TensorFlow2教程-用keras構建自己的網路層

1 構建一個簡單的網路層

我們可以通過繼承tf.keras.layer.Layer，實現一個自訂的網路層。

In [1]:
```
from __future__ import absolute_import, division, print_function
import tensorflow as tf
tf.keras.backend.clear_session()
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
```

In [3]:
```
# 定義網路層就是：設置網路權重和輸出到輸入的計算過程
class MyLayer(layers.Layer):
    def __init__(self, input_dim=32, unit=32):
        super(MyLayer, self).__init__()

        w_init = tf.random_normal_initializer()
        # 權重變數
        self.weight = tf.Variable(initial_value=w_init(
            shape=(input_dim, unit), dtype=tf.float32), trainable=True)

        b_init = tf.zeros_initializer()
        # 偏置變數
        self.bias = tf.Variable(initial_value=b_init(
            shape=(unit,), dtype=tf.float32), trainable=True)

    def call(self, inputs):
        # 全連接網路
        return tf.matmul(inputs, self.weight) + self.bias

x = tf.ones((3,5))
my_layer = MyLayer(5, 4)
out = my_layer(x)
print(out)
```

```
tf.Tensor(
[[ 0.021577   -0.04662052 -0.12161195  0.08813772]
 [ 0.021577   -0.04662052 -0.12161195  0.08813772]
 [ 0.021577   -0.04662052 -0.12161195  0.08813772]], shape=(3, 4), dtype=float3
2)
```

按上面構建網路層，圖層會自動跟蹤權重w和b，當然我們也可以直接用add_weight的方法構建權重

In [5]:
```
class MyLayer(layers.Layer):
    def __init__(self, input_dim=32, unit=32):
        super(MyLayer, self).__init__()
        # 使用add_weight添加網路變數，使其可追蹤
        self.weight = self.add_weight(shape=(input_dim, unit),
                                     initializer=keras.initializers.RandomNormal(
                                     trainable=True)
        self.bias = self.add_weight(shape=(unit,),
                                   initializer=keras.initializers.Zeros(),
                                   trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.weight) + self.bias

x = tf.ones((3,5))
my_layer = MyLayer(5, 4)
out = my_layer(x)
print(out)
```

```
tf.Tensor(
[[ 0.146505    0.03690553 -0.09940173  0.05251381]
 [ 0.146505    0.03690553 -0.09940173  0.05251381]
 [ 0.146505    0.03690553 -0.09940173  0.05251381]], shape=(3, 4), dtype=float3
2)
```

也可以設置不可訓練的權重

In [7]:
```
class AddLayer(layers.Layer):
    def __init__(self, input_dim=32):
        super(AddLayer, self).__init__()
        # 只存儲，不訓練的變數
        self.sum = self.add_weight(shape=(input_dim,),
                                   initializer=keras.initializers.Zeros(),
                                   trainable=False)


    def call(self, inputs):
        self.sum.assign_add(tf.reduce_sum(inputs, axis=0))
        return self.sum

x = tf.ones((3,3))
my_layer = AddLayer(3)
out = my_layer(x)
print(out.numpy())
out = my_layer(x)
print(out.numpy())
print('weight:', my_layer.weights)
print('non-trainable weight:', my_layer.non_trainable_weights)
print('trainable weight:', my_layer.trainable_weights)
```

```
[3. 3. 3.]
[6. 6. 6.]
weight: [<tf.Variable 'Variable:0' shape=(3,) dtype=float32, numpy=array([6.,
6., 6.], dtype=float32)>]
non-trainable weight: [<tf.Variable 'Variable:0' shape=(3,) dtype=float32, nump
y=array([6., 6., 6.], dtype=float32)>]
trainable weight: []
```

