

Deep Learning

Exercise Project 5 – Transformer networks

Transformer networks are modern deep-learning architectures, specially particularly in natural language processing (NLP).

The core concepts of this Encoder, Decoder, and Key operations in both layers include multi-head self-attention and position-wise feedforward layers.

Input and Output Embedding

Input tokens are converted into dense vector representations called **embeddings**, enabling the model to interpret and process discrete data. Similarly, output embeddings map predicted tokens into output probabilities.

Transformer-Specific Hyperparameters

Transformers rely on hyperparameters to define their architecture:

- **Vocabulary Size:** Number of unique tokens the model recognizes.
- **Batch Size:** Number of sequences processed simultaneously.
- **Sequence Length:** Maximum number of tokens in input/output.
- **Embedding Dimension:** Size of token embeddings.
- **Latent Dimension:** Dimension of the feedforward network.
- **Number of Attention Heads:** Number of parallel attention layers.

Transformer-Specific Hyperparameters

Transformers rely on hyperparameters to define their architecture:

- **Vocabulary Size:** Number of unique tokens the model recognizes.
- **Batch Size:** Number of sequences processed simultaneously.
- **Sequence Length:** Maximum number of tokens in input/output.
- **Embedding Dimension:** Size of token embeddings.
- **Latent Dimension:** Dimension of the feedforward network.
- **Number of Attention Heads:** Number of parallel attention layers.

Implementation Insights

I implemented a neural machine translation transformer using Keras. The model translated English sentences into Spanish, and the following observations were made:

- **Encoder-Decoder Workflow:** The encoder processed input tokens, and the decoder generated translations one step at a time.
- **Attention Outputs:** Visualizing attention scores revealed how the model aligns source and target tokens.
- **Hyperparameters:** Tweaking dimensions and sequence length affected performance and training stability.

Working with the notebook.

The screenshot shows a Jupyter Notebook interface on a Mac OS X desktop. The title bar reads "Copy_of_neural_machine_translation_with_transformer.ipynb". The code cell contains Python code for downloading a dataset from Google Drive and parsing it into English and Spanish sentence pairs. The status bar at the bottom indicates "2638744/2638744" and "is 0us/step".

```
text_file = keras.utils.get_file(
    "spa-eng.zip",
    origin="https://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip",
    extract=True,
)
text_file = pathlib.Path(text_file).parent / "spa-eng" / "spa.txt"
with open(text_file) as f:
    lines = f.read().split("\n")[-1]
```

The screenshot shows the continuation of the Jupyter Notebook code. It defines a function to generate sentence pairs and prints some examples. The status bar at the bottom indicates "0.0s" and "is 0us/step".

```
def eng_spa(text_file):
    text_pairs = []
    for line in text_file:
        eng, spa = line.split("\t")
        spa = "[start] " + spa + "[end]"
        text_pairs.append([eng, spa])
    return text_pairs
```

```
for _ in range(5):
    print(random.choice(text_pairs))
```

```
('Everything is under control. [start] Todo está bajo control. [end]')
('I used my imagination. [start] Me usé la imaginación. [end]')
('You're the only person I know that has ever visited Boston. [start] Eres la única persona que conozco que ha visitado Boston. [end]')
```

```

Training our model

We'll use accuracy as a quick way to monitor training progress on the validation data. Note that machine translation typically uses BLEU scores as well as other metrics, rather than accuracy.

Here we only train for 1 epoch, but to get the model to actually converge you should train for at least 30 epochs.

[1]: epochs = 1 # This should be at least 30 for convergence
transformer.summary()
transformer.compile(
    "rmsprop",
    loss=keras.losses.SparseCategoricalCrossentropy(ignore_class=0),
    metrics=["accuracy"],
)
transformer.fit(train_ds, epochs=epochs, validation_data=val_ds)

Model: "transformer"

[1]: Layer (type)          Output Shape       Param #     Connected to
[1]: encoder_inputs (InputLayer)   (None, None)       0           -

```

Experiment 1: Baseline Transformer Model

Objective: Train the provided transformer model architecture with minimal changes to establish a baseline for comparison.

