TensorFlow Developer Certificate Notes

Contents

Introduction:

- **tf.constant()** is not mutable, but **tf.Variable()** is by using the *.assign()* method on the var object.
- You must set both the global **tf.random.set_seed()** and function **seed=** parameter to get reproducible results for shuffle function.
- We can *add dimensions* to a tensor whilst keeping the same information (*newaxis* and *expand_dims* have same output).

```
rank_3_tensor = rank_2_tensor[..., tf.newaxis] # "..." means "all dims prior to"
rank_2_tensor, rank_3_tensor # shape (2, 2), shape (2, 2, 1)
tf.expand_dims(rank_2_tensor, axis=-1) # "-1" means last axis (2, 2, 1)
```

- **tf.reshape()** will change the shape in the order they appear (top left to bottom right) and **tf.transpose()** simply flips the matrix.
- We can reduce tensor sizes in memory by changing the datatype (i.e. float32 cast to float16).
- We can perform aggregation on tensors by using **reduce()_[action]** and using min, max, mean, sum, etc. We can also find positional arguments using **tf.argmin()** or **tf.argmax()**.

Neural Network Classification:

- We can create a **learning rate callback** to update our learning rate during training.

```
# Create a learning rate scheduler callback
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch:
                1e-4 * 10**(epoch/20))
```

- Traverse a set of learning rate values starting from 1e-4, increasing by 10**(epoch/20) every epoch.
- Note that learning rate exponentially increases as epochs increases.
- We can use a plot to determine the **ideal learning rate**, which we want to take the value where loss is still decreasing but not quite flattened out. It is the value around 10x smaller than the lowest point (refer to notebook for graph and point selection).

```
lrs = 1e-4 * (10 ** (np.arange(100)/20))
plt.figure(figsize=(10, 7))
plt.semilogx(lrs, history.history["loss"]) # x-axis (lr) to be log scale
```

1 Transfer Learning

1.1 Feature Extraction

- We can log the performance of multiple models, then view and compare these models in a visual way on a **TensorBoard**. It saves a model's training performance to a specified *log_dir*.

```
def create_tensorboard_callback(dir_name, experiment_name):
  log_dir = dir_name + "/" + experiment_name + "/" +
            datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
  tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir)
  print(f"Saving TensorBoard log files to: {log_dir}")
  return tensorboard_callback
```

- We can also save a model as it trains so you can stop training if needed and come back to continue off where you left using **Model Checkpointing**. By default, metric monitored is *validation loss*.

```
cp_path = "model_checkpoint_name_here/checkpoint.ckpt"

# Create a ModelCheckpoint callback that saves the model's weights only
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath=cp_path,
  save_weights_only=True, # False to save the entire model
  save_best_only=False, # True to save only best model instead of every epoch
  save_freq="epoch", # save every epoch
  verbose=1)
```

- **Feature Extraction** is when you take the weights a pretrained model has learned and adjust its outputs to be more suited to your problem (keep layers frozen except new output layers).

1.2 Fine Tuning

- The **GlobalAveragePooling2D** layer take the average of the outputs of the model (across the inner axis) and reduces it into a **feature vector** that is then passed to our final **Dense** layer, which then gives us our final output. For example, a tensor of shape (2, 4, 5, 3) will be reduced into shape (2, 3).

- Images are best preprocessed on the GPU where as text and structured data are more suited to be preprocessed on the CPU. Image data augmentation only happens during training so we can still export our whole model and use it elsewhere. And if someone else wanted to train the same model as us, including the same kind of data augmentation, they could.

- We can create a **Data Augmentation** layer for our model using the Sequential API and the *tf.keras.layers.experimental.preprocessing* layers. Note that this layer is turned off for predicting.

```
data_augmentation = keras.Sequential([
  preprocessing.RandomFlip("horizontal"),
  preprocessing.RandomRotation(0.2),
  ... # zoom, width, rotation, normalize, etc.
], name = "data_augmentation")

input_shape = (224, 224, 3)
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False # freeze model layers

inputs = layers.Input(shape=input_shape, name="input_layer")
x = data_augmentation(inputs)
x = base_model(x, training=False)
x = layers.GlobalAveragePooling2D(name="global_average_pooling_layer")(x)
outputs = layers.Dense(10, activation="softmax", name="output_layer")(x)
model = keras.Model(inputs, outputs)
```

- In **Fine Tuning** we will unfreeze deeper layers in the model in order to learn more problem specific features for our dataset. Generally, the amount we unfreeze is determined by how much data we have.
- Click here for how to resume training after unfreezing layers and plotting the history.

1.3 Scaling Up

- We can used **Mixed Precision** in order to improve our models performance on GPU by using a mix of float32 and float16 data types to use less memory where possible and in turn run faster (using less memory per tensor means more tensors can be computed on simultaneously). Note that this doesn't work for all hardware (must have score of 7.0+, see *supported hardware* in above link).

```
from tensorflow.keras import mixed_precision

# set global policy to mixed precision
mixed_precision.set_global_policy(policy="mixed_float16")
```

- Note that in the final output layer, it is required to specify the *dtype=tf.float32* and use the **Activation** layer instead of Dense when using mixed precision.

```
base_model = tf.keras.applications.EfficientNetB0(include_top=False)
base_model.trainable = False # freeze base model layers

inputs = layers.Input(shape=input_shape, name="input_layer")
x = base_model(inputs, training=False) # set base_model to inference mode only
x = layers.GlobalAveragePooling2D(name="pooling_layer")(x)
x = layers.Dense(len(class_names))(x) # want one output neuron per class
# Separate activation of output layer so we can output float32 activations
outputs = layers.Activation("softmax", dtype=tf.float32, name="sm_float32")(x)
model = tf.keras.Model(inputs, outputs)

for layer in model.layers:
  print(layer.dtype_policy) # Check the dtype policy of layers
```

2 Natural Language Processing

- **Text Vectorization Layer** - maps input sequence to numbers (convert words to number pairing).
- **Embedding** - Turns mapping of text vectors to embedding matrix (finds how words relate).
- **RNN cell(s)** - find patterns in sequences (usually an *LSTM* layer with *tanh* activation).

2.1 Text to Numbers

- **Tokenization** - A straight mapping from word or character or sub-word to a numerical value. There are three main levels of tokenization:
  1. **word-level** - every word in a sequence is a single token.
  2. **character-level** - convert A-Z to 1-26, single token.
  3. **sub-word** - mix of the previous two, break words into smaller chunks so every word is considered multiple tokens.
- **Embeddings** - An embedding is a representation of natural language which can be learned. Representation comes in the form of a **feature vector**. You can either create an embedding layer built on our text, or use a pre-learned layer that has been trained on a large corpus.

2.1.1 Text Vectorization

```
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization

text_vectorizer = TextVectorization(max_tokens=10000, # how many words in the vocab
  standardize="lower_and_strip_punctuation", # how to process text
  split="whitespace", # how to split tokens
  ngrams=None, # create groups of n-words?
  output_mode="int", # how to map tokens to numbers
  output_sequence_length=15, # how long should the output sequence of tokens be?
  pad_to_max_tokens=True)

text_vectorizer.adapt(train_sentences) # map training data to vectorizer
```

- For *max_tokens* (the number of words in the vocabulary), multiples of 10,000 or the exact number of unique words in your text are common values.
- For *output_sequence_length* we could use the average number of tokens per observation.

```
round(sum([len(i.split()) for i in train_sentences])/len(train_sentences)) # 15
```

- We can check the unique tokens in the vocabulary and the most/least common words.

```
words_in_vocab = text_vectorizer.get_vocabulary()
top_5 = words_in_vocab[:5] # ['', '[UNK]', 'the', 'a', 'in']
bottom_5 = words_in_vocab[-5:]
# ['pages', 'paeds', 'pads', 'padres', 'paddytomlinson1']
```

2.1.2 Embedding Layer

- A word's numeric representation can be improved as a model goes through data samples, so our embedding layer turns each token into a vector of shape *output_dim*.

```
from tensorflow.keras import layers

embedding = layers.Embedding(input_dim=len(text_vectorizer.get_vocabulary()),
  output_dim=128, # set size of embedding vector
  embeddings_initializer="uniform", # default, intialize randomly
  input_length=15) # how long is each input
```

```
sample_embed = embedding(text_vectorizer([random_sentence]))
sample_embed # <tf.Tensor: shape=(1, 15, 128), dtype=float32, numpy=array([[[...]]])
```

- Each token gets turned into a length 128 feature vector. Above we have 1 observation, 15 tokens for the observation, and each token is a vector of size 128.

