Homework 07

Table of contents

Noted

Problem 2

26

We have a residual layer with the following reverse graph

```
%load_ext tikzmagic
```

```
%%tikz -f svg --size=200,150
\node (dx) at (3, 0) {$\frac{\partial e}{\partial x}$};
\node[draw, rectangle] (dfx) at (4, 0) {$f(x)$};
\node (dy) at (5, 0) {$\frac{\partial e}{\partial y}$};
\path[draw, ->] (dy) -- (dfx) -- (dx);
\path[draw, ->] (dy) |- (4, 0.5) -| (dx);
```

To calculate $\frac{\partial e}{\partial x}$ we will need to apply the chain rule through $f(x)$ and sum on the residual merge. This gives

$$\frac{\partial e}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial e}{\partial y}$$

27

28

Problem 3

The Tiny ImageNet dataset can be downloaded here Extract the zip onto a fast disk drive. First we will set up the python environment and imports

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import os

# For TFRecord demo
tf.logging.set_verbosity(tf.logging.INFO)
#tf.enable_eager_execution()

# Train on secondary GPU
os.environ['CUDA_VISIBLE_DEVICES'] = '1'

DATASET_ROOT = '/home/tidal/tiny-imagenet-200/tiny-imagenet-200'
TFRECORD = os.path.join(DATASET_ROOT, 'tiny_imagenet')
TRAIN_DIR = os.path.join(DATASET_ROOT, 'train')
```

Looking at the directory structure of the dataset, we see that there are subdirectory for training, validating, and testing. The training set contains 200 classes where images of a class are grouped by directory.

```
def process_class(writer, path, label):
    for image in os.listdir(path):
        img_raw = open(img_file, 'rb').read()
        feature = {
            'label' : _int64_feature(label),
            'image_raw' : _bytes_feature(img_raw)
        }
        example = tf.train.Example(features=tf.train.Features(feature=feature))
        writer.write(example.SerailizeToString())
```

Preprocessing

We can import the training set with preprocessing as follows (documentation available here.)

We will use the `ImageDataGenerator`. This works will on the training set by default

```
TRAIN_PATH = os.path.join(DATASET_ROOT, 'train')
TEST_PATH = os.path.join(DATASET_ROOT, 'test')
VAL_PATH = os.path.join(DATASET_ROOT, 'val')

TRAINING_IMAGE_SIZE        = 64
TRAINING_ZERO_PADDED_SIZE  = 64
TRAINING_BATCH_SIZE        = 128
```

```
# Define a data generator with preprocessing for the training set
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        samplewise_center=True,
        samplewise_std_normalization=True,
        horizontal_flip=True,
        data_format='channels_first',
        rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        TRAIN_PATH,
        target_size=(TRAINING_IMAGE_SIZE, TRAINING_IMAGE_SIZE),
        batch_size=TRAINING_BATCH_SIZE,
        class_mode='binary')
```

But for the validation and testing sets the `ImageDataGenerator` expects directory for each label. We must manually fix labels

```
# A limited preprocessing generator for the test/validation set
test_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        samplewise_center=True,
        samplewise_std_normalization=True,
        data_format='channels_first',
        rescale=1./255)

val_generator = test_datagen.flow_from_directory(
        VAL_PATH,
        target_size=(TRAINING_IMAGE_SIZE, TRAINING_IMAGE_SIZE),
        batch_size=TRAINING_BATCH_SIZE,
        class_mode=None)
```

Problem 4

Understood

Problem 5 - Modifying Resnet

We can construct Resnet using a subclassed approach. This involves creating modular blocks of layers that can be reused as needed, thus increasing code reuseability and ease of maintainance.

Specifically, we subclass `tf.keras.Model` and implement the methods `__init__()` and `call()`. Our choice of `__init__()` method will define the the types of layers in this block, but says nothing about how they are connected. In the `call()` method we will define the connections between layers. This method takes an input as a parameter and returns an ouput that represents the feature maps after a forward pass through all layers in the block.

Basic Block

First we will define the fundamental CNN style 2D convolution block of Resnet, ie

Note that the number of filters and the kernel size are parameterized, and that parameter packs `*args, **kwargs` are forwarded to the convolution layer. This is important as it enables the reuse of this model for the various types of convolutions that we will need.

