HW05 CNN

3180101041 杨锐

1.软件开发说明

1.1 开发环境

- @MacOS
- Python == 3.7.6
- Numpy == 1.8.1
- tensorflow == 2.2.0
- Matplotlib.pyplot == 3.1.2
- Click == 7.0

1.2 运行方式

```
# 训练mnist数据，将模型保存到本地，并测试数据，输出测试准确率
python cnn.py mnist --option train
# 读取本地已训练好的模型，测试数据，输出测试准确率
python cnn.py mnist --option test
# 训练cifar-10数据，将模型保存到本地，并测试数据，输出测试准确率
python cnn.py cifar_10 --option train
# 读取本地已训练好的模型，测试数据，输出测试准确率
python cnn.py cifar_10 --option test
```

2.算法设计说明

2.1 LeNet-5

1)读取Mnist数据集，并进行预处理

- 读取数据到本地，返回训练图像、训练标签、测试图像、测试标签:

```
# 加载训练数据
(train_images, train_labels), (test_images,
                               test_labels) = datasets.mnist.load_data()
```

由于LeNet-5网络第一层卷积的输入尺寸是32x32x1，而mnist数据集读取图片尺寸为28x28，因此在进行训练之前我们需要对图片进行如下预处理：

- 将像素的值标准化至0到1的区间内：

```
train_images, test_images = train_images / 255.0, test_images / 255.0
```

- 将28 x 28的图片填充为32 x 32

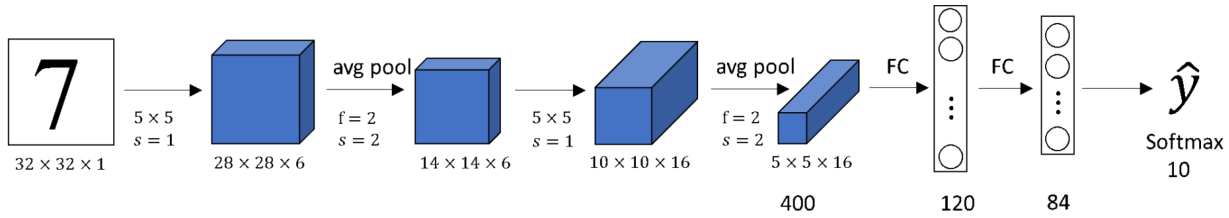
```
train_images = np.pad(
    train_images, ((0, 0), (2, 2), (2, 2)), 'constant')
test_images = np.pad(test_images, ((0, 0), (2, 2), (2, 2)), 'constant')
```

- 将32x32的图片扩展为32x32x1

```
# 将32 x 32展开为32 x 32 x 1
tmp_train = []
tmp_test = []
for train_image in train_images:
    tmp_train.append(train_image.reshape(32, 32, 1))
for test_image in test_images:
    tmp_test.append(test_image.reshape(32, 32, 1))
return np.array(tmp_train), np.array(tmp_test)
```

