CS 282 Deep Neural Networks **Spring 2023**

Lecture 14: Sequence-to-Sequence, Attention

Lecturer: Anant Sahai *Scribe: Francis Geng*

1 . Sequence-to-Sequence Problems

1.1 Definition

A sequence-to-sequence problem, often abbreviated as seq2seq, refers to a class of tasks where the goal is to map an input sequence of data to an output sequence, typically with different lengths or structures. These tasks often involve predicting or generating a sequence based on the given input sequence, with both the input and output sequences consisting of discrete elements, such as words, characters, or tokens. Translating a sentence from one language to another, where the input sequence consists of words in the source language, and the output sequence is a translation in the target language, is a common example of a sequence-to-sequence problem.

1.2 Classical RNN?

As mentioned in previous lectures, Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed for processing sequences of data. They are particularly well-suited for tasks that involve temporal or sequential information, such as natural language processing, speech recognition, and time series prediction. It's natural to consider using RNNs for sequence-to-sequence problems because RNNs are known to be useful for processing sequential information. However, there are several challenges and limitations associated with using RNNs for sequence-to-sequence problems. We'll be using machine translation as an example here to prove the point.

1. Sequential processing: RNNs process input sequences one element at a time, which can result in slow computation for long sentences.

2. Fixed-length context: RNNs have a limited context window, which can make it difficult for them to accurately capture the relationships between words and phrases in longer sentences. This can lead to poor translations when there's a need to consider the broader context of the sentence.

3. Alignment issues: In some cases, the order of words in the source and target languages might be different, making it challenging for RNNs to learn the correct alignment between source and target words. This can result in translations that are not accurate or grammatically correct.

Therefore, we should consider using another architecture that can take an entire sequence as input and emit an entire sequence as output and overcome the challenges that classical RNNs encountered.

1.3 A Solution: Encoder-Decoder Architecture

1.3.1 Encoder

When humans read a sentence, we read one word at a time. Therefore, we design the RNN encoder to do the same thing. As shown in Figure 14.1, each word is inputted into the encoder blocks sequentially and each block outputs a hidden state that is used as an input to the next block. Each block shares the same weight.

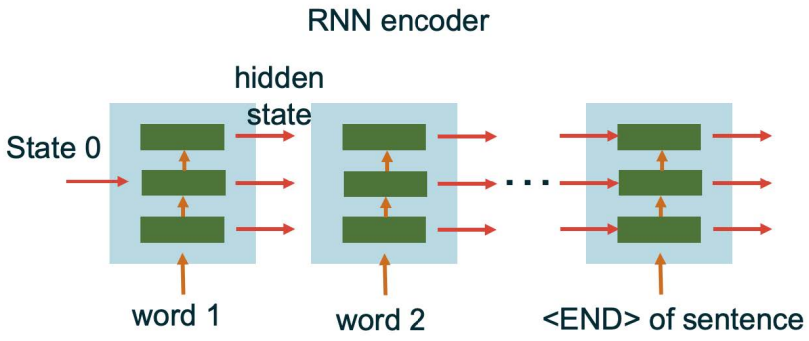

Figure 14.1: A RNN encoder used to process sequential data [1].

1.3.2 Decoder

In a typical sequence-to-sequence model, the input sequence is first processed by an encoder, which generates an encoded input representation that captures the relevant information from the input. To pass this encoded input from the encoder to the decoder, we could either pass the entire state or pass it through a multi-layer perceptron. The RNN decoder then takes this input representation and generates the output sequence one element at a time, using its recurrent structure to maintain an internal state that is updated at each time step. This allows the RNN decoder to capture dependencies and relationships between elements in the output sequence.