```
class ResnetBasic(tf.keras.Model):

    def __init__(self, filters, kernel_size, *args, **kwargs):
        super(ResnetBasic, self).__init__()
        self.batch_norm = layers.BatchNormalization()
        self.relu = layers.ReLU()
        self.conv2d = layers.Conv2D(
                filters,
                kernel_size,
                padding='same',
                data_format='channels_last',
                activation=None,
                use_bias=False,
                *args,**kwargs)

    def call(self, inputs):
        x = self.batch_norm(inputs)
        x = self.relu(x)
        return self.conv2d(x)
```

Standard Bottleneck

Recognizing this, we can define a bottleneck layer. Again, the number of input feature maps is parameterized. We no longer parameterize the kernel dimensions, as these are intrinsic to this type of block.

```
class Bottleneck(tf.keras.Model):

    def __init__(self, Ni, *args, **kwargs):

        # Parent constructor call
        super(Bottleneck, self).__init__(*args, **kwargs)

        # Three residual convolution blocks
        self.residual_filter_1 = ResnetBasic(int(Ni/4), (1,1))
        self.residual_filter_2 = ResnetBasic(int(Ni/4), (3,3))
        self.residual_filter_3 = ResnetBasic(Ni, (1,1))
        # Merge operation
        self.merge = layers.Add()

    def call(self, inputs):

        # Residual forward pass
        res = self.residual_filter_1(inputs)
        res = self.residual_filter_2(res)
        res = self.residual_filter_3(res)

        # Combine residual pass with identity
        return self.merge([inputs, res])
```

Special Bottleneck

We can define the special bottleneck layer by subclassing the `Bottleneck` class as follows.

```
class SpecialBottleneck(Bottleneck):

    def __init__(self, Ni, *args, **kwargs):

        # Layers that also appear in standard bottleneck
        super(SpecialBottleneck, self).__init__(Ni, *args, **kwargs)

        # Add convolution layer along main path
        self.main = layers.Conv2D(
                Ni,
                (1, 1),
                padding='same',
                data_format='channels_last',
                activation=None,
                use_bias=False)

    def call(self, inputs):

        # Residual forward pass as with normal bottleneck
        res = self.residual_filter_1(inputs)
        res = self.residual_filter_2(res)
        res = self.residual_filter_3(res)

        # Convolution on main forward pass
        main = self.main(inputs)

        # Merge residual and main
        return self.merge([main, res])
```

Downsampling

Next we need to define the downsampling layer.

```
class Downsample(tf.keras.Model):

    def __init__(self, Ni, *args, **kwargs):
        super(Downsample, self).__init__(*args, **kwargs)

        # Three residual convolution blocks
        self.residual_filter_1 = ResnetBasic(int(Ni/2), (1,1), strides=(2,2))
        self.residual_filter_2 = ResnetBasic(int(Ni/2), (3,3))
        self.residual_filter_3 = ResnetBasic(2*Ni, (1,1))

        # Convolution on main path
        self.main = ResnetBasic(2*Ni, (1,1), strides=(2,2))

        # Merge operation for residual and main
        self.merge = layers.Add()

    def call(self, inputs):

        # Residual forward pass
        res = self.residual_filter_1(inputs)
        res = self.residual_filter_2(res)
        res = self.residual_filter_3(res)

        # Main forward pass
        main = self.main(inputs)

        # Merge residual and main
        return self.merge([main, res])
```

Final Model

Finally, we can assemble these blocks into the final model. Note that the tail and other simple layers are defined within the Resnet model class, rather than being subclassed as we did with the other building blocks. This choice came down to the simplicity of tail and other non-subclassed layers.

Also worth noting is the use of `layers.GlobalAveragePooling2D`. There is no `keras.layers.reduce_mean()` layer, but this operation represents global average pooling so we simply choose the correct layer class.

```
class Resnet(tf.keras.Model):

    def __init__(self, classes, filters, levels, *args, **kwargs):
        super(Resnet, self).__init__()
        # Lists to hold various layers
        self.blocks = list()

        # Tail
        self.tail = layers.Conv2D(
                filters,
                (3, 3),
                padding='same',
                data_format='channels_last',
                use_bias=False,
                name='tail')

        # Special bottleneck layer with convolution on main path
        self.level_0_special = SpecialBottleneck(filters)

        # Loop through levels and their parameterized repeat counts
        for level, repeats in enumerate(levels):
            for block in range(repeats):
                # Append a bottleneck block for each repeat
                name = 'bottleneck_%i_%i' % (level, block)
                layer = Bottleneck(filters, name=name)
                self.blocks.append(layer)