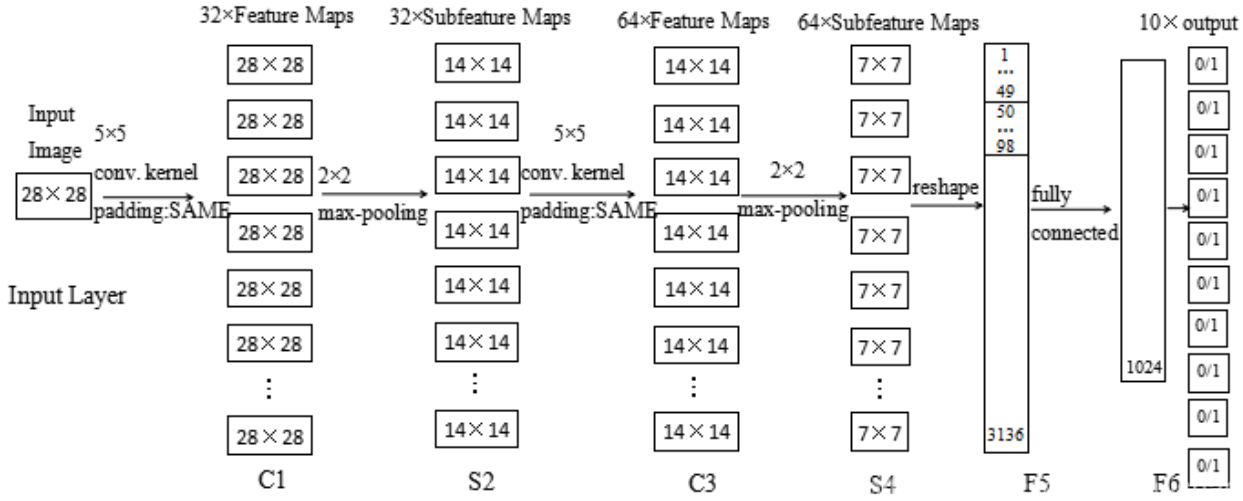
2) 构建LeNet-5

LeNet-5是经典的CNN，其结构示意图如下：



具体来说，该网络一共7层（不包括输入）

- C1层是具有6个5x5的卷积核的卷积层，特征映射的大小为28x28，可以防止输入图像的信息落入卷积核的边界。
- S2是子采样/池化层，可输出6个大小为14x14的特征图。每个要素图中的每个像元都连接到C1中相应要素图中的2x2邻域。
- C3层是具有16个5-5卷积内核的卷积层。前六个C3特征图的输入是S2中三个特征图的每个连续子集，接下来六个特征图的输入来自四个连续子集的输入，接下来三个特征图的输入来自四个不连续子集。最后，最后一个特征图的输入来自S2的所有特征图。
- S4层与S2相似，大小为2x2，输出为16个5x5特征图。
- C5层是一个卷积层，具有120个大小为5x5的卷积核。每个单元都连接到S4的所有16个特征图上的5 * 5邻域。在此，由于S4的特征图大小也为5x5，因此C5的输出大小为1 * 1。因此S4和C5已完全连接。 C5被标记为卷积层而不是全连接层，因为如果lenet-5输入变大并且其结构保持不变，则其输出大小将大于1x1，即不是全连接层（后续实验中采取flatten+全连接）。
- F6层完全连接到C5，并输出了84个特征图
- 最后一层做一个softmax，将特征归一化到10，对应手写字体的10个分类。



在具体实现中，使用tensorflow.keras提供的api:

```
def create_model(input_size=(32, 32, 1), kernel_size=(5, 5)):
    model = models.Sequential()

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    model.add(layers.Conv2D(filters=6, kernel_size=kernel_size, activation='relu',
input_shape=input_size))
    # Pooling. Input = 28x28x6. Output = 14x14x6.
    model.add(layers.AveragePooling2D())

    # Layer 2: Convolutional. Output = 10x10x16.
    model.add(layers.Conv2D(filters=16, kernel_size=kernel_size, activation='relu'))
    # Pooling. Input = 10x10x16. Output = 5x5x16.
    model.add(layers.AveragePooling2D())

    # Flatten. Input = 5x5x16. Output = 400.
    model.add(layers.Flatten())
    # Layer 3: Fully Connected. Input = 400. Output = 120.
    model.add(layers.Dense(units=120, activation='relu'))
    # Layer 4: Fully Connected. Input = 120. Output = 84.
    model.add(layers.Dense(units=84, activation='relu'))
    # Layer 5: Fully Connected. Input = 84. Output = 10.
    model.add(layers.Dense(units=10, activation='softmax'))

    return model
```

- `models.Sequential()`定义了一个模型序列，通过`model.add()`添加每一层的结构
- `layers.Conv2D(filters=?,kernel_size=(?,?),activation='?',input_shape=?)`定义了卷积层
  - `filters`定义卷积核的数量。
  - `kernel_size`定义卷积核的宽度和高度，深度一定和上一层保持一致（注意和filters区分开）。具体来说，当输入图像为32x32x3时，kernel_size = 5x5，则卷积核的大小为5x5x3，即RGB三通道。
  - `activation`定义激活函数。每一个神经元都必须指定激活函数，对于CNN来说，卷积层和全连接隐藏层通常使用整流线性单元（*ReLu*），输出层通常使用归一化指数函数(*Softmax*)

