CSE 5819 Assignment #7

by: Aayushi Verma (uef24001)

This assignment is organized as follows:

1 Part 1: ChatGPT Self-Learning (20pts)

You can start from the following prompts, but you need to create subsequent questions that attempt to understand the basic concepts of machine learning.

1.1 What is a recurrent neural network?

A recurrent neural network (RNN) is a type of artificial neural network designed for sequence data. Unlike feedforward neural networks, RNNs have connections that form directed cycles, allowing information to persist over time. This makes them particularly suited for tasks where context or sequential information is important, such as time series prediction, natural language processing, and speech recognition.

Mathematically, an RNN processes a sequence of inputs $x_1, x_2, \ldots, x_T$ by maintaining a hidden state h_t at each time step t. The hidden state is updated as follows:

$$h_t = f(W_h h_{t-1} + W_x x_t + b)$$

where:

- h_t is the hidden state at time t,
- h_{t-1} is the hidden state from the previous time step,
- x_t is the input at time t,
- W_h and W_x are weight matrices,
- b is a bias vector,
- f is a non-linear activation function, such as `tanh` or `ReLU`.

RNNs can suffer from issues like vanishing and exploding gradients during training, which can be addressed using variants like Long Short-Term Memory (LSTM) networks or Gated Recurrent Units (GRUs).

1.2 How many kinds of recurrent neural networks?

Recurrent neural networks (RNNs) can be categorized into several types based on how they process sequences and the structure of their connections. The main kinds of RNNs are:

- **Vanilla RNNs:** The basic form of RNNs where each neuron uses simple activation functions like `tanh` or `ReLU`. They are suitable for simple sequence modeling but struggle with long-term dependencies due to vanishing gradients.
- **Long Short-Term Memory (LSTM):** A type of RNN that uses special gating mechanisms (input, forget, and output gates) to retain information over longer sequences. LSTMs mitigate the vanishing gradient problem.
- **Gated Recurrent Units (GRU):** A simplified version of LSTMs with fewer gates (update and reset gates). GRUs are computationally efficient and effective for many tasks.
- **Bidirectional RNNs:** RNNs that process input sequences in both forward and backward directions, enabling the use of future context as well as past context. This is particularly useful for tasks like text translation or speech recognition.
- **Sequence-to-Sequence (Seq2Seq) RNNs:** RNN architectures with an encoder-decoder structure. The encoder compresses the input sequence into a fixed representation, which the decoder then uses to generate an output sequence. These are common in machine translation and text summarization.
- **Clockwork RNNs:** A variation of RNNs where different neurons operate at different time scales, allowing efficient modeling of long sequences with varying time dependencies.
- **Attention-Based RNNs:** RNNs augmented with attention mechanisms to selectively focus on specific parts of the input sequence, improving performance on tasks with long input sequences.

Each type is designed to address specific limitations or enhance performance in certain sequence modeling tasks.

1.3 What are issues with recurrent neural networks?

Recurrent neural networks (RNNs) face several challenges that can impact their performance and usability in sequence modeling tasks. The main issues include:

- **Vanishing Gradient Problem:** During backpropagation through time (BPTT), gradients can become extremely small, especially for long sequences. This makes it difficult for the RNN to learn long-term dependencies as the gradients effectively vanish.
- **Exploding Gradient Problem:** Conversely, gradients can grow exponentially large during training, causing numerical instability and making the optimization process unreliable. Gradient clipping is often used to address this issue.
- **Difficulty with Long-Term Dependencies:** RNNs struggle to capture dependencies between distant elements in a sequence. This limits their effectiveness for tasks requiring long-term memory.
- **Computational Inefficiency:** RNNs process sequences sequentially, which limits parallelization during training and inference. This can make them slower compared to other models like transformers.
- **Overfitting:** RNNs have a high capacity to model sequential data, making them prone to overfitting, especially when the dataset is small or noisy.
- **Sensitivity to Input Length:** RNNs may not generalize well to sequences of lengths significantly different from those seen during training.
- **Difficulty in Training:** Training RNNs can be challenging due to their complex architecture and issues like gradient instability. They also require careful initialization and hyperparameter tuning.
- **Loss of Context in Vanilla RNNs:** Standard RNNs can forget important context over long sequences, which can reduce their effectiveness on complex tasks. Variants like LSTMs and GRUs help mitigate this issue but add computational complexity.