當定義網路時不知道網路的維度是可以重寫build()函數，用獲得的shape構建網路

In [5]:
```
class MyLayer(layers.Layer):
    def __init__(self, unit=32):
        super(MyLayer, self).__init__()
        self.unit = unit

    def build(self, input_shape):
        # 在build時獲取input_shape
        self.weight = self.add_weight(shape=(input_shape[-1], self.unit),
                                      initializer=keras.initializers.RandomNormal(
                                      trainable=True)
        self.bias = self.add_weight(shape=(self.unit,),
                                    initializer=keras.initializers.Zeros(),
                                    trainable=True)

    def call(self, inputs):
        return tf.matmul(inputs, self.weight) + self.bias


my_layer = MyLayer(3)
x = tf.ones((3,5))
out = my_layer(x)
print(out)
my_layer = MyLayer(3)

x = tf.ones((2,2))
out = my_layer(x)
print(out)
```

```
tf.Tensor(
[[-0.25201735  0.09862914  0.06587204]
 [-0.25201735  0.09862914  0.06587204]
 [-0.25201735  0.09862914  0.06587204]], shape=(3, 3), dtype=float32)
tf.Tensor(
[[-0.0270178  -0.03847811 -0.09622537]
 [-0.0270178  -0.03847811 -0.09622537]], shape=(2, 3), dtype=float32)
```

2 使用子層遞迴構建網路層

可以在自訂網路層中調用其他自訂網路層

In [6]:
```
class MyBlock(layers.Layer):
    def __init__(self):
        super(MyBlock, self).__init__()
        # 其他自訂網路層
        self.layer1 = MyLayer(32)
        self.layer2 = MyLayer(16)
        self.layer3 = MyLayer(2)
    def call(self, inputs):
        h1 = self.layer1(inputs)
        h1 = tf.nn.relu(h1)
        h2 = self.layer2(h1)
        h2 = tf.nn.relu(h2)
        return self.layer3(h2)

my_block = MyBlock()
print('trainable weights:', len(my_block.trainable_weights))
y = my_block(tf.ones(shape=(3, 64)))
# 構建網路在build()裡面，所以執行了才有網路
print('trainable weights:', len(my_block.trainable_weights))
```

```
trainable weights: 0
trainable weights: 6
```

可以通過構建網路層的方法來收集loss，並可以遞迴呼叫。

In [7]:
```
class LossLayer(layers.Layer):

  def __init__(self, rate=1e-2):
    super(LossLayer, self).__init__()
    self.rate = rate

  def call(self, inputs):
    # 添加loss
    self.add_loss(self.rate * tf.reduce_sum(inputs))
    return inputs

class OutLayer(layers.Layer):
    def __init__(self):
        super(OutLayer, self).__init__()
        self.loss_fun=LossLayer(1e-2)
    def call(self, inputs):
        # 就一個loss層
        return self.loss_fun(inputs)

my_layer = OutLayer()
print(len(my_layer.losses)) # 還未call
y = my_layer(tf.zeros(1,1))
print(len(my_layer.losses)) # 執行call之後
y = my_layer(tf.zeros(1,1))
print(len(my_layer.losses)) # call之前會重新置0
```

```
0
1
1
```

如果中間調用了keras網路層，裡面的正則化loss也會被加入進來

In [8]:
```
class OuterLayer(layers.Layer):

    def __init__(self):
        super(OuterLayer, self).__init__()
        # 子層中正則化loss也會添加到總的loss中
        self.dense = layers.Dense(32, kernel_regularizer=tf.keras.regularizers.l2

    def call(self, inputs):
        return self.dense(inputs)


my_layer = OuterLayer()
y = my_layer(tf.zeros((1,1)))
print(my_layer.losses)
print(my_layer.weights)
```

```
[<tf.Tensor: id=266, shape=(), dtype=float32, numpy=0.0020732714>]
[<tf.Variable 'outer_layer/dense/kernel:0' shape=(1, 32) dtype=float32, numpy=
array([[ 0.18419057,  0.41972226,  0.06816366,  0.3822255 , -0.18456893,
        -0.25967044, -0.3724193 , -0.10103354,  0.28944224, -0.38763577,
         0.26489502,  0.40765905,  0.3051821 ,  0.3081022 , -0.02279559,
         0.16751188,  0.23520029,  0.28544015,  0.22495598,  0.21490109,
         0.28766167, -0.2586367 , -0.04058033, -0.22604927, -0.12887421,
         0.10930204,  0.41669363,  0.1015256 ,  0.0400646 ,  0.12960178,
        -0.04219085, -0.32611725]], dtype=float32)>, <tf.Variable 'outer_layer/
dense/bias:0' shape=(32,) dtype=float32, numpy=
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
      dtype=float32)>]
```

