This is a companion notebook for the book Deep Learning with Python, Second Edition. For readability, it only contains runnable code blocks and section titles, and omits everything else in the book: text paragraphs, figures, and pseudocode.

If you want to be able to follow what's going on, I recommend reading the notebook side by side with your copy of the book.

This notebook was generated for TensorFlow 2.6.

The Transformer architecture

Understanding self-attention

Generalized self-attention: the query-key-value model

Multi-head attention

The Transformer encoder

Getting the data

In [1]:

```
!curl -O https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
!rm -r aclImdb/train/unsup
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 80.2M  100 80.2M    0     0  31.6M      0  0:00:02  0:00:02 --:--:-- 31.6M
```

Preparing the data

In [2]:

```
import os, pathlib, shutil, random
from tensorflow import keras
batch_size = 32
base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
text_only_train_ds = train_ds.map(lambda x, y: x)
```

```
Found 20000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
Found 25000 files belonging to 2 classes.
```

Vectorizing the data

In [3]:

```
from tensorflow.keras import layers

max_length = 600
max_tokens = 20000
text_vectorization = layers.TextVectorization(
    max_tokens=max_tokens,
    output_mode="int",
    output_sequence_length=max_length,
)
text_vectorization.adapt(text_only_train_ds)

int_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_val_ds = val_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
int_test_ds = test_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

Transformer encoder implemented as a subclassed `Layer`

In [4]:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None):
        if mask is not None:
            mask = mask[:, tf.newaxis, :]
        attention_output = self.attention(
            inputs, inputs, attention_mask=mask)
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

Using the Transformer encoder for text classification

In [5]:

```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, None)]            0

 embedding (Embedding)       (None, None, 256)         5120000

 transformer_encoder (Transf  (None, None, 256)        543776
 ormerEncoder)

 global_max_pooling1d (Globa  (None, 256)              0
 lMaxPooling1D)

 dropout (Dropout)           (None, 256)               0

 dense_2 (Dense)             (None, 1)                 257

=================================================================
Total params: 5,664,033
Trainable params: 5,664,033
Non-trainable params: 0
_________________________________________________________________
```

Training and evaluating the Transformer encoder based model

In [6]:

```
callbacks = [
    keras.callbacks.ModelCheckpoint("transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20, callbacks=callbacks)
model = keras.models.load_model(
    "transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

```
Epoch 1/20
625/625 [==============================] - 47s 66ms/step - loss: 0.4911 - accuracy: 0.7756 - val_loss: 0.3203 - val_accuracy: 0.8612
Epoch 2/20
625/625 [==============================] - 42s 67ms/step - loss: 0.3109 - accuracy: 0.8674 - val_loss: 0.2910 - val_accuracy: 0.8760
Epoch 3/20
625/625 [==============================] - 43s 69ms/step - loss: 0.2364 - accuracy: 0.9059 - val_loss: 0.2638 - val_accuracy: 0.8884
Epoch 4/20
625/625 [==============================] - 42s 67ms/step - loss: 0.1850 - accuracy: 0.9265 - val_loss: 0.2847 - val_accuracy: 0.8858
Epoch 5/20
625/625 [==============================] - 42s 67ms/step - loss: 0.1480 - accuracy: 0.9430 - val_loss: 0.2781 - val_accuracy: 0.8908
Epoch 6/20
625/625 [==============================] - 42s 67ms/step - loss: 0.1204 - accuracy: 0.9541 - val_loss: 0.3203 - val_accuracy: 0.8788
Epoch 7/20
625/625 [==============================] - 42s 67ms/step - loss: 0.1019 - accuracy: 0.9626 - val_loss: 0.4704 - val_accuracy: 0.8632
Epoch 8/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0822 - accuracy: 0.9696 - val_loss: 0.4025 - val_accuracy: 0.8768
Epoch 9/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0679 - accuracy: 0.9753 - val_loss: 0.7426 - val_accuracy: 0.8464
Epoch 10/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0544 - accuracy: 0.9815 - val_loss: 0.6133 - val_accuracy: 0.8540
Epoch 11/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0437 - accuracy: 0.9844 - val_loss: 0.6142 - val_accuracy: 0.8680
Epoch 12/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0379 - accuracy: 0.9873 - val_loss: 0.6738 - val_accuracy: 0.8488
Epoch 13/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0285 - accuracy: 0.9906 - val_loss: 0.8983 - val_accuracy: 0.8618
Epoch 14/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0283 - accuracy: 0.9923 - val_loss: 0.8841 - val_accuracy: 0.8416
Epoch 15/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0164 - accuracy: 0.9945 - val_loss: 0.8874 - val_accuracy: 0.8660
Epoch 16/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0217 - accuracy: 0.9941 - val_loss: 0.9952 - val_accuracy: 0.8548
Epoch 17/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0156 - accuracy: 0.9958 - val_loss: 1.1246 - val_accuracy: 0.8524
Epoch 18/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0145 - accuracy: 0.9961 - val_loss: 1.1337 - val_accuracy: 0.8556
Epoch 19/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0107 - accuracy: 0.9969 - val_loss: 0.9671 - val_accuracy: 0.8576
Epoch 20/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0109 - accuracy: 0.9967 - val_loss: 1.2125 - val_accuracy: 0.8580
782/782 [==============================] - 24s 30ms/step - loss: 0.2914 - accuracy: 0.8767
Test acc: 0.877
```