These challenges have led to the development of RNN variants (e.g., LSTMs, GRUs) and alternative architectures like transformers, which are better suited for handling long-range dependencies and parallel computation.

1.4 What is a gated recurrent unit?

A Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem and improve the efficiency of modeling long-term dependencies in sequence data. GRUs simplify the structure of Long Short-Term Memory (LSTM) units by using fewer gates while maintaining comparable performance.

The GRU uses two primary gates:

- **Update Gate:** Determines how much of the past information needs to be passed along to the future.
- **Reset Gate:** Decides how much of the past information to forget.

The mathematical operations for a GRU at time step t are as follows:

$$
\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\tilde{h}_t &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

where:

- z_t is the update gate vector,
- r_t is the reset gate vector,
- $\tilde{h}_t$ is the candidate hidden state,
- h_t is the final hidden state,

- σ is the sigmoid activation function,
- $\odot$ represents element-wise multiplication,
- W_z, W_r, W_h and U_z, U_r, U_h are weight matrices,
- b_z, b_r, b_h are bias vectors,
- x_t is the input at time t,
- h_{t-1} is the hidden state from the previous time step.

GRUs are computationally efficient and perform well on tasks requiring long-term memory, making them popular for applications like speech recognition, time-series analysis, and natural language processing.

1.5 What is a Long Short-Term Memory (LSTM) network?

A Long Short-Term Memory (LSTM) network is a specialized type of recurrent neural network (RNN) designed to effectively learn long-term dependencies in sequence data. LSTMs address the vanishing gradient problem that traditional RNNs face, enabling them to retain information over long sequences.

The LSTM achieves this by using a memory cell and three gates:

- **Forget Gate:** Decides what information to discard from the cell state.
- **Input Gate:** Determines what new information to add to the cell state.
- **Output Gate:** Controls what information to output from the cell state to the hidden state.

The mathematical operations for an LSTM at time step t are as follows:

$$\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

where:

- f_t is the forget gate vector,
- i_t is the input gate vector,
- $\tilde{c}_t$ is the candidate cell state,
- c_t is the updated cell state,
- o_t is the output gate vector,
- h_t is the hidden state,
- σ is the sigmoid activation function,
- tanh is the hyperbolic tangent activation function,
- $\odot$ represents element-wise multiplication,
- W_f, W_i, W_c, W_o and U_f, U_i, U_c, U_o are weight matrices,
- b_f, b_i, b_c, b_o are bias vectors,
- x_t is the input at time t,
- h_{t-1} and c_{t-1} are the hidden state and cell state from the previous time step.

LSTMs are widely used in applications such as natural language processing, time series forecasting, speech recognition, and video analysis due to their ability to model long-term dependencies effectively.

1.6 What is an autoencoder?

An autoencoder is a type of artificial neural network designed to learn an efficient, compressed representation of data, typically for dimensionality reduction or feature extraction. It consists of two main components:

- **Encoder:** Maps the input data x to a latent-space representation z, typically of lower dimensionality, using one or more neural network layers.
- **Decoder:** Reconstructs the input data x from the latent representation z, attempting to minimize the reconstruction error.

Mathematically, the autoencoder can be represented as:

$$z = f_{\text{encoder}}(x) = \sigma(W_e x + b_e)$$
$$\hat{x} = f_{\text{decoder}}(z) = \sigma(W_d z + b_d)$$

where:

- x is the input data,
- z is the latent representation,
- $\hat{x}$ is the reconstructed data,
- W_e, b_e are the weights and biases of the encoder,
- W_d, b_d are the weights and biases of the decoder,
- σ is an activation function, such as `ReLU` or `sigmoid`.

The goal of training an autoencoder is to minimize the reconstruction loss, typically using a metric like mean squared error (MSE):

$$\mathcal{L} = \|x - \hat{x}\|^2$$

Autoencoders come in various types, including:

- **Denoising Autoencoders:** Train the network to reconstruct clean input data from noisy data.
- **Sparse Autoencoders:** Encourage sparsity in the latent representation z for feature selection.
- **Variational Autoencoders (VAEs):** Use probabilistic techniques to learn a latent space with a structured distribution.
- **Convolutional Autoencoders:** Use convolutional layers for image data to learn spatially meaningful latent representations.

Autoencoders are widely used in applications such as anomaly detection, image denoising, data compression, and as a pretraining step for other machine learning models.

