

CSCI-GA 2572: Deep Learning, Fall 2025

Convolutional Neural Networks and Recurrent Neural Networks

Aditeya Baral
N19186654

Problem 1.1: Convolutional Neural Networks

Problem a

Solution

We are given an input image of size 21×12 , a convolutional kernel of size 4×5 , stride $S = 4$, and no padding ($P = 0$). Let $H_{\text{in}}, W_{\text{in}}$ denote the height and width of the input image, K_h, K_w denote the height and width of the kernel, S the stride, and P the padding. The output dimensions of a convolution are computed as,

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - K_h + 2P}{S} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - K_w + 2P}{S} \right\rfloor + 1$$

Substituting the given values:

$$H_{\text{out}} = \left\lfloor \frac{21 - 4 + 2(0)}{4} \right\rfloor + 1 = \lfloor 4.25 \rfloor + 1 = 5$$

$$W_{\text{out}} = \left\lfloor \frac{12 - 5 + 2(0)}{4} \right\rfloor + 1 = \lfloor 1.75 \rfloor + 1 = 2$$

Thus, the output dimension after applying the convolution is:

$$\boxed{5 \times 2}$$

Problem b

Solution

Let the input have dimension $C \times H \times W$, where C is the number of input channels, H is the height, and W is the width. Let the convolutional layer have kernel size $K \times K$, padding P , stride S , dilation D , and F filters. When dilation is applied, the effective kernel size becomes larger than the actual kernel size. The effective kernel size is computed as,

$$K_{\text{eff}} = K + (K - 1)(D - 1) = D(K - 1) + 1$$

The output height H_{out} and width W_{out} are then computed as,

$$H_{\text{out}} = \left\lfloor \frac{H - K_{\text{eff}} + 2P}{S} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W - K_{\text{eff}} + 2P}{S} \right\rfloor + 1$$

Substituting the effective kernel size:

$$H_{\text{out}} = \left\lfloor \frac{H - D(K - 1) - 1 + 2P}{S} \right\rfloor + 1 = \left\lfloor \frac{H + 2P - D(K - 1) - 1}{S} \right\rfloor + 1$$

$$W_{\text{out}} = \left\lfloor \frac{W + 2P - D(K - 1) - 1}{S} \right\rfloor + 1 = \left\lfloor \frac{W + 2P - D(K - 1) - 1}{S} \right\rfloor + 1$$

The number of filters F determines the number of output channels. Thus, the output dimension is,

$$F \times H_{\text{out}} \times W_{\text{out}}$$

where

$$H_{\text{out}} = \left\lfloor \frac{H + 2P - D(K - 1) - 1}{S} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W + 2P - D(K - 1) - 1}{S} \right\rfloor + 1$$

Problem c

Problem i

Solution

We are given an input $x[n] \in \mathbb{R}^5$ with $1 \leq n \leq 7$, meaning we have a length 7 sequence with 5 channels at each position. We consider the convolutional layer f_W with 1 filter, kernel size 3, stride 2, no dilation, and no padding. The weight is $W \in \mathbb{R}^{1 \times 5 \times 3}$, with no bias and no non-linearity. For a 1D convolution, the output sequence length is given by:

$$L_{\text{out}} = \left\lfloor \frac{L_{\text{in}} - K}{S} \right\rfloor + 1$$

where L_{in} is the input sequence length, L_{out} is the output sequence length, K is the kernel size, and S is the stride. Substituting $L_{\text{in}} = 7$, $K = 3$, $S = 2$ we obtain,

$$L_{\text{out}} = \left\lfloor \frac{7 - 3}{2} \right\rfloor + 1 = \lfloor 2 \rfloor + 1 = 3$$

Since we have 1 filter, the output has 1 channel and length 3. Thus, $f_W(x) \in \mathbb{R}^{1 \times 3}$.

Let $W[f, c, m]$ denote the weight for filter f , input channel $c \in \{1, 2, 3, 4, 5\}$, and kernel position $m \in \{0, 1, 2\}$. For our convolutional layer with kernel size 3, we can apply the cross-correlation equation $s[n] = (x * k)[n] = \sum_m x[n + m]k[m]$ to compute the output at position i by applying the kernel centered at input position $2i - 1$ (accounting for stride = 2),

$$f_W(x)[1, i] = \sum_{c=1}^5 \sum_{m=0}^2 W[1, c, m] \cdot x[c, 2i - 1 + m]$$

for $i \in \{1, 2, 3\}$. Thus, $f_W(x)$ is defined by,

$$f_W(x) \in \mathbb{R}^{1 \times 3}, \quad f_W(x)[1, i] = \sum_{c=1}^5 \sum_{m=0}^2 W[1, c, m] \cdot x[c, 2i - 1 + m], \quad i \in \{1, 2, 3\}$$

Problem ii

Solution

We need to find the dimension and expression for $\frac{\partial f_W(x)}{\partial W}$. From part (i), we have $f_W(x) \in \mathbb{R}^{1 \times 3}$ and $W \in \mathbb{R}^{1 \times 5 \times 3}$, with the expression:

$$f_W(x)[1, i] = \sum_{c=1}^5 \sum_{m=0}^2 W[1, c, m] \cdot x[c, 2i - 1 + m]$$

where

- $f_W(x)$ has dimensions: 1×3 , with indices $[1, i]$ where $i \in \{1, 2, 3\}$
- W has dimensions: $1 \times 5 \times 3$, with indices $[1, c, m]$ where $c \in \{1, 2, 3, 4, 5\}$, $m \in \{0, 1, 2\}$

