RNN Theoretical Questions

Q1: **What's the difference between *Convolutional Neural Networks (CNN)* and *Recurrent Neural Networks (RNN)* and in which cases would use each one?**

Answer

Convolutional neural nets apply a convolution to the data before using it in fully connected layers.

- They are best used in cases where you want *positional invariance*, that is to say, you want features to be captured regardless of where they are in the input sample.
- Think of a picture with all sorts of animals in it. If you apply a convolutional neural net to classify whether there is a cat in the picture, it will identify the cat no matter what position in the picture the cat is (at the top, the bottom, left or right). This is very useful for *image classification*.

Recurrent neural nets are neural networks that *keep state* between input samples. They remember previous input samples and use those to help classify the current input sample.

- They are most useful when the *order of your data is important*. So for instance in speech (previous words do help identify the current word), video (frames are ordered) and also text processing.
- Generally speaking, problems related to time-series data (data with a timestamp on them) are good candidates to be solved well with recurrent neural nets.

Source: stats.stackexchange.com

Q2: **What are the uses of using *RNN* in NLP?**

Answer

- The **RNN** is a *stateful* neural network, which means that it not only retains information from the previous layer but also from the previous pass. Thus, this neuron is said to have *connections* between passes, and through time.
- For the **RNN** the order of the input matters due to being stateful. The same words with different orders will yield different outputs.
- **RNN** can be used for *unsegmented, connected* applications such as *handwriting recognition* or *speech recognition*.

Source: en.wikipedia.org

Q3: **Why are *RNNs (Recurrent Neural Network)* better than *MLPs* at predicting Time Series Data?**

Answer

- In an **RNN**, the output of the *previous state* is passed as an input to the *current state*.
- There is a *temporal relationship* in the way in which input is processed in an *RNN*. It can understand how the current state was achieved based on the previous values, i.e. value at time-step `t` is a result of value at time-steps `t-1`, `t-2` , and so on.
- In *DNN*, there is no temporal relationship in the way input is processed. Values at time-steps `t`, `t-1`, `t-2` ... are all treated distinctly and not as a continuation of the previous time-step values.

Source: ai.stackexchange.com

Q4: **How many dimensions must the inputs of an *RNN* layer have? What does each dimension represent? What about its outputs?**

Answer

An *RNN* layer must have three-dimensional inputs:

- The first dimension is the *batch dimension* (its size is the batch size.
- The second dimension represents the *time* (its size is the number of time steps),
- And the third dimension holds the *inputs at each time step* (its size is the number of input features per time step).

For example, if you want to process a batch containing `3` time series of `10` time steps each, with `2` values per time step (e.g., the temperature and the wind speed), the shape will be `[3, 10, 2]`.

The outputs are also three-dimensional, with the same first two dimensions, but the last dimension is equal to the *number of neurons*. For example, if an RNN layer with `32` neurons processes the batch we just discussed, the output will have a shape of `[3, 10, 32]`.

Source: www.amazon.com

Q5: **What's the difference between *Traditional Feedforward Networks* and *Recurrent Neural Networks*?**

Answer

The main difference is in how the input data is taken in by the model.

- ***Traditional feedforward neural networks*** take in a fixed amount of input data *all at the same time* and produce a fixed amount of output *each time*.
- ***Recurrent neural networks*** do not consume all the input data at once. Instead, they take them in one *at a time* and in a sequence:
  - At each step, the RNN does a series of *calculations* before producing an *output*. The output, known as the *hidden state*, is then **combined** with the next *input* in the sequence to produce another output.
  - It may seem that a different *RNN* cell is being used at each time step in the network, but the underlying principle of Recurrent Neural Networks is that the RNN cell is actually the exact same one and reused throughout.
  - This process continues until the model is programmed to finish or the *input sequence* ends.

