Documentation of Machine Learning Models for Sequence Processing

Ziad Amer & Ahmed Ashour

September 6, 2024

1 Introduction

This document outlines the development and training of various machine learning models designed to handle sequence processing tasks, specifically targeting machine translation between English and French. We discuss the architectural choices, training processes, and optimizations applied throughout the projects.

2 Model Architectures

2.1 GRU-based Seq2Seq Model

In this implementation, we employ a Gated Recurrent Unit (GRU)-based Seq2Seq model for English-to-French translation. The GRU units are chosen for their ability to handle sequential data efficiently while requiring fewer parameters than LSTMs, making them faster to train.

2.1.1 Model Architecture

Encoder: The encoder processes English input sentences by first embedding them into dense vectors using an embedding layer. These embeddings are then passed through a GRU layer, which produces the hidden state that will be used to initialize the decoder. Unlike LSTMs, GRUs do not maintain a separate cell state; they only use a hidden state:

```
encoder_inputs = Input(shape=(max_eng_len,))
encoder_embedding = Embedding(eng_vocab_size, embedding_dim)(encoder_inputs)
encoder_gru, state_h = GRU(latent_dim, return_state=True)(encoder_embedding)
encoder_states = [state_h]
```

Decoder: The decoder receives the French input sequences (shifted by one token) and the encoder's final hidden state as the initial state. It also consists of an embedding layer followed by a GRU layer. The decoder outputs are passed through a dense layer with a softmax activation function to predict the next word in the sequence:

```
decoder_inputs = Input(shape=(max_fra_len-1,))
decoder_embedding = Embedding(fra_vocab_size, embedding_dim)(decoder_inputs)
decoder_gru = GRU(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, state_h_dec = decoder_gru(decoder_embedding, initial_state=
    encoder_states)
decoder_dense = Dense(fra_vocab_size, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

2.1.2 Training Process

The model is compiled using the Adam optimizer and sparse categorical cross-entropy loss function, which is ideal for multi-class classification tasks. Several callbacks were employed during training, including:

- **EarlyStopping**: To halt training when validation loss stops improving.
- **ModelCheckpoint**: To save the best-performing model during training.
- **LearningRateScheduler**: To reduce the learning rate as training progresses.

```
model.compile(optimizer=Adam(learning_rate=0.1),
              loss='sparse_categorical_crossentropy')

history = model.fit(
    [eng_sequences, fra_input_sequences],
    np.expand_dims(fra_output_sequences, -1),
    epochs=20,
    batch_size=256,
    validation_split=0.2,
    callbacks=[early_stopping, model_checkpoint, lr_scheduler, tensorboard]
)
```

2.1.3 Performance Evaluation

The training and validation loss were tracked to monitor the model's performance. The GRU-based Seq2Seq model is able to learn the complex relationships between English and French sentences effectively, as demonstrated by the smooth reduction in both training and validation losses.

2.2 LSTM-based Seq2Seq Model

The Long Short-Term Memory (LSTM)-based Seq2Seq model is designed to handle sequence-to-sequence tasks such as machine translation by encoding a source sequence (in English) and decoding it into a target sequence (in French). This architecture leverages LSTM units in both the encoder and decoder to capture long-term dependencies within the sequences.

2.2.1 Model Architecture

Encoder: The encoder processes the input English sentences using an embedding layer followed by an LSTM layer. The LSTM outputs a hidden state (h) and a cell state (c), which are passed to the decoder as the initial states:

```
encoder_inputs = Input(shape=(max_eng_len,))
encoder_embedding = Embedding(eng_vocab_size, embedding_dim)(encoder_inputs)
encoder_lstm, state_h, state_c = LSTM(latent_dim, return_state=True)(
    encoder_embedding)
encoder_states = [state_h, state_c]
```

Decoder: The decoder takes the French input sequences (shifted by one position to predict the next word) along with the encoder's final states as its initial states. It uses another LSTM layer followed by a dense layer with a softmax activation to generate the target sequences.

```
decoder_inputs = Input(shape=(max_fra_len-1,))
decoder_embedding = Embedding(fra_vocab_size, embedding_dim)(decoder_inputs)
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_embedding, initial_state=encoder_states)
decoder_dense = Dense(fra_vocab_size, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

2.2.2 Training Process

The model was compiled with the Adam optimizer and trained using sparse categorical cross-entropy loss, which is suitable for multi-class classification tasks. The training used early stopping to prevent overfitting, model checkpointing to save the best model, and a learning rate scheduler to adjust the learning rate dynamically:

```
model.compile(optimizer=Adam(learning_rate=0.1), loss='
    sparse_categorical_crossentropy')
history = model.fit(
    [eng_sequences, fra_input_sequences],
    np.expand_dims(fra_output_sequences, -1),
    epochs=20,
    batch_size=256,
    validation_split=0.2,
    callbacks=[early_stopping, model_checkpoint, lr_scheduler, tensorboard]
)
```

2.2.3 Performance Evaluation

The model's performance was monitored using the loss and validation loss over time. The model exhibited a decreasing loss trend, indicating effective learning. This LSTM-based Seq2Seq model captures sequential dependencies and is capable of learning translation mappings between English and French. The model's success is demonstrated by the reduction in training and validation losses.

