Ishaan Salaskar
(Research Scientist Applicant)

February 18th, 2023

AI @ UIUC

NLP Polynomial Expansion:

Abstract Overview:

Taking into consideration how the goal of the project was to demonstrate knowledge of Natural Language Processing, operating in the context of predicting the expanded form of the polynomial based on its factored form served as an effective method of indicating knowledge of this field. My overall approach of handling this task came down to utilizing the artificial intelligence construct of Recurrent Neural Networks.

In essence, a recurrent neural network (RNN) is a type of neural network designed to process sequential data, such as time series and typically natural language text. While feedforward networks process input data in a static manner, RNN's provide more dynamic capabilities in the form of processing input sequences with varying lengths. A significant advantage of these sequence-to-sequence networks is their ability to manage data passed "one step at a time," meaning they take into account the current input, as well as the previous inputs as well, ultimately resulting in a more optimized training model.

Quick Note on Dependencies

- Numpy (np) provided an optimal method of devising multi-dimensional arrays
- TensorFlow (tf) served as a solid framework for developing the actual RNNs
    - The Keras API within tf included ideal functions for tokenization and vectorization of the data

Implementation Methodology:

Data Organization - Converting raw data into usable arrays

Tokenization - Split arrays into understandable units

Vectorization - Convert tokens into machine-readable vectors
Model Training - Training the RNN to predict the expanded form based on the factored

- Data Organization: Due to how the “train.txt” file, containing all the data, was simply devised of all the factors on the left side, and expansions on the right side, this raw data could not have been processed by the RNN. As a result, the provided function “load_file()” effectively created the necessary iterables that could be used further.
- Tokenization: While the previously mentioned load_file() function divided the data into arrays, this next step provided greater ease for training the network. Tokenization is a common process in natural language processing as it allows for programmers to convert large chunks of sentences and string data into more processable and comprehensive bits that can be assigned meaning. In this case, both the factored and expanded expressions were tokenized, to identify each number, set of parentheses, operators, as well as functions like sine and cosine. These tokens were then internally stored into a dictionary with a unique ID assigned to each. This allows for the network to distinguish between the elements of an inputted factored form better.
- Vectorization: Tokenizing the data does create more detailed information to be inputted into the neural network, however, most machine learning algorithms can only be inputted numerical data to operate on and train from. As a result an additional step of converting all the tokens, which are still string characters, to their respective numerical representation is quite necessary. Also known as word embedding, this process generally makes training the model more efficient.
- Model Training: Following the previous steps, this final step involves actually training the model with the given information. Providing the model with a sample input, the RNN works through the provided tokens sequentially to predict the expanded output.

Architecture Details:

- Within the predict() function, after importing all the necessary data into “factors” and “expansions”, the data was divided into 80% being allocated towards training the model, with the remaining 20% being utilized for testing the model. The “80/20 Train/Test Split” is widely used in the context of training neural networks, as it serves as an excellent method of effectively dividing the data to provide abundance in both training and testing.
- The Tokenization and Vectorization step was next applied, using the Keras function tf.keras.layers.TextVectorization(). By inputting arguments such as the “max_tokens=4000”, “output_mode=‘int’”, and “output_sequence_length = MAX_SEQUENCE LENGTH”, the function took the inputted data and tokenized according the operators/operands/parentheses, bounded the process with a maximum token vocabulary, and finally vectorized each of these tokens into a numeral state.

```
source_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=MAX_SEQUENCE_LENGTH,
)
target_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=MAX_SEQUENCE_LENGTH + 1,
)
```

- Next the functions “format_dataset()” as well as “make_dataset()” were implemented to devise the two iterables “train_ds" as well as “val_ds”. These respective functions further work with the data to create an array consisting only of the training data, as well as an array solely consisting of the test data. Furthermore, within make_dataset(), the map() function is utilized to apply the previously mentioned format_dataset() function to every item in the inputted array, in a parallel manner to optimize the program.

- Finally, the implementation of the RNN utilized an alternative architectural design known as Gated Recurrent Units (GRU). As discussed previously, RNNs work on the concept of "timesteps", or units of time that capture data sequentially. However, a significant issue that emerges with this design is the vanishing gradient problem. Taking into account how each consecutive hidden layer utilizes data from the previous, the problem primarily arises when the error function, with respect to the parameters, becomes increasingly small as it propagates through time, thus making it quite difficult for the model to learn long-term data. The GRU architecture instead tackles this problem by utilizing a gate function that selectively updates and forgets certain information.

```
source = keras.Input(shape=(None,), dtype="int64", name="Factors")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
encoded_source = layers.Bidirectional(layers.GRU(latent_dim),
merge_mode="sum")(x)

past_target = keras.Input(shape=(None,), dtype="int64", name="Expansions")
x = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(past_target)
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
x = decoder_gru(x, initial_state=encoded_source)
x = layers.Dropout(0.5)(x)
target_next_step = layers.Dense(vocab_size, activation="softmax")(x)
seq2seq_rnn = keras.Model([source, past_target], target_next_step)
```

- As indicated in the code above, the embedding layer is created using keras.layers.
- The bidirectional GRU architecture uses inputs from both past inputs and anticipated inputs to predict the output. Since the model is bidirectional, it consists of a separate set of weights and biases for both the feedforward and backward moving states.
- Ultimately, the following parameters were utilized to compile the model

```
seq2seq_rnn.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])
seq2seq_rnn.fit(train_ds, epochs=7, validation_data=val_ds)
```