Machine Learning & Deep Learning

Tutorial

MSCV/ESIREM

Antoine Lavault
antoine.lavault@u-bourgogne.fr

RNN

Any question or exercise marked with a "*" is typically more technical or goes further into developing the tools and notions seen during class.

***

Problem 1

Basic Concepts

1. Draw a diagram of an RNN.
2. Draw a diagram for an LSTM.
3. Draw a diagram for a GRU.
4. What is the concept of attention in neural networks?
5. How the concept of attention is used inside of a Transformer?

Problem 2

Review of RNNs In the following exercise, we will use the following notation. Denote the input sequence as $x_t ß R^k$ for $t \in \{1, ..., T\}$, and output of the network be$yt \in R^m$ for $t \in \{1, ..., T\}$.

In the following example, we construct a ”vanilla” many-to-many RNN consisting of a node that updates the hidden state h_t and produces an output y_t at each timestep with the following equations:

$$\begin{cases} h_t = \tanh(W_{h,h} h_{t-1} + W_{x,h} x_t + B_h) \\ y_t = W_{h,y} h_t + B_y \end{cases}$$

Where h_t is the time step of a hidden state (one can think of h_{t-1} as the previous hidden state), $W_{.,.}$ be the set of weights (for example, $W_{x,h}$ represents the weight matrix that accepts an input vector and produces a new hidden state), y_t be the output at timestep t and $B_.$ the bias terms.

1. Why are vanishing or exploding gradients an issue for RNNs?

   A significant issue with the vanilla RNN is that they suffer from vanishing/exploding gradients similar to issues with deep feedforward networks. An RNN can be unrolled into a (deep) feedforward network.

   At each timestep, the hidden state h_t is multiplied by W. At the last timestep, h_t is multiplied by W^T (matrix power). This means that depending on the singular values of the matrix W, the gradients of the loss with respect to W may become very large or very small as they pass back down the unrolled network. Additionally, the $\tanh$ activation at each step can also contribute to the vanishing gradient problem.

2. Let's show this property in an overly basic example. Consider the following 1D RNN with no nonlinearities, a 1D hidden state, and 1D inputs u_t at each timestep. (Note: There is only a single parameter w and no bias).

   (a) What recurrent relation does h_t follow?

   $h_t = w(u_t + h_{t-1})$

   (b) Draw the computational graph for 3 time steps, starting at $t = 1$. Assume $h_0 = 0$ and give the hidden state at time step 2 that we will call y.

   $y = h_3 = w(u_3 + h_2) = wu_3 + w(w(u_2 + h_1)) = wu_3 + w^2u_2 + w^3u_1$

   (c) Using the expression for y from the previous question, compute dy/dw and dy/du_1

   (d) Discuss this result compared to the first question of the exercise

3. Complete the class given below:

```python
import numpy as np
class VanillaRNN:
    def __init__(self):
        self.hidden_state = np.zeros((3, 3))
        self.W_hh = np.random.randn(3, 3)
        self.W_xh = np.random.randn(3, 3)
        self.W_hy = np.random.randn(3, 3)
        self.Bh = np.random.randn(3)
        self.By = np.random.randn(3)
    def forward(self, x):
        # Processes the input at a single timestep
        # and updates the hidden state
        self.hidden_state = np.tanh(...)
        self.output = np.dot(...) + ...
        return self.output
```

4. How does an LSTM mitigate the gradient vanishing/explosion?

Problem 3

GRU A GRU has a Reset gate and an Update gate in addition to the hidden state. We will describe in more detail what happens with the new gates in the GRU and why they are called like this.

In GRUs, the forget gate decides what information should be discarded from the previous state, and the update gate decides how much of the new state will be a blend of the last state and the candidate state.