Source: blog.floydhub.com

Q6: **What are the main *difficulties* when training RNNs? How can you handle them?**

Answer

The two main difficulties when training RNNs are **unstable gradients** (exploding or vanishing) and a **very limited short-term memory**. These problems both get worse when dealing with long sequences.

To alleviate the unstable gradients problem, we can:

- Use a smaller learning rate.
- Use a *saturating activation function* such as the hyperbolic tangent (which is the default), and possibly use *gradient clipping*, *Layer Normalization*, or *dropout* at each time step.

To tackle the limited short-term memory problem, we can use a *Long Short-Term Memory* layer or a *Gated recurrent unit* layer.

Source: www.amazon.com

Q7: **What types of Recurrent Neural Networks (RNN) do you know?**

Answer

There are different types of *recurrent neural networks* with varying architectures. Some examples are:

- **One to one**: Here there is a single `(x1,y1)` pair. Traditional neural networks employ a one to one *architecture*.
- **One to many**: In one to *many* networks, a single input at `x1` can produce multiple outputs, e.g., `y10`, `y11`, `y12` Music generation is an example area, where one to many networks are employed.
- **Many to one**: In this case, many inputs from different time steps produce a single *output*. For example, `x1`, `x1+1`, `x1+2` can produce a single output. Such networks are employed in sentiment analysis or emotion detection, where the class label depends upon a sequence of words.
- **Many to many**: There are many possibilities for many to many. An example is shown below, where two inputs produce three outputs. Many to many networks are applied in machine translation, e.g. English to French or vice versa translation systems.

Source: machinelearningmastery.com

Q8: **What's the difference between *Recurrent Neural Networks* and *Recursive Neural Networks*?**

Answer

- **Recurrent Neural Networks**:
  - It is used for sequential inputs where the *time factor* is the main differentiating factor between the elements of the sequence, that's why it's commonly used in *time-series*.
  - When we *unfold the network*, at each time step, it accepts the *user input* at that *time step* and the output of the hidden layer that was computed at the *previous time step*.
  - The weights are shared (and *dimensionality remains constant*) along the *length of the sequence*.
- **Recursive Neural Networks**:
  - Is more like a *hierarchical network* where there is really no *time aspect* to the input sequence but the *input* has to be processed *hierarchically* in a tree fashion.
  - Is very used in *NLP*, where the way to learn a parse tree of a sentence is *recursively* taking the output of the operation performed on a *smaller chunk* of the text.
  - Here, the weights are shared (and *dimensionality remains constant*) at *every node*.

Source: stats.stackexchange.com

Q9: **Why would you use *Encoder-Decoder RNNs* vs *plain sequence-to-sequence RNNs for automatic translation*?**

Answer

A **plain sequence-to-sequence RNN** would start translating a sentence immediately after reading the first word of a sentence, while an **Encoder-Decoder RNN** will first *read the whole sentence* and then translate it.

In general, if you translate a sentence one word at a time, the result will be terrible. For example, the french sentence *"Je vous en prie"* means *"You are welcome"* but if you translate it one word at a time using *plain sequence-to-sequence RNN*, you get *"I you in pray"* which it does not have sense. So in automatic translation cases is much better to use *Encoder-Decoder RNNs* to read the whole sentence first and then translate it.

Source: www.amazon.com

Q10: **What's the difference between *Stateful RNN* vs *Stateless RNN*? What are their pros and cons?**

Answer

- In a **stateless RNNs**, on each training iteration the model starts with a *hidden state full of zeros*, then it updates this state at each *time step*, and after the last *time step*, it throws it away, as it is not needed anymore.
- A **stateful RNNs** *preserve this final state* after processing one training batch and use it as the *initial state* for the next training batch, this way the model can learn long-term patterns.
- Given the inner work of *stateless RNNs*, they can only *capture* patterns whose length is less than, or equal to, the size of the *windows* the RNN is trained. Conversely, **stateful RNNs can capture *longer-term patterns***.
- However, implementing a *stateful RNN* is much harder, especially preparing the dataset properly.
- Moreover, ***stateful RNNs do not always work better***, in part because consecutive batches are not *independent and identically distributed* (IID) and Gradient Descent is not fond of non-IID datasets.

Source: www.amazon.com

Q11: **When would you use *MLP*, *CNN*, and *RNN*?**

Answer

- **Multilayer Perceptrons**, or MLPs for short are the classical type of neural network. They are very flexible and can be used *generally* to learn a *mapping from inputs to outputs*, however, they are perhaps more suited to *classification* and *regression* problems.
- **Convolutional Neural Networks**, or CNNs, were developed and are best used for *image classification*. But they can also be used generally with data that has a *spatial structure*, such as a sequence of words, and can be used for *document classification*.
- **Recurrent Neural Network** or RNNs, was developed for *sequence prediction* and is well suited for problems that have a sequence of input observations or a sequence of output observations. They are suitable for *text data*, *audio data*, and similar applications.

Source: machinelearningmastery.com

Q12: **How is the *Transformer Network* better than *CNNs* and *RNNs*?**

Answer

- With **RNN**, you have to *go word by word* to access to the cell of the last word. If the network is formed with a long reach, it may take several steps to remember, each masked state (output vector in a word) depends on the previous masked state. This becomes a *major problem* for GPUs. This sequentiality is an obstacle to the parallelization of the process. In addition, in cases where such sequences are too long, the model tends to forget the contents of the distant positions one after the other or to mix with the contents of the following positions. In general, whenever ***long-term dependencies*** are involved, we know that **RNN** suffers from the ***Vanishing Gradient Problem***.
- Early efforts were trying to solve the dependency problem with ***sequential convolutions*** for a solution to the **RNN**. A long sequence is taken and the convolutions are applied. The disadvantage is that **CNN** approaches require many layers to capture long-term dependencies in the sequential data structure, without ever succeeding or making the network so large that it would eventually become impractical.
- The ***Transformer*** presents a new approach, it proposes to encode each word and apply the ***mechanism of attention*** in order to connect two distant words, then the *decoder* predicts the sentences according to all the words preceding the current word. This workflow can be parallelized, accelerating learning and solving the ***long-term dependencies*** problem.

Source: medium.com

Q13: **Compare *Feed-forward* and *Recurrent* Neural Network**

Answer

- ***Feed-forward*** neural networks allow the signals to travel in *one direction only*. There are *no feedback loops* (the *output* does not affect the *input* of the same layer). This type of network only associate inputs with outputs.
- **Recurrent** neural networks have signals travelling in *both directions* by introducing loops in the network. The computations derived from an earlier input is fed back into the network which gives them a kind of *memory*. This memory allows them to process *sequences* of input very well. That is why RNN is used in text summarization, machine translation, etc. where the sequence of input is very important.

Source: stats.stackexchange.com

Q14: **Explain the intuition behind *RNN* having a *Vanishing Gradient Problem*?**

Problem

How does *LSTM* solve this problem?

Answer

- When the *weight matrix* is multiplied successively at various time-steps it causes an instability in the form of exploding or vanishing gradients.
- Recurrent neural networks use multiplicative updates of the weights, so it can only learn over short sequences due to the exploding and vanishing gradients. Hence, *RNNs are endowed with good short-term memory but bad long-term memory*.
- Changing the *recurrence equation* for the hidden vector with the use of LSTM with long-term memory improves the bad long-term memory of RNNs. The operations of the LSTM are designed to have *fine-grained control* over the state written into this long-term memory.
- LSTMs have changed *recurrence conditions of how* hidden states are propagated. To do this, there is an additional hidden vector which is referred to as *cell state*. Cell state is a kind of *long-term memory* that retains at least a part of the information in earlier states by using a combination of partial *forgetting* and *incremental* operations on the previous cell states.
- In LSTM, the presence of a *forget gate*, along with the additive property of *cell gradients* enables the network to update the parameter in such a way that the different sub gradients do not agree and hence the gradients do not become zero.

Source: medium.datadriveninvestor.com

Q15: **How to *calculate the output* of a Recurrent Neural Network (RNN)?**

Answer

A recurrent neural network looks very much like a *feedforward neural network*, except it also has connections pointing backward. Let's look at the simplest possible *RNN*, composed of *just one neuron receiving inputs, producing an output, and sending that output back to itself*, as shown in the Figure `1`.

Now to see the inner work of this tiny neuron, we *unroll the network over the time axis*, as shown in the Figure `2`. We observe that in the time step `t`, the recurrent neuron (marked with the *red arrow*) receives the inputs `x(t)` as well as its own output from the previous time step, `y(t-1)`, and successively, `y(t-1)` is the output of getting `x(t-1)` and `y(t-2)` as inputs, and so on.

Now consider a *layer* of recurrent neurons. As before, the same principle remains: at each time step `t`, every neuron receives both the *input vector* `x(t)` and the *output vector* from the previous time step `y(t-1)`.

This recurrent layer has two sets of weights: one for the inputs `x(t)` and the other for the outputs of the previous time step, `y(t-1)`. Let's call these weight matrices `Wx` and `Wy` respectively, then the *output vector* of the whole recurrent layer can then be computed as:

where `b` is the bias vector and `ϕ(·)` is the activation function, e.g., *ReLU*.

Source: www.oreilly.com

Q16: **How does *LSTM* compare to *RNN*?**

Answer

- **Long Short-Term Memory** (LSTM) is a type of *RNN* architecture. LSTM has feedback connectors and it can process whole *sequences* of data.
- *LSTMs* were developed to deal with the *vanishing gradient problem* that can be encountered when training traditional RNNs.
- *RNNs* have *hidden states* which serve as the *memory* of the *RNN*. *LSTMs* have both cell states and hidden states. The *cell state* has the ability to *add or remove* information from the cell, regulated by *gates*.

Source: stats.stackexchange.com

Q17: **Explain how a *Recurrent Architecture* for leveraging visual attention works**

Answer

An architecture for leveraging visual attention is explained below:

1. **Glimpse sensor**: It creates a retina-like representation of the image. The glimpse sensor is conceptually assumed to not have full access to the image, and is able to access only a *small portion* of the image in high-resolution. It behaves much like a *real human eye*.
2. **Glimpse network**: The glimpse network contains the glimpse sensor and encodes both the glimpse location and the glimpse representation into hidden spaces using *linear layers*. Subsequently, the two are combined into a single hidden representation using another *linear layer*.
3. **Recurrent neural network**: The recurrent neural network is the main network that is creating the action-driven outputs in each time-stamp. The recurrent neural network includes the *glimpse network*, and therefore it includes the *glimpse sensor* as well. Rewards are associated with the *output action*.

Source: www.amazon.com

Q18: **Why would you want to use `1D` convolutional layers in an *RNN*?**

Answer

Recall that an RNN layer is fundamentally *sequential*: in order to compute the outputs at time step `t`, it has to first compute the outputs at all earlier time steps. This makes it impossible to parallelize. On the other hand, a `1D` convolutional layer lends itself well to *parallelization* since it does not hold a *state* between time steps. In other words, it has no *memory*: the output at any time step can be computed based only on a small window of values from the inputs without having to know all the past values.

Moreover, since a `1D` convolutional layer is not recurrent, it suffers less from unstable gradients. Therefore, one or more `1D` convolutional layers can be useful in an RNN to efficiently *preprocess the inputs*, for example to reduce their temporal resolution (*downsampling*) and thereby help the RNN layers detect *long-term patterns*. In fact, it is possible to use only convolutional layers, for example by building a *WaveNet* architecture.

Source: www.amazon.com