浅谈keras的扩展性：自定义keras

个人简介： wedo实验君, 数据分析师；热爱生活，热爱写作

1. 自定义keras

keras是一种深度学习的API，能够快速实现你的实验。keras也集成了很多预训练的模型，可以实现很多常规的任务，如图像分类。TensorFlow 2.0之后tensorflow本身也变的很keras化。

另一方面，keras表现出高度的模块化和封装性，所以有的人会觉得keras不易于扩展， 比如实现一种新的Loss，新的网络层结构； 其实可以通过keras的基础模块进行快速的扩展，实现更新的算法。

本文就keras的扩展性，总结了对layer，model和loss的自定义。

2. 自定义keras layers

layers是keras中重要的组成部分，网络结构中每一个组成都要以layers来表现。keras提供了很多常规的layer, 如Convolution layers，pooling layers， activation layers， dense layers等， 我们可以通过继承基础layers来扩展自定义的layers。

2.1 base layer

layer实了输入tensor和输出tensor的操作类，以下为base layer的5个方法，自定义layer只要重写这些方法就可以了。

- **init**(): 定义自定义layer的一些属性
- build(self, input_shape)： 定义layer需要的权重weights
- call(self, *args, **kwargs)：layer具体的操作，会在调用自定义layer自动执行
- get_config(self)：layer初始化的配置，是一个字典dictionary。
- compute_output_shape(self,input_shape)：计算输出tensor的shape

2.2 例子

```
# 标准化层
class InstanceNormalize(Layer):
    def __init__(self, **kwargs):
        super(InstanceNormalize, self).__init__(**kwargs)
        self.epsilon = 1e-3
    def call(self, x, mask=None):
        mean, var = tf.nn.moments(x, [1, 2], keep_dims=True)
        return tf.div(tf.subtract(x, mean), tf.sqrt(tf.add(var, self.epsilon)))


    def compute_output_shape(self,input_shape):
        return input_shape

# 调用
inputs = keras.Input(shape=(None, None, 3))
x = InstanceNormalize()(inputs)
```

```
# 可以通过add_weight() 创建权重
class SimpleDense(Layer):

  def __init__(self, units=32):
      super(SimpleDense, self).__init__()
      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

# 调用
inputs = keras.Input(shape=(None, None, 3))
x = SimpleDense(units=64)(inputs)
```

3. 自定义keras model

我们在定义完网络结构时，会把整个工作流放在`keras.Model`， 进行`compile()`, 然后通过`fit()`进行训练过程。执行`fit()`的时候，执行每个batch size data的时候，都会调用`Model`中`train_step(self, data)`

```
from keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential()

model.add(Dense(units=64, input_dim=100))
model.add(Activation("relu"))
model.add(Dense(units=10))
model.add(Activation("softmax"))
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=
['accuracy'])

model.fit(x_train, y_train, epochs=5, batch_size=32)
```

当你需要自己控制训练过程的时候，可以重写`Model`的`train_step(self, data)`方法

```
class CustomModel(keras.Model):
    def train_step(self, data):
        # Unpack the data. Its structure depends on your model and
        # on what you pass to `fit()`.
        x, y = data

        with tf.GradientTape() as tape:
            y_pred = self(x, training=True)  # Forward pass
            # Compute the loss value
            # (the loss function is configured in `compile()`)
            loss = self.compiled_loss(y, y_pred,
regularization_losses=self.losses)

        # Compute gradients
        trainable_vars = self.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
        # Update weights
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))
        # Update metrics (includes the metric that tracks the loss)
        self.compiled_metrics.update_state(y, y_pred)
        # Return a dict mapping metric names to current value
        return {m.name: m.result() for m in self.metrics}

import numpy as np

# Construct and compile an instance of CustomModel
inputs = keras.Input(shape=(32,))
outputs = keras.layers.Dense(1)(inputs)
model = CustomModel(inputs, outputs)
model.compile(optimizer="adam", loss="mse", metrics=["mae"])

# Just use `fit` as usual
x = np.random.random((1000, 32))
y = np.random.random((1000, 1))
model.fit(x, y, epochs=3)
```

4. 自定义keras loss

keras实现了交叉熵等常见的loss，自定义loss对于使用keras来说是比较常见，实现各种魔改loss，如focal loss。

我们来看看keras源码中对loss实现

```
def categorical_crossentropy(y_true, y_pred):
    return K.categorical_crossentropy(y_true, y_pred)

def mean_squared_error(y_true, y_pred):
    return K.mean(K.square(y_pred - y_true), axis=-1)
```

可以看出输入是groud true `y_true`和预测值`y_pred`， 返回为计算loss的函数。自定义loss可以参照如此模式即可。

```
def focal_loss(weights=None, alpha=0.25, gamma=2):
    r"""Compute focal loss for predictions.
        Multi-labels Focal loss formula:
            FL = -alpha * (z-p)^gamma * log(p) -(1-alpha) * p^gamma * log(1-p)
                 ,which alpha = 0.25, gamma = 2, p = sigmoid(x), z = 
target_tensor.
    # https://github.com/ailias/Focal-Loss-implement-on-
Tensorflow/blob/master/focal_loss.py
    Args:
     prediction_tensor: A float tensor of shape [batch_size, num_anchors,
        num_classes] representing the predicted logits for each class
     target_tensor: A float tensor of shape [batch_size, num_anchors,
        num_classes] representing one-hot encoded classification targets
     weights: A float tensor of shape [batch_size, num_anchors]
     alpha: A scalar tensor for focal loss alpha hyper-parameter
     gamma: A scalar tensor for focal loss gamma hyper-parameter
    Returns:
        loss: A (scalar) tensor representing the value of the loss function
    """

    def _custom_loss(y_true, y_pred):
        sigmoid_p = tf.nn.sigmoid(y_pred)
        zeros = array_ops.zeros_like(sigmoid_p, dtype=sigmoid_p.dtype)

        # For poitive prediction, only need consider front part loss, back part is 
0;
        # target_tensor > zeros <=> z=1, so poitive coefficient = z - p.
        pos_p_sub = array_ops.where(y_true > zeros, y_true - sigmoid_p, zeros)

        # For negative prediction, only need consider back part loss, front part 
is 0;
        # target_tensor > zeros <=> z=1, so negative coefficient = 0.
        neg_p_sub = array_ops.where(y_true > zeros, zeros, sigmoid_p)
        per_entry_cross_ent = - alpha * (pos_p_sub ** gamma) * 
tf.log(tf.clip_by_value(sigmoid_p, 1e-8, 1.0)) \
                              - (1 - alpha) * (neg_p_sub ** gamma) * tf.log(
            tf.clip_by_value(1.0 - sigmoid_p, 1e-8, 1.0))
        return tf.reduce_sum(per_entry_cross_ent)

    return _custom_loss
```

5. 总结

本文分享了keras的扩展功能，扩展功能其实也是实现Keras模块化的一种继承实现。

总结如下：

- 继承Layer实现自定义layer， 记住`bulid()` `call()`
- 继续Model实现`train_step`定义训练过程，记住梯度计算`tape.gradient(loss, trainable_vars)`，权重更新`optimizer.apply_gradients`, 计算evaluate `compiled_metrics.update_state(y, y_pred)`
- 魔改loss，记住groud true `y_true`和预测值`y_pred`输入，返回loss function