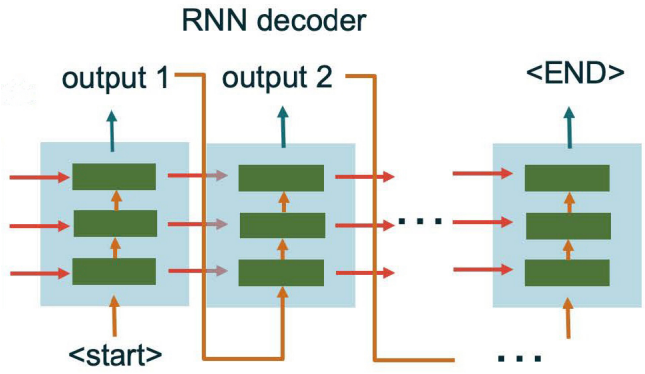

Figure 14.2: A RNN decoder used to generate the output sequence given encoded data [1].

1.3.3 Training: Teacher Forcing

During training for the decoder, because we have access to the ground-truth output and its tokens at each time step, we can use the ground-truth tokens as input for the next time step. This approach helps the model to learn the correct dependencies more quickly, as it avoids compounding errors from previous time steps. This approach is also called teacher forcing.

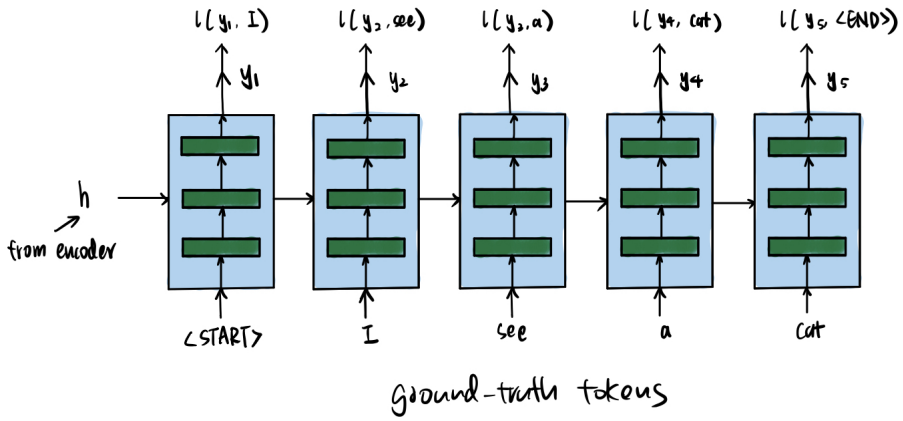

Figure 14.3: RNN Decoder with goal output "I see a cat" [3]

Here's how this works in detail:

1. The decoder is initialized with the encoded input representation from the encoder and a special start-of-sequence (SOS) token, shown as <START> in Figure 14.3.
2. At each time step, the decoder takes the ground-truth token and its hidden state as input, and generates the next token in the output sequence.
3. This generated token is then fed into a loss function (we'll discuss how loss functions work in this context in a bit) that computes the error of the generated token compared to the ground truth token at that time step. For the next time step, the decoder will once again take in the ground-truth token along with the updated hidden state to generate a new token.
4. This process is repeated until an end-of-sequence (EOS) token (shown as <END> in Figure 14.3) is generated, which shall not be fed back into the decoder.

Questions:
1. Will inputting the ground-truth output allow for "cheating"?
A: In Figure 14.2, it can be observed that there are no arrows that indicate a connection from a recurrent neural network block containing the actual output value to the previous block that predicts the output and calculates the loss. This prevents any possibility of the previous block being influenced by the ground truth and prevents any "contamination" of the predictions.

2. What is the role of the loss function during training?
A: Using backpropagation, the combined effect of these losses (in the decoder) is going to put pressure on the hidden state (passed from the encoder), which will then back propagate through the encoder to do weight updates via weight sharing on all the time steps (you'll see this again in 1.3.4).

