1 Project Description

This project involves creating multiple parameter models of ResNet for predicting gravitational waves generated in the universe. Specific data and project details can be found at g2net-gravitational-wave-detection. The innovative approach is to construct 6 ResNet models, ranging from ResNet16 to ResNet152, in one go without the need for rebuilding in the code. For detailed data and code outputs, please refer to my github.

2 Import Python Packages

```
import numpy as np
import tensorflow as tf
print(tf.__version__)
from tensorflow import keras
from tensorflow.keras import layers,Sequential,optimizers
import random
import pandas as pd
from matplotlib import pyplot as plt
#import torch
import math
import collections
import os
import logging
from datetime import datetime
```

3 Import Datasets

```
model_weights_file = '..//content//model//resnet_model.ckpt' # reloading
↪  training weight save time
train_df = pd.read_csv('/content/training_labels.csv')
test_df = pd.read_csv('/content/sample_submission.csv')
def get_file_path(idx, train = True):
    path = "../content/g2net-image/"
    if train:
        path += 'train/'
    else:
        path += 'test/'

    path += idx+'.npy'
    return path

train_df['path'] = train_df['id'].apply(get_file_path, train = True)
test_df['path'] = test_df['id'].apply(get_file_path, train = False)
```

4 Preprocessing Data

```
num_classes = 1
batch_size = 500

def load_and_preprocess_from_label(path, label):
    path = path.numpy()
    image = np.load(path.decode()).astype(np.float32)
    image = (image - 0.75) * 4
    image = tf.cast(image, tf.float32)
    image = tf.convert_to_tensor(image, tf.float32)
    image = tf.squeeze(image)
    label = tf.expand_dims(label, -1)
    label = tf.cast(label, tf.int32)
    label = tf.convert_to_tensor(label, tf.int32)

    return image, label

def preprocess_test(path, labelid):
    path = path.numpy()
    image = np.load(path.decode()).astype(np.float32)
    image = (image - 0.75) * 4
    image = tf.cast(image, tf.float32)
    image = tf.convert_to_tensor(image, tf.float32)
    image = tf.squeeze(image)
    labelid = tf.convert_to_tensor(labelid, tf.string)
    return image, labelid

allPath,allPathTest = tf.convert_to_tensor(train_df['path'],
↪  dtype=tf.string), \
    tf.convert_to_tensor(test_df['path'], dtype=tf.string)
train_target,test_target =
↪  tf.convert_to_tensor(train_df['target'].to_numpy(), dtype=tf.int32), \
    tf.convert_to_tensor(test_df['id'].to_numpy(), dtype=tf.string)

train_dataset = tf.data.Dataset.from_tensor_slices((allPath, train_target))
test_dataset = tf.data.Dataset.from_tensor_slices((allPathTest,
↪  test_target))

train_dataset = train_dataset.map(lambda x,y:
↪  tf.py_function(load_and_preprocess_from_label,[x, y],[tf.float32,
↪  tf.int32]))
train_dataset = train_dataset.batch(batch_size)
test_dataset = test_dataset.map(lambda x,y:
↪  tf.py_function(preprocess_test,[x, y],[tf.float32, tf.string]))
test_dataset = test_dataset.batch(batch_size)

for x, y in train_dataset.take(1):
    print(np.max(x),np.min(x), x.shape, y.shape)
```

