csc413 a4

lv aty

December 2024

1 1.1.2

Firstly, for n layers RNN:

$$f(x) = \text{ReLU}(W_n \cdot \text{ReLU}(W_{n-1} \cdot \text{ReLU}(W_{n-2} \cdots \text{ReLU}(W_1 \cdot x))))$$

Define:

$$\begin{aligned} h_1 &= \text{ReLU}(W_1 \cdot x), \\ h_2 &= \text{ReLU}(W_2 \cdot h_1), \\ &\vdots \\ h_n &= f(x) = \text{ReLU}(W_n \cdot h_{n-1}). \end{aligned}$$

We know:

$$\text{ReLU}(z) = \begin{cases} z, & \text{if } z > 0, \\ 0, & \text{if } z < 0. \end{cases}$$

$$\frac{d}{dz}\text{ReLU}(z) = \begin{cases} 1, & \text{if } z > 0, \\ 0, & \text{if } z < 0. \end{cases}$$

Then:

$$\begin{aligned} \frac{d}{dx} f(x) &= \frac{d\,\text{ReLU}(W_n h_{n-1})}{dh_{n-1}} \cdot \frac{d\,\text{ReLU}(W_{n-1} h_{n-2})}{dh_{n-2}} \cdots \frac{d\,\text{ReLU}(W_1 x)}{dx} \\ &= W_n(\text{or } 0) \cdot W_{n-1}(\text{or } 0) \cdots W_1(\text{or } 0) \\ &= \prod_{i=1}^{n} W_i(\text{or } 0). \end{aligned}$$

$$\left|\frac{d}{dx}f(x)\right| = 0 \text{ or } \prod_{i=1}^{n}|W_i|$$

Since Wi does not equal to 1 anymore.
When any input of ReLU is negative, the gradient becomes zero at that layer, immediately causing the gradient to vanish.
When $|W_i| > 1$, as you backpropagate, the magnitude of the gradient grows exponentially with the number of layers. This is an explosion scenario.
When $|W_i| < 1$, as you backpropagate, the magnitude of the gradient shrinks exponentially with the number of layers. This is a vanishing scenario.

2 1.2.1

Recall:

$$\frac{\partial}{\partial z}\text{sigmoid}(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \text{sigmoid}(z)\cdot(1-\text{sigmoid}(z))$$

Recall that the maximum of $\frac{\partial}{\partial z}\text{sigmoid}(z)$ is:

$$\frac{1}{4} \quad \text{at } z = 0$$

From hint: When we have n layers, the input-output Jacobian is the multiplication of per-layer Jacobians.
From hint:

$$C = AB \implies \sigma_{\max}(C) = \sigma_{\max}(A)\cdot\sigma_{\max}(B)$$

Thus, the maximum singular value of a product of matrices is at most the product of their maximum singular values.

At one time step:
For $x_{t+1} = \text{sigmoid}(Wx_t)$:

$$\frac{\partial x_{t+1}}{\partial x_t} = \frac{\partial\,\text{sigmoid}(Wx_t)}{\partial Wx_t}\cdot\frac{\partial Wx_t}{\partial x_t}$$

Suppose:

$$
\begin{aligned}
Wx_t &= \begin{bmatrix} W_{11} & \cdots & W_{1n} \\ \vdots & \ddots & \vdots \\ W_{n1} & \cdots & W_{nn} \end{bmatrix} \begin{bmatrix} x_{t1} \\ \vdots \\ x_{tn} \end{bmatrix} \\
&= \begin{bmatrix} W_{11}x_{t1} + \cdots + W_{1n}x_{tn} \\ \vdots \\ W_{n1}x_{t1} + \cdots + W_{nn}x_{tn} \end{bmatrix} \\
&= \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}.
\end{aligned}
$$

$$
\operatorname{sigmoid}(Wx_t) = \begin{bmatrix} \operatorname{sigmoid}(z_1) \\ \vdots \\ \operatorname{sigmoid}(z_n) \end{bmatrix}
$$

$$
\begin{aligned}
\frac{\partial \operatorname{sigmoid}(Wx_t)}{\partial Wx_t} &= \begin{bmatrix} \frac{\partial \operatorname{sigmoid}(z_1)}{\partial z_1} & \cdots & \frac{\partial \operatorname{sigmoid}(z_1)}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \operatorname{sigmoid}(z_n)}{\partial z_1} & \cdots & \frac{\partial \operatorname{sigmoid}(z_n)}{\partial z_n} \end{bmatrix} \\
&= \begin{bmatrix} \operatorname{sigmoid}'(z_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \operatorname{sigmoid}'(z_n) \end{bmatrix} \\
&= \operatorname{diag}(\operatorname{sigmoid}'(Wx_t)).
\end{aligned}
$$