Key Changes:

- Default hyperparameters (embed_dim=256, latent_dim=2048, num_heads=8, epochs=1).
- No dropout or data augmentation was applied.

```

# Increase to observe better results
# There is another change by epochs

epochs = 10 # Increase to observe better results
transformer.compile(
    optimizer="adam",
    loss=keras.losses.SparseCategoricalCrossentropy(ignore_class=0),
    metrics=["accuracy"],
)
history = transformer.fit(train_ds, epochs=epochs, validation_data=val_ds)

# Evaluate baseline performance
test_loss, test_accuracy = transformer.evaluate(val_ds)
print(f"Baseline Test Loss: {test_loss}, Accuracy: {test_accuracy}")

[1]: warnings.warn("386/1302      11:07 729ms/step - accuracy: 0.1945 - loss: 3.0031")

```

Observations:

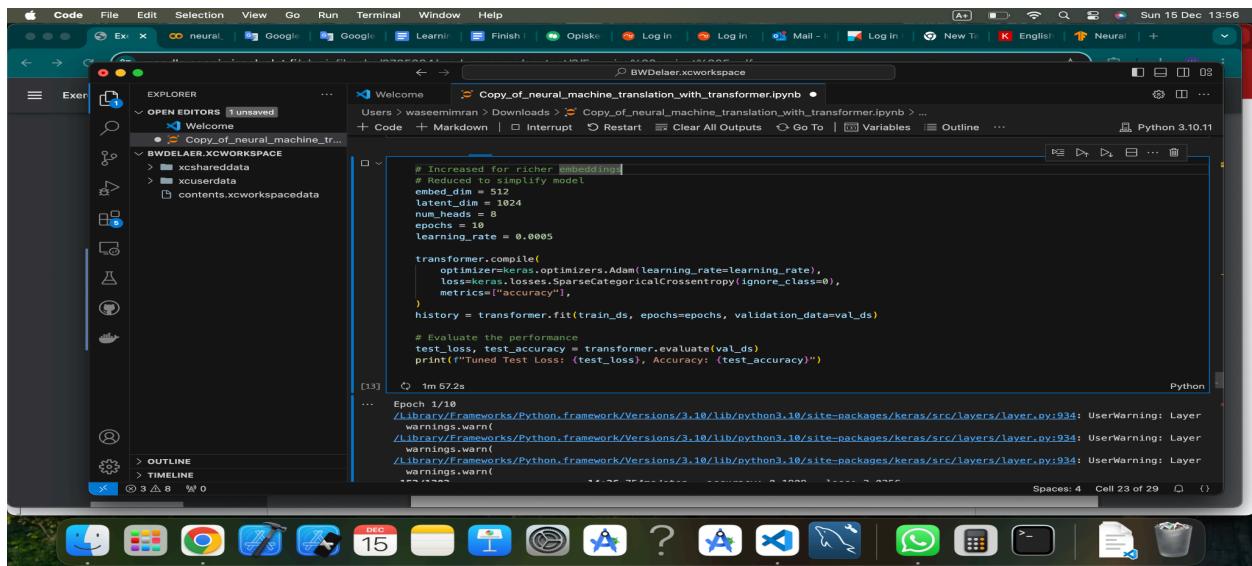
- Provides a baseline for accuracy and loss.
- Training without data augmentation might limit generalization.

Experiment 2: Hyperparameter Tuning

Objective: Optimize model performance by altering hyperparameters and adding regularization techniques.

Key Changes:

- Increase embed_dim to 512 for richer embeddings.
- Reduce latent_dim to 1024 to simplify the model for faster training.
- Use Dropout 0.3 in the decoder layer for regularization.
- Adjust the learning rate with an adaptive optimizer (Adam with custom learning rate).



