HW to Chapter 19 “Recurrent Neural Networks”

**Non-programming Assignment**

1. What are recurrent neural networks (RNN) and why are they needed?
2. What do time steps play in recurrent neural networks?
3. What are the types of recurrent neural networks?
4. What is the loss function for RNN defined?
5. How do forward and back-propagation of RNN work?
6. What are the most common activation functions for RNN?
7. Describe bidirectional recurrent neural networks (BRRN) and explain why they are needed.
8. Describe Deep recurrent neural networks (DRRN) and explain why they are needed.

Answer

**1. What are RNNs and why are they needed?**

RNNs are a type of neural network designed for sequential data, meaning they can process inputs that have an order, like time-series data, speech, and text. RNNs are basically neural networks with a memory. Unlike standard feedforward networks that treat each input independently, RNNs maintain an internal state (or “memory”) that captures information about what's been processed so far.

We need them because so much data in the real world is sequential - think speech, text, time series, videos, etc. Regular neural networks are not so great at handling sequential dependencies. For example, trying to predict the next word in “The clouds are in the \_\_\_” without remembering the earlier words – will be pretty much impossible. This is where RNNs come in play, allowing the model to remember previous steps and then being able to predict the next steps and so on.

The key intuition is that RNNs process information in loops, allowing information to persist. This lets them learn patterns across time steps, which is crucial for tasks like language modeling, speech recognition, and translation.

**2. What do time steps play in RNNs?**

Time steps represent different points in a sequence, allowing an RNN to process inputs one at a time while keeping track of past information. Time steps are basically the discrete points in your sequence for a time series data. In an RNN, we unroll the network across these time steps - each time step corresponds to an element in your sequence. It is like reading a sentence: each word (time step) influences the understanding of the next word. The network updates its hidden state at each step, gradually forming a memory of the sequence.

For example, if you’re processing the sentence “I love neural networks”:

* Time step 1: “I”
* Time step 2: “love”
* Time step 3: “neural”
* Time step 4: “networks”

The magic of RNNs is that at each time step, the network makes a prediction based not just on the current input, but also on the hidden state from the previous time step. This hidden state acts like a compressed summary of everything the network has seen so far.

**3. What are the types of recurrent neural networks?**

RNNs can handle different types of input-output relationships depending on the structure of the data. Here are the main types:

1. **One-to-One**: This is the simplest form of RNN, where a single input is mapped to a single output. It behaves like a traditional neural network with a fixed input and output size. Since there's no sequential dependency here, it's mostly used in tasks like image classification, where each image corresponds to a single label.

2. **One-to-Many**: In this case, a single input generates a sequence of outputs. The input remains fixed, but the model produces multiple outputs over time. A good example is music generation, where a single theme or note can be expanded into an entire melody, or image captioning, where a single image is used to generate a sequence of descriptive words.

3. **Many-to-One**: Here, a sequence of inputs is processed to produce a single output. The RNN takes in a series of data points and condenses their information into one final result. This is commonly used in sentiment analysis, where a model reads an entire sentence (multiple words) before classifying it as positive or negative, or in text classification tasks like spam detection.

4. **Many-to-Many**: This type of RNN is designed to process both sequential inputs and sequential outputs. It takes in a sequence and produces a corresponding sequence of outputs. A classic example is named entity recognition (NER), where a model reads a sentence and labels each word as a person, location, or organization.

Many-to-Many RNNs can be further divided into:

* Full Many-to-Many: The input and output sequences are of the same length (e.g., real-time speech recognition).
* Partial Many-to-Many: The input and output sequences have different lengths. A common example is machine translation, where a sentence in one language is converted into another, often with a different number of words.

**4. How is the loss function for RNN defined?**

In RNNs, the loss is calculated at each time step and then either summed or averaged over all time steps. The choice of loss function depends on the type of task:

* **For sequence classification** (e.g., text classification, speech recognition), we typically focus on the final time step’s output. A common choice here is **cross-entropy loss**, which measures how well the model classifies the sequence into the correct category.
* **For sequence generation or prediction** (e.g., time-series forecasting, text generation), the model makes a prediction at each time step, so the loss is computed at every step. In regression-based tasks, **Mean Squared Error (MSE)** is often used to measure how far the predicted values deviate from the actual values.