2.3 Transformer Model

The Transformer model, introduced by [4], represents a significant advancement in sequence-to-sequence tasks by leveraging attention mechanisms to capture contextual relationships within the data. Unlike Recurrent Neural Networks (RNNs), Transformers do not rely on sequential data processing, allowing for greater parallelization and efficiency.

2.3.1 Architecture Overview

The Transformer architecture consists of an encoder and a decoder, each comprising multiple layers of multi-head attention and feed-forward neural networks. The primary components of the Transformer model implemented in this project are as follows:

- **Embedding and Positional Encoding**: Both the encoder and decoder begin with embedding layers that convert input tokens into dense vectors. Since Transformers lack a sense of order inherent in RNNs, positional encoding is added to inject information about the position of each token in the sequence.
- **Multi-Head Attention**: This mechanism allows the model to focus on different parts of the input sequence simultaneously, capturing various aspects of the contextual relationships.
- **Feed-Forward Networks**: After attention, each layer includes a fully connected feed-forward network to process the attended information further.
- **Layer Normalization and Residual Connections**: These techniques stabilize and accelerate the training process by normalizing the inputs and adding shortcut connections around each sub-layer.

2.3.2 Positional Encoding

Since the Transformer model does not process input data sequentially, positional encoding is essential to provide the model with information about the position of each token in the sequence. The positional encoding is defined as:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{1}$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{2}$$

where:

- pos is the position of the token in the sequence.
- i is the dimension index.
- d_{model} is the dimensionality of the embeddings.

This formulation allows the model to easily learn to attend by relative positions since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

2.3.3 Attention Mechanism

The core of the Transformer model is the attention mechanism, specifically the **Scaled Dot-Product Attention**. Given queries (Q), keys (K), and values (V), the attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{3}$$

where d_k is the dimensionality of the keys. The scaling by $\sqrt{d_k}$ prevents the dot products from growing too large, which can push the softmax function into regions with extremely small gradients.

Multi-Head Attention extends this concept by running multiple attention mechanisms in parallel, allowing the model to jointly attend to information from different representation subspaces at different positions. The outputs of these attention heads are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O \tag{4}$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{5}$$

and W_i^Q, W_i^K, W_i^V, and W^O are learned projection matrices.

2.3.4 Model Implementation

The Transformer model was implemented using TensorFlow and Keras, following the standard architecture with modifications tailored to the English-French translation task. Below are the key components of the implementation:

Positional Encoding Function The positional encoding is implemented as follows:

```
def positional_encoding(max_len, d_model):
    angle_rads = np.arange(max_len)[:, np.newaxis] / np.power(10000,
        (2 * (np.arange(d_model)[np.newaxis, :] // 2)) / d_model)
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
    return tf.cast(angle_rads, dtype=tf.float32)
```

Encoder Layer Each encoder layer consists of a multi-head attention mechanism followed by a feed-forward network, with layer normalization and residual connections applied after each sub-layer:

```
def encoder_layer(d_model, num_heads, dff, maximum_position_encoding):
    inputs = Input(shape=(max_eng_len,))

    # Embedding and Positional Encoding
    embedding = Embedding(eng_vocab_size, d_model)(inputs)
    pos_encoding = positional_encoding(maximum_position_encoding, d_model)
    embedding *= tf.math.sqrt(tf.cast(d_model, tf.float32))
    embedding += pos_encoding[:max_eng_len, :]

    # Multi-Head Attention
    attention = MultiHeadAttention(num_heads=num_heads, key_dim=d_model)(embedding,
    embedding)
    attention = LayerNormalization(epsilon=1e-6)(attention + embedding)

    # Feed Forward Network
    ffn_output = Dense(dff, activation='relu')(attention)
    ffn_output = Dense(d_model)(ffn_output)
    ffn_output = LayerNormalization(epsilon=1e-6)(ffn_output + attention)

    return Model(inputs, ffn_output)
```