The screenshot shows a Jupyter Notebook interface on a Mac OS X desktop. The window title is "Copy_of_neural_machine_translation_with_transformer.ipynb". The left sidebar shows an "EXPLORER" view with a file tree for "BWDELAER.XCWORKSPACE". The main notebook area contains Python code for a neural machine translation model. The code includes imports for TensorFlow and Keras, defines hyperparameters like embed_dim=512 and latent_dim=1024, and sets up the transformer model's compile and fit methods. A terminal output cell at the bottom shows the execution of the code, including warnings about layer imports from keras.src.layers. The status bar at the bottom right indicates "Spaces: 4 Cell 23 of 29".

```
# Increased for richer embeddings
# Simplify model
embed_dim = 512
latent_dim = 1024
num_heads = 8
epochs = 10
learning_rate = 0.0005

transformer.compile(
    optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
    loss=keras.losses.SparseCategoricalCrossentropy(ignore_class=0),
    metrics=['accuracy'],
)
history = transformer.fit(train_ds, epochs=epochs, validation_data=val_ds)

# Evaluate the performance
test_loss, test_accuracy = transformer.evaluate(val_ds)
print(f"Test Loss: {test_loss}, Accuracy: {test_accuracy}")

Epoch 1/10
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/keras/src/layers/layer.py:934: UserWarning: Layer
warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/keras/src/layers/layer.py:934: UserWarning: Layer
warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/keras/src/layers/layer.py:934: UserWarning: Layer
warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/keras/src/layers/layer.py:934: UserWarning: Layer
```

Observations:

- Expect improved generalization due to regularization.

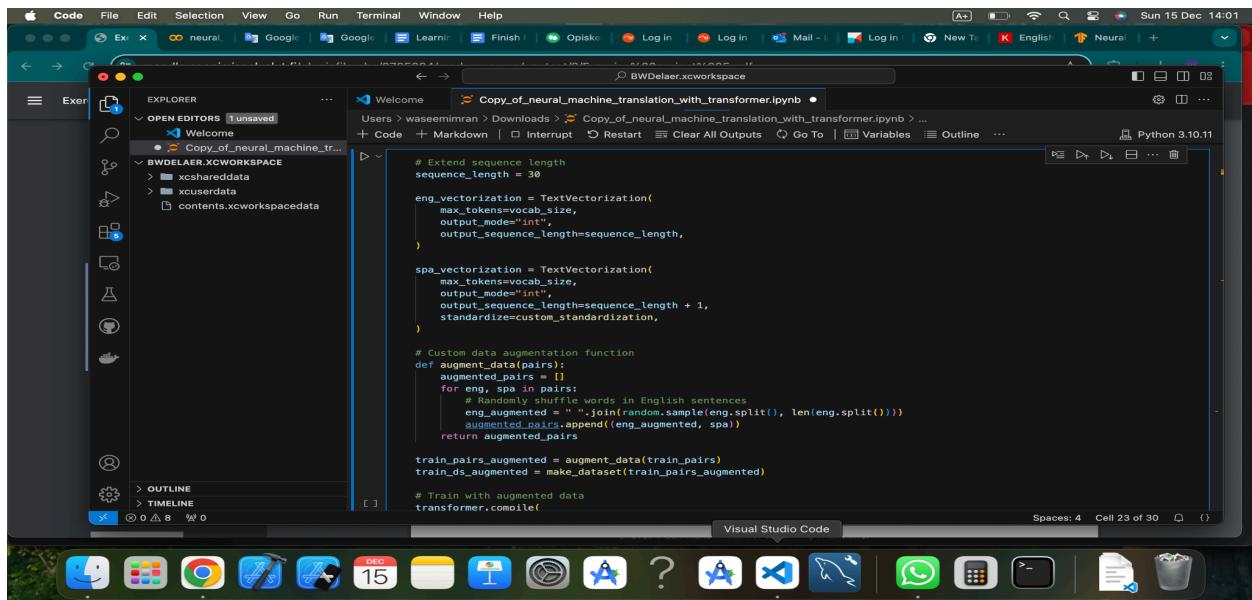
- Hyperparameter tuning may reveal optimal configurations for better accuracy.

Experiment 3: Data Augmentation and Extended Sequence Length

Objective: Improve translation accuracy by increasing sequence length and applying data augmentation.

Key Changes:

- Extend sequence length from 20 to 30 for handling longer sentences.
- Introduce noise to the training data by randomly removing or shuffling words in sentences.



