Natural Language Processing with RNNs and Attention

Character RNN

- We can train an RNN to predict the next character in a sentence.
- This character RNN can generate novel text, one character at a time.
    - We can use Keras's **TextVectorization()** layer for our character RNN.
        - Its *split* argument should be set to "character" for character-level encoding, and we can also specify the *standardize* argument to "lower" to convert the text to lowercase.
        - TextVectorization() maps each character to an integer, starting at 2. The value 0 is reserved for padding, and 1 is for unknown characters.
    - We can convert this long sequence into a dataset of windows using the **window()** method.
        - We are trying to predict the next character, so the *shift* argument should be 1.
        - We can use the **flat_map()** method to turn this into a 1D array if necessary.
    - We can use a Sequential() model with an Embedding() layer, a GRU() layer, and a Dense() output layer with a softmax activation function.
        - We must use sparse categorical cross-entropy as the loss function.

Generating Fake Text

- To generate new text using the character RNN, we can feed it text, make it predict the most likely next letter, append it to the text, make the model predict the next letter, etc.
    - This greedy decoding algorithm does not work well since the text won't be diverse.
- Instead, we can sample the next character randomly with a probability equal to the estimated probability.
    - TensorFlow's **random.categorical()** function can do this for us, but it accepts logits, so we may want to use **tf.math.log()** to get the logits.
- We can control the logits by the temperature to have more control over the diversity of the generated text.
    - A low temperature favors high-probability characters, while a high temperature gives all characters an equal probability.
    - Lower temperatures are preferred for generating rigid and precise text, such as mathematical equations.
    - Higher temperatures are preferred for generating more diverse and creative text.

- To generate more convincing text, we can sample only from the top k characters or the smallest set of top characters whose probability exceeds some threshold. This is called nucleus sampling.

Stateful RNN

- For stateless RNNs, like the model above, the model starts with a hidden state entire of zeros.
  - The model updates the hidden state at each time step and abandons it after the last step.
  - What if we used the final state after processing a training batch and use it as the initial state for the next training batch?
- Stateful RNNs can learn long-term patterns despite only backpropagating through short sequences.
  - When we use the window() function on a Keras dataset, we must set the *shift* argument to the window length.
  - Also, we should not shuffle the dataset since we need to feed each batch's final state to the next batch's initial state.
  - Lastly, we must use a batch size of 1.
  - We can set the *stateful* argument to true when creating a recurrent layer. We must also set the first layer's *batch_input_shape* argument.

Sentiment Analysis

- Sentiment analysis is the process of classifying negative and positive text.
  - A perfect example of this is classifying movie reviews.
- Building a sentiment analysis model requires separating the text into words instead of characters.
  - Keras's TextVectorization() layer automatically uses whitespace as a delimiter.
  - Some tasks may require a different method of separating words since not all languages function similarly.
    - Byte pair encoding (BPE) splits the training set into individual characters (including spaces) and merges the most frequent adjacent pairs.
    - Subword regularization tokenizes words with some randomness.
  - Since the model classifies a binary task, the output layer will use the sigmoid function. Similarly, we can use binary cross-entropy as the loss function.
- The model may suffer from limited memory even if we use a GRU() layer.

Masking

- A masked model ignores padding tokens, ensuring we have batches of equal-length sentences.
  - In Keras, we can set the *mask_zero* argument to true when creating the Embedding() layer.
    - If the model does not start with an Embedding() layer, we can use the **Masking()** layer.