The Jacobian $\frac{\partial f_W(x)}{\partial W}$ represents how each element of the output $f_W(x)$ changes with respect to each element of W and has one entry for each pair of (output element, weight element)

$$\frac{\partial f_W(x)}{\partial W} \in \mathbb{R}^{1 \times 3 \times 1 \times 5 \times 3}$$

Taking the partial derivative of $f_W(x)[1, i]$ with respect to $W[1, c', m']$:

$$\frac{\partial f_W(x)[1, i]}{\partial W[1, c', m']} = \frac{\partial}{\partial W[1, c', m']} \left(\sum_{c=1}^5 \sum_{m=0}^2 W[1, c, m] \cdot x[c, 2i - 1 + m] \right)$$

Since the summation is linear in W (no bias and no non-linearity), only the term where $c = c'$ and $m = m'$ contributes to the derivative:

$$\frac{\partial f_W(x)[1, i]}{\partial W[1, c', m']} = x[c', 2i - 1 + m']$$

Thus, we obtain the following expression,

$$\boxed{\frac{\partial f_W(x)}{\partial W} \in \mathbb{R}^{1 \times 3 \times 1 \times 5 \times 3}, \quad \frac{\partial f_W(x)[1, i]}{\partial W[1, c, m]} = x[c, 2i - 1 + m]}$$

for $i \in \{1, 2, 3\}$, $c \in \{1, 2, 3, 4, 5\}$, $m \in \{0, 1, 2\}$.

Problem iii

Solution

We need to find the dimension and expression for $\frac{\partial f_W(x)}{\partial x}$. From part (i), we have $f_W(x) \in \mathbb{R}^{1 \times 3}$ and $x \in \mathbb{R}^{5 \times 7}$, with the expression:

$$f_W(x)[1, i] = \sum_{c=1}^5 \sum_{m=0}^2 W[1, c, m] \cdot x[c, 2i - 1 + m]$$

where

- $f_W(x)$ has dimensions: 1×3 , with indices $[1, i]$ where $i \in \{1, 2, 3\}$
- x has dimensions: 5×7 , with indices $[c, n]$ where $c \in \{1, 2, 3, 4, 5\}$, $n \in \{1, 2, \dots, 7\}$

The Jacobian $\frac{\partial f_W(x)}{\partial x}$ represents how each element of the output $f_W(x)$ changes with respect to each element of the input x and has one entry for each pair of (output element, input element)

$$\frac{\partial f_W(x)}{\partial x} \in \mathbb{R}^{1 \times 3 \times 5 \times 7}$$

Taking the partial derivative of $f_W(x)[1, i]$ with respect to $x[c', n']$:

$$\frac{\partial f_W(x)[1, i]}{\partial x[c', n']} = \frac{\partial}{\partial x[c', n']} \left(\sum_{c=1}^5 \sum_{m=0}^2 W[1, c, m] \cdot x[c, 2i-1+m] \right)$$

Since the summation is linear in x , the derivative is non-zero only when $c = c'$ (matching channel) and $2i-1+m = n'$ (matching position). This gives $m = n' - 2i + 1$. The derivative is non-zero only when $m \in \{0, 1, 2\}$, which means $0 \leq n' - 2i + 1 \leq 2$, or equivalently $2i-1 \leq n' \leq 2i+1$. This implies that only input elements within the kernel window for output position i contribute to the derivative and all other positions have zero derivative:

$$\frac{\partial f_W(x)[1, i]}{\partial x[c', n']} = \begin{cases} W[1, c', n' - 2i + 1] & \text{if } n' \in \{2i-1, 2i, 2i+1\} \\ 0 & \text{otherwise} \end{cases}$$

Thus, we obtain the following expression,

$$\boxed{\frac{\partial f_W(x)}{\partial x} \in \mathbb{R}^{1 \times 3 \times 5 \times 7}, \quad \frac{\partial f_W(x)[1, i]}{\partial x[c, n]} = \begin{cases} W[1, c, n - 2i + 1] & \text{if } n \in \{2i-1, 2i, 2i+1\} \\ 0 & \text{otherwise} \end{cases}}$$

for $i \in \{1, 2, 3\}$, $c \in \{1, 2, 3, 4, 5\}$, $n \in \{1, 2, \dots, 7\}$.

Problem iv

Solution

From previous parts, we have $f_W(x) \in \mathbb{R}^{1 \times 3}$, $W \in \mathbb{R}^{1 \times 5 \times 3}$, and from part (ii):

$$\frac{\partial f_W(x)[1, i]}{\partial W[1, c, m]} = x[c, 2i-1+m]$$

for $i \in \{1, 2, 3\}$, $c \in \{1, 2, 3, 4, 5\}$, $m \in \{0, 1, 2\}$.

Since the loss ℓ is a scalar and $W \in \mathbb{R}^{1 \times 5 \times 3}$, the gradient $\frac{\partial \ell}{\partial W}$ must have the same dimensions as W :

$$\frac{\partial \ell}{\partial W} \in \mathbb{R}^{1 \times 5 \times 3}$$

By the chain rule, we can compute the gradient of the loss with respect to the weights by summing over all output elements:

$$\frac{\partial \ell}{\partial W[1, c, m]} = \sum_{i=1}^3 \frac{\partial \ell}{\partial f_W(x)[1, i]} \cdot \frac{\partial f_W(x)[1, i]}{\partial W[1, c, m]}$$

Substituting the expression from part (ii):

$$\frac{\partial \ell}{\partial W[1, c, m]} = \sum_{i=1}^3 \frac{\partial \ell}{\partial f_W(x)[1, i]} \cdot x[c, 2i-1+m]$$

Comparison with part (i):*Similarities:*

- Both expressions involve the same input values $x[c, 2i - 1 + m]$
- Both have a summation structure that connects outputs and weights through the input
- The indexing pattern $2i - 1 + m$ (accounting for stride 2 and kernel positions) appears in both

Differences:

- Part (i); the forward pass computes the output by multiplying weights with inputs

$$f_W(x)[1, i] = \sum_{c=1}^5 \sum_{m=0}^2 W[1, c, m] \cdot x[c, 2i - 1 + m]$$

This sums over channels c and kernel positions m for each output position i .

- Part (iv); The backward pass computes weight gradients by multiplying upstream gradients with inputs

$$\frac{\partial \ell}{\partial W[1, c, m]} = \sum_{i=1}^3 \frac{\partial \ell}{\partial f_W(x)[1, i]} \cdot x[c, 2i - 1 + m]$$

This sums over output positions i for each weight element $W[1, c, m]$.

- The summation indices are swapped: part (i) sums over (c, m) for each i , while part (iv) sums over i for each (c, m) . This demonstrates the transpose relationship in backpropagation through convolutional layers.

Thus, we obtain the following expression,

$$\boxed{\frac{\partial \ell}{\partial W} \in \mathbb{R}^{1 \times 5 \times 3}, \quad \frac{\partial \ell}{\partial W[1, c, m]} = \sum_{i=1}^3 \frac{\partial \ell}{\partial f_W(x)[1, i]} \cdot x[c, 2i - 1 + m]}$$

for $c \in \{1, 2, 3, 4, 5\}$, $m \in \{0, 1, 2\}$.

Problem 1.2: Recurrent Neural Networks

Problem 1.2.1 Part 1

Problem a

Solution

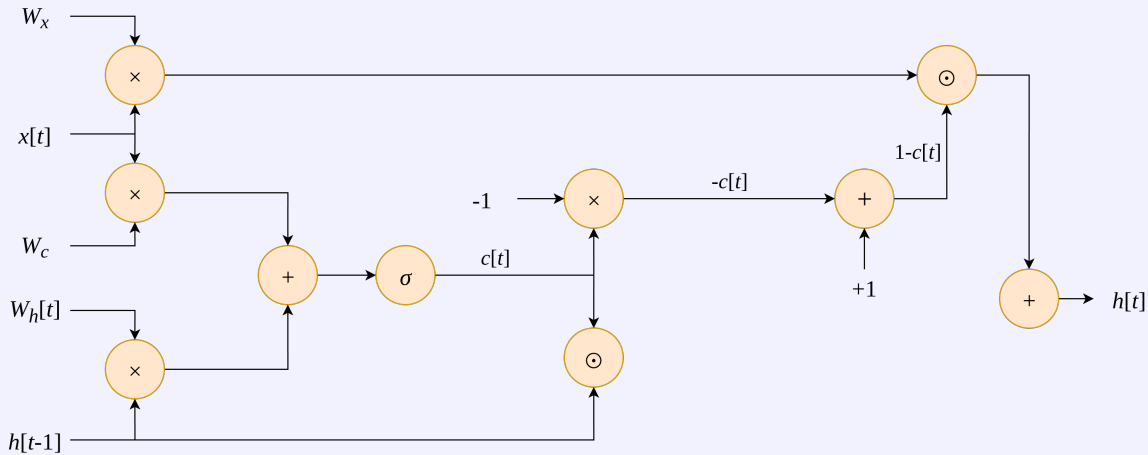


Figure 1: RNN architecture showing the computation of hidden state $h[t]$ through gating mechanism $c[t]$, which blends the previous hidden state $h[t-1]$ with the transformed current input $W_x x[t]$.

Problem b

Solution

From equation (1), we know that: $c[t] = \sigma(W_c x[t] + W_h h[t-1])$. We can find the dimension of $c[t]$ by analyzing the dimensions of the operands of the above expression.

Dimension of $W_c x[t]$

- $W_c \in \mathbb{R}^{m \times n}$
- $x[t] \in \mathbb{R}^n$
- $W_c x[t] \in \mathbb{R}^m$

Dimension of $W_h h[t-1]$

- $W_h \in \mathbb{R}^{m \times m}$
- $h[t-1] \in \mathbb{R}^m$
- $W_h h[t-1] \in \mathbb{R}^m$

Both terms have dimension \mathbb{R}^m , which implies that $W_c x[t] + W_h h[t-1] \in \mathbb{R}^m$. Since σ is applied element-wise:

$$c[t] = \sigma(W_c x[t] + W_h h[t-1]) \in \mathbb{R}^m$$

We can verify the same with equation (2): $h[t] = c[t] \odot h[t-1] + (1 - c[t]) \odot W_x x[t]$. For the Hadamard products to be valid:

- $c[t] \odot h[t-1]$ requires $c[t]$ and $h[t-1]$ to have the same dimension

- Since $h[t-1] \in \mathbb{R}^m$, we must have $c[t] \in \mathbb{R}^m$
- Similarly, $(1 - c[t]) \odot W_x x[t]$ requires $(1 - c[t]) \in \mathbb{R}^m$ and $W_x x[t] \in \mathbb{R}^m$

Thus, the dimension of $c[t]$ is:

$$c[t] \in \mathbb{R}^m$$

Problem c

Solution

Since the loss ℓ is a scalar and $W_x \in \mathbb{R}^{m \times n}$, the gradient must have the same dimensions as W_x :

$$\frac{\partial \ell}{\partial W_x} \in \mathbb{R}^{m \times n}$$

From equation (2), W_x appears in the term $(1 - c[t]) \odot W_x x[t]$. We first compute the local gradient $\frac{\partial h[t]}{\partial W_x}$. For element $W_x[i, j]$, we have:

$$\frac{\partial h[t]_i}{\partial W_x[i, j]} = \frac{\partial}{\partial W_x[i, j]} \left(c[t]_i h[t-1]_i + (1 - c[t]_i) \sum_{k=1}^n W_x[i, k] x[t]_k \right)$$

The first term does not depend on W_x . For the second term, only the $k = j$ term in the sum contributes:

$$\frac{\partial h[t]_i}{\partial W_x[i, j]} = (1 - c[t]_i) \cdot x[t]_j$$

By the chain rule, summing over all time steps where W_x affects the output:

$$\frac{\partial \ell}{\partial W_x[i, j]} = \sum_{t=1}^K \frac{\partial \ell}{\partial h[t]_i} \cdot \frac{\partial h[t]_i}{\partial W_x[i, j]} = \sum_{t=1}^K \frac{\partial \ell}{\partial h[t]_i} \cdot (1 - c[t]_i) \cdot x[t]_j$$

In matrix form, this can be written as:

$$\frac{\partial \ell}{\partial W_x} = \sum_{t=1}^K \left(\frac{\partial \ell}{\partial h[t]} \right)^T \odot (1 - c[t]) x[t]^T$$

where the Hadamard product $(\frac{\partial \ell}{\partial h[t]})^T \odot (1 - c[t])$ gives a column vector in $\mathbb{R}^{m \times 1}$, and the outer product with $x[t]^T$ produces an $m \times n$ matrix.

Similarities between backward and forward pass:

Similarities:

- Both involve the gating term $(1 - c[t])$ which controls the contribution of the input
- Both use the input vectors $x[t]$
- Both accumulate information across time steps
- The same gate values that control information flow in the forward pass also appear in the backward pass

Forward pass (equation 2): Computes $h[t]$ by combining gated previous state and gated new input:

$$h[t] = c[t] \odot h[t-1] + (1 - c[t]) \odot W_x x[t]$$

Backward pass: Computes gradient by accumulating over time, using upstream gradients and the same gate values:

$$\frac{\partial \ell}{\partial W_x} = \sum_{t=1}^K \left(\frac{\partial \ell}{\partial h[t]} \right)^T \odot (1 - c[t]) x[t]^T$$

The key similarity is that the gate $(1 - c[t])$ plays the same role in both: *it modulates how much the input $x[t]$ contributes at each time step*, whether in the forward computation of $h[t]$ or in the backward computation of gradients.

Problem d

Solution

For an RNN, the gradient of the loss with respect to a hidden state $h[t]$ depends on the product of Jacobians along the sequence:

$$\frac{\partial \ell}{\partial h[t]} = \sum_{k=t}^K \frac{\partial \ell}{\partial h[k]} \frac{\partial h[k]}{\partial h[t]}, \quad \frac{\partial h[k]}{\partial h[t]} = \prod_{\tau=t+1}^k \frac{\partial h[\tau]}{\partial h[\tau-1]}.$$

This product can shrink or grow exponentially, leading to vanishing or exploding gradients. From equation (2), the derivative of $h[t]$ with respect to $h[t-1]$ is:

$$\frac{\partial h[t]}{\partial h[t-1]} = \underbrace{\text{diag}(c[t])}_{\text{direct path}} + \underbrace{\text{diag}(h[t-1] - W_x x[t]) \cdot \text{diag}(\sigma'(W_c x[t] + W_h h[t-1])) \cdot W_h}_{\text{indirect path through gate}}$$

where $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ is the derivative of the sigmoid, with $\sigma'(z) \in (0, 0.25]$ for all z .

Direct path: Differentiating the first term with respect to $h[t-1]$:

$$\frac{\partial(c[t] \odot h[t-1])}{\partial h[t-1]} = \text{diag}(c[t])$$

Indirect path through $c[t]$: The second and first terms both depend on $c[t]$, which depends on $h[t-1]$ through equation (1). We have:

$$\frac{\partial h[t]}{\partial c[t]} = h[t-1] - W_x x[t]$$

From equation (1), $c[t] = \sigma(W_c x[t] + W_h h[t-1])$:

$$\frac{\partial c[t]}{\partial h[t-1]} = \text{diag}(\sigma'(W_c x[t] + W_h h[t-1])) \cdot W_h$$

By the chain rule:

$$\text{diag}(h[t-1] - W_x x[t]) \cdot \frac{\partial c[t]}{\partial h[t-1]} = \text{diag}(h[t-1] - W_x x[t]) \cdot \text{diag}(\sigma'(\cdot)) \cdot W_h$$

The first term $\text{diag}(c[t])$ is elementwise multiplication by $c[t] \in (0, 1)$ and the second term involves W_h and the derivative of the sigmoid $\sigma'(z)$ which is usually small. This network is *less susceptible* to vanishing and exploding gradients compared to standard RNNs, but *not immune*.

- *Vanishing gradients*: Possible if
 - The gate values $c[t]$ are consistently small (near 0), making the direct path contribute little to gradient flow
 - Sigmoid saturation: when the argument to σ has large magnitude, $\sigma'(z) \approx 0$, blocking the indirect gradient path
 - The weight matrix W_h has small eigenvalues (spectral norm $\|W_h\| < 1$), and the gate values are not consistently near 1
 - The product $\prod_{\tau=t+1}^k \frac{\partial h[\tau]}{\partial h[\tau-1]}$ shrinks exponentially over long sequences
- *Exploding gradients*: Possible if
 - The weight matrix W_h has large eigenvalues (spectral norm $\|W_h\| \gg 1$), and the indirect path contributes significantly through the gate computation
 - The term $\text{diag}(h[t-1] - W_x x[t])$ has large magnitudes, amplifying gradients through the indirect path
 - The product of Jacobians $\prod_{\tau=t+1}^k \frac{\partial h[\tau]}{\partial h[\tau-1]}$ grows exponentially

Thus, this RNN behaves similarly to a standard RNN in this respect; the gating $c[t]$ can *mitigate* vanishing gradients (like in LSTM/GRU), but does not fully prevent it.

Yes, this network can suffer from vanishing or exploding gradients, but is less susceptible than standard RNNs.

Problem 1.2.2 Part 2

Problem a

Solution

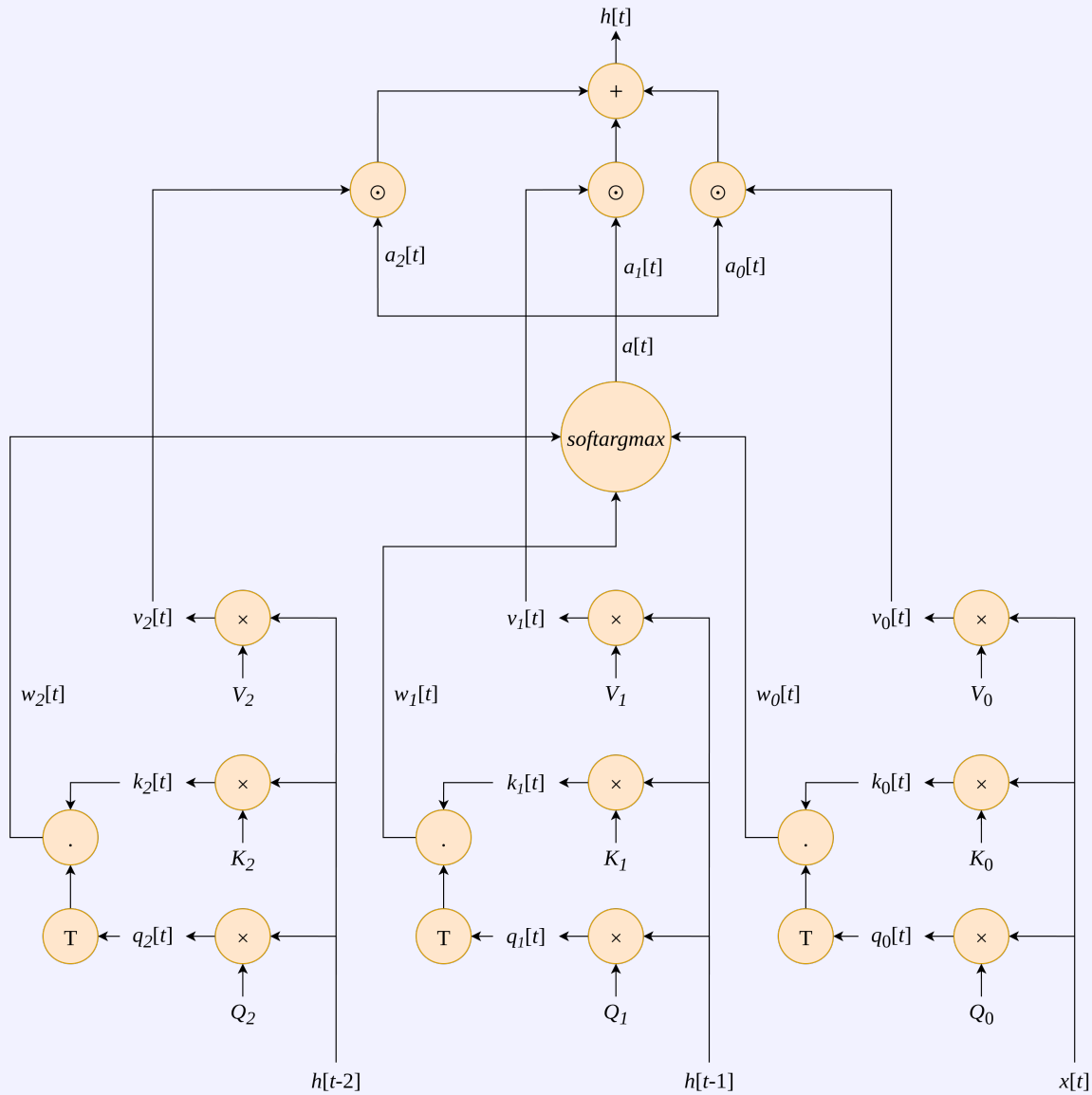


Figure 2: AttentionRNN(2) architecture showing the computation of hidden state $h[t]$ through attention over the current input $x[t]$ and previous hidden states $h[t-1]$, $h[t-2]$. Queries, keys, and values are linearly projected, and attention weights are computed via softmax over dot-product scores.

Problem b

Solution

From equation (7), we have $a[t] = \text{softargmax}([w_0[t], w_1[t], w_2[t]])$. The input to the softargmax function is a vector containing three scalar values: $w_0[t]$, $w_1[t]$, and $w_2[t]$. From equation (6), each $w_i[t]$ is computed as:

$$w_i[t] = q_i[t]^\top k_i[t]$$

Since $q_i[t], k_i[t] \in \mathbb{R}^n$, the dot product $q_i[t]^\top k_i[t]$ produces a scalar. Therefore:

$$w_i[t] \in \mathbb{R}, \quad i \in \{0, 1, 2\}$$

The softargmax function takes a vector of dimension d and outputs a probability distribution (a vector) of the same dimension d . Since the input to softargmax is:

$$[w_0[t], w_1[t], w_2[t]] \in \mathbb{R}^3$$

The output must be:

$$a[t] = [a_0[t], a_1[t], a_2[t]] \in \mathbb{R}^3$$

where $a_i[t] \geq 0$ and $\sum_{i=0}^2 a_i[t] = 1$. Therefore:

$$a[t] \in \mathbb{R}^3$$

Problem c

Solution

To extend AttentionRNN(2) to AttentionRNN(k), we generalize the network to attend over the current input $x[t]$ and the last k hidden states $h[t-1], h[t-2], \dots, h[t-k]$. This gives us $k+1$ query-key-value triplets (indexed from $i=0$ to $i=k$). The system of equations for AttentionRNN(k) is:

$$q_0[t], q_1[t], \dots, q_k[t] = Q_0 x[t], Q_1 h[t-1], \dots, Q_k h[t-k] \quad (3')$$

$$k_0[t], k_1[t], \dots, k_k[t] = K_0 x[t], K_1 h[t-1], \dots, K_k h[t-k] \quad (4')$$

$$v_0[t], v_1[t], \dots, v_k[t] = V_0 x[t], V_1 h[t-1], \dots, V_k h[t-k] \quad (5')$$

$$w_i[t] = q_i[t]^\top k_i[t] \quad \text{for } i = 0, 1, \dots, k \quad (6')$$

$$a[t] = \text{softargmax}([w_0[t], w_1[t], \dots, w_k[t]]) \quad (7')$$

$$h[t] = \sum_{i=0}^k a_i[t] v_i[t] \quad (8')$$

where $x[t], h[t] \in \mathbb{R}^n$, and $Q_i, K_i, V_i \in \mathbb{R}^{n \times n}$ for $i = 0, 1, \dots, k$. We define $h[t] = 0$ for $t < 1$.

Problem d

Solution

To extend AttentionRNN(k) to AttentionRNN(∞), we need the network to attend over the current input $x[t]$ and *all* past hidden states $h[t-1], h[t-2], \dots, h[1]$. Since we cannot have infinitely many parameter matrices, we use *weight sharing*: we use the same set of parameters for all past hidden states. We define two sets of parameters:

- (Q_0, K_0, V_0) for the current input $x[t]$
- (Q_h, K_h, V_h) for all past hidden states $h[t-i]$ where $i \geq 1$

The system of equations for AttentionRNN(∞) is:

$$q_0[t] = Q_0 x[t], \quad q_i[t] = Q_h h[t-i] \text{ for } i = 1, \dots, t \quad (3'')$$

$$k_0[t] = K_0 x[t], \quad k_i[t] = K_h h[t-i] \text{ for } i = 1, \dots, t \quad (4'')$$

$$v_0[t] = V_0 x[t], \quad v_i[t] = V_h h[t-i] \text{ for } i = 1, \dots, t \quad (5'')$$

$$w_i[t] = q_i[t]^\top k_i[t] \quad \text{for } i = 0, 1, \dots, t \quad (6'')$$

$$a[t] = \text{softmax}([w_0[t], w_1[t], \dots, w_t[t]]) \quad (7'')$$

$$h[t] = \sum_{i=0}^t a_i[t] v_i[t] \quad (8'')$$

where $x[t], h[t] \in \mathbb{R}^n$, and $Q_0, K_0, V_0, Q_h, K_h, V_h \in \mathbb{R}^{n \times n}$. We define $h[t] = 0$ for $t < 1$. All past hidden states $h[t-i]$ for $i \geq 1$ share the same parameters (Q_h, K_h, V_h) , rather than having separate parameters Q_i, K_i, V_i for each i .

Problem e

Solution

We know from equation (8):

$$h[t] = \sum_{i=0}^2 a_i[t] v_i[t] = a_0[t] v_0[t] + a_1[t] v_1[t] + a_2[t] v_2[t]$$

Applying the product rule to each term:

$$\frac{\partial h[t]}{\partial h[t-1]} = \sum_{i=0}^2 \left(v_i[t] \frac{\partial a_i[t]}{\partial h[t-1]} + a_i[t] \frac{\partial v_i[t]}{\partial h[t-1]} \right)$$

Compute $\frac{\partial v_i[t]}{\partial h[t-1]}$: From equations (3)-(5), only $v_1[t]$ depends directly on $h[t-1]$:

$$v_0[t] = V_0 x[t] \quad \Rightarrow \quad \frac{\partial v_0[t]}{\partial h[t-1]} = 0$$

$$v_1[t] = V_1 h[t-1] \quad \Rightarrow \quad \frac{\partial v_1[t]}{\partial h[t-1]} = V_1$$

$$v_2[t] = V_2 h[t-2] \quad \Rightarrow \quad \frac{\partial v_2[t]}{\partial h[t-1]} = 0$$

Compute $\frac{\partial a_i[t]}{\partial h[t-1]}$: From equation (7), $a[t] = \text{softmax}([w_0[t], w_1[t], w_2[t]])$. The derivative of softmax (or softargmax) is:

$$\frac{\partial a_i[t]}{\partial w_j[t]} = a_i[t] (\delta_{ij} - a_j[t])$$

where δ_{ij} is the Kronecker delta. By the chain rule:

$$\frac{\partial a_i[t]}{\partial h[t-1]} = \sum_{j=0}^2 \frac{\partial a_i[t]}{\partial w_j[t]} \frac{\partial w_j[t]}{\partial h[t-1]} = \sum_{j=0}^2 a_i[t] (\delta_{ij} - a_j[t]) \frac{\partial w_j[t]}{\partial h[t-1]}$$

Compute $\frac{\partial w_j[t]}{\partial h[t-1]}$: From equation (6), $w_j[t] = q_j[t]^\top k_j[t]$. Only $w_1[t]$ depends on $h[t-1]$:

$$\begin{aligned} w_0[t] &= q_0[t]^\top k_0[t] \Rightarrow \frac{\partial w_0[t]}{\partial h[t-1]} = 0 \\ w_1[t] &= q_1[t]^\top k_1[t] = (Q_1 h[t-1])^\top (K_1 h[t-1]) = h[t-1]^\top Q_1^\top K_1 h[t-1] \\ w_2[t] &= q_2[t]^\top k_2[t] \Rightarrow \frac{\partial w_2[t]}{\partial h[t-1]} = 0 \end{aligned}$$

For $w_1[t] = h[t-1]^\top Q_1^\top K_1 h[t-1]$, this is a quadratic form. Taking the derivative:

$$\frac{\partial w_1[t]}{\partial h[t-1]} = h[t-1]^\top (Q_1^\top K_1 + K_1^\top Q_1)$$

Combine results

Substituting into the expression for $\frac{\partial a_i[t]}{\partial h[t-1]}$:

$$\frac{\partial a_i[t]}{\partial h[t-1]} = a_i[t](\delta_{i1} - a_1[t]) \frac{\partial w_1[t]}{\partial h[t-1]} = a_i[t](\delta_{i1} - a_1[t]) h[t-1]^\top (Q_1^\top K_1 + K_1^\top Q_1)$$

Now, substituting back into the main expression:

$$\begin{aligned} \frac{\partial h[t]}{\partial h[t-1]} &= \sum_{i=0}^2 \left(v_i[t] \frac{\partial a_i[t]}{\partial h[t-1]} + a_i[t] \frac{\partial v_i[t]}{\partial h[t-1]} \right) \\ &= \sum_{i=0}^2 a_i[t](\delta_{i1} - a_1[t]) v_i[t] h[t-1]^\top (Q_1^\top K_1 + K_1^\top Q_1) + a_1[t] V_1 \\ &= \left[\sum_{i=0}^2 a_i[t](\delta_{i1} - a_1[t]) v_i[t] \right] h[t-1]^\top (Q_1^\top K_1 + K_1^\top Q_1) + a_1[t] V_1 \end{aligned}$$

Simplifying the sum:

$$\begin{aligned} \sum_{i=0}^2 a_i[t](\delta_{i1} - a_1[t]) v_i[t] &= a_1[t](1 - a_1[t]) v_1[t] - a_0[t] a_1[t] v_0[t] - a_2[t] a_1[t] v_2[t] \\ &= a_1[t] [(1 - a_1[t]) v_1[t] - a_0[t] v_0[t] - a_2[t] v_2[t]] \\ &= a_1[t] \left[v_1[t] - \sum_{j=0}^2 a_j[t] v_j[t] \right] \\ &= a_1[t] (v_1[t] - h[t]) \end{aligned}$$

Therefore:

$$\frac{\partial h[t]}{\partial h[t-1]} = a_1[t] (v_1[t] - h[t]) h[t-1]^\top (Q_1^\top K_1 + K_1^\top Q_1) + a_1[t] V_1$$

Problem f**Solution**

We need to find $\frac{\partial \ell}{\partial h[T]}$ for $\text{AttentionRNN}(k)$ given that we know $\frac{\partial \ell}{\partial h[t]}$ and $\frac{\partial h[t]}{\partial h[T]}$ for all $t > T$. By the chain rule, the gradient at time T receives contributions from all future time steps where $h[t]$ depends on $h[T]$:

$$\frac{\partial \ell}{\partial h[T]} = \left. \frac{\partial \ell}{\partial h[T]} \right|_{\text{direct}} + \sum_{t=T+1}^K \frac{\partial \ell}{\partial h[t]} \frac{\partial h[t]}{\partial h[T]}$$

where $\left. \frac{\partial \ell}{\partial h[T]} \right|_{\text{direct}}$ represents the direct contribution if the loss ℓ depends explicitly on $h[T]$.

In $\text{AttentionRNN}(k)$, each hidden state $h[t]$ depends on the current input $x[t]$ and the last k hidden states: $h[t-1], h[t-2], \dots, h[t-k]$. Therefore, $h[t]$ depends on $h[T]$ only if $h[T]$ is within the k -step attention window of time t . This occurs when $T \in \{t-k, t-k+1, \dots, t-1\}$, which is equivalent to $t \in \{T+1, T+2, \dots, T+k\}$. For $t > T+k$, we have $\frac{\partial h[t]}{\partial h[T]} = 0$ because $h[T]$ is more than k steps in the past and is outside the attention window, so it does not affect $h[t]$.

Since $\frac{\partial h[t]}{\partial h[T]} = 0$ for all $t > T+k$, the summation only needs to include terms where $h[t]$ actually depends on $h[T]$:

$$\sum_{t=T+1}^K \frac{\partial \ell}{\partial h[t]} \frac{\partial h[t]}{\partial h[T]} = \sum_{t=T+1}^{\min(T+k, K)} \frac{\partial \ell}{\partial h[t]} \frac{\partial h[t]}{\partial h[T]}$$

The upper limit $\min(T+k, K)$ accounts for two cases: if $T+k \leq K$, then the sum goes from $T+1$ to $T+k$ (all k future states within the window), and if $T+k > K$, then the sum goes from $T+1$ to K (only the remaining states until the end of the sequence).

Therefore:

$$\boxed{\frac{\partial \ell}{\partial h[T]} = \left. \frac{\partial \ell}{\partial h[T]} \right|_{\text{direct}} + \sum_{t=T+1}^{\min(T+k, K)} \frac{\partial \ell}{\partial h[t]} \frac{\partial h[t]}{\partial h[T]}}$$

Problem 1.3: Debugging Loss Curves

Problem 1

Solution

The spikes in the loss curve are caused by the fact that the plotted loss is the *loss of the last batch* in each epoch, rather than an average across all batches. The `train()` and `test()` functions return `loss.item()` from the last processed batch, making the epoch-level plotted value highly sensitive to single-batch outliers. Occasionally, a particular batch produces a very large loss due to several possible factors:

- The batch may contain hard sequences that are difficult to classify
- An unlucky class mix that doesn't represent the typical distribution
- Transient numeric instability such as gradient spikes during backpropagation
- The data generator creates new random sequences each epoch, so the difficulty and characteristics of the last batch vary significantly across epochs
- When the model has learned certain patterns confidently but encounters a batch where those patterns don't apply well, it may make confident but incorrect predictions, resulting in particularly high loss values

Thus, the plotted point represents only a *single snapshot rather than the overall epoch performance*. The variability in the difficulty of this last batch across different epochs creates the irregular spike pattern observed in the loss curve. The model's performance across all batches in an epoch is likely much smoother than what the plotted curve suggests. A proper implementation would average the loss across all batches within each epoch to provide a more representative and stable metric of training progress.

Problem 2

Solution

The loss spikes can exceed the initial loss value because of how `CrossEntropyLoss` *penalizes confident incorrect predictions more severely than random guessing*. Since the plotted loss represents only the last batch of each epoch rather than the average across all batches, we can observe extreme values when that particular batch happens to be unusually difficult or unrepresentative of the overall dataset.

At initialization (epoch 0), the model makes essentially random predictions. For a 4-class classification problem, a random classifier assigns probability ≈ 0.25 to each class. The expected `CrossEntropyLoss` for the correct class is:

$$\text{Loss}_{\text{initial}} = -\log(0.25) = \log(4) \approx 1.386$$

This matches the observed initial loss of approximately 1.4 in the plot. However, as training progresses, the model learns patterns and becomes increasingly confident in its predictions. When the model encounters a batch where its learned patterns don't apply well (for instance, batches containing unusually difficult sequences, rare patterns the model hasn't seen often, or a class distribution very different from the typical training data), it may *assign very high probability to an incorrect class and very low probability to the correct class*. For example, if the model predicts the wrong class with 95% confidence, the loss becomes:

$$\text{Loss}_{\text{confident wrong}} = -\log(0.05) \approx 3.0$$

If the predicted probability for the correct class is even lower (say, 0.01), the loss can spike to:

$$\text{Loss}_{\text{very confident wrong}} = -\log(0.01) \approx 4.6$$

This explains why the spikes reach values of 3-4, significantly higher than the initial loss: a confident but incorrect prediction produces much larger loss than a uniformly uncertain (random) prediction. The `CrossEntropyLoss` function *penalizes overconfident mistakes severely*, which is exactly what we observe in these spikes when the last batch happens to contain sequences that violate the model's learned patterns.

Problem 3

Solution

The spikes in the loss curve can be fixed by modifying how loss is tracked and reported during training. The primary issue is that the current implementation only records the loss from the last batch of each epoch, which creates high variance and misleading visualizations.

We should update the `train()` and `test()` functions to accumulate loss values across all batches within an epoch and return the average. Currently, only the loss from the final batch is returned. Instead, we should initialize a variable to accumulate the total loss at the start of each epoch, and after computing the loss for each batch, add `loss.item()` to the running total. After processing all batches, divide the accumulated total by the number of batches to compute the average loss and return this average loss. This ensures the plotted loss represents the model's average performance across the entire epoch rather than a single, potentially unrepresentative batch. The same modification should be applied to the `test()` function for consistency.

We can also make additional improvements such as:

- *Gradient clipping*: Add gradient norm clipping before the optimizer step to prevent gradient explosions that can destabilize training. This caps the magnitude of gradients and reduces sensitivity to outlier batches.
- *Learning rate scheduling*: Implement learning rate decay to reduce the learning rate when training plateaus. This helps the model converge more smoothly and reduces large parameter updates that can cause loss spikes.
- *Increasing batch size*: Larger batches provide more stable gradient estimates and reduce the variance in loss values across batches.

Problem 4

Solution

Before training starts (epoch 0), the model has randomly initialized weights and has not learned any patterns from the data. Therefore, it makes essentially random predictions.

Initial Loss Value:

The task is a 4-class sequence classification problem with classes Q, R, S, and U, as defined in the notebook. The model uses `CrossEntropyLoss` as the loss function. For a randomly initialized model, the output logits are approximately uniform, meaning after the softmax operation, each class receives roughly equal probability. For the correct class, the model assigns probability $\approx 1/4 = 0.25$.

The **CrossEntropyLoss** for a single prediction is defined as:

$$\text{Loss} = -\log(p_{\text{correct}})$$

where p_{correct} is the predicted probability for the correct class. With random initialization and uniform predictions:

$$\text{Loss}_{\text{initial}} = -\log(0.25) = -\log(1/4) = \log(4) \approx 1.386$$

Observing the left plot (Loss vs Epoch), the initial loss at epoch 0 is approximately 1.4, which matches this theoretical expectation for a 4-class classification problem with random predictions.

Initial Accuracy Value:

For a 4-class classification problem, a model making random predictions has an equal probability of selecting any of the four classes. The probability of randomly guessing the correct class is:

$$\text{Accuracy}_{\text{initial}} = \frac{1}{4} = 0.25 = 25\%$$

Observing the right plot (Accuracy vs Epoch), the initial accuracy at epoch 0 is approximately 25%, confirming that the untrained model performs at chance level, randomly selecting among the four possible classes.

Thus, the initial values reflect random chance performance before any learning has occurred:

- *Loss* ≈ 1.4 : Corresponds to $-\log(0.25) = \log(4) \approx 1.386$, the expected cross-entropy loss when uniformly guessing among 4 classes
- *Accuracy* $\approx 25\%$: Corresponds to $1/4$, the probability of randomly selecting the correct class out of 4 possible classes

The observed values may differ slightly from theoretical expectations due to random weight initialization and batch sampling variability.