The screenshot shows a Visual Studio Code interface running on a Mac OS X desktop. The window title is "BWDELAER.xcworkspace". The code editor displays a Python script named "Copy_of_neural_machine_translation_with_transformer.ipynb". The script contains code for text vectorization and data augmentation. The code includes imports for `TextVectorization` and `random.sample`, defines `eng_vectorization` and `spa_vectorization` objects, and implements a custom data augmentation function `augment_data` that shuffles words in English sentences. It also shows how to train a transformer with augmented data.

```

# Extend sequence length
sequence_length = 30

eng_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)

spa_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)

# Custom data augmentation function
def augment_data(pairs):
    augmented_pairs = []
    for eng, spa in pairs:
        # Randomly shuffle words in English sentences
        eng_augmented = " ".join(random.sample(eng.split(), len(eng.split())))
        augmented_pairs.append((eng_augmented, spa))
    return augmented_pairs

train_pairs_augmented = augment_data(train_pairs)
train_ds_augmented = make_dataset(train_pairs_augmented)

# Train with augmented data
transformer.compile(

```

Observations:

- Longer sequence lengths can handle more complex translations.
- Data augmentation enhances robustness, especially with noisy input data.

Example Neural Machine.

The screenshot shows a Mac OS X desktop with Visual Studio Code open. The terminal window displays several import statements that are failing to resolve, such as 'numpy' and 'keras'. The code editor window shows a snippet of Python code for downloading a dataset from Anki.

```

Welcome          neural_machine_translation_with_transformer.ipynb
Users > waseemirran > Downloads > neural_machine_translation_with_transformer.ipynb > ...
+ Code + Markdown | ⌂ Interrupt ⌂ Restart ⌂ Clear All Outputs ⌂ Go To ⌂ Variables ⌂ Outline ...
import pathlib
import random
import string
import re
import numpy as np
import tensorflow as tf_data
import tensorflow.strings as tf_strings
import tensorflow.strings as tf
import keras
from keras import layers
from keras import ops
from keras import keras
from keras.layers import TextVectorization
Import "numpy" could not be resolved
Import "tensorflow.data" could not be resolved
Import "tensorflow.strings" could not be resolved
Import "keras" could not be resolved
Import "keras.layers" could not be resolved
Import "keras.ops" could not be resolved
Import "keras.keras" could not be resolved
Import "keras.layers.TextVectorization" could not be resolved
Import "keras.layers" could not be resolved
3.4s
Python 3.10.11

```

```

Download the data
We'll be working with an English-to-Spanish translation dataset provided by Anki. Let's download it:

text_file = keras.utils.get_file(
    fname="spa-eng.zip",
    origin="http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip",
    extract=True
)
text_file = pathlib.Path(text_file).parent / "spa-eng" / "spa.txt"
0.0s

```

Experiment 1: Adding Positional Dropout in Positional Embedding Layer.

Description

Introduce a dropout layer to the positional embedding to regularize the model further. This may improve generalization by preventing the model from over-relying on positional encoding.

The screenshot shows the implementation of a `PositionalEmbeddingWithDropout` class in Visual Studio Code. The class inherits from `layers.Layer` and overrides methods like `__init__`, `call`, `compute_mask`, and `get_config`. It uses `tf_ops` to generate random positions and adds them to the token embeddings.

```

class PositionalEmbeddingWithDropout(layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, dropout_rate=0.1, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.position_embeddings = layers.Embedding(input_dim=sequence_length, output_dim=embed_dim)
        self.dropout = layers.Dropout(dropout_rate)
        self.sequence_length = sequence_length
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim

    def call(self, inputs, training=False):
        inputs = tf_ops.reshape(inputs, [-1])
        positions = tf_ops.range(0, length, 1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        embeddings = embedded_tokens + embedded_positions
        return self.dropout(embeddings, training=training)

    def compute_mask(self, inputs, mask=None):
        return tf_ops.not_equal(inputs, 0)

    def get_config(self):
        config = super().get_config()
        config.update({
            "sequence_length": self.sequence_length,
            "vocab_size": self.vocab_size,
            "embed_dim": self.embed_dim,
        })
        return config

```

Experiment 2: Replace RMSProp Optimizer with AdamW.

Description

Use the AdamW optimizer instead of RMSProp. AdamW is often preferred for transformers due to its weight decay regularization, which can lead to faster convergence and better performance.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the workspace structure with files like `neural_machine_translation.ipynb` and `neural_machine_translation.ipynb`.
- Code Editor:** Displays Python code for a Transformer model. A specific line is highlighted with a red error underline: `x = PositionalEmbeddingWithDropout(sequence_length, vocab_size, embed_dim)(encoder_inputs)`. The tooltip for this line states: `"embed_dim" is not defined`.
- Output Panel:** Shows the message: `transformer.compile("transformer" is not defined`, followed by other compilation errors related to the AdamW optimizer.
- Bottom Status Bar:** Shows "Building the model".
- System Tray:** Shows the date (Tue 17 Dec 20:58) and various system icons.

