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*.

```python
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*.

```python
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.

```python
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).

```python
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.

```python
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