For example, in **language modeling**, cross-entropy loss is applied at each time step to evaluate how well the model predicts the next word in a sequence. The model is then updated to minimize this loss, improving its predictions over time.

**5. How do forward and back-propagation of RNN work?**

**Forward propagation** in RNNs: The input is processed one time step at a time, updating the hidden state and generating outputs at each step.

1. At each time step t, take input x\_t and previous hidden state h\_{t-1}
2. Compute current hidden state: h\_t = tanh(W\_h\*h\_{t-1} + W\_x\*x\_t + b\_h)
3. Compute output: y\_t = softmax (W\_y\*h\_t + b\_y)
4. Move to next time step and repeat

**Backpropagation Through Time (BPTT)**: This is where RNNs get tricky! Since RNNs deal with sequences, errors are backpropagated through all previous time steps. Since the parameters are shared across time steps, we need to accumulate gradients across the entire sequence:

1. Compute loss at each time step
2. Calculate gradient of loss with respect to output
3. Propagate error backwards through time, accumulating gradients for shared weights
4. Update weights using the accumulated gradients

The main challenge with BPTT is the vanishing/exploding gradient problem. As you propagate back through many time steps, gradients either vanish (multiply by small numbers repeatedly) or explode (multiply by large numbers). This is why regular RNNs struggle with long-term dependencies and why LSTMs and GRUs were developed.

**6. What are the most common activation functions for RNN?**

In traditional RNNs, these are the most common activation functions:

1. **Tanh (hyperbolic tangent)**: The classic choice for the hidden state in regular RNNs. Its output range (-1 to 1) helps prevent exploding activations and having both positive and negative outputs helps the dynamics of the network but suffer from vanishing gradients.
2. **Sigmoid**: Often used in gating mechanisms (like in LSTMs and GRUs) because its output range (0 to 1) is perfect for representing “how much” to let through a gate.
3. **ReLU**: Sometimes used in variations of modern RNN, especially in deep RNNs, as it can help with the vanishing gradient problem.

For the final output layer, the activation depends on the task:

* **softmax**: For classification problems (predicts probabilities across classes)
* **sigmoid**: For binary classification
* **linear/identity**: For regression problems

**7. Bidirectional Recurrent Neural Networks (BRNNs)**

BRNNs are pretty clever because they process sequences in both directions simultaneously (both forward and backward directions). You have two separate RNNs:

* One processes the sequence from start to end (forward)
* The other processes it from end to start (backward)
* The outputs of both networks are combined (usually concatenated) to form the final output

This means the network has access to both past and future context, making it particularly useful for tasks like speech recognition and text translation, where understanding the full sequence (not just past information) is important.

Why are they needed? Because context often flows in both directions! In natural language, the meaning of a word depends not just on previous words but also on future words. For example, in "The bank is by the river," knowing "river" comes later helps disambiguate "bank."

BRNNs are mainly used in tasks like:

* Named entity recognition
* Machine translation
* Speech recognition
* Sentiment analysis

They’ve become standard in NLP for tasks where you have access to the complete sequence at once. However, they can’t be used for real-time sequence generation since they need the entire sequence upfront.

**8. Deep Recurrent Neural Networks (DRNNs)**

DRNNs stack multiple layers of recurrent networks on top of each other allowing the model to learn more abstract and complex features.

In a standard RNN, we have: Input → RNN layer → Output

In a deep RNN, we might have: Input → RNN layer 1 → RNN layer 2 → ... → RNN layer N → Output

Each RNN layer processes the sequence from the previous layer, creating increasingly abstract representations. The hidden state from one layer becomes the input to the next layer.

Why are they needed? For the same reason we use deep networks in other architectures - to learn hierarchical representations of the data**. The lower layers can learn basic patterns, while higher layers capture more complex, abstract patterns.** This is particularly useful for:

* Complex sequence modeling tasks
* Speech recognition
* Machine translation
* Video analysis

The challenge with DRNNs is they're computationally expensive and can be difficult to train. They often require techniques like residual connections, layer normalization, or careful initialization strategies to train effectively.