1.3.4 Training: Autoregression

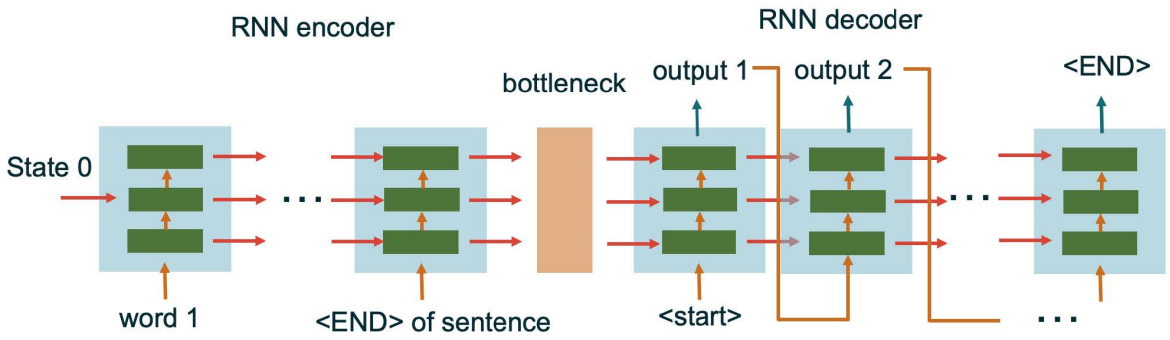

Figure 14.4: Encoder and Decoder in RNN [1].

Autoregression is a technique often employed to ensure that the model learns to generate meaningful sequences based on the available context. With autoregression, instead of using the ground-truth tokens as input to the decoder at each time stamp, tokens generated by the decoder itself are used. Figure 14.4 would actually be a good example of autoregression because all the outputs of the decoder are fed into the decoder at the following time step (see the orange line between output 1 and the second RNN block in the decoder).

If the decoder's output at a specific time step is incorrect (different from the ground-truth token), the next RNN block will have less high quality gradients, and the even poorer quality gradients at the RNN blocks following. These gradients are all going to be applied to the shared weights so using autoregression will typically slow down training. However, using noisy inputs are not necessarily bad; in practice, people will randomly send the predicted output into the next RNN block instead of the correct output during training to take advantage of the regularizing effect of autoregression. It helps the RNN blocks be less dependent on the details of the true outputs and make the actual outputs it tends to generate more robust.

Questions:
1. How do we do data augmentation for language data?
A: In the context of languages, we add random noises to the vector representation of words which need not correspond to any natural disturbance that you might expect to see in practice. The nature of data augmentations that capture natural variability in language is not as obvious as the data augmentation techniques we have used for images.

2. What if the predicted outcome is not the"END" token where it's supposed to end?
A: We don't continue to the next time step anyways during training because we know we're supposed to get the "END" token instead. One way to explain this is that if you insert "END" into the decoder, you will not be able to get gradients from it since there is no more label and the loss will not be valid.

3. What if the predicted outcome is the "END" token before it's supposed to end and we want it to actually continue predicting?
A: If we're using the teacher forcing technique, we can compute the loss between the "END" token and ground-truth token for that time step just like every other time step. If we're using the autoregression approach, an "END" token will be sent to the next block and this will still have a valid loss.

1.3.4 Training: Backpropagation

Backpropagation in an RNN encoder-decoder model is an extension of the standard backpropagation algorithm used for training feedforward neural networks. It involves computing gradients of the loss function with respect to the model's weights (for both encoder and decoder) and updating the weights to minimize the loss. We use backpropagation through time (BPTT) to account for the recurrent structure of the RNNs.

Here's a high-level overview of how backpropagation works here:

- Forward pass: The input sequence is passed through the encoder, generating an encoded representation that captures the input's relevant information, followed by the process described in 1.3.3.
- Backward pass:

  1. The gradient of the loss with respect to the decoder's output is computed.

  2. The gradient is then propagated backward through the decoder, computing the gradients with respect to its weights and hidden states at each time step.

  3. Once the gradient reaches the initial hidden state of the decoder (i.e., encoded input representation), it continues to propagate backward through the encoder.

  4. The gradients with respect to the encoder's weights and hidden states at each time step are computed, similar to the decoder.

- Weight updates: Use the computed gradients to update the encoder and decoder's weights in a manner that minimizes the loss function, improving the model's performance on the task.
- Iteration: Repeat the steps above for multiple epochs, using different input-output pairs from the training dataset to caliberate the weights, minimize the loss and improve the model's ability to generate accurate output sequences.