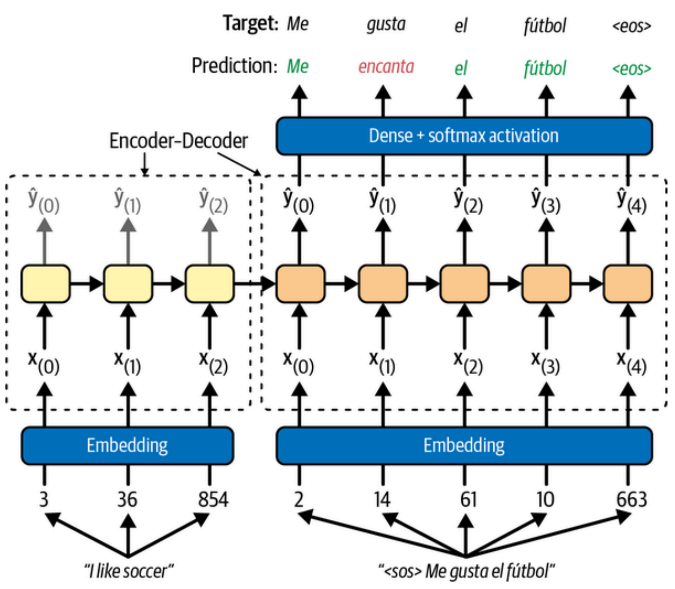
    - If a layer’s *supports_masking* argument is set to true, the mask is propagated to the next layer until it encounters a layer whose masking argument is set to false.
        - The SimpleRNN(), GRU(), LSTM(), Bidirectional(), Dense(), and a few others support masking; the Conv1D() layer does not.
    - Alternatively, we can feed the model with ragged tensors. We can set the layer’s *ragged* argument to true when creating the TextVectorization() layer.
        - We must use the layer’s adapt() method on the training set.

Reusing Pretrained Embeddings and Language Models

- We could use pre-trained word embeddings for sentiment analysis even if they were trained on another task.
    - Several words have the same meaning (e.g., “awesome” and “amazing” are both positive regardless of where they are used).
    - Google’s Word2vec embeddings, Stanford’s GloVe embeddings, and Facebook’s FastText embeddings are popular for pre-trained embeddings.
    - Pre-trained embeddings have limits: some words differ in meaning depending on the context.
    - We could even use unsupervised pre-training for NLP tasks.
- Many pre-trained language models are accessible through TensorFlow Hub.

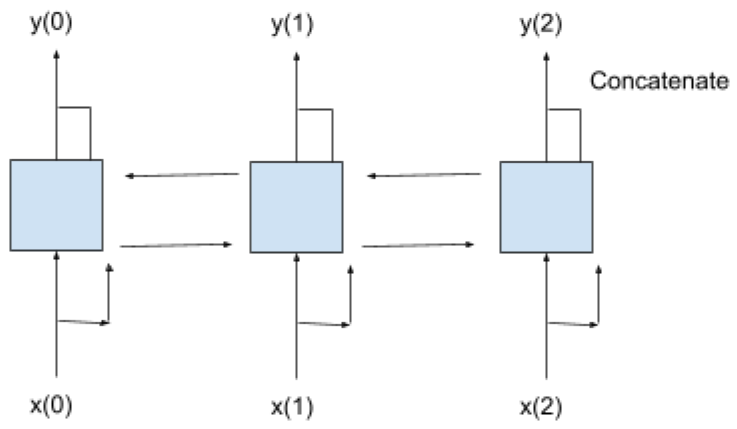
An Encoder-Decoder Network for Neural Machine Translation

- We can use NMT models to translate sentences from one language to another.
    - English sentences are fed as inputs to the encoder, and the decoder outputs the Spanish translations.
    - During training, the decoder is given as input the word that it should have output at the previous step, regardless of what it actually outputs. This is called teacher forcing.
    - For the first word, the decoder is given the start-of-sequence (SOS) token, and the decoder is expected to end the sentence with an end-of-sequence (EOS) token.
    - Each word is initially represented by its ID. Then, an Embedding() layer returns the word embedding, fed to the encoder and decoder.
    - At each step, the decoder outputs a score for each word in the output vocabulary (i.e., Spanish), and then the softmax activation function turns these scores into probabilities.

- 

- Note that some languages may have unusual characters (e.g., Spanish's ¡ and ¿), so we must remove those from the dataset.
- Also, we must add an SOS and EOS marker to the target language dataset.
- We will need two TextVectorization() layers, one per language.
  - We need sentences of equal length; we can use masking.
- We cannot call the model's predict() method to translate new sentences.
  - The decoder expects words predicted at the previous time step as inputs.
  - We can write a custom function that keeps track of the previous output and feeds it to the encoder at the next step.

Bidirectional RNNs

- A regular recurrent layer is causal, meaning it cannot look into the future.
- One solution is to run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left, then combine their outputs at each time step, typically by concatenating them. This is a bidirectional recurrent layer.
  - 

  - Keras has a **Bidirectional()** layer, which accepts an LSTM or GRU layer as an input.