- *ReLu*：对于负的输入，输出为0；对于正的输入，直接输出。对于CNN，抑制负的神经元能够显著减少卷积层的计算量。缺点时一旦该神经元输入为负，将永远被抑制，无法进行任何训练
- *Softmax*: 一个神经元的输出取决于该神经元的输入占总的神经元的比例，即范围是[0:1]。优点之一是输出小，因此梯度不会过大。具体计算公式是：

$$y_i = exp(x_i) / \sum_j exp(x_j)$$

- `layers.AveragePooling2D()`定义了池化层，常用的有max-pooling和mean-pooling，LeNet-5采用的是mean-pooling，即取n×n区域内像素的均值。
- `layers.Flatten()`将输入特征转换为一维特征
- `layers.Dense()`定义了全连接，`units()`定义了该层神经元数量。对于输出层，我们通常将神经元设置为期望结果数量，并设置激活函数为*Softmax*，将输出归一化到*[0:1]*的概率。

3）设置模型损失函数和优化器

```
# 编译模型
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(
                  from_logits=True),
              metrics=['accuracy'])
```

- 损失函数：这里使用了交叉熵来计算损失函数，loss越小，模型越精确，交叉熵具体公式为：$H_y(y) = -\sum_i y_i' \log(y_i)$
- 使用Adam优化器来进行训练，其中学习率默认为0.001。

4）训练模型

传入训练图像和目标输出，设定训练轮树/为10。将训练得到的模型权重保存在本地，以供测试使用。

```
# 训练模型 epochs = 10
history = model.fit(train_images, train_labels, epochs=10)
# 保存权重到本地
model.save_weights('./mnist_models/mnist_model')
```

5) 测试模型准确率

读取保存在本地的模型权重，测试识别准确率

```
model = create_model()
# 载入模型权重
model.load_weights('./mnist_models/mnist_model')
# 测试模型
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=1)
print(test_loss, test_acc)
```

2.2 改进的CNN

LeNet-5对Mnist的识别率在10个epoches训练之后就能达到0.9852。但当运用到cifar-10上时，识别率仅能达到50%左右。主要原因在于cifar-10的数据相比Mnist细节更加丰富，因此我们适当调小了滤波器的尺寸，增加了每一层卷积核的数量以描述更复杂的特征。

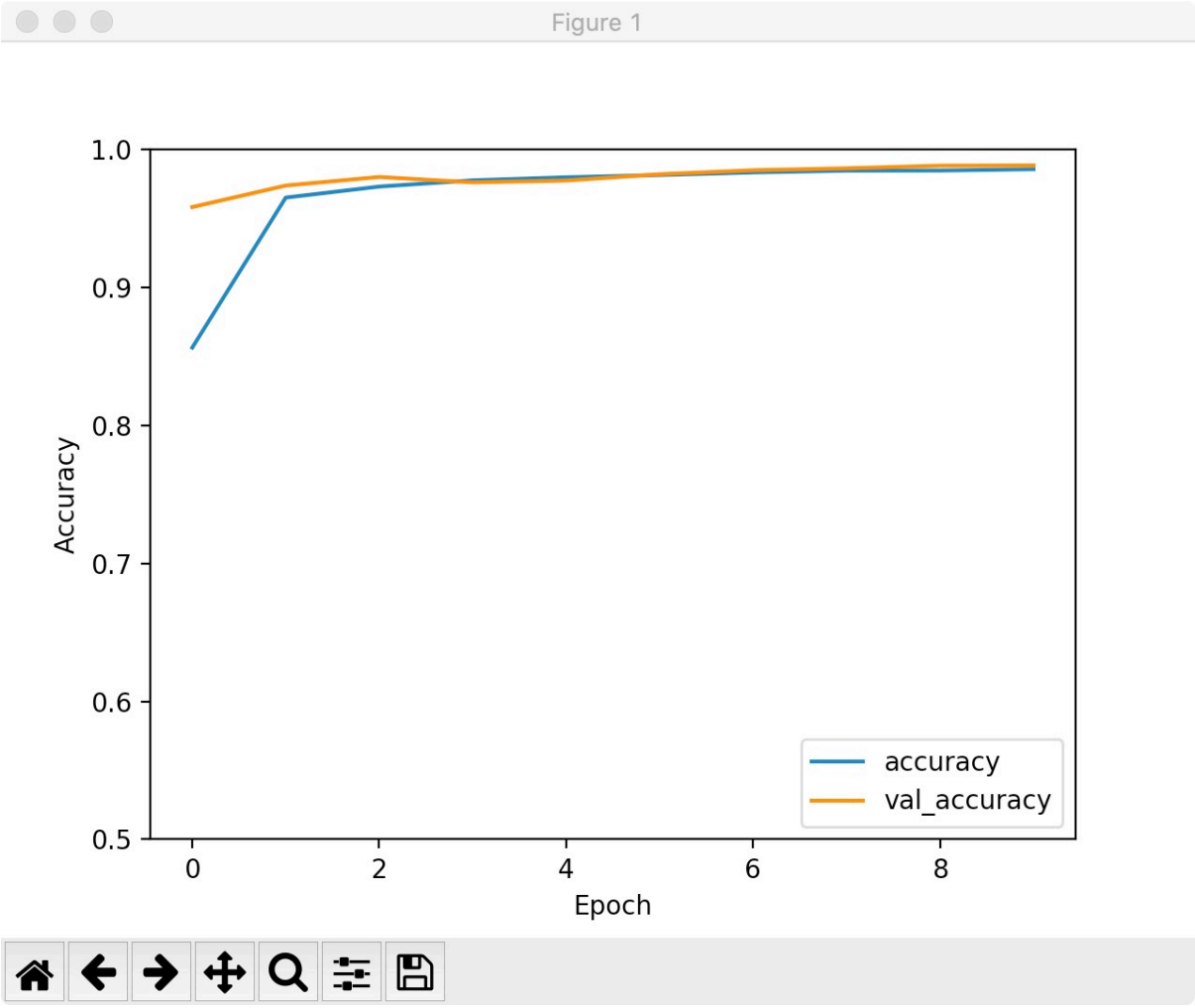
```
def create_better_model():
    model = models.Sequential()

    model.add(layers.Conv2D(
        32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(64, (3, 3), activation='relu'))

    model.add(layers.Flatten())

    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))

    return model
```

在epoches = 10的时候，识别率能够达到70%。不太令人满意，但对于这样简单的CNN，我们已经可以接受。之后我继续增加的了网络层数和每一层的参数，结果变化不大，甚至可以发现在epoches超过10之后，网络出现了过拟合（训练数据集准确率提高，但验证数据集的识别率反复波动），详细信息在结果与分析中给出。

3.实验结果分析

Mnist

- 训练结果：

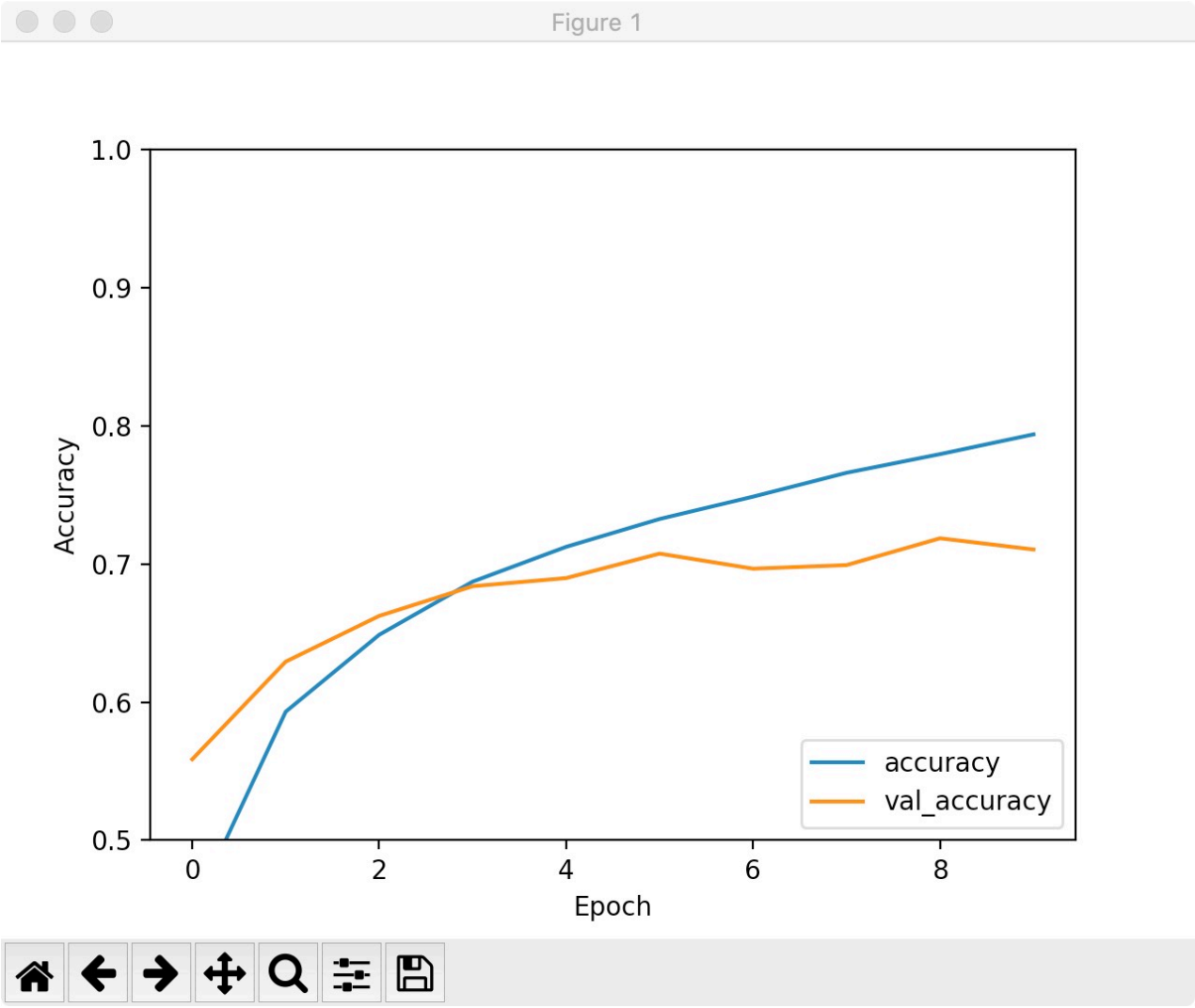


- 识别结果：0.9884

```
(cnn) python cnn.py mnist --option train
2021-01-07 21:19:31.122020: I tensorflow/core/platform/cpu_feature_guard.cc:143] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
2021-01-07 21:19:31.137482: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x7fa76146b670 initialized for platform Host (this does not guarantee that XLA will be used). Devices:
2021-01-07 21:19:31.137503: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
Epoch 1/10
1875/1875 [==============================] - 14s 8ms/step - loss: 1.6090 - accuracy: 0.8565 - val_loss: 1.5048 - val_accuracy: 0.9583
Epoch 2/10
1875/1875 [==============================] - 14s 8ms/step - loss: 1.4971 - accuracy: 0.9652 - val_loss: 1.4878 - val_accuracy: 0.9739
Epoch 3/10
1875/1875 [==============================] - 14s 7ms/step - loss: 1.4888 - accuracy: 0.9732 - val_loss: 1.4813 - val_accuracy: 0.9801
Epoch 4/10
1875/1875 [==============================] - 15s 8ms/step - loss: 1.4838 - accuracy: 0.9777 - val_loss: 1.4852 - val_accuracy: 0.9762
Epoch 5/10
1875/1875 [==============================] - 15s 8ms/step - loss: 1.4816 - accuracy: 0.9799 - val_loss: 1.4836 - val_accuracy: 0.9775
Epoch 6/10
1875/1875 [==============================] - 15s 8ms/step - loss: 1.4798 - accuracy: 0.9815 - val_loss: 1.4786 - val_accuracy: 0.9821
Epoch 7/10
1875/1875 [==============================] - 15s 8ms/step - loss: 1.4779 - accuracy: 0.9834 - val_loss: 1.4764 - val_accuracy: 0.9850
Epoch 8/10
1875/1875 [==============================] - 15s 8ms/step - loss: 1.4766 - accuracy: 0.9847 - val_loss: 1.4747 - val_accuracy: 0.9864
Epoch 9/10
1875/1875 [==============================] - 15s 8ms/step - loss: 1.4763 - accuracy: 0.9848 - val_loss: 1.4732 - val_accuracy: 0.9883
Epoch 10/10
1875/1875 [==============================] - 14s 8ms/step - loss: 1.4754 - accuracy: 0.9858 - val_loss: 1.4729 - val_accuracy: 0.9885
313/313 - 1s - loss: 1.4729 - accuracy: 0.9885
0.9884999990463257
(cnn) python cnn.py cifar-10 --option train
```