5 Building ResNet Model

```
class BasicBlock(layers.Layer):

    def __init__(self, filter_num, strides=1):
        super(BasicBlock, self).__init__()

        self.conv1 = layers.Conv2D(filter_num, kernel_size=3,
        ↪  strides=strides, padding='same', bias=False)
        self.bn1 = layers.BatchNormalization()
        self.relu = layers.Activation('relu')

        self.conv2 = layers.Conv2D(filter_num, kernel_size=3, strides=1,
        ↪  padding='same', bias=False)
        self.bn2 = layers.BatchNormalization()

        if strides != 1:
            self.downsample = Sequential()
            self.downsample.add(layers.Conv2D(filter_num, kernel_size=1,
            ↪  strides=strides, bias=False))
        else:
            self.downsample = lambda x:x

    def call(self, inputs, training=None):

        out = self.conv1(inputs)
        out = self.bn1(out, training=training)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out, training=training)

        identity = self.downsample(inputs)

        output = layers.add([out, identity])
        output = tf.nn.relu(output)

        return output

class Bottleneck(layers.Layer):
    expansion = 4
    def __init__(self, filter_num, strides=1):
        super(Bottleneck, self).__init__()

        self.conv1 = layers.Conv2D(filter_num, kernel_size=1,
        ↪  use_bias=False)
        self.bn1 = layers.BatchNormalization()
        self.relu = layers.Activation('relu')

        self.conv2 = layers.Conv2D(filter_num, kernel_size=3,
        ↪  strides=strides, use_bias=False, padding='same')
        self.bn2 = layers.BatchNormalization()
        self.relu = layers.Activation('relu')

        self.conv3 = layers.Conv2D(filter_num * self.expansion,
        ↪  kernel_size=1, strides=1, use_bias=False, padding='same')
        self.bn3 = layers.BatchNormalization()

        if strides != 1 or filter_num != filter_num * self.expansion:
            self.downsample = Sequential()
            self.downsample.add(layers.Conv2D(filter_num * self.expansion,
            ↪  kernel_size=1, strides=strides, use_bias=False))
        else:
            self.downsample = lambda x:x

    def call(self, inputs, training=None):
        out = self.conv1(inputs)
        out = self.bn1(out, training=training)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out, training=training)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out, training=training)

        identity = self.downsample(inputs)

        output = layers.add([out, identity])
        output = tf.nn.relu(output)

        return output
class ResNet(keras.Model):

    def __init__(self, layer_dims, num_classed=10, is_neck=False):
    ↪  #[2,2,2,2]
        super(ResNet, self).__init__()

        self.stem = Sequential([layers.Conv2D(filters=64, kernel_size=7,
        ↪  strides=2, use_bias=False),
                                layers.BatchNormalization(),
                                layers.Activation('relu'),
                                layers.MaxPool2D(pool_size=3, strides=2,
                                ↪  padding='same')])

        if is_neck:
            self.layer1 = self.build_resblockneck(64, layer_dims[0])
            self.layer2 = self.build_resblockneck(128, layer_dims[1],
            ↪  strides=2)
            self.layer3 = self.build_resblockneck(256, layer_dims[2],
            ↪  strides=2)
```

```
            self.layer4 = self.build_resblockneck(512, layer_dims[3],
            ↪  strides=2)
        else:
            self.layer1 = self.build_resblock(64, layer_dims[0])
            self.layer2 = self.build_resblock(128, layer_dims[1], strides=2)
            self.layer3 = self.build_resblock(256, layer_dims[2], strides=2)
            self.layer4 = self.build_resblock(512, layer_dims[3], strides=2)

        self.avgpool = layers.GlobalAveragePooling2D()
        self.fc = layers.Dense(num_classes, activation='sigmoid')

    def call(self, inputs, training=None):
        x = self.stem(inputs, training=training)
        x = self.layer1(x, training=training)
        x = self.layer2(x, training=training)
        x = self.layer3(x, training=training)
        x = self.layer4(x, training=training)

        x = self.avgpool(x)
        x = self.fc(x)

        return x

    def build_resblock(self, filter_num, blocks, strides=1):

        res_blocks = Sequential()
        res_blocks.add(BasicBlock(filter_num, strides))

        for _ in range(1, blocks):
            res_blocks.add(BasicBlock(filter_num, strides=1))

        return res_blocks

    def build_resblockneck(self, filter_num, blocks, strides=1):

        res_blocks = Sequential()
        res_blocks.add(Bottleneck(filter_num, strides))

        for _ in range(1, blocks):
            res_blocks.add(Bottleneck(filter_num, strides=1))

        return res_blocks
```

6 Create Multiple Models and Select One to Use

```
def resnet16():
    return ResNet([1, 2, 2, 1],is_neck=True)

def resnet18():
    return ResNet([2, 2, 2, 2],is_neck=True)

def resnet34():
    return ResNet([3, 4, 6, 3])

def resnet50():
    return ResNet([3, 4, 6, 3], is_neck=True)

def resnet101():
    return ResNet([3, 4, 23, 3], is_neck=True)

def resnet152():
    return ResNet([3, 8, 36, 3], is_neck=True)


resnet = resnet16()
resnet.build(input_shape=(None, 69, 129, 3))
resnet.summary()
if os.path.exists(model_weights_file + '.index'):
    resnet.load_weights(model_weights_file)
    print('load weights.')
```