Then

$$
\begin{aligned}
\frac{\partial x_{t+1}}{\partial x_t} &= \frac{\partial \operatorname{sigmoid}(Wx_t)}{\partial Wx_t} \cdot \frac{\partial Wx_t}{\partial x_t} \\
&= \operatorname{diag}(\operatorname{sigmoid}'(Wx_t)) \cdot W
\end{aligned}
$$

$$
\sigma_{\max}\left(\frac{\partial x_{t+1}}{\partial x_t}\right) = \sigma_{\max}(\operatorname{diag}(\operatorname{sigmoid}'(Wx_t))) \cdot \sigma_{\max}(W)
$$

As $\operatorname{diag}(\operatorname{sigmoid}'(Wx_t))$ is a diagonal matrix:

$$
\sigma_{\max}(\operatorname{diag}(\operatorname{sigmoid}'(Wx_t))) = \max\{\operatorname{sigmoid}'(Wx_t)\}
$$

As we have known:

$$
\frac{\partial}{\partial z}\operatorname{sigmoid}(z) \leq \frac{1}{4}
$$

So:

$$\sigma_{\max}(\text{diag}(\text{sigmoid}'(Wx_t))) \leq \frac{1}{4}$$

From the given:

$$\sigma_{\max}(W) = \frac{1}{4}$$

$$\sigma_{\max}\left(\frac{\partial x_{t+1}}{\partial x_t}\right) = \sigma_{\max}(\text{diag}(\text{sigmoid}'(Wx_t))) \cdot \sigma_{\max}(W)$$

$$\leq \frac{1}{4} \cdot \frac{1}{4}$$

$$\leq \frac{1}{16}$$

$$\frac{\partial x_n}{\partial x_1} = \frac{\partial \,\text{sigmoid}(Wx_{n-1})}{\partial x_{n-1}} \cdot \frac{\partial \,\text{sigmoid}(Wx_{n-2})}{\partial x_{n-2}} \cdots \frac{\partial \,\text{sigmoid}(Wx_1)}{\partial x_1}$$

Then:

$$\sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \left(\frac{1}{16}\right)^{n-1}$$

As the singular value must be positive by definition:

$$0 \leq \sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \left(\frac{1}{16}\right)^{n-1}$$

3 1.3.1

By hint, substitute in the kernel function into Eq 1.1:

$$\begin{aligned}
\alpha_i &= \frac{\sum_{j=1}^{n} \sin(Q_i, K_j) V_j}{\sum_{j=1}^{n} \sim (Q_i, K_j)} \\
&= \frac{\sum_{j=1}^{n} \phi(Q_i)^T \phi(K_j) V_j}{\sum_{j=1}^{n} \phi(Q_i)^T \phi(K_j)} \\
&= \frac{\phi(Q_i)^T \sum_{j=1}^{n} \phi(K_j) V_j}{\phi(Q_i)^T \sum_{j=1}^{n} \phi(K_j)}.
\end{aligned}$$

For $\sum_{j=1}^{n} \phi(K_j)V_j$ in the numerator and $\sum_{j=1}^{n} \phi(K_j)$ in the denominator:
This requires going through $j = 1, \ldots, n$.
The complexity is $O(n)$.
Then for each i (each query), compute:

$$\phi(Q_i)^T \sum_{j=1}^{n} \phi(K_j)V_j \text{ and } \phi(Q_i)^T \sum_{j=1}^{n} \phi(K_j)$$

will still be $O(1)$.

So the dominating term is still $O(n)$. Thus, by expressing the similarity function as a kernel inner product and grouping terms by their subscript, we have factored the original $O(n^2)$ operation into a set of $O(n)$ operations plus a set of $O(1)$-per-query computations, bringing the overall complexity for computing all α_i down to $O(n)$.

4 1.3.2

Suppose P is an $n \times n$ matrix, and rank $= k$.

Let:

$$P = U\Sigma W^T,$$

where $U, W \in \mathbb{R}^{n\times k}$ are orthogonal matrices, and $\Sigma \in \mathbb{R}^{k\times k}$ is a diagonal matrix with singular values.

$$\text{Attention} = PV = U\Sigma W^T V,$$

where $V \in \mathbb{R}^{n\times d}$.

Step 1: Compute $W^T V$, which equals computing the matrix multiplication of $d \times n$ and $n \times k$.

$$\text{Complexity will be } O(nkd).$$

Step 2: Compute $\Sigma(W^T V)$, which equals computing the matrix multiplication of $k \times k$ and $d \times k$.

$$\text{Complexity will be } O(dk^2).$$

As $k \ll n$, $O(dk^2)$ is negligible compared to $O(nkd)$.

Step 3: Compute $U(\Sigma W^T V)$, which equals computing the matrix multiplication of $n \times k$ and $d \times k$.

$$\text{Complexity will be } O(ndk).$$

Therefore, in total, the dominant term is $O(ndk)$.

So it is possible to compute self-attention in $O(ndk)$ time.