Decoder Layer The decoder layer extends the encoder by including an additional multi-head attention mechanism that attends to the encoder's output:

```
def decoder_layer(d_model, num_heads, dff, maximum_position_encoding):
    inputs = Input(shape=(max_fra_len-1,))
    enc_output = Input(shape=(max_eng_len, d_model))

    # Embedding and Positional Encoding
    embedding = Embedding(fra_vocab_size, d_model)(inputs)
    pos_encoding = positional_encoding(maximum_position_encoding, d_model)
    embedding *= tf.math.sqrt(tf.cast(d_model, tf.float32))
    embedding += pos_encoding[:max_fra_len-1, :]

    # Multi-Head Attention (Self-attention)
    attention1 = MultiHeadAttention(num_heads=num_heads, key_dim=d_model)(embedding,
    embedding)
    attention1 = LayerNormalization(epsilon=1e-6)(attention1 + embedding)

    # Multi-Head Attention (Encoder-Decoder Attention)
    attention2 = MultiHeadAttention(num_heads=num_heads, key_dim=d_model)(attention1,
     enc_output)
    attention2 = LayerNormalization(epsilon=1e-6)(attention2 + attention1)

    # Feed Forward Network
    ffn_output = Dense(dff, activation='relu')(attention2)
    ffn_output = Dense(d_model)(ffn_output)
    ffn_output = LayerNormalization(epsilon=1e-6)(ffn_output + attention2)

    return Model([inputs, enc_output], ffn_output)
```

Model Compilation and Training The complete Transformer model is constructed by connecting the encoder and decoder, followed by a final dense layer with softmax activation for output prediction. The model is compiled using the Adam optimizer and trained with the sparse categorical cross-entropy loss function.

```
# Hyperparameters
d_model = 256
num_heads = 8
dff = 512
maximum_position_encoding = max(max_eng_len, max_fra_len)

```

```
# Build the encoder and decoder
encoder_inputs = Input(shape=(max_eng_len,))
encoder = encoder_layer(d_model, num_heads, dff, maximum_position_encoding)
encoder_outputs = encoder(encoder_inputs)

decoder_inputs = Input(shape=(max_fra_len-1,))
decoder = decoder_layer(d_model, num_heads, dff, maximum_position_encoding)
decoder_outputs = decoder([decoder_inputs, encoder_outputs])

# Final linear and softmax layer
outputs = Dense(fra_vocab_size, activation='softmax')(decoder_outputs)

# Define the model
model = Model([encoder_inputs, decoder_inputs], outputs)
model.compile(optimizer=Adam(learning_rate=0.1), loss='
    sparse_categorical_crossentropy')
```

2.3.5 Training Process

The model was trained on a dataset of 5000 English-French sentence pairs. Early stopping and model checkpointing were employed to prevent overfitting and save the best-performing model. A learning rate scheduler was also used to adjust the learning rate dynamically during training.

```
# Callbacks
early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
model_checkpoint = keras.callbacks.ModelCheckpoint('seq2seq_transformer.keras',
    save_best_only=True)

def scheduler(epoch, lr):
    return float(lr * tf.math.exp(-0.1))

lr_scheduler = keras.callbacks.LearningRateScheduler(scheduler)
tensorboard = keras.callbacks.TensorBoard(log_dir='logs')

# Training
history = model.fit(
    [eng_sequences, fra_input_sequences],
    np.expand_dims(fra_output_sequences, -1),
    epochs=20,
    batch_size=256,
    validation_split=0.2,
    callbacks=[early_stopping, model_checkpoint, lr_scheduler, tensorboard]
)
```

2.3.6 Performance Evaluation

The training and validation loss were monitored over epochs to assess the model's learning progress and generalization capabilities. The loss curves typically exhibit a decreasing trend, indicating effective learning, while validation loss helps in identifying potential overfitting.

```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='validation')
plt.legend()
plt.show()
```

2.3.7 Conclusion

The Transformer model effectively captures the intricate dependencies between English and French sentences through its attention mechanisms and parallel processing capabilities. The implementation demonstrates the model's ability to learn translation mappings efficiently, as evidenced by the decreasing loss during training. Future work may involve experimenting with deeper architectures, different hyperparameters, or larger datasets to further enhance performance.

2.4 Pretrained T5-small Model for English-to-French Translation

We utilized the pretrained T5-small model from Huggingface for English-to-French translation. T5 (Text-to-Text Transfer Transformer) is a model that formulates every NLP task as a text-to-text problem. The T5-small variant is a lightweight version with 60 million parameters, making it suitable for faster experimentation on moderate-sized datasets.

In our case, the model was fine-tuned on a subset of 5000 English and French sentence pairs from the dataset. The input English sentences were preprocessed by adding the prefix "translate English to French: " before tokenization to indicate the translation task to the model.

2.4.1 Model Architecture

The T5 model is based on the standard Transformer architecture. Like the original Transformer, it employs an encoder-decoder structure, where both the encoder and decoder consist of multiple layers of self-attention and feed-forward networks.

Attention Mechanism: The attention mechanism in T5 is based on the scaled dot-product attention, defined as in Equation 3.