To define this problem, we introduce the following quantities:

- x_t, the input vector at time step t.
- h_{t-1}, the previous hidden state.
- W_z, U_z, the weights for the update gate for input and previous hidden state respectively.
- W_r, U_r, the weights for the reset gate for input and previous hidden state, respectively.
- W_h, U_h, the weights for the candidate state for input and previous hidden state respectively.
- b_z, b_r, and b_h, the bias terms for the update gate, reset gate, and candidate state respectively.

1. Recall the equations used in the most general form of GRU.
2. If $x_t = [0.5, -0.1]$, $h_{t-1} = [0.2, 0.4]$, $W_z = U_z = W_r = U_r = W_h = U_h = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & -0.5 \end{bmatrix}$, and $b_z = b_r = b = [0.01, -0.01]$, calculate z_t, r_t, $\tilde{h}_t$, and h_t.
3. Discuss the role of the sigmoid function in the update and reset gate operations.
4. Explain how the values of z_t influence the final hidden state h_t.
5. What would happen to h_t if z_t is a vector of zeros? What if it is a vector of ones?

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

2. Calculations for the given values are found in the script below:

```python
import numpy as np

# Given data for the exercise
x_t = np.array([0.5, -0.1])  # input vector
h_t_minus_1 = np.array([0.2, 0.4])  # previous hidden state
W_z = U_z = W_r = U_r = W_h = U_h = np.array([[0.1, 0.2], [0.3, -0.5]])  # weights
b_z = b_r = b_h = np.array([0.01, -0.01])  # biases


# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# Hyperbolic tangent function
def tanh(x):
    return np.tanh(x)


# Update gate calculation
z_t = sigmoid(np.dot(W_z, x_t) + np.dot(U_z, h_t_minus_1) + b_z)
```

```
22
23  # Reset gate calculation
24  r_t = sigmoid(np.dot(W_r, x_t) + np.dot(U_r, h_t_minus_1) + b_r)
25
26  # Candidate state calculation
27  candidate_h_t = tanh(np.dot(W_h, x_t) + np.dot(U_h, (r_t * h_t_minus_1)) + b_h)
28
29  # Final hidden state calculation
30  h_t = (1 - z_t) * h_t_minus_1 + z_t * candidate_h_t
31
32  z_t, r_t, candidate_h_t, h_t
```

3. The sigmoid function is used in the update and reset gate operations to restrict the output between 0 and 1. This regulates the extent to which information is retained or discarded, allowing the GRU to decide which information is relevant at each time step.

4. The values of z_t influence the final hidden state h_t by determining the ratio at which the previous hidden state h_{t-1} is combined with the new candidate state $\tilde{h}_t$. When elements of z_t are close to 1, more of the previous hidden state is kept; when they are close to 0, more of the candidate state forms the new hidden state.

5. If z_t is a vector of zeros, the final hidden state h_t would be entirely composed of the new candidate state $\tilde{h}_t$, effectively forgetting the previous hidden state. If z_t is a vector of ones, the final hidden state h_t would be identical to the previous hidden state h_{t-1}, completely ignoring the candidate state. This demonstrates the gate's role in preserving past information and incorporating new information.

Problem 4

Attention Mechanisms for Sequence Modelling Sequence-to-Sequence is a powerful paradigm for formulating machine learning problems. As long as we can formulate a problem as a mapping from a sequence of inputs to a sequence of outputs, we can use sequence-to-sequence models to solve it. For example, in machine translation, we can formulate the problem as a mapping from a sequence of words in one language to a sequence of words in another language. While some RNN architectures we previously covered can maintain a memory of the previous inputs/outputs, to compute the output, the memory states need to encompass information of many previous states, which can be difficult, especially when performing tasks with long-term dependencies. To understand the limitations of vanilla RNN architectures, we consider changing the case of a the sentence is given a prompt token. For example, given a mixed case sequence like "<U> I am a student", the model should identify this as an upper-case task based on token <U> and convert it to "I AM A STUDENT". Similarly, given "<L> I am a student", the lower-case task is to convert it to "i am a student". We can formulate this task as a character-level sequence-to-sequence problem, where the input sequence is the mixed case sentence, and the output sequence is the desired case sentence. In this exercise, we use an encoder-decoder architecture to solve the task. The encoder is a vanilla RNN that takes the input sequence as input and outputs a sequence of hidden states. The decoder is also a vanilla RNN that inputs the last hidden state from the encoder and outputs the desired case sentence (generally called the context vector).