5 1.4.1

$$\text{Con1D}(x; W)_i = \sum_{j=-1}^{1} x_{i+j} W_{j+2}$$

$$= x_{i-1}W_1 + x_i W_2 + x_{i+1}W_3$$

$$= 2x_{i-1} + x_{i+1}.$$

We can see (intuitively):

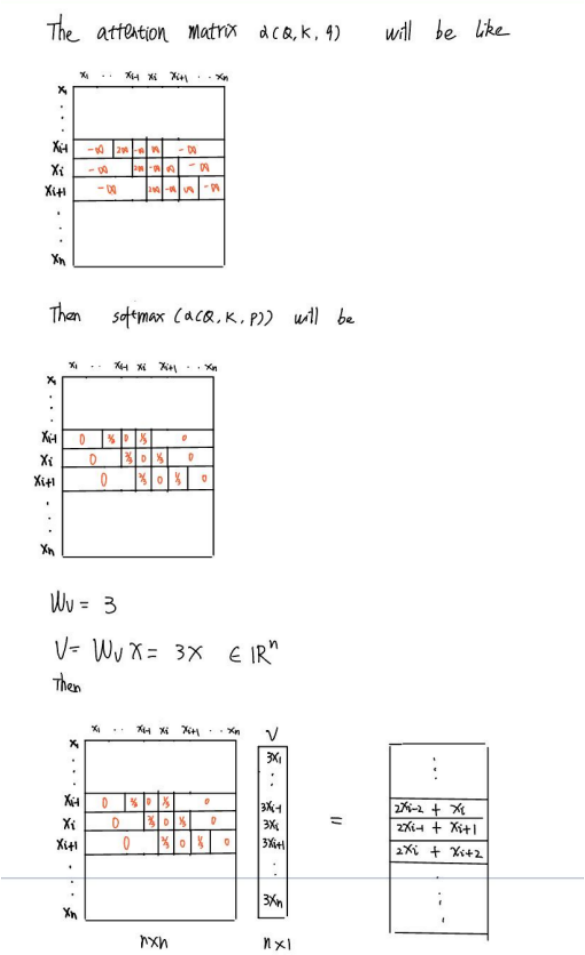
For every x_i, we want it to pay twice attention to the front x_{i-1} and one attention to x_{i+1} after it.

$$x \in \mathbb{R}^n$$

Let

$$W_Q = 1, \quad Q = W_Q x = x \in \mathbb{R}^n,$$
$$W_K = 1, \quad K = W_K x = x \in \mathbb{R}^n.$$

$$P_{i-j} = \begin{cases} -\infty, & |i-j| \geq 2 \text{ or } i = j, \\ 2\infty, & i - j = 1, \\ \infty, & i - j = -1. \end{cases}$$



For RelAttn(W_Q, W_K, W_V, X, q)$_i$

$= 2x_{i-1} + x_{i+1}$

$=$ Conv1D$(x; W)_i$

6 1.4.2

We can see (intuitively):

For every x_i, we want it to pay equal attention to $x_{i-k} \sim x_{i+k}$.

$$x \in \mathbb{R}^n,$$

Let

$$W_Q = 1, \quad Q = W_Q x = x \in \mathbb{R}^n,$$
$$W_K = 1, \quad K = W_K x = x \in \mathbb{R}^n.$$

$$P_{i-j} = \begin{cases} -\infty, & |i-j| > k, \\ 0, & |i-j| \le k. \end{cases}$$

$$\alpha_{ij}(Q, K, P) = \frac{Q_i K_j}{\sqrt{d_k}} + P_{i-j}$$

$$= \frac{x_i x_j}{\sqrt{d_k}} + P_{i-j}.$$

For each i, j (from 1 to n),

$$\alpha_i = [\alpha_{i1}, \ldots, \alpha_{in}].$$

As $x_i \ge 0$,

$\alpha_{ij} \ge 0$ for $i - k \le j \le i + k$
$\alpha_{ij} = -\infty$ for $j > i + k$ or $j < i - k$.

Since adding $P_{i-j} = 0$ for $i - k \le j \le i + k$ does not affect the location of maximum.
Then apply arg max instead of softmax. $\arg\max(\alpha_i(Q, K, P))$ will be a one-hot vector, where the position of 1 is the position of maximum of $x_{i-k} \sim x_{i+k}$.

Then:

$$\arg\max(\alpha_i(Q, K, P)) \cdot V = \max_{-k \le m \le k} x_{m+i}.$$

7 2.1.1

For $X = \{0, e_1, \ldots, e_d\}$, where $e_1, \ldots, e_d$ are the standard normal basis:

$$\implies X \subseteq \mathbb{R}^d$$
$$\forall x \in X, \quad \|x\|_2 = 1$$

Let $\epsilon = \frac{1}{3}\delta$.

