation set
                val_acc = solver.check_accuracy(data['X_val'],data['y_val'])
                print ('batch_size = %d, lr = %f, hidden size = %f,
    Valid_accuracy: %f' %(bs, lr, hidden_size,val_acc))
                if val_acc > best_acc:
                    best_acc = val_acc
                    best_model = net
```

Listing 24: FC Net grid search

4 GAT

4.1 theory

对于一个图，我们将其表示为邻接矩阵 $adj \in R^{N \times N}$，每个节点 i 都有其特征表示向量 $h_i \in R^D$，在单个 attention 中，设 $W \in R^{in \times out}$ 为共有的可学习的参数，对于直接连接在节点向量上的 attention 层中 in = D，$\alpha \in R^{2 \cdot out}$ 为注意力系数向量也是可学习的，在前向计算中，隐藏层的向量由以下公式计算：

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T\left[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j\right]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T\left[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k\right]\right)\right)}$$

$$\vec{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}\mathbf{W}\vec{h}_j\right)$$

对于多头 attention，我们只需设立多个 W α，将计算出的每个节点的隐藏层向量拼接起来即可。

对于此多头 attention 层是最后一层，就不对其做拼接操作，而是直接对 K（头数）个隐藏层向量齐求均值

4.2 codes

GAT 的搭建（based torch）：

```python
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)]
```

```python
        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)  # 第二层(最后一层)的attention layer

    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)  # cat hidden vector of every node
        x = F.dropout(x, self.dropout, training=self.training)
        x = F.elu(self.out_att(x, adj))   # final attention
        return F.log_softmax(x, dim=1)
```

Listing 25: GAT

```python
    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)))  # concat(V,NeigV)
        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)
        a_input = self._prepare_attentional_mechanism_input(Wh)  # 每一个节点和所有节点，特征。(Vall, Vall, feature)
        e = self.leakyrelu(torch.matmul(a_input, self.a).squeeze(2))
        # 之前计算的是一个节点和所有节点的attention，其实需要的是连接的节点的attention系数
        zero_vec = -9e15*torch.ones_like(e) #
        attention = torch.where(adj > 0, e, zero_vec)    # 无边相连即为-\inf,有边相连，设置值为\alpha_{i,j}将邻接矩阵中小于0的变成负无穷 e^-\inf = 0
        attention = F.softmax(attention, dim=1)  # 按行求softmax。 将系数归
```

```
一, sum(axis=1) == 1
        attention = F.dropout(attention, self.dropout, training=self.
training)
        h_prime = torch.matmul(attention, Wh)   # 聚合邻居函数, 加权平均

        if self.concat:
            return F.elu(h_prime)
        else:
            return h_prime

    def _prepare_attentional_mechanism_input(self, Wh): # 计算Wh_i || Wh_j
        N = Wh.size()[0]
        Wh_repeated_in_chunks = Wh.repeat_interleave(N, dim=0)
        Wh_repeated_alternating = Wh.repeat(N, 1)
        all_combinations_matrix = torch.cat([Wh_repeated_in_chunks,
Wh_repeated_alternating], dim=1)
        return all_combinations_matrix.view(N, N, 2 * self.out_features)
```

Listing 26: Attention layer

对于训练过程，我们可以传入训练集的图结构和节点向量矩阵，在测试的时候将邻接矩阵变幻，通过 mask 获得测试节点的 label。

```
model.train()
optimizer.zero_grad()

features1 = features[0:1000]
adj1 = adj[0:1000]
adj1 = adj1.T
adj1 = adj1[0:1000]
adj1 = adj1.T
labels1 = labels[0:1000]
# print("feature1:",features1.shape)
# print("adj1:",adj1.shape)
output = model(features1, adj1)  # GAT模块
# loss_train = F.nll_loss(output[idx_train], labels[idx_train])
# acc_train = accuracy(output[idx_train], labels[idx_train])

loss_train =  F.nll_loss(output, labels1)
acc_train = accuracy(output, labels1)


# 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])
```

Listing 27: train & test