1. Consider a simple encoder-decoder architecture with a single hidden layer in the encoder and decoder. The encoder takes the input sequence and outputs a sequence of hidden states. The decoder inputs the last hidden state from the encoder and outputs the desired case sentence.

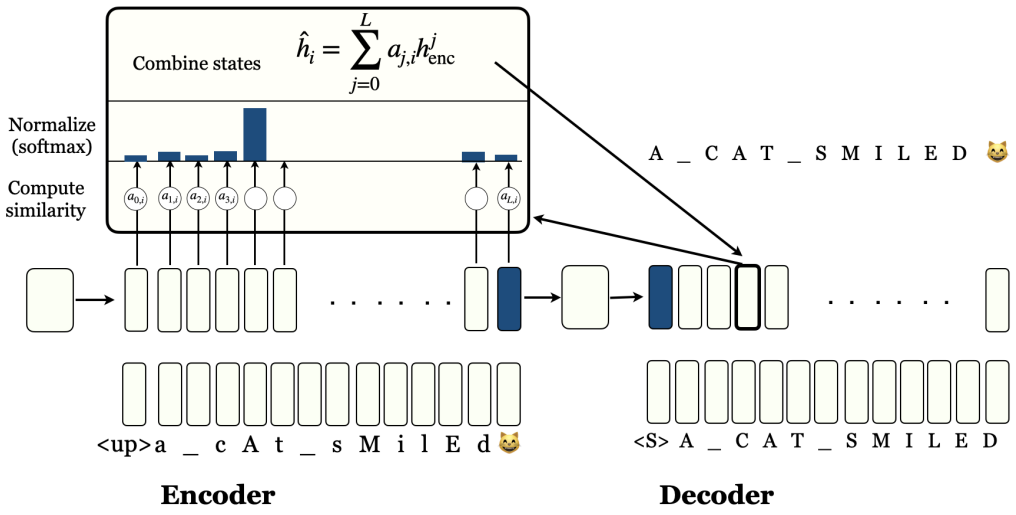

Figure 1: Attention Mechanism for Sequence Modelling with RNNs

(a) Make a diagram of the architecture

(b) What information must be stored in the hidden state to perform the upper-case/lower-case task? Are there any limitations to this architecture? One major limitation of the architecture is the encoder bottleneck-activation that is passed to the decoder. This means that the hidden state at the last time step should contain information about the **entire** input sequence. This can be difficult, especially when performing tasks with long-term dependencies (e.g., the task-identifier token is at the beginning of the sentence).

2. Instead of storing all the information in the hidden state, we can use attention to selectively store information. The idea of attention is to query the encoder hidden states with a query vector and use the resulting attention weights to compute a weighted sum of the hidden states. This weighted sum is then used as the input to the readout layer that computes the output token at each time step of the decoder.

   How does adding attention (fig. 1) allow the model to bypass the information bottleneck? In particular, what information in the following modules would allow the model to perform the capitalization task ?

   - Encoder Weights
   - Attention Scores
   - Bottleneck Activations
   - Decoder Weights

   - The encoder weights need to learn a representation of the position of a particular token in the input-sequence.
   - The attention scores compute the similarity between the decoder "query" vector and the hidden states of the encoder. It can perform the task as long as it scores the token at the same index as the query vector.
   - The bottleneck activation no longer needs to store information about the entire input sequence, since we are allowed to perform a look-up with the attention scores.

- The decoder weights learn to count, which is used to identify which token in the output-sequence we are decoding.