Apply JL-Lemma on $X = \{0, e_1, \ldots, e_d\}$, where $e_1, \ldots, e_d$ are the standard normal basis:

$$\implies X \subseteq \mathbb{R}^d, \quad |X| = n = d + 1.$$

For $k \in \mathbb{N}$:

$$k \geq \frac{8 \ln n}{\epsilon^2}$$
$$k \geq \frac{8 \ln n}{\frac{1}{9}\delta^2}$$
$$k \geq \frac{72 \ln n}{\delta^2}$$
$$k \geq \frac{72 \ln(d+1)}{\delta^2}.$$

$$\frac{k\delta^2}{72} \geq \ln(d+1)$$

$$e^{\frac{k\delta^2}{72}} \geq d + 1$$

$$d \leq e^{\frac{k\delta^2}{72}} - 1.$$

By JL Lemma
There exists a matrix $M \in \mathbb{R}^{k \times d}$, such that for all $x_i, x_j \in X$:

$$(1-\epsilon)\|x_i - x_j\|_2^2 \leq \|Mx_i - Mx_j\|_2^2 \leq (1+\epsilon)\|x_i - x_j\|_2^2.$$

By Hint

$$\|x - y\|_2^2 = \|x\|_2^2 + \|y\|_2^2 - 2x^T y.$$

For unit vectors, i.e., $\|x\|_2^2 = \|y\|_2^2 = 1$:

$$\|x - y\|_2^2 = 2 - 2x^T y.$$

Thus

$$(1-\epsilon)\|x_i - x_j\|_2^2 \leq \|Mx_i - Mx_j\|_2^2 \leq (1+\epsilon)\|x_i - x_j\|_2^2$$

$$(1-\epsilon)(2 - 2x_i^T x_j) \leq \|Mx_i - Mx_j\|_2^2 \leq (1+\epsilon)(2 - 2x_i^T x_j).$$

As $M \in \mathbb{R}^{k \times d}$, $k \leq d$, M transforms $x \in \mathbb{R}^d$ to a smaller dimensional space $Mx \in \mathbb{R}^k$. Preserving $\|x_i - x_j\|_2$ approximately means we preserve $x_i^T x_j$, meaning we preserve the almost-orthogonality property.

$$(1-\epsilon)(2 - 2x_i^T x_j) \leq \|Mx_i - Mx_j\|_2^2 \leq (1+\epsilon)(2 - 2x_i^T x_j).$$

As $x_i, x_j \in X = \{0, e_1, \ldots, e_d\}$:
$\implies$ Any $x_i^T x_j = 0$.

$\implies 2(1-\epsilon) \leq \|Mx_i - Mx_j\|_2^2 \leq 2(1+\epsilon).$
$\implies 2(1-\epsilon) \leq \|Mx_i\|_2^2 + \|Mx_j\|_2^2 - 2Me_i^T Me_j \leq 2(1+\epsilon).$

Additionally, when $x_i = 0$, $x_j \neq 0$:

$$2(1-\epsilon) \leq \|Me_j\|_2^2 \leq 2(1+\epsilon), \quad \text{for } j = 1 \sim d.$$

Next, again

$$2(1-\epsilon) \leq \|Mx_i\|_2^2 + \|Mx_j\|_2^2 - 2(Mx_i)^T(Mx_j) \leq 2(1+\epsilon).$$
$$2(1-\epsilon) - \|Mx_i\|_2^2 - \|Mx_j\|_2^2 \leq -2(Mx_i)^T(Mx_j) \leq 2(1+\epsilon) - \|Mx_i\|_2^2 - \|Mx_j\|_2^2.$$

1. When $x_i = 0$, $x_j \neq 0$:
$$-4\epsilon \leq -2(Mx_i)^T(Mx_j) \leq 4\epsilon.$$
2. When $x_i \neq 0$, $x_j \neq 0$:
$$-6\epsilon \leq -2(Mx_i)^T(Mx_j) \leq 6\epsilon.$$

In total, we only care about case 2:

$$-6\epsilon \leq -2(Mx_i)^T(Mx_j) \leq 6\epsilon.$$

$$-3\epsilon \leq (Mx_i)^T(Mx_j) \leq 3\epsilon.$$

$$|Me_i^T Me_j| \leq 3\epsilon = \delta.$$

In conclusion:

For:
$$U = \left\{ \frac{Me_1}{\|Me_1\|}, \frac{Me_2}{\|Me_2\|}, \ldots, \frac{Me_d}{\|Me_d\|} \right\}.$$

Argument 1: is automatically satisfied:

$$\forall u \in U, \quad \|u\|_2 = 1.$$

Argument 2:

$$\forall u_i, u_j \in U, \quad \text{if } u_i \neq u_j, \quad |Me_i^T Me_j| \leq 3\epsilon = \delta.$$

Argument 3: Want To Show:

$$|U| \in \Omega\left(e^{\frac{n\delta^2}{c}}\right), \quad \text{i.e., } |U| \geq c_1 e^{\frac{n\delta^2}{c}} \quad \text{for some positive } c_1.$$