    - This layer will contain four states, which is problematic. To fix this, we can concatenate the two short-term and long-term states (i.e., using **tf.concat()**).

Beam Search

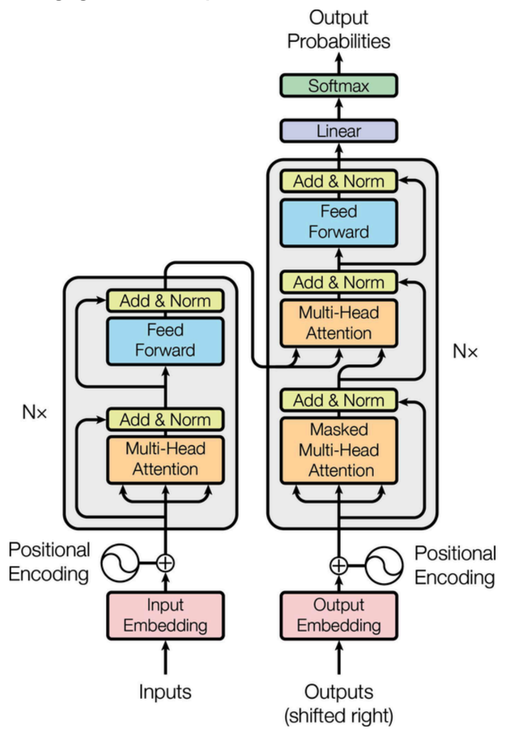
- Sometimes, the model may make mistakes predicting the next word and cannot return and fix it.
- Beam search keeps track of the k most promising sentences.
    - At each decoder step, beam search tries to extend the list by one word, keeping only the k most likely sentences.
    - The parameter k is called the beam width.
    - We compute the probabilities of each next possible word by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes.
- Due to the RNN's limited short-term memory, the model may still perform poorly at translating long sentences.

Attention Mechanisms

- The attention mechanism allows the decoder to focus on the appropriate words at each time step.
    - For example, at the time step where the decoder needs to output the word "fútbol," it will focus its attention on the word "soccer."
    - This shortens the path from an input word to its translation.
- The encoder-decoder network, with the addition of the attention mechanism, lessens the impact of the RNN's short-term memory limitations.
    - We send the encoder's outputs to the decoder, not just the encoder's final hidden state and the previous target word at each step.
    - At each time step, the decoder's memory cell computes a weighted sum of all the encoder outputs.
        - The weight a(t, i) is the weight of the i^{th} encoder output at the t^{th} decoder time step.
        - For example, if the weight a(3, 2) is larger than the weights a(3, 0) and a(3, 1), the decoder will pay more attention to the encoder's output for the 2nd word.
    - A small neural network called the alignment model (or attention layer) computes these weights.
        - This layer outputs a score for each encoder output, measuring how well each output is aligned with the decoder's previous hidden state.
        - All scores pass through a softmax layer to get a final weighted score for each encoder output. This mechanism is called Bahdanau attention. Sometimes, it is also called concatenative or additive attention.
    - Another attention mechanism is multiplicative attention.
        - The main difference is that multiplicative attention uses the dot product to compute the score.

        - This variant performs better than concatenative attention.
    - Keras provides the **Attention()** layer for multiplicative attention and **AdditiveAttention()** for concatenative attention.

The Transformer Architecture

- A transformer is a non-recurrent model, so it does not suffer as much from the vanishing of exploding gradients problem as RNNs.
    - 

    - The left part is the encoder, and the right is the decoder.
        - We must feed the sentences of the input language to the encoder and the sentences of the target language to the decoder.
        - The encoder transforms the inputs until each word's representation perfectly captures the word's meaning in the sentence's context.
        - The decoder transforms each word representation in the translated sentence into a word representation of the next word in the translation.
    - Since these layers are time-distributed, each word is treated independently.
        - The encoder's multi-head attention layer updates each word representation by paying attention to all other words in the same sentence.
        - The decoder's masked multi-head attention layer acts similarly but ignores words located after it.

Positional Encodings

- Positional encodings are dense vectors that encode a word's position within a sentence
    - The i^{th} positional encoding is added to the word embedding of the i^{th} word in the sentence.

    - We can use an Embedding layer and make it encode all the positions from 0 to the maximum sequence length in the batch, and then add the result to the word embeddings.
    - Since each position has a unique encoding, the model has access to the absolute position of each word in a sentence.
    - Keras doesn't support a positional encoding layer, but we can create a custom class that supports masking.

Multi-head Attention

- Multi-head attention is based on scaled dot-product attention.
    - The equation is $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_{keys}}})V$ where
        - Q is a matrix containing one row per query.
        - K is a matrix containing one row per key.
        - V is a matrix containing one row per value.
        - d_{keys} is the number of dimensions of each query and each key.
    - If we set the *use_scale* argument to true when creating an Attention() layer, Keras will create an additional parameter that allows the layer to learn how to scale the similarity scores properly.
    - An Attention() layer's inputs are similar to Q, K, and V but with an extra batch dimension.
- Multi-head attention is a bunch of scaled dot-product attention layers, each preceded by a linear transformation of the values, keys, and queries.
    - It applies multiple different linear transformations of the values, keys, and queries.
    - This allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word's characteristics (e.g. if it's a verb, if it's in the present tense, etc.)
    - Keras provides a **MultiHeadAttention()** layer.
        - This layer does not support automatic masking but accepts an *attention_mask* argument.
        - We may need two masks: one for the padding (we can use the **tf.math.not_equal()** method) and one for causal layers (we can use the **tf.linalg.band_part()** method).

An Avalanche of Transformer Models

- Many transformer models have been developed.
    - Radford's GPT paper:
        - The tasks were quite diverse: they included text classification, entailment (whether sentence A imposes, involves, or implies sentence B as a necessary consequence), similarity (e.g., "Nice weather today" is very similar to "It is sunny"), and question answering (given a few paragraphs

        of text giving some context, the model must answer some multiple-choice questions).
    - Google’s Bidirectional Encoder Representations from Transformers (BERT):
        - Similar architecture to GPT but with nonmasked multi-head attention layers only.
        - Masked language model: Each word in a sentence has a probability of being masked, and the model is trained to predict the masked words.
        - Next sentence prediction: A model that can predict whether two sentences are consecutive.
    - Hugging Face’s DistilBERT model:
        - Transfers knowledge from a teacher to a student model, usually much smaller than the teacher model.
        - The student model uses the teacher’s predicted probabilities for each training instance as targets.
        - This is more effective than training the student model from scratch.
    - Google’s Pathways Language model:
        - Uses 540 billion parameters and over 6,000 TPUs.
        - This model is a standard transformer, using decoders only (i.e., with masked multi-head attention layers), with just a few tweaks.
        - The model uses the chain of thought prompting technique, where the model is asked a question and answers correctly.

Vision Transformers

- In visual attention, a convolutional neural network first processes the image and outputs some feature maps. A decoder RNN with an attention mechanism generates the caption, one word at a time.
    - At each decoder time step (i.e., each word), the decoder uses the attention model to focus on just the right part of the image.
    - This allows us to understand what led the model to produce its output. This way, we can adjust the model’s training data accordingly.
- Vision transformers chop the image into little 16 × 16 squares and treat the sequence of squares as a sequence of word representations.
    - The resulting sequence of vectors can then be treated just like a sequence of word embeddings: this means adding positional embeddings and passing the result to the transformer.
    - This model performed better than CNNs but needed several times the amount of data.
        - This occurs due to transformers not having as many inductive biases.
        - An inductive bias is an implicit assumption that models make due to their architecture.
        - For example, linear models assume that the data is linear.
        - The more inductive biases a model has, assuming they are correct, the less training data the model will require.

- Generally, we won't have to implement transformers ourselves since we can use pre-trained models.

Hugging Face's Transformers Library

- Hugging Face is an AI company that provides open-source tools for NLP, vision, etc.
- We can access the tools via the **Transformers** library.
    - We can use the **pipeline()** function to specify the task we want to perform (classification, sentiment analysis, etc.)
    - Then, we can use the **classifier()** function to provide input that the library can use for its outputs (we must provide some input to the function).
    - We can specify a specific model with the pipeline function's *model* argument.
- We can fine-tune these models the same way we would in Keras.
    - Note that since the model outputs logits instead of probabilities, we must use **tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)** instead of specifying the *loss* argument during compilation.