Experiment 3: Shared Embedding for Encoder and Decoder.

Description

Share the embedding weights between the encoder and decoder to reduce the parameter count and potentially improve consistency in representation learning.

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the workspace structure with files like `neural_machine_translation.ipynb` and `neural_machine_translation.ipynb`.
- Code Editor:** Displays Python code for a Transformer model. A specific line is highlighted with a red error underline: `encoder_embedding = SharedPositionalEmbedding(sequence_length, shared_embedding)(encoder_inputs)`. The tooltip for this line states: `"encoder_inputs" is not defined`.
- Output Panel:** Shows the message: `decoder_embedding = SharedPositionalEmbedding(sequence_length, shared_embedding)(decoder_inputs)`. The tooltip for this line states: `"decoder_inputs" is not defined`.
- Bottom Status Bar:** Shows "Building the model".
- System Tray:** Shows the date (Tue 17 Dec 20:59) and various system icons.

Transfer flow explanation.

1. Input Tokenization:

Input text is tokenized and converted into numerical representations (embeddings).

2. Positional Encoding:

Positional information is added to embeddings to retain the order of words.

3. Encoder Stack:

Multiple encoder layers process the input embeddings using self-attention and feed-forward layers.

- **Self-Attention:** Captures relationships between all tokens in the input sequence.
- **Feed-Forward Layer:** Applies dense transformations to refine token representations.

4. Decoder Stack:

The decoder takes the target sequence (shifted right) and the encoder's output.

- **Masked Self-Attention:** Ensures predictions are based only on prior tokens.
- **Cross-Attention:** Focuses on relevant parts of the encoder's output.

5. Output Layer:

The decoder generates the next token probabilities using a final dense layer with softmax.

This flow repeats until the entire sequence is generated. The model is trained end-to-end using loss between predicted and true tokens.

Comparison with my experiments

Input Tokenization

- **Flow Description:** Text is tokenized and converted to numerical embeddings.

- **Experiment:** The TextVectorization layer is used for tokenization and numerical embedding generation. Custom standardization removes punctuation and unwanted characters.

2. Positional Encoding

- **Flow Description:** Adds positional information to embeddings to encode word order.
- **Experiment:** Implemented via the PositionalEmbedding layer, which combines token embeddings and position embeddings.

3. Encoder Stack

- **Flow Description:** Uses self-attention and feed-forward layers to process embeddings.
- **Experiment:** Implemented as the TransformerEncoder class, which includes:
 - **Self-Attention:** Modeled via layers.MultiHeadAttention.
 - **Feed-Forward Layer:** A dense projection (keras.Sequential) with ReLU and another dense layer.
 - **Layer Normalization:** Used to stabilize training after each step.

4. Decoder Stack

- **Flow Description:** Takes shifted target sequences and encoder outputs to predict the next token.
- **Experiment:** Implemented as the TransformerDecoder class, with:
 - **Masked Self-Attention:** Enforces causality using a causal mask in the get_causal_attention_mask method.
 - **Cross-Attention:** Utilizes encoder outputs to refine decoder predictions.
 - **Feed-Forward Layers:** Similar to the encoder but adapted for the decoder.

5. Output Layer

- **Flow Description:** Final dense layer with softmax generates token probabilities.
- **Experiment:** The output layer of the decoder uses layers.Dense with softmax activation to compute probabilities over the vocabulary.

6. Training and Inference

- **Flow Description:** The model is trained end-to-end using loss functions and predictions are generated iteratively.
- **Experiment:**
 - Loss is computed using SparseCategoricalCrossentropy, ignoring padding tokens.
 - The decode_sequence function demonstrates iterative token generation using the trained model.

Differences:

- The **flow explanation** focuses on high-level concepts, while the **experiment code** gives concrete implementations.
- The **flow diagram** abstracts away implementation details like how masks or embeddings are built, while the **experiment** defines them explicitly.
- The flow suggests the general pipeline, whereas the experiments include dataset preparation and model-specific configurations.