1.3.5 Loss Function: Cross-entropy Loss for Language Problems

In order to choose an appropriate loss function for the encoder-decoder model for language problems, we need to look at the representation of data for sequence-to-sequence tasks. There are two main representations for data: 1. regression-like where we're trying to predict a continuous value, 2. classification-like where we're trying to predict something from a discrete set. Spoken language exhibits both discrete and continuous aspects. While specific words or phonemes can be well approximated as discrete, features such as intonation, pausing, and emphasis are actually quite continuous in nature. Therefore, actual language is a blend of both discrete and continuous elements, and the two mutually influence each other. However, written language data, which we actually have access to in these problems, is discrete. Therefore, the cross-entropy loss is well-suited for sequence-to-sequence tasks, where the goal is to generate a sequence of discrete elements, such as words or characters, based on a given input sequence. When we use cross-entropy loss, the model generates a probability distribution over the possible tokens at each time step of the output sequence. An issue arises when we use autoregression during inference. Even though it is fine to admit a probability distribution during training, a probability distribution cannot be used as an input to the next block in application, a specific token is necessary.

There are three ways to solve this:
1. Pick the token with the highest probability
2. Sample a token from the probability distribution
3. Beam search by keeping a few possibilities/enumerating all possibilities

Questions:
1. What should we do if we have a different type of seq2seq problem?
A: If we ask a robot to predict a trajectory, we will need to use squared loss to account for the continuous nature of the output (e.g. a sequence of real numbers for trajectory). We can still use autoregression during training in this case, which is the same approach as our approach to a linguistic problem (i.e. input the predicted trajectory vs. input the ground-truth trajectory at a specific time into the decoder). In some problems, there will also be a third sequence which is a contextual sequence. In the robotics case, the contextual sequence will be your sensor measurements for your robots and these sequences will be additional inputs to the decoder.

2. If you're going to be learning a machine translation system and you want to translate from English to French then can you train it on just English to French data?
A: Yes, you can always train with end-to-end data on the specific case.

3. Would it be helpful to have English-to-Spanish data?
A: It should be helpful because English-to-Spanish data is telling us at least something about English, so it could help the encoder. This becomes really relevant when you want the translator from English to a small tribal language for which you have very limited data. Although you may not have enough data for direct translation, there is still hope. If you have a translator for all languages, you can learn the correct representations and fill in translations for language pairs even if you have no data for them.

2 . Attention [1]

2.1 Motivation for Attention

The U-Net architecture is specifically designed for image processing, utilizing skip connections between the encoder and decoder to capture both local and global context information in the input image. These skip connections enable the decoder to recover fine-grained details of the input image, leading to high-quality output. In addition to these skip connections, the natural correspondence that exists in many image-image problems guides the search for relevant information. For example, in semantic segmentation, the corresponding pixel in the original image holds local, fine-grained information that is relevant to a specific pixel in the output image. This correspondence plays a crucial role in the success of the U-Net architecture for image processing.

On the other hand, for seq2seq problems, the Encoder-Decoder architecture without U-Net compresses the input data into a low-dimensional representation before passing it through a bottleneck, leading to the loss of fine-grained information. To overcome this issue, it is necessary to find another way to pass local, fine-grained information from the encoder to the decoder.

In language processing, different languages have varying word orders, making the routing of information context-dependent. U-Net architecture is ineffective in capturing the sequential context information in language data, as it is designed to process data in a convolutional manner, which is not suitable for sequential data such as sentences or text. Therefore, context-dependent routing of information is necessary, and static routing methods, such as indexing into an array at a known position or direct mapping between input and output, are unsuitable.

To address this issue, the goal is to add a "memory" to the existing Encoder-Decoder architecture that allows the input information to be stored and retrieved at the decoder. This approach will ensure that the decoder has access to the right piece of information for every position, facilitating the generation of high-quality output.

2.2 Attention structure: queries, keys and values

In the last section, we explained the motivation for "attention" and how it behaves similarly to "memory" in a computer. There are two methods of storing memory. One is like an "array," where we can store the value at an address and retrieve it later by looking into that address. The other method is like a "hash table" or "dictionary" where we have a key that corresponds to a specific value and can retrieve the value by querying the key. Attention will use the hash table structure.