3 其他網路層配置

3.1 使自己的網路層可以序列化

In [1]:
```
class Linear(layers.Layer):

    def __init__(self, units=32, **kwargs):
        super(Linear, self).__init__(**kwargs)
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),
                                 initializer='random_normal',
                                 trainable=True)
        self.b = self.add_weight(shape=(self.units,),
                                 initializer='random_normal',
                                 trainable=True)
    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

    def get_config(self):
        # 獲取網路配置，用於實現序列化
        config = super(Linear, self).get_config()
        config.update({'units':self.units})
        return config


layer = Linear(125)
config = layer.get_config()
print(config)
# 從配置中構建網路，（已知網路結構，不知超參的情況）
new_layer = Linear.from_config(config)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6b595bb01b21> in <module>
----> 1 class Linear(layers.Layer):
      2 
      3     def __init__(self, units=32, **kwargs):
      4         super(Linear, self).__init__(**kwargs)
      5         self.units = units

NameError: name 'layers' is not defined
```

如果在反序列化中(從配置中構建網路)需要更大的靈活性，可以重寫from_config方法。

In [10]:
```
def from_config(cls, config):
    return cls(**config)
```

3.2 配置訓練時特有參數

有一些網路層， 如BatchNormalization層和Dropout層，在訓練和推理中具有不同的行為，對於此類層，則需要在方法中使用train等參數進行控制。

In [11]:
```
class MyDropout(layers.Layer):
    def __init__(self, rate, **kwargs):
        super(MyDropout, self).__init__(**kwargs)
        self.rate = rate
    def call(self, inputs, training=None):
        return tf.cond(training,
                       lambda: tf.nn.dropout(inputs, rate=self.rate),
                      lambda: inputs)
```

4 構建自己的模型

通常，我們使用Layer類來定義內部計算塊，並使用Model類來定義外部模型 - 即要訓練的物件。

Model類與Layer的區別：

- 它對外開放內置的訓練，評估和預測函數（model.fit(),model.evaluate(),model.predict()）。
- 它通過model.layers屬性開放其內部網路層清單。
- 它對外開放保存和序列化API。

4.1 自訂模型

下面通過構建一個變分自編碼器（VAE），來介紹如何構建自己的網路， 並使用內置的函數進行訓練。

In [12]:
```
# 採樣網路
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5*z_log_var) * epsilon
# 編碼器
class Encoder(layers.Layer):
    def __init__(self, latent_dim=32,
                intermediate_dim=64, name='encoder', **kwargs):
        super(Encoder, self).__init__(name=name, **kwargs)
        self.dense_proj = layers.Dense(intermediate_dim, activation='relu')
        self.dense_mean = layers.Dense(latent_dim)
        self.dense_log_var = layers.Dense(latent_dim)
        self.sampling = Sampling()

    def call(self, inputs):
        h1 = self.dense_proj(inputs)
        # 獲取z_mean和z_log_var
        z_mean = self.dense_mean(h1)
        z_log_var = self.dense_log_var(h1)
        # 進行採樣
        z = self.sampling((z_mean, z_log_var))
        return z_mean, z_log_var, z

# 解碼器
class Decoder(layers.Layer):
    def __init__(self, original_dim,
                 intermediate_dim=64, name='decoder', **kwargs):
        super(Decoder, self).__init__(name=name, **kwargs)
        self.dense_proj = layers.Dense(intermediate_dim, activation='relu')
        self.dense_output = layers.Dense(original_dim, activation='sigmoid')
    def call(self, inputs):
        # 兩層全連接網路
        h1 = self.dense_proj(inputs)
        return self.dense_output(h1)

# 變分自編碼器
class VAE(tf.keras.Model):
    def __init__(self, original_dim, latent_dim=32,
                intermediate_dim=64, name='encoder', **kwargs):
        super(VAE, self).__init__(name=name, **kwargs)

        self.original_dim = original_dim
        self.encoder = Encoder(latent_dim=latent_dim,
                              intermediate_dim=intermediate_dim)
        self.decoder = Decoder(original_dim=original_dim,
                              intermediate_dim=intermediate_dim)
    def call(self, inputs):
        # 編碼
        z_mean, z_log_var, z = self.encoder(inputs)
        # 解碼
        reconstructed = self.decoder(z)
        # 獲取損失函數
```

```
        kl_loss = -0.5*tf.reduce_sum(
            z_log_var-tf.square(z_mean)-tf.exp(z_log_var)+1)
        self.add_loss(kl_loss)
        return reconstructed
```