1.7 How many kinds of autoencoders?

Autoencoders come in various types, each designed to address specific tasks or constraints. The main kinds of autoencoders include:

- **Vanilla Autoencoder:** A basic autoencoder with fully connected layers in the encoder and decoder. It is primarily used for dimensionality reduction or reconstruction tasks.
- **Denoising Autoencoder:** Trains the autoencoder to reconstruct clean input data from a noisy version of the data, making it effective for tasks like image denoising or data cleaning.
- **Sparse Autoencoder:** Encourages sparsity in the latent representation by adding a sparsity penalty to the loss function. It is useful for feature extraction and selection.
- **Contractive Autoencoder:** Adds a penalty to the loss function to make the latent space more robust to small changes in the input, improving its generalization ability.

- **Variational Autoencoder (VAE):** Uses a probabilistic framework to learn a structured latent space, enabling generation of new data samples by sampling from the latent distribution.
- **Convolutional Autoencoder:** Uses convolutional layers instead of fully connected layers, making it well-suited for image data. It learns spatially meaningful latent representations.
- **Stacked Autoencoder:** Combines multiple autoencoders in a layered manner, where each layer's output serves as the input to the next. This deep architecture enables learning hierarchical features.
- **Undercomplete Autoencoder:** Has a bottleneck in the architecture where the latent representation is of lower dimensionality than the input, forcing the network to learn efficient data representations.
- **Overcomplete Autoencoder:** Has a latent representation with a higher dimensionality than the input, often paired with regularization techniques to avoid simply copying the input to the output.
- **Multimodal Autoencoder:** Trained on data from multiple modalities (e.g., images and text) to learn a joint representation across different types of data.
- **Generative Autoencoder:** Combines autoencoder principles with generative models to produce new samples from the latent space, often used in applications like image generation.

Each type of autoencoder is tailored to specific tasks, making autoencoders a versatile tool in deep learning for applications like data compression, anomaly detection, feature extraction, and generative modeling.

1.8 What is a convolutional neural network?

A Convolutional Neural Network (CNN) is a specialized type of deep neural network designed for processing data with a grid-like topology, such as images. CNNs are particularly effective in tasks like image classification, object detection, and segmentation, as they can automatically and adaptively learn spatial hierarchies of features from data.

A CNN consists of the following key components:

- **Convolutional Layers:** Apply convolution operations to extract local features from the input data. A convolution operation involves sliding a filter (or kernel) over the input and computing element-wise multiplications and summations to produce a feature map. Mathematically, the convolution operation can be expressed as:

  $$(f * g)(x, y) = \sum_i \sum_j f(i, j) \cdot g(x - i, y - j)$$

  where f is the filter, g is the input, and (x, y) denotes spatial positions.
- **Activation Functions:** Apply non-linear functions such as `ReLU` (Rectified Linear Unit) to introduce non-linearity, allowing the network to model complex relationships.
- **Pooling Layers:** Reduce the spatial dimensions of feature maps, making the network computationally efficient and robust to small variations in the input. Common pooling techniques include max pooling and average pooling.
- **Fully Connected Layers:** Connect all neurons in the previous layer to each neuron in the current layer, combining the extracted features for making predictions.
- **Dropout Layers:** Introduce regularization by randomly setting a fraction of the neurons' outputs to zero during training, preventing overfitting.
- **Softmax Layer:** Used in the output layer for multi-class classification problems, converting the raw scores into probabilities.

The key advantages of CNNs include:

- **Local Connectivity:** Leverages the spatial structure of data, allowing efficient feature extraction.
- **Parameter Sharing:** Reuses the same filter across different regions of the input, reducing the number of parameters and computational cost.
- **Translation Invariance:** Detects features regardless of their position in the input.

CNNs are widely used in applications such as image recognition, video analysis, medical imaging, natural language processing (with 1D convolutions), and more.

1.9 What is the convolution operator?

The convolution operator is a mathematical operation used to extract features from an input by combining it with a filter (or kernel). It is widely used in convolutional neural networks (CNNs) to process data with spatial or temporal structure, such as images or time-series data.

Mathematically, for a 2D input $I(x, y)$ and a filter $K(i, j)$, the convolution operation is defined as:

$$(I * K)(x, y) = \sum_i \sum_j K(i, j) \cdot I(x - i, y - j)$$

Here:

- $I(x, y)$: Input data (e.g., an image),
- $K(i, j)$: Filter (or kernel), a small matrix of learnable weights,
- x, y: Spatial coordinates of the output,
- i, j: Spatial indices of the filter.

In the context of CNNs, the operation is often implemented using the following steps:

- Slide the filter K over the input I at every position (x, y).
- Compute the element-wise product between the filter and the corresponding region of the input.
- Sum the resulting values to produce a single output value for position (x, y).

Properties of the Convolution Operator:

- **Local Receptive Field:** The filter focuses on a small region of the input at each step, capturing local patterns.
- **Parameter Sharing:** The same filter is applied across the entire input, reducing the number of learnable parameters.
- **Translation Invariance:** The operation detects features regardless of their position in the input.

Variants in Practice:

- **Strided Convolution:** Introduces a step size (stride) for moving the filter across the input, controlling the resolution of the output.
- **Padding:** Adds a border of zeros around the input to preserve spatial dimensions or avoid losing edge information.
- **1D and 3D Convolution:** Used for processing 1D signals (e.g., time-series data) or 3D inputs (e.g., video data) respectively.

The convolution operator is central to CNNs, enabling the detection of spatially meaningful features such as edges, textures, and patterns in data.

1.10 What does the output of an encoder look like?

The output of an encoder in a neural network, particularly in the context of autoencoders, sequence-to-sequence models, or transformers, typically consists of a lower-dimensional, compressed representation of the input data. This output is often referred to as the "latent space" or "encoded representation." The key characteristics of the encoder's output are as follows:

- **Lower Dimensionality:** The encoder maps the input data to a representation with fewer dimensions than the input. For example, an image of size $256 \times 256 \times 3$ may be encoded into a vector of size 128 or smaller.
- **Abstract Features:** The encoder's output captures high-level, abstract features of the input data, depending on the task. For example, in an image encoder, it may represent visual patterns, textures, or objects in a compressed form.

- **Vector or Tensor Representation:** The output is often a vector (1D) or a tensor (2D or higher), depending on the architecture. In the case of an autoencoder, the latent representation may be a flat vector, whereas in models like convolutional autoencoders, it could be a 3D tensor with spatial structure preserved.
- **Encoded Representation (Latent Space):** The encoder attempts to preserve essential features of the input in a compact form. This representation is used by the decoder (in autoencoders) or other parts of the network (in sequence models or transformers) to reconstruct the original input or generate an output based on the learned features.
- **Probabilistic Output (in Variational Autoencoders):** In the case of variational autoencoders (VAEs), the encoder produces parameters of a probability distribution (mean and variance) rather than a deterministic representation, allowing for the generation of diverse samples by sampling from the latent distribution.

Mathematically, the output of an encoder can be expressed as:

$$z = f_{\text{encoder}}(x)$$

where:

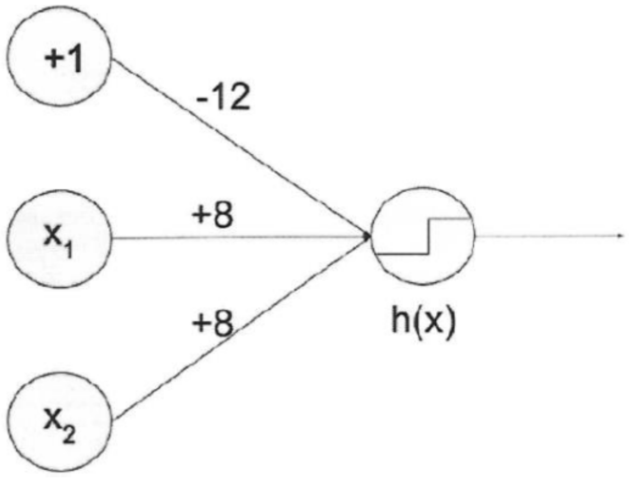
- x is the input data (e.g., an image or a sequence),
- z is the latent representation (the output of the encoder),
- f_{encoder} is the encoding function.

In summary, the output of an encoder is a compact, abstract representation of the input data that captures its most salient features, often in a lower-dimensional space, and can be used for reconstruction, classification, or other tasks depending on the model.

2 Problems

2.1 Question One

[logic neural network] [10 pts] You are given the following neural network which takes two binary valued inputs x1, x2, and the activation function is the threshold function (or indicator function, h(x) = 1 if x¿0; or 0 otherwise). Which of the following logic function does it compute, OR, AND, Negative AND, or XOR? Please provide your argument (draw a truth table).



We are given $h(x)$, which outputs the following:

$$\begin{aligned} z &= w \cdot x + b \\ &= 1 \cdot 12 + x_1 \cdot 8 + x_2 \cdot 8 \\ &= -12 + 8x_1 + 8x_2 \end{aligned} \tag{1}$$

Then according to the condition of $h(x)$, we obtain the following truth table for each potential value {0, 1} for x_1 and x_2:

x_1	x_2	z	$h(x)$
0	0	-12	0
0	1	-4	0
1	0	-4	0
1	1	4	1

Since the only time $h(x) = 1$ is when both x_1 and x_2 are 1, this corresponds to the AND logic function.

2.2 Question Two

[Forward and backward calculation] We have a function which takes a two-dimensional input $\mathbf{x} = (x_1, x_2)$ and has two parameters $\mathbf{w} = (w_1, w_2)$ given by:

$$f(\mathbf{x}; \mathbf{w}) = \sigma(\sigma(x_1 w_1) \cdot w_2 + x_2)$$

where

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

We use backpropagation to estimate the right parameter values. We start by setting the parameters $w_1 = 1$, $w_2 = 2$. Assume that we are given a training point $x_1 = 1$, $x_2 = 0$, and $y = 5$.

(2a) [10 pts] Please calculate the prediction for the point $x_1 = 1$, $x_2 = 0$ given the current weight values of the model.

We can re-write the $f(\mathbf{x}; \mathbf{w})$ to

$$o_1 = \sigma(x_1 w_1)$$

$$o_2 = \sigma(o_1 w_2 + x_2)$$

and output o_2.
Substituting the values, we get:

$$\begin{aligned} o_1 &= \sigma(x_1 w_1) \\ &= \sigma(1 \cdot 1) \\ &= \frac{1}{1+e^{-1}} \\ &\approx 0.7311 \end{aligned} \tag{2}$$

$$\begin{aligned} o_2 &= \sigma(o_1 w_2 + x_2) \\ &= \sigma(0.7311 \cdot 2 + 0) \\ &= \sigma(1.4622) \\ &= \frac{1}{1+e^{-1.4622}} \\ &\approx 0.8119 \end{aligned} \tag{3}$$

$\therefore o_2 \approx 0.8119$.

(2b) [10 pts] Please find the value of the gradient of the squared loss $E(\mathbf{w}) = (y^* - y)^2$ where $y^* = o_2$ is the output (or prediction) of the model.

Note that the gradient contains two entries

$$\left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}\right].$$

We start by first finding $\frac{\partial E}{\partial o_2}$:

$$\begin{aligned} \frac{\partial E}{\partial o_2} &= 2(o_2 - y) \\ &= 2(0.8119 - 5) \\ &= -8.3762 \end{aligned} \tag{4}$$

We then find $\frac{\partial E}{\partial w_2}$ using the chain rule:

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial o_2}\frac{\partial o_2}{\partial w_2} \tag{5}$$

We first find $\frac{\partial o_2}{\partial w_2}$:

$$\frac{\partial o_2}{\partial w_2} = \sigma'(o_1 w_2 + x_2) \cdot o_1 \tag{6}$$

where $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, and $z = o_1 w_2 + x_2 = 1.4622$. Then we have:

$$\begin{aligned} \frac{\partial o_2}{\partial w_2} &= \left(\frac{1}{1+e^{-z}}\Big(1 - \frac{1}{1+e^{-z}}\Big)\right) \cdot o_1 \\ &= \left(\frac{1}{1+e^{-1.4622}}\Big(1 - \frac{1}{1+e^{-1.4622}}\Big)\right) \cdot 0.8119 \\ &= 0.1240 \end{aligned} \tag{7}$$

Subbing this back into Eq. 5:

$$\begin{aligned} \frac{\partial E}{\partial w_2} &= \frac{\partial E}{\partial o_2} \cdot \frac{\partial o_2}{\partial w_2} \\ &= -8.3762 \cdot 0.1240 \\ &= -1.0386 \end{aligned} \tag{8}$$

We now find the gradient with respect to w_1:

$$\begin{aligned}\frac{\partial o_2}{\partial o_1} &= w_2(1-o_2)o_2 \\ &= 2\cdot(1-0.8119)\cdot 0.8119 \\ &\approx 0.3054 \end{aligned} \tag{9}$$

$$\begin{aligned}\frac{\partial o_1}{\partial w_1} &= x_1(1-o_1)o_1 \\ &= 1\cdot(1-0.7311)\cdot 0.7311 \\ &\approx 0.1966 \end{aligned} \tag{10}$$

$$\begin{aligned}\frac{\partial E}{\partial w_1} &= \frac{\partial E}{\partial o_2}\cdot\frac{\partial o_2}{\partial o_1}\cdot\frac{\partial o_1}{\partial w_1} \\ &= -8.3762\cdot 0.3054\cdot 0.1966 \\ &\approx -0.5029 \end{aligned} \tag{11}$$

Therefore the final gradient with respect to **w** is:

$$\begin{aligned}\nabla_{\mathrm{w}} E &= \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}\right] \\ &= [-0.5029, -1.0386] \end{aligned} \tag{12}$$

2.3 Long Short-term Memory (LSTM) [20 pts]

We are given two data points with 2 different timesteps.

At the timestamp $t = 1$, we have data point $(x_{1,1}, x_{1,2}, y_1) = (0.3, 0.6, 0.2)$.

At the timestamp $t = 2$, we have data point $(x_{2,1}, x_{2,2}, y_2) = (0.1, 1.0, 0.4)$.

Here $x_{t,1}$ and $x_{t,2}$ are 2 input variables, y_t is the output variable, t is the time step.

Consider the traditional LSTM model. Initially, we have the following internal weight vectors and bias variables as follows:

$$\mathbf{W}_f = \begin{pmatrix}0.8\\0.4\\0.1\end{pmatrix}, \quad \mathbf{b}_f = 0.2$$

$$\mathbf{W}_i = \begin{pmatrix}0.9\\0.8\\0.7\end{pmatrix}, \quad \mathbf{b}_i = 0.5$$

$$\mathbf{W}_a = \begin{pmatrix}0.4\\0.2\\0.1\end{pmatrix}, \quad \mathbf{b}_a = 0.3$$

$$\mathbf{W}_o = \begin{pmatrix}0.6\\0.4\\0.1\end{pmatrix}, \quad \mathbf{b}_o = 0.2$$

In the model, we have the following gate variables. For each $t = 1, 2, \ldots$:

- Forget gate variable (weight) f_t;
- Input gate variable (weight) i_t;
- Candidate cell state $\tilde{C}_t$;
- Cell state variable C_t;
- Output gate variable o_t;
- Final output variable $\tilde{y}_t$ (this may have a different notation from our lecture slides).

Suppose that $y_0 = s_0 = 0$ and $C_0 = 0$.

Consider the input forward propagation step only.

(3.a) [10 pts]

What are the values of the above gate variables (1 – 6) when $t = 1$ and when $t = 2$? Please show each answer up to 4 decimal places.

(3.a) [10 pts] Answer

We compute the values of the gate variables for each timestep $t = 1$ and $t = 2$. The formulae for the gate variables are:

$$
\begin{aligned}
f_t &= \sigma(\mathbf{W}_f^\top \mathbf{x}_t + \mathbf{b}_f) \\
i_t &= \sigma(\mathbf{W}_i^\top \mathbf{x}_t + \mathbf{b}_i) \\
\tilde{C}_t &= \tanh(\mathbf{W}_a^\top \mathbf{x}_t + \mathbf{b}_a) \\
C_t &= f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \\
o_t &= \sigma(\mathbf{W}_o^\top \mathbf{x}_t + \mathbf{b}_o) \\
\tilde{y}_t &= o_t \cdot \tanh(C_t)
\end{aligned}
$$