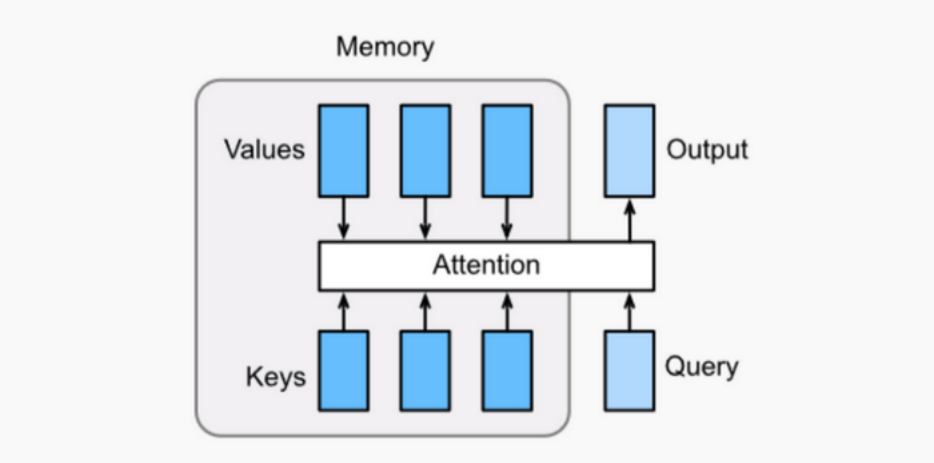

Figure 14.5: Scheme of attention. Query, Key, and Value [2].

Unlike the traditional hash table, now we can have a query vector that is not always exactly matched to keys in the hash table. So we need to design a differentiable hash table which not only can get an appropriate forward value, but also backpropagate gradients to make it work with training.

2.2.1 Attention design ideas

1. Idea[-1]: Let's look for an exact match of query **q** to a key **k** and return a value

Problem: When we initialize the model, we set a random key, if we try to get the value with a random query, we'd always fail. Additionally, we are unable to calculate the gradient using back-propagation since we will not get any matches even if we change the keys and queries by a small amount, so there is no gradient. Due to this shortcoming, for the next idea, we are hoping to have an approximate hash table.

2. Idea[0]: Scan for the closest match of query to key in the hash table, and return the value.

Problem: Now we can retrieve a value and the gradient can reach the value. However, we still have the problem of calculating a gradient for the query and key. Note that we are using the approximation of query and key. This means that changing the query or the key by a small amount will result in the same closest match and the same value, so the gradients will be 0 and cannot update the weights. Due to this shortcoming, for the next idea, we are hoping to have a "softened" hash table.

3. Idea[1]: Scan for the closest matches of query to the keys. Then return a weighted average of the values.

Since the weights now depend on both the query and the keys, we have gradients on query, keys and values. In this perspective, attention can be thought of as "queryable pooling", because keys and query values are also learnable parameters, but there are no learnable weights in the attention mechanism itself. As shown in Figure 14.6, the hidden states in the encoder are used as keys and values, and the hidden state from a decoder layer is used as a query. The attention layer will output the weighted value (where closer matches get higher weights) which is used together with the hidden state from the decoder to generate the output. This output is then fed into the next decoder layer.