The softmax operation ensures that the model attends to different parts of the input sequence by assigning appropriate weights to each token.

Feed-Forward Network: Each attention block is followed by a position-wise feed-forward network, which is applied independently to each position:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{6}$$

where:

- x is the input to the feed-forward network.
- W_1, W_2 are learned weight matrices.
- b_1, b_2 are learned biases.

This helps the model capture non-linear relationships between the tokens.

2.4.2 Fine-tuning Process

The fine-tuning process involved training the T5-small model using TensorFlow's dataset API for efficient batch processing. The model was trained for three epochs using the Adam optimizer, with the learning rate set to 3×10^{-5}. The training objective is to minimize the cross-entropy loss between the predicted French tokens and the true target tokens.

For each English sentence, the corresponding French translation is treated as the target sequence. During training, the model learns to map the source sentence to the target sentence token by token.

The Adam optimizer is used with the following update rule:

$$\theta_t = \theta_{t-1} - \eta \frac{m_t}{\sqrt{v_t}} \tag{7}$$

where:

- η is the learning rate.
- m_t is the first moment (mean of the gradients).
- v_t is the second moment (uncentered variance of the gradients).

After training, the model was saved for future inference. During inference, sentences are passed to the model, and the output is decoded into the translated French sentence. The following Python code demonstrates the inference process:

```
>>> translate("Hello, how are you?")
Hallo, comment ca va?
```

2.4.3 Translation Inference

For inference, the input sentence is tokenized and encoded into input IDs, which are passed to the model. The decoder generates the translated output by autoregressively predicting each token. Beam search or greedy decoding can be used to generate translations, depending on the desired trade-off between speed and quality. The output sequence is then decoded back into human-readable text.

The overall architecture of T5 allows for rapid prototyping and evaluation of translation tasks using pretrained models. The training process leverages transfer learning, which allows the model to build on prior knowledge from large-scale pretraining and apply it to the specific task of English-to-French translation.

2.4.4 Advantages of T5

Using the T5-small model offered several advantages:

- **Flexibility**: T5's text-to-text formulation allows it to handle a wide range of NLP tasks, including translation, without requiring task-specific changes to the architecture.
- **Transfer Learning**: By leveraging a pretrained model, we were able to achieve competitive performance on our dataset with limited data and training time.
- **Efficiency**: The T5-small model is computationally efficient compared to larger models, making it ideal for experimentation with limited resources.

This approach allowed for rapid prototyping and evaluation of the translation task using a powerful pretrained model, which was fine-tuned effectively to handle English-to-French translation.

3 Dataset and Preprocessing

The dataset comprises pairs of English and French sentences. We used Keras Tokenizer for text processing, converting text data into sequences of integers and padding them to ensure uniform input size.

4 Training

Models were compiled with Adam optimizer and sparse categorical cross-entropy loss function. We employed callbacks like EarlyStopping and ModelCheckpoint for better training control.

5 Code Snippets

5.1 Tokenization and Padding

```
eng_tokenizer = Tokenizer(filters='')
eng_tokenizer.fit_on_texts(input_texts)
eng_sequences = eng_tokenizer.texts_to_sequences(input_texts)
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_sequences = pad_sequences(eng_sequences, maxlen=max_eng_len, padding='post')
```

5.2 Model Setup - GRU Example

```
encoder_inputs = Input(shape=(max_eng_len,))
encoder_embedding = Embedding(eng_vocab_size, embedding_dim)(encoder_inputs)
encoder_gru, state_h = GRU(latent_dim, return_state=True)(encoder_embedding)
encoder_states = [state_h]

decoder_inputs = Input(shape=(max_fra_len-1,))
decoder_embedding = Embedding(fra_vocab_size, embedding_dim)(decoder_inputs)
decoder_gru = GRU(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, state_h_dec = decoder_gru(decoder_embedding, initial_state=
    encoder_states)
decoder_dense = Dense(fra_vocab_size, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

6 Results and Discussion

- **Accuracy and Loss**: All models show improvement in accuracy over epochs, with corresponding decreases in loss. The LSTM model had a notably rough start but improved significantly, highlighting the impact of initial conditions and learning rate adjustments.
- **Valuation Metrics**: Validation accuracy and loss were also recorded to assess model generalizability. In most cases, validation loss decreased, but some fluctuations indicate potential overfitting or instability in training dynamics.
- **Sparse Categorical Crossentropy**: This metric remained relatively stable, suggesting that the models were learning effectively to classify sequences correctly across different contexts.

7 Conclusion

The document captures the methodology and technical details involved in the development of sequence processing models, serving as a comprehensive guide for further experimentation and review.

References

[1] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing*, 2014.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[3] Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2019.

[4] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems*, 2017.