For the first timestep $t = 1$, we have:

$$
\begin{aligned}
f_1 &= \sigma(0.8 \cdot 0.3 + 0.4 \cdot 0.6 + 0.1 \cdot 0 + 0.2) \\
&= \sigma(0.24 + 0.24 + 0 + 0.2) \\
&= \sigma(0.68) \\
&= 0.6637 \qquad (13)
\end{aligned}
$$

$$
\begin{aligned}
i_1 &= \sigma(0.9 \cdot 0.3 + 0.8 \cdot 0.6 + 0.7 \cdot 0 + 0.5) \\
&= \sigma(0.27 + 0.48 + 0 + 0.5) \\
&= \sigma(1.25) \\
&= 0.7773 \qquad (14)
\end{aligned}
$$

$$
\begin{aligned}
\tilde{C}_1 &= \tanh(0.4 \cdot 0.3 + 0.2 \cdot 0.6 + 0.1 \cdot 0 + 0.3) \\
&= \tanh(0.12 + 0.12 + 0 + 0.3) \\
&= \tanh(0.54) \\
&= 0.4930 \qquad (15)
\end{aligned}
$$

$$
\begin{aligned}
C_1 &= f_1 \cdot C_0 + i_1 \cdot \tilde{C}_1 \\
&= 0.6637 \cdot 0 + 0.7773 \cdot 0.4930 \\
&= 0.3831 \qquad (16)
\end{aligned}
$$

$$
\begin{aligned}
o_1 &= \sigma(0.6 \cdot 0.3 + 0.4 \cdot 0.6 + 0.1 \cdot 0 + 0.2) \\
&= \sigma(0.18 + 0.24 + 0 + 0.2) \\
&= \sigma(0.62) \\
&= 0.6508 \qquad (17)
\end{aligned}
$$

$$
\begin{aligned}
\tilde{y}_1 &= o_1 \cdot \tanh(C_1) \\
&= 0.6508 \cdot \tanh(0.3831) \\
&= 0.6508 \cdot 0.3651 \\
&= 0.2377 \qquad (18)
\end{aligned}
$$

For the second timestep $t = 2$, we have:

$$
\begin{aligned}
f_2 &= \sigma(0.8 \cdot 0.1 + 0.4 \cdot 1.0 + 0.1 \cdot 0 + 0.2) \\
&= \sigma(0.08 + 0.4 + 0 + 0.2) \\
&= \sigma(0.68) \\
&= 0.6637 \qquad (19)
\end{aligned}
$$

$$\begin{aligned} i_2 &= \sigma(0.9 \cdot 0.1 + 0.8 \cdot 1.0 + 0.7 \cdot 0 + 0.5) \\ &= \sigma(0.09 + 0.8 + 0 + 0.5) \\ &= \sigma(1.39) \\ &= 0.8006 \end{aligned} \tag{20}$$

$$\begin{aligned} \tilde{C}_2 &= \tanh(0.4 \cdot 0.1 + 0.2 \cdot 1.0 + 0.1 \cdot 0 + 0.3) \\ &= \tanh(0.04 + 0.2 + 0 + 0.3) \\ &= \tanh(0.54) \\ &= 0.4930 \end{aligned} \tag{21}$$

$$\begin{aligned} C_2 &= f_2 \cdot C_1 + i_2 \cdot \tilde{C}_2 \\ &= 0.6637 \cdot 0.3831 + 0.8006 \cdot 0.4930 \\ &= 0.2542 + 0.3945 \\ &= 0.6487 \end{aligned} \tag{22}$$

$$\begin{aligned} o_2 &= \sigma(0.6 \cdot 0.1 + 0.4 \cdot 1.0 + 0.1 \cdot 0 + 0.2) \\ &= \sigma(0.06 + 0.4 + 0 + 0.2) \\ &= \sigma(0.66) \\ &= 0.6590 \end{aligned} \tag{23}$$

$$\begin{aligned} \tilde{y}_2 &= o_2 \cdot \tanh(C_2) \\ &= 0.6590 \cdot \tanh(0.6487) \\ &= 0.6590 \cdot 0.5713 \\ &= 0.3765 \end{aligned} \tag{24}$$

(3.b) [10 pts]

What are the error values (in terms of absolute error between true value y_t and prediction $\tilde{y}_t$) of the final output variables when $t = 1$ and $t = 2$? Please show each answer up to 4 decimal places.

For each time step t, we have the absolute errors:

$$\begin{aligned} \text{Error at } t = 1 &= |y_1 - \tilde{y}_1| \\ &= |0.2 - 0.2377| \\ &= 0.0377 \end{aligned} \tag{25}$$

$$\begin{aligned} \text{Error at } t = 2 &= |y_2 - \tilde{y}_2| \\ &= |0.4 - 0.3765| \\ &= 0.0235 \end{aligned} \tag{26}$$