7 Setting Training Parameters

```
optimizer = tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
)

import time
def format_seconds(sec):
    m, s = divmod(sec, 60)
    h, m = divmod(m, 60)
    return ("%02d:%02d:%02d" % (h, m, s))

start = time.perf_counter()
time.sleep(1)
dur = time.perf_counter()
dif = dur - start
print(format_seconds(dif))
```

8 Build Training and Testing Function

```
def computeAcc(label, pred):
    pred = pred // 0.5
    correct = tf.reduce_sum(tf.cast(tf.equal(tf.cast(pred, tf.int32),
    ↪  tf.cast(label, tf.int32)),tf.int32))
    return correct

def train(epoch, logStep = 1, saveStep = 1000):
    total_num, total_loss, total_correct = 0, 0, 0
    cur_num, cur_loss, cur_correct  = 0, 0, 0
    cnt = train_df.count()[0] // batch_size
    scale = cnt // 30
    start = time.perf_counter()
    for step,(x,y) in enumerate(train_dataset):
        with tf.GradientTape() as tape:
            logits = resnet(x,training=True)
            #loss
            loss = tf.losses.binary_crossentropy(y, logits)
            loss = tf.reduce_mean(loss)

        #print(loss.trainable_variables)
        grads = tape.gradient(loss, resnet.trainable_variables)
        #print(grads)
        optimizer.apply_gradients(zip(grads, resnet.trainable_variables))

        correct = computeAcc(y, logits)

#         cur_num += x.shape[0]
#         cur_loss += loss * x.shape[0]
#         cur_correct += int(correct)
        total_num += x.shape[0]
        total_correct += int(correct)
        total_loss += loss

#         if step % logStep == 0:
#             acc = cur_correct / cur_num
#             #logging.getLogger().info(epoch, step,'/',total_num, ' loss:',
↪  float(cur_loss / cur_num),'acc=', acc)
#             cur_correct = 0
#             cur_num = 0
#             cur_loss = 0

        if step % saveStep == 0 and step != 0:
            resnet.save_weights(model_weights_file)
            print('saved weights.')

        a = "*" * (step // scale)
        b = "." * ((cnt - step) // scale)
        c = (step / cnt) * 100
        dur = (time.perf_counter() - start)
        start = time.perf_counter()
        print("\r epoch({:^d}) now used time ls:{:^.5f} acc:{:^.3f} total
        ↪  loss:{:^.5f} Acc:{:^.3f} {:^3.2f}% [{}->{}]{:^.2f}s need time:{}
        ↪  {:^d}   "\
              .format(epoch, loss, correct / batch_size, total_loss / (step
              ↪  + 1), total_correct / total_num, c,a,b,dur \
              ,format_seconds(dur * (cnt - step)), total_num),end = "")

    #acc = total_correct / total_num
    #print(epoch, 'acc=', acc,'total_num=', total_num)

    resnet.save_weights(model_weights_file)
    print('saved weights.')

def test(epoch):
    y_pred = []
    ids = []
    test_num = 0
    start = time.perf_counter()
    proCnt = 20
    for step,(x,y) in enumerate(test_dataset):

        #print(step)
        out = resnet(x, training=False)
        test_num += x.shape[0]

        y_pred.extend(tf.squeeze(out).numpy())
        ids.extend(y.numpy().astype(str))

        cnt = test_df.count()[0] // batch_size
        scale = cnt // proCnt
        a = "*" * (step // scale)
        b = "." * ((cnt - step) // scale)
        c = (step / cnt) * 100
        dur = (time.perf_counter() - start)
        start = time.perf_counter()
        print("\r{:^3.1f}%[{}->{}]{:^.2f}s need time:".format(c,a,b,dur) +
        ↪  format_seconds(dur * (cnt - step)),end = "")

    submission = pd.DataFrame({"id": ids, "target": y_pred})
    print(submission)
    submission.to_csv("submission("+ str(epoch)+").csv", index=False)
```

9 Training

```
for epoch in range(15): # range() could define training epoches
    train_dataset = train_dataset.shuffle(buffer_size=1)
    train(epoch, 1, 3000)
    test(epoch)
```