We know: as long as:

$$|U| = d \leq e^{\frac{k\delta^2}{72}} - 1.$$

the above proof works.
So let $|U| = d = e^{\frac{k\delta^2}{72}} - 1$. Set $k = n$, $c > 72$ (since c is arbitrary): such that

$$\implies |U| \in \Omega\left(e^{\frac{n\delta^2}{c}}\right).$$

Programming

3.1

```python
class GraphConvolution(nn.Module):
    """
    A Graph Convolution Layer (GCN)
    """

    def __init__(self, in_features, out_features, bias=True):
        """
        * `in_features`, $F$, is the number of input features per node
        * `out_features`, $F'$, is the number of output features per node
        * `bias`, whether to include the bias term in the linear layer. Default=True
        """
        super(GraphConvolution, self).__init__()
        # TODO: initialize the weight W that maps the input feature (dim F ) to output feature (dim F')
        # hint: use nn.Linear()
        ############ Your code here ##################################
        self.linear = nn.Linear(in_features, out_features, bias=bias)



        ##############################################################

    def forward(self, input, adj):
        # TODO: transform input feature to output (don't forget to use the adjacency matrix
        # to sum over neighbouring nodes )
        # hint: use the linear layer you declared above.
        # hint: you can use torch.spmm() sparse matrix multiplication to handle the
        #       adjacency matrix
        ############ Your code here ##################################
        input = self.linear(input)
        output = torch.spmm(adj, input)
        return output



        ##############################################################
```

3.2

```python
class GCN(nn.Module):
    '''
    A two-layer GCN
    '''
    def __init__(self, nfeat, n_hidden, n_classes, dropout, bias=True):
        """
        * `nfeat`, is the number of input features per node of the first layer
        * `n_hidden`, number of hidden units
        * `n_classes`, total number of classes for classification
        * `dropout`, the dropout ratio
        * `bias`, whether to include the bias term in the linear layer. Default=True
        """

        super(GCN, self).__init__()
        # TODO: Initialization
        # (1) 2 GraphConvolution() layers.
        # (2) 1 Dropout layer
        # (3) 1 activation function: ReLU()
        ############ Your code here ##################################
        self.gc1 = GraphConvolution(nfeat, n_hidden, bias=bias)
        self.gc2 = GraphConvolution(n_hidden, n_classes, bias=bias)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()

        ##############################################################

    def forward(self, x, adj):
        # TODO: the input will pass through the first graph convolution layer,
        # the activation function, the dropout layer, then the second graph
        # convolution layer. No activation function for the
        # last layer. Return the logits.
        ############ Your code here ##################################
        x = self.gc1(x, adj)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.gc2(x, adj)
        return x
        ##############################################################
```

3.3

```
Epoch: 0091 loss_train: 0.7919 acc_train: 0.8286 loss_val: 1.1238 acc_val: 0.6254 time: 0.0026s
Epoch: 0092 loss_train: 0.7797 acc_train: 0.8429 loss_val: 1.1160 acc_val: 0.6293 time: 0.0025s
Epoch: 0093 loss_train: 0.7539 acc_train: 0.8786 loss_val: 1.1086 acc_val: 0.6336 time: 0.0026s
Epoch: 0094 loss_train: 0.8075 acc_train: 0.8286 loss_val: 1.1020 acc_val: 0.6386 time: 0.0026s
Epoch: 0095 loss_train: 0.7681 acc_train: 0.8500 loss_val: 1.0956 acc_val: 0.6421 time: 0.0026s
Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892 acc_val: 0.6452 time: 0.0026s
Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826 acc_val: 0.6491 time: 0.0026s
Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760 acc_val: 0.6515 time: 0.0026s
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val: 0.6488 time: 0.0026s
Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643 acc_val: 0.6515 time: 0.0026s
Optimization Finished!
Total time elapsed: 2.1294s
Test set results: loss= 1.0643 accuracy= 0.6515
```