Cifar-10

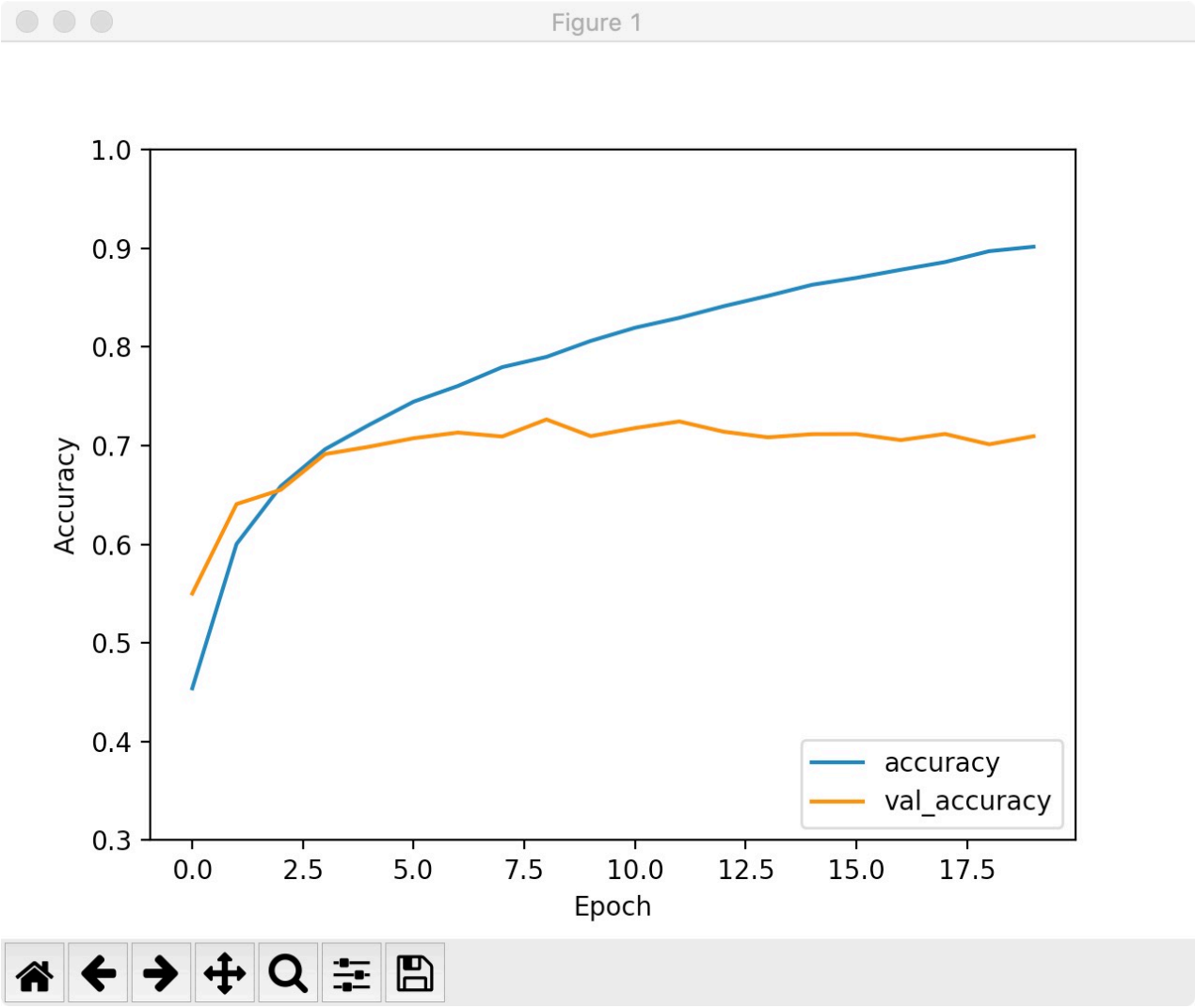
- 10epoches训练结果：



- 10epoches识别结果：0.7105

```
2021-01-06 17:07:29.193149: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
Epoch 1/10
1563/1563 [==============================] - 29s 19ms/step - loss: 1.5120 - accuracy: 0.4484 - val_loss: 1.2272 - val_accuracy: 0.5586
Epoch 2/10
1563/1563 [==============================] - 30s 19ms/step - loss: 1.1496 - accuracy: 0.5931 - val_loss: 1.0504 - val_accuracy: 0.6293
Epoch 3/10
1563/1563 [==============================] - 30s 19ms/step - loss: 1.0020 - accuracy: 0.6487 - val_loss: 0.9622 - val_accuracy: 0.6624
Epoch 4/10
1563/1563 [==============================] - 30s 19ms/step - loss: 0.8912 - accuracy: 0.6872 - val_loss: 0.9006 - val_accuracy: 0.6839
Epoch 5/10
1563/1563 [==============================] - 28s 18ms/step - loss: 0.8205 - accuracy: 0.7124 - val_loss: 0.8914 - val_accuracy: 0.6898
Epoch 6/10
1563/1563 [==============================] - 29s 19ms/step - loss: 0.7597 - accuracy: 0.7326 - val_loss: 0.8462 - val_accuracy: 0.7075
Epoch 7/10
1563/1563 [==============================] - 30s 19ms/step - loss: 0.7088 - accuracy: 0.7488 - val_loss: 0.8776 - val_accuracy: 0.6966
Epoch 8/10
1563/1563 [==============================] - 30s 19ms/step - loss: 0.6644 - accuracy: 0.7660 - val_loss: 0.8901 - val_accuracy: 0.6992
Epoch 9/10
1563/1563 [==============================] - 29s 18ms/step - loss: 0.6213 - accuracy: 0.7795 - val_loss: 0.8608 - val_accuracy: 0.7186
Epoch 10/10
1563/1563 [==============================] - 30s 19ms/step - loss: 0.5826 - accuracy: 0.7939 - val_loss: 0.9000 - val_accuracy: 0.7105
313/313 - 1s - loss: 0.9000 - accuracy: 0.7105
0.7105000019073486
```

- 20epoches训练结果：



- 20epoches识别结果：0.7095

```
➜  cnn python cnn.py cifar-10 --option train
2021-01-07 21:48:44.064942: I tensorflow/core/platform/cpu_feature_guard.cc:143] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
2021-01-07 21:48:44.080962: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x7fa7787f0480 initialized for platform Host (this does not guarantee that XLA will be used). Devices:
2021-01-07 21:48:44.081008: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
Epoch 1/20
1563/1563 [==============================] - 28s 18ms/step - loss: 1.4982 - accuracy: 0.4539 - val_loss