2.2 Creating Multiple Models

2.2.1 Model 0: Naive Bayes

Create a Scikit-Learn Pipeline using the TF-IDF (term frequency-inverse document frequency) formula to convert our words to numbers and then model them with the Multinomial Naive Bayes algorithm.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# Create tokenization and modelling pipeline
model_0 = Pipeline([
  ("tfidf", TfidfVectorizer()), # convert words to numbers using tfidf
  ("clf", MultinomialNB()) # model the text
])

model_0.fit(train_sentences, train_labels)
```

2.2.2 Model 1: Simple Dense Model

We will create a single layer dense model. It'll take our text and labels as input, tokenize the text, create an embedding, find the average of the embedding (using Global Average Pooling) and then pass the average through a fully connected layer with one output unit and a sigmoid activation function.

```
inputs = layers.Input(shape=(1,), dtype="string") # inputs are 1-D strings
x = text_vectorizer(inputs) # turn the input text into numbers
x = embedding(x) # create an embedding of the numerized numbers
x = layers.GlobalAveragePooling1D()(x) # lower the dimensionality of the embedding
outputs = layers.Dense(1, activation="sigmoid")(x) #binary classification
model_1 = tf.keras.Model(inputs, outputs, name="model_1_dense")
```

2.2.3 Extra: Visualize Learned Embeddings

- Now that we have trained an embedding layer, we can save the weights and visualize them using the Embedding Projector Tool. To see how to save weights and metadata (needed for projector tool) to .tsv files, see the TensorFlow Tutorial for Saving Word Embeddings
- With these embeddings, we can see if similar words are grouped together and how the model interprets these words (not how we interpret them).

2.2.4 Model 2: RNN with LSTM

- A **RNN** allows the model to take information from the past to help with the future, meaning it can take into consideration the previous words to determine the meaning of the given word. There are many different types of RNNs:
  - *One to one*: one input, one output, such as image classification.
  - *One to many*: one input, many outputs, such as image captioning (image input, caption output).
  - *Many to one*: many inputs, one outputs, such as binary text classification.
  - *Many to many*: many inputs, many outputs, such as machine translation or speech to text.
- An **LSTM** (Long Short Term Memory) is a variant of an RNN which allows for both feedforward and feedback, as well as processing entire sequences of data at once.

```
from tensorflow.keras import layers

inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = embedding(x) # shape: (None, 15, 128)
x = layers.LSTM(64)(x) # return vector for whole sequence, shape: (None, 64)
# x = layers.Dense(64, activation="relu")(x) # optional on top of LSTM
outputs = layers.Dense(1, activation="sigmoid")(x)
model_2 = tf.keras.Model(inputs, outputs, name="model_2_LSTM")
```

- NOTE: we can stack LSTM cells as long as *return_sequences=True* is set in layer parameters.
- LSTM number of parameters: 4*(embedding_size + LSTM_units + 1) * LSTM_units

2.2.5 Model 3: RNN with GRU

- A **GRU** (Gated Recurrent Unit) aims to solve the vanishing gradient problem that often occurs in RNNs. The GRU will have less trainable parameters compared to the LSTM.
- Again we can stack GRU cells as long as *return_sequences=True* is set in layer parameters.

```
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = embedding(x)
x = layers.GRU(64)(x) # return vector for whole sequence
# x = layers.Dense(64, activation="relu")(x) # optional after GRU
outputs = layers.Dense(1, activation="sigmoid")(x)
model_3 = tf.keras.Model(inputs, outputs, name="model_3_GRU")
```

2.2.6 Model 4: Bidirectional RNN Model

- A **bidirectional** RNN will process the sequence from left to right and then again from right to left. This can improve performance but comes at the cost of longer training time and double the number of trainable model parameters.

```
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = embedding(x)
x = layers.Bidirectional(layers.LSTM(64, return_sequences=True))(x)
x = layers.Bidirectional(layers.GRU(64))(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model_4 = tf.keras.Model(inputs, outputs, name="model_4_Bidirectional")
```

2.2.7 Model 5: CNN for Text

- Sequences come in the form of 1-dimensional data (string of text), so using a CNN will require 1D layers (temporal convolution) rather than 2D.
- We can think of CNN *filters* as ngram detectors, each filter specializing in a closely-related family of ngrams. *Max-pooling* over time extracts the relevant ngrams for making a decision.

```
inputs = layers.Input(shape=(1,), dtype="string")
x = text_vectorizer(inputs)
x = embedding(x)
x = layers.Conv1D(filters=32, kernel_size=5, activation="relu")(x)
x = layers.GlobalMaxPool1D()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model_5 = tf.keras.Model(inputs, outputs, name="model_5_Conv1D")
```

2.2.8 Model 6: Transfer Learning (Pretrained Embeddings)

-