            # Downsample and double feature maps at end of level
            name = 'downsample_%i' % (level)
            layer = Downsample(filters, name=name)
            self.blocks.append(layer)
            filters *= 2

        # encoder - level 2 special block x1
        # input:  256 x   8 x 8
        # output: 256 x   8 x 8
        self.level2_batch_norm = layers.BatchNormalization()
        self.level2_relu = layers.ReLU()

        # Decoder - global average pool and fully connected
        self.global_avg = layers.GlobalAveragePooling2D(
                data_format='channels_last')
        self.dense = layers.Dense(classes,
                use_bias=True)


    def call(self, inputs):
        x = self.tail(inputs)
        x = self.level_0_special(x)
        # Loop over layers by level
        for layer in self.blocks:
            x = layer(x)

        # Finish up specials in level 2
        x = self.level2_batch_norm(x)
        x = self.level2_relu(x)

        # Decoder
        x = self.global_avg(x)
        return self.dense(x)
```

Using the Model

Now that we have defined a subclassed model, we need to incorproate it into a training / testing environment. This is where the beauty of the subclassed approach comes in. In our case we want construct Resnet modified for Tiny Imagenet, where the modifications are as follows:

- Third level of residual blocks + downsampling
- Full and half width versions

Our Resnet class accepts an interable of integers to define the number of repeats at each level. As such, we need only add an integer for the number of repeats at level 3 to our constructor call. Similarly, we can scale the number of feature maps as needed to adjust width.

Lastly we will provide the number of classes in Tiny Imagenet, ie 200.

```
# Add another level
standard_levels = [4, 6, 3]
new_level_count = 2
modified_levels = standard_levels + [new_level_count]

# Define full and half width feature map count
full_width = 64
half_width = full_width / 2

# Tiny Imagenet properties
NUM_CLASSES = 200
INPUT_SHAPE = [3, 64, 64]

model = Resnet(NUM_CLASSES, full_width, modified_levels)
```

Note that `model` returned by our class constructor is callable. Thus our forward pass mapping inputs to outputs is invoked by “calling” `model` on the inputs and storing the returned outputs. Note that a call on model is simply calling the

`Resnet.call()` method we wrote earlier. More on this when we get to training.

Finally, we can build the model for the appropriate input shape and get a summary of the included layers

```
input_shape = (
        TRAINING_BATCH_SIZE,
        3,
        TRAINING_IMAGE_SIZE,
        TRAINING_IMAGE_SIZE
)
model.build(input_shape)
model.summary()
```

Problem 6 - Saving Validation

Now we can construct a training loop with the following additional features

- Saving validation loss/accuracy on a per epoch basis
- Checkpointing after each epoch with ability to restore from checkpoint

First we will define training hyperparameters to be used later on

```
TRAINING_SHUFFLE_BUFFER    = 5000
TRAINING_NUM_EPOCHS        = 72
TRAINING_MOMENTUM          = 0.9                     # currently not used
TRAINING_REGULARIZER_SCALE = 0.1                     # currently not used
TRAINING_LR_INITIAL        = 0.01
TRAINING_LR_SCALE          = 0.1
TRAINING_LR_EPOCHS         = 64
TRAINING_LR_STAIRCASE      = True
TRAINING_MAX_CHECKPOINTS   = 5
TRAINING_CHECKPOINT_FILE   = os.path.join('log','tiny-imgnet')
```

Defining Metrics

Next we define an accuracy and loss metric, as well as an optimizer

```
# accuracy
metrics = ['accuracy']

# loss
loss = tf.losses.sparse_softmax_cross_entropy


STEP_SIZE_TRAIN=train_generator.n//train_generator.batch_size
```

```
# optimizer
optimizer = tf.train.AdamOptimizer()



model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
callback = tf.keras.callbacks.TensorBoard(
        log_dir='./logs',
        write_graph=True)

model.fit_generator(
        generator=train_generator,
        shuffle=True,
        callbacks=[callback],
        steps_per_epoch=STEP_SIZE_TRAIN,
        epochs=TRAINING_NUM_EPOCHS)
```

Checkpointing...

```
checkpoint_prefix = os.path.join(TRAINING_CHECKPOINT_FILE, "ckpt")
status = checkpoint.restore(tf.train.latest_checkpoint(checkpoint_directory))
for _ in range(num_training_steps):
  optimizer.minimize( ... )  # Variables will be restored on creation.
status.assert_consumed()  # Optional sanity checks.
checkpoint.save(file_prefix=checkpoint_prefix)
```