3.4

```python
class GraphAttentionLayer(nn.Module):

    def __init__(self, in_features: int, out_features: int, n_heads: int,
                 is_concat: bool = True,
                 dropout: float = 0.6,
                 alpha: float = 0.2):
        """
        in_features: F, the number of input features per node
        out_features: F', the number of output features per node
        n_heads: K, the number of attention heads
        is_concat: whether the multi-head results should be concatenated or averaged
        dropout: the dropout probability
        alpha: the negative slope for leaky relu activation
        """
        super(GraphAttentionLayer, self).__init__()

        self.is_concat = is_concat
        self.n_heads = n_heads

        if is_concat:
            assert out_features % n_heads == 0
            self.n_hidden = out_features // n_heads
        else:
            self.n_hidden = out_features

        # TODO: initialize the following modules:
        # (1) self.W: Linear layer that transform the input feature before self attention.
        # You should NOT use for loops for the multiheaded implementation (set bias = Flase)
        # (2) self.attention: Linear layer that compute the attention score (set bias = Flase)
        # (3) self.activation: Activation function (LeakyReLU whith negative_slope=alpha)
        # (4) self.softmax: Softmax function (what's the dim to compute the summation?)
        # (5) self.dropout_layer: Dropout function(with ratio=dropout)
        ################ your code here ######################
        self.W = nn.Linear(in_features, self.n_hidden * n_heads, bias=False)
        self.attention = nn.Linear(2 * self.n_hidden, 1, bias=False)
        self.activation = nn.LeakyReLU(alpha)
        self.softmax = nn.Softmax(dim=1)
        self.dropout_layer = nn.Dropout(dropout)
```

```python
    def forward(self, h: torch.Tensor, adj_mat: torch.Tensor):
        # Number of nodes
        n_nodes = h.shape[0]

        # TODO:
        # (1) calculate s = Wh and reshape it to [n_nodes, n_heads, n_hidden]
        #     (you can use tensor.view() function)
        # (2) get [s_i || s_j] using tensor.repeat(), repeat_interleave(), torch.cat(), tensor.view()
        # (3) apply the attention layer
        # (4) apply the activation layer (you will get the attention score e)
        # (5) remove the last dimension 1 use tensor.squeeze()
        # (6) mask the attention score with the adjacency matrix (if there's no edge, assign it to -inf)
        #     note: check the dimensions of e and your adjacency matrix. You may need to use the function unsqueeze()
        # (7) apply softmax
        # (8) apply dropout_layer
        ############## Your code here ######################################## (with help of ChatGPT)
        s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)
        s_i = s.repeat_interleave(n_nodes, dim=0)
        s_j = s.repeat(n_nodes, 1, 1)
        s_cat_ij = torch.cat([s_i, s_j], dim=-1)
        s_cat_ij = s_cat_ij.view(n_nodes,n_nodes,self.n_heads, self.n_hidden*2)


        e = self.attention(s_cat_ij) #[n_nodes, n_nodes, n_heads, 1]
        e_scores = self.activation(e)
        e_scores = e_scores.squeeze(-1) #[n_nodes, n_nodes, n_heads]

        # (3)
        adj_mat = adj_mat.unsqueeze(-1)
        e_scores = e_scores.masked_fill(adj_mat == 0, -np.inf)
        a = self.softmax(e_scores)
        a = self.dropout_layer(a)

        ########################################################################
```

```
    # Summation
    h_prime = torch.einsum('ijh,jhf->ihf', a, s) #[n_nodes, n_heads, n_hidden]


    # TODO: Concat or Mean
    # Concatenate the heads
    if self.is_concat:
        ############## Your code here ##########################################
        h_prime = h_prime.reshape(n_nodes, self.n_heads * self.n_hidden)
        return h_prime


        ########################################################################
    # Take the mean of the heads (for the last layer)
    else:
        ############## Your code here ##########################################
        h_prime = torch.mean(h_prime, dim=1)
        return h_prime



        ########################################################################
```

3.5

```
Epoch: 0089 loss_train: 0.9947 acc_train: 0.7786 loss_val: 1.1557 acc_val: 0.7368 time: 0.2108s
Epoch: 0090 loss_train: 1.0510 acc_train: 0.7786 loss_val: 1.1492 acc_val: 0.7364 time: 0.2117s
Epoch: 0091 loss_train: 1.0177 acc_train: 0.8071 loss_val: 1.1429 acc_val: 0.7379 time: 0.2116s
Epoch: 0092 loss_train: 0.9753 acc_train: 0.7571 loss_val: 1.1366 acc_val: 0.7399 time: 0.2117s
Epoch: 0093 loss_train: 0.9582 acc_train: 0.7786 loss_val: 1.1302 acc_val: 0.7395 time: 0.2118s
Epoch: 0094 loss_train: 1.0284 acc_train: 0.7357 loss_val: 1.1238 acc_val: 0.7403 time: 0.2112s
Epoch: 0095 loss_train: 0.9910 acc_train: 0.7929 loss_val: 1.1173 acc_val: 0.7410 time: 0.2115s
Epoch: 0096 loss_train: 1.0008 acc_train: 0.7714 loss_val: 1.1111 acc_val: 0.7410 time: 0.2115s
Epoch: 0097 loss_train: 0.8956 acc_train: 0.7929 loss_val: 1.1050 acc_val: 0.7410 time: 0.2115s
Epoch: 0098 loss_train: 0.8817 acc_train: 0.8000 loss_val: 1.0991 acc_val: 0.7403 time: 0.2113s
Epoch: 0099 loss_train: 0.8774 acc_train: 0.8286 loss_val: 1.0932 acc_val: 0.7414 time: 0.2112s
Epoch: 0100 loss_train: 0.9441 acc_train: 0.7929 loss_val: 1.0872 acc_val: 0.7426 time: 0.2115s
Optimization Finished!
Total time elapsed: 21.4710s
Test set results: loss= 1.0872 accuracy= 0.7426
```

3.6

From the output, we can see the test accuracy of GAT(74.26%) is higher than test accuracy(65.15%) of GCN. Also, the total time elapsed of GAT is much more than GCN.