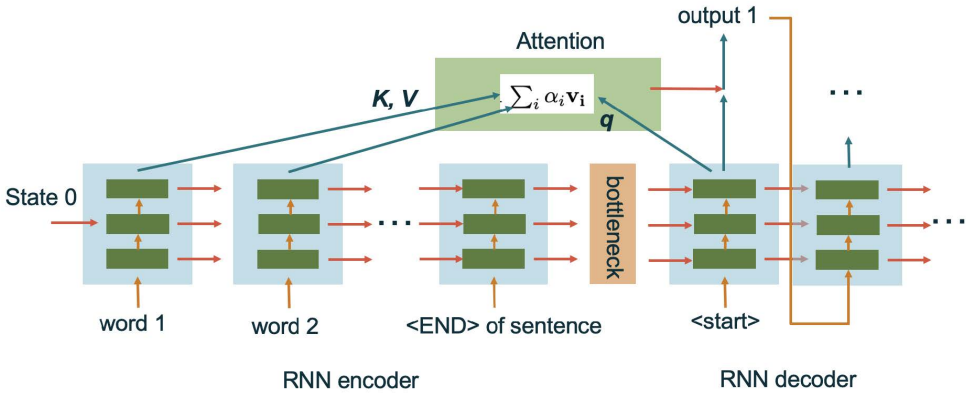

Figure 14.6: Scheme of attention in RNN layers [1].

2.3 Math behind attention

With a proper similarity function, the weighted average will be

$$\textbf{Weighted Average} = \sum_{i=1}^{n} Sim(\textbf{query}, \textbf{key}_i)\mathbf{v}_i$$

where *Sim* is the similarity function, **query** or $\mathbf{q}$ stands for query vector, $\mathbf{key_i}$ or $\mathbf{k_i}$ means ith key vector, and $\mathbf{v_i}$ is the value corresponding to the ith value vector. Vectors/scalars are represented as bold/plain letters.

We have seen similar ideas before - softmax and kernel-based methods. What should our similarity function be?

Inner product ($\mathbf{q^T k}$) or radial basis function (RBF) ($\exp(-\gamma\|\mathbf{q}-\mathbf{k}\|^2)$) are good choices. In fact, RBF is also rooted in correlation. If we expand the squared norm component in RBF, we can see that

$$\|\mathbf{q}-\mathbf{k}\|^2 = \mathbf{q}^T\mathbf{q} + \mathbf{k}^T\mathbf{k} - 2\mathbf{q}^T\mathbf{k}$$

Since $\mathbf{q}$ and $\mathbf{k}$ will be normalized (explained below), the only interesting part is $\mathbf{q}^T\mathbf{k}$, which is same as the dot product we calculated for correlation.

Note that the inner product should be normalized because the inner product will change in its relative absolute magnitude based on how big the dimensions are.

In "attention," we will use the normalized inner product:

$$e_i = \frac{\mathbf{q^T k_i}}{\sqrt{d}}$$

where d is the dimensions of $\mathbf{q}$ and $\mathbf{k}$.

Aside: Why do we normalize by $\sqrt{d}$ instead of d?

Normalizing by d ensures the similarity value never exceeds 1. However, to assign all probability mass to one item using softmax, the corresponding score e_i must be large or even infinite. By capping the scores, it becomes impossible to allocate all mass to one item, which significantly reduces expressive power.

For random initialization, we can consider $\mathbf{q^T k_i}$ as the sum of d independent and identically distributed (IID) random variables, and the Central Limit Theorem (CLT) tells us that this dot product has a standard deviation that is proportional to the square root of the number of variables being added together (d). Normalizing the dot product by dividing by the square root of the dimension helps keep the scores within a reasonable range, allowing the softmax function to produce a more balanced distribution.

After that, we use softmax to compute the weights:

$$\alpha_i = \frac{\exp(e_i)}{\sum_j \exp(e_j)}$$

Then "attention" will return $\sum_i \alpha_i v_i$ upon query.

What we wish this lecture also had to make things clearer?

I wish this lecture had included an introduction to how language is represented or how words are embedded in a real-world encoder-decoder model used in a computational setting.

References

[1] Chen, X., & Choi, Y. (2022). Fall 2022 EECS 182 Lecture 14, Attention/self-supervision [Lecture notes]. University of California, Berkeley. https://www.eecs182.org

[2] Madhav, M. [madhav_mi]. (2020, August 21). [Photograph of a beautiful landscape]. Twitter. https://twitter.com/madhav_mi/status/1296534709657190400

[3] Sahai, A. (2023). Spring 2023 EECS 182 Lecture 14, Attention/self-supervision [Image shown in class, digitized by the scribe]. University of California, Berkeley. https://www.eecs182.org