Using positional encoding to re-inject order information

Implementing positional embedding as a subclassed layer

In [7]:

```python
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.token_embeddings = layers.Embedding(
            input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

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

    def get_config(self):
        config = super().get_config()
        config.update({
            "output_dim": self.output_dim,
            "sequence_length": self.sequence_length,
            "input_dim": self.input_dim,
        })
        return config
```

Putting it all together: A text-classification Transformer

Combining the Transformer encoder with positional embedding

In [8]:

```python
vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("full_transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20, callbacks=callbacks)
model = keras.models.load_model(
    "full_transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder,
                    "PositionalEmbedding": PositionalEmbedding})
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, None)]            0

 positional_embedding (Posit  (None, None, 256)        5273600
 ionalEmbedding)

 transformer_encoder_1 (Tran  (None, None, 256)        543776
 sformerEncoder)

 global_max_pooling1d_1 (Glo  (None, 256)              0
 balMaxPooling1D)

 dropout_1 (Dropout)         (None, 256)               0

 dense_7 (Dense)             (None, 1)                 257

=================================================================
Total params: 5,817,633
Trainable params: 5,817,633
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
625/625 [==============================] - 44s 68ms/step - loss: 0.4701 - accuracy: 0.7872 - val_loss: 0.3867 - val_accuracy: 0.8432
Epoch 2/20
625/625 [==============================] - 43s 68ms/step - loss: 0.2321 - accuracy: 0.9118 - val_loss: 0.2833 - val_accuracy: 0.8942
Epoch 3/20
625/625 [==============================] - 43s 68ms/step - loss: 0.1747 - accuracy: 0.9350 - val_loss: 0.2945 - val_accuracy: 0.8888
Epoch 4/20
625/625 [==============================] - 43s 68ms/step - loss: 0.1449 - accuracy: 0.9477 - val_loss: 0.3108 - val_accuracy: 0.8918
Epoch 5/20
625/625 [==============================] - 43s 68ms/step - loss: 0.1234 - accuracy: 0.9564 - val_loss: 0.3126 - val_accuracy: 0.8876
Epoch 6/20
625/625 [==============================] - 43s 68ms/step - loss: 0.1051 - accuracy: 0.9633 - val_loss: 0.4117 - val_accuracy: 0.8864
Epoch 7/20
625/625 [==============================] - 42s 67ms/step - loss: 0.0902 - accuracy: 0.9692 - val_loss: 0.4598 - val_accuracy: 0.8800
Epoch 8/20
625/625 [==============================] - 43s 69ms/step - loss: 0.0797 - accuracy: 0.9729 - val_loss: 0.3973 - val_accuracy: 0.8794
Epoch 9/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0740 - accuracy: 0.9758 - val_loss: 0.5528 - val_accuracy: 0.8784
Epoch 10/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0649 - accuracy: 0.9784 - val_loss: 0.6417 - val_accuracy: 0.8788
Epoch 11/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0607 - accuracy: 0.9793 - val_loss: 0.4812 - val_accuracy: 0.8800
Epoch 12/20
625/625 [==============================] - 42s 68ms/step - loss: 0.0532 - accuracy: 0.9824 - val_loss: 0.4809 - val_accuracy: 0.8730
Epoch 13/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0476 - accuracy: 0.9836 - val_loss: 0.7396 - val_accuracy: 0.8704
Epoch 14/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0434 - accuracy: 0.9867 - val_loss: 0.7362 - val_accuracy: 0.8768
Epoch 15/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0398 - accuracy: 0.9872 - val_loss: 0.7013 - val_accuracy: 0.8740
Epoch 16/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0354 - accuracy: 0.9887 - val_loss: 0.9323 - val_accuracy: 0.8674
Epoch 17/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0309 - accuracy: 0.9897 - val_loss: 0.7896 - val_accuracy: 0.8704
Epoch 18/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0279 - accuracy: 0.9908 - val_loss: 0.8036 - val_accuracy: 0.8730
Epoch 19/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0242 - accuracy: 0.9924 - val_loss: 0.6579 - val_accuracy: 0.8762
Epoch 20/20
625/625 [==============================] - 43s 68ms/step - loss: 0.0223 - accuracy: 0.9934 - val_loss: 0.8889 - val_accuracy: 0.8742
782/782 [==============================] - 25s 32ms/step - loss: 0.3098 - accuracy: 0.8798
Test acc: 0.880
```

When to use sequence models over bag-of-words models?