Why:

GAT achieves higher accuracy because it learns to assign adaptive importance attention to different neighbors, which allows more informative context of graph structure to be considered into the model. GCN simply treats all neighbors equally and assign equal weights. However, this attention mechanism introduces additional complexity and memory usage, leading to longer training times.

4.

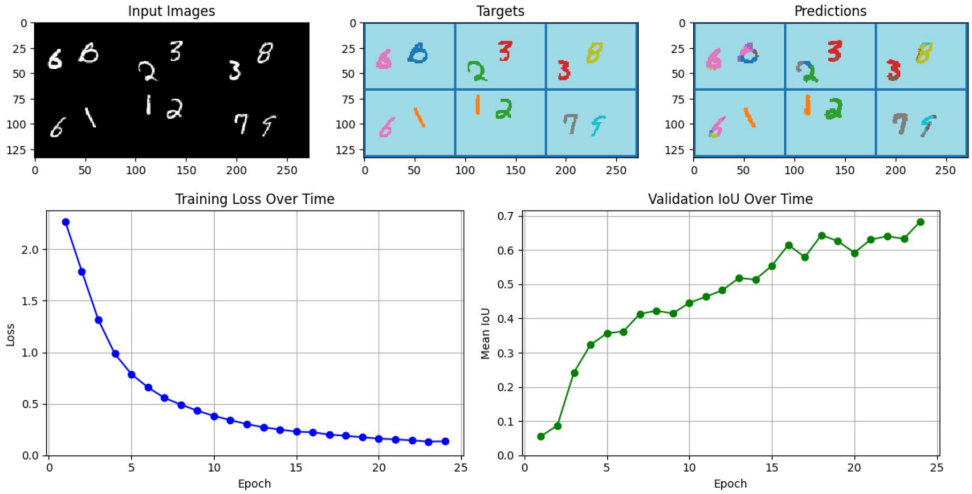
```python
class Down(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super(Down, self).__init__()
        ### YOU'RE CODE HERE ###
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.max_pool = nn.MaxPool2d(kernel_size=2)

    def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        """
        Args:
            x: Input tensor of shape (B, C, H, W)
        Returns:
            down: Downsampled tensor of shape (B, C_out, H/2, W/2)
            skip_connection: Skip connection tensor of shape (B, C_out, H, W) that goes
                through the convolution layers but skips the max pooling layer.
        """
        assert x.ndim == 4, f"Expected 4D input tensor, got shape {x.shape}"
        ### YOU'RE CODE HERE ####
        down = self.max_pool(self.relu(self.bn(self.conv(x))))
        skip_connection = self.relu(self.bn(self.conv(x)))

        expexted_size = tuple(s // 2 for s in x.shape[2:])
        assert down.shape[2:] == expexted_size, \
            f"Expected downsampled shape {expexted_size}, got {down.shape[2:]}"
        assert skip_connection.shape[2:] == x.shape[2:], \
            f"Expected skip connection shape {x.shape[2:]}, got {skip_connection.shape[2:]}"
        return down, skip_connection
```

```python
class Up(nn.Module):
    def __init__(self, in_channels: int, skip_channels: int, out_channels: int):
        super(Up, self).__init__()
        #### YOU'RE CODE HERE ####
        self.conv = nn.Conv2d(in_channels+skip_channels, out_channels, kernel_size=3, padding=1)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.block = nn.Sequential(
            self.conv,
            self.bn,
            self.relu
        )
        self.conv_transpose = nn.ConvTranspose2d(in_channels = in_channels, out_channels=in_channels, kernel_size=2, stride=2)



    def forward(self, x: torch.Tensor, skip_connection: torch.Tensor) -> torch.Tensor:
        """
        Args:
            x: Input tensor of shape (B, C_in, H, W)
            skip_connection: Skip connection tensor of shape (B, C_skip, H, W)
        Returns:
            Output tensor of shape (B, C_out, H*2, W*2)
        """
        assert x.ndim == 4, f"Expected 4D input tensor, got shape {x.shape}"
        assert skip_connection.ndim == 4, f"Expected 4D skip connection tensor, got shape {skip_connection.shape}"
        expected_skip_size = tuple(2 * s for s in x.shape[2:])
        assert skip_connection.shape[2:] == expected_skip_size, \
            f"Skip connection hight {skip_connection.shape[2:]} must be {expected_skip_size}"

        ### YOU'RE CODE HERE ####
        x = self.conv_transpose(x)
        x = torch.cat((x, skip_connection), dim=1)
        x = self.block(x)

        return x
```

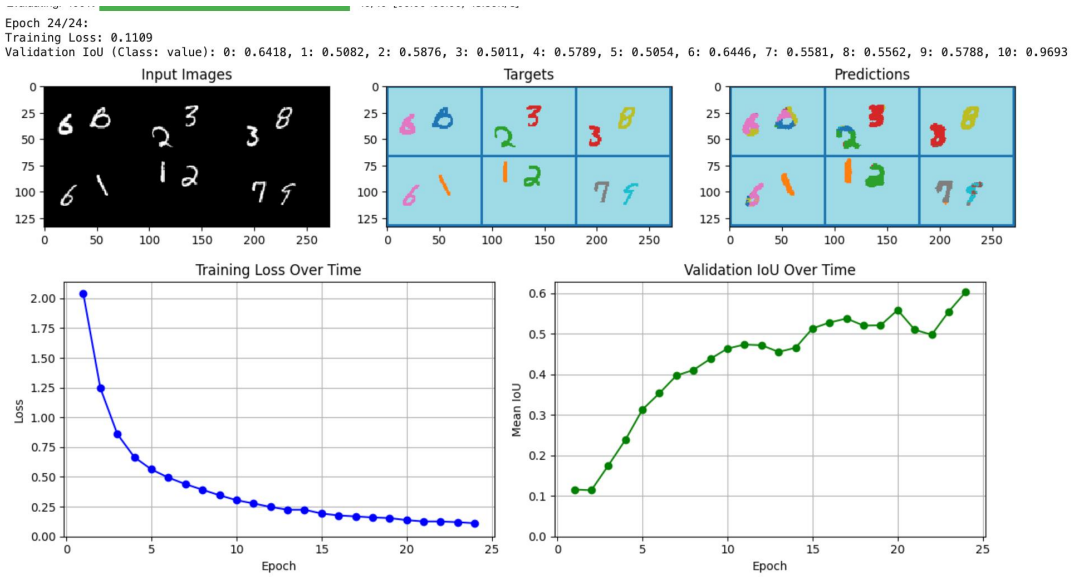
```python
class UNet(nn.Module):
    def __init__(self, in_channels: int = 1, out_channels: int = 11, disable_skip_connections: bool = False):
        """
        Args:
            in_channels: Number of input channels
            out_channels: Number of output channels
            disable_skip_connections: Whether to set all the skip connections to zero.
        """
        super(UNet, self).__init__()

        self.disable_skip_connections = disable_skip_connections

        ### YOU'RE CODE HERE ###
        self.down1 = Down(in_channels, 64)
        self.down2 = Down(64, 128)
        self.up1 = Up(64, 128, 32)
        self.up2 = Up(32, 64, 32)
        self.final_conv = nn.Conv2d(128, 64, kernel_size=3, padding=1)
        self.final_bn = nn.BatchNorm2d(64)
        self.final_relu = nn.ReLU()
        self.con_block = nn.Sequential(
            self.final_conv,
            self.final_bn,
            self.final_relu
        )
        self.final_conv2 = nn.Conv2d(32, out_channels, kernel_size=1)


    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Args:
            x: Input tensor of shape (B, C, H, W)
        Returns:
            Output tensor of shape (B, num_classes, H, W)
        """
        assert x.ndim == 4, f"Expected 4D input tensor, got shape {x.shape}"
        input_shape = x.shape[2:]

        ### YOU'RE CODE HERE ###
        x, skip1 = self.down1(x)
        x, skip2 = self.down2(x)
        x= self.con_block(x)
        if self.disable_skip_connections:
          skip1 = torch.zeros_like(skip1)
          skip2 = torch.zeros_like(skip2)

        x = self.up1(x, skip2)
        x = self.up2(x, skip1)

        output = self.final_conv2(x)
        assert output.shape[2:] == input_shape, \
            f"Output shape {output.shape[2:]} must match input shape {input_shape}"
        return output
```

When skip connection enabled.

```
Epoch 24/24:
Training Loss: 0.1354
Validation IoU (Class: value): 0: 0.7256, 1: 0.6364, 2: 0.6574, 3: 0.6231, 4: 0.6536, 5: 0.5939, 6: 0.7230, 7: 0.5714, 8: 0.7054, 9: 0.6456, 10: 0.9852
```



When skip connection disabled.



U-Net without skip connection has a lower training loss, but also a lower IoU(ratio between area of overlap and area of union) than U-Net with skip connection.

Why:
Without skip connections, U-Net can more easily minimize the training loss by learning overly smooth or simplified segmentations. However, without the direct transfer of high-resolution spatial features from early layers (skip connections), the reconstructions lack details. As a result, even though no skip connections can make training loss lower, the segmentation quality and IoU is bad because it cannot accurately capture fine boundaries in early layers.