訓練VAE

In [13]:

```
(x_train, _), _ = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
vae = VAE(784,32,64)
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

vae.compile(optimizer, loss=tf.keras.losses.MeanSquaredError())
vae.fit(x_train, x_train, epochs=3, batch_size=64)
```

```
Train on 60000 samples
Epoch 1/3
60000/60000 [==============================] - 4s 58us/sample - loss: 1.0678
Epoch 2/3
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0695
Epoch 3/3
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0680
```

Out[13]:

```
<tensorflow.python.keras.callbacks.History at 0x7f88d00ac2b0>
```

自己編寫訓練方法

In [15]:
```
train_dataset = tf.data.Dataset.from_tensor_slices(x_train)
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(64)

original_dim = 784
vae = VAE(original_dim, 64, 32)

# 優化器
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
# 損失函數
mse_loss_fn = tf.keras.losses.MeanSquaredError()
# 評價指標
loss_metric = tf.keras.metrics.Mean()

# 訓練迴圈
for epoch in range(3):
    print('Start of epoch %d' % (epoch,))

    # 每批次訓練
    for step, x_batch_train in enumerate(train_dataset):
        with tf.GradientTape() as tape:
            # 前向傳播
            reconstructed = vae(x_batch_train)
            # 計算loss
            loss = mse_loss_fn(x_batch_train, reconstructed)
            loss += sum(vae.losses)  # Add KLD regularization loss
        # 計算梯度
        grads = tape.gradient(loss, vae.trainable_variables)
        # 反向傳播
        optimizer.apply_gradients(zip(grads, vae.trainable_variables))
        # 統計指標
        loss_metric(loss)
        # 輸出
        if step % 100 == 0:
            print('step %s: mean loss = %s' % (step, loss_metric.result()))
```

```
Start of epoch 0
step 0: mean loss = tf.Tensor(192.1773, shape=(), dtype=float32)
step 100: mean loss = tf.Tensor(6.5825667, shape=(), dtype=float32)
step 200: mean loss = tf.Tensor(3.3554409, shape=(), dtype=float32)
step 300: mean loss = tf.Tensor(2.2692108, shape=(), dtype=float32)
step 400: mean loss = tf.Tensor(1.7223562, shape=(), dtype=float32)
step 500: mean loss = tf.Tensor(1.3939205, shape=(), dtype=float32)
step 600: mean loss = tf.Tensor(1.1746095, shape=(), dtype=float32)
step 700: mean loss = tf.Tensor(1.0176468, shape=(), dtype=float32)
step 800: mean loss = tf.Tensor(0.8996144, shape=(), dtype=float32)
step 900: mean loss = tf.Tensor(0.8074596, shape=(), dtype=float32)
Start of epoch 1
step 0: mean loss = tf.Tensor(0.77757883, shape=(), dtype=float32)
step 100: mean loss = tf.Tensor(0.70945406, shape=(), dtype=float32)
step 200: mean loss = tf.Tensor(0.6533016, shape=(), dtype=float32)
step 300: mean loss = tf.Tensor(0.60621214, shape=(), dtype=float32)
step 400: mean loss = tf.Tensor(0.5661074, shape=(), dtype=float32)
step 500: mean loss = tf.Tensor(0.5315464, shape=(), dtype=float32)
step 600: mean loss = tf.Tensor(0.50146633, shape=(), dtype=float32)
```

```
step 700: mean loss = tf.Tensor(0.4750789, shape=(), dtype=float32)
step 800: mean loss = tf.Tensor(0.4516706, shape=(), dtype=float32)
step 900: mean loss = tf.Tensor(0.43074456, shape=(), dtype=float32)
Start of epoch 2
step 0: mean loss = tf.Tensor(0.42340422, shape=(), dtype=float32)
step 100: mean loss = tf.Tensor(0.40543476, shape=(), dtype=float32)
step 200: mean loss = tf.Tensor(0.38920242, shape=(), dtype=float32)
step 300: mean loss = tf.Tensor(0.3744474, shape=(), dtype=float32)
step 400: mean loss = tf.Tensor(0.3610152, shape=(), dtype=float32)
step 500: mean loss = tf.Tensor(0.34867495, shape=(), dtype=float32)
step 600: mean loss = tf.Tensor(0.33732668, shape=(), dtype=float32)
step 700: mean loss = tf.Tensor(0.326859, shape=(), dtype=float32)
step 800: mean loss = tf.Tensor(0.3171746, shape=(), dtype=float32)
step 900: mean loss = tf.Tensor(0.30814958, shape=(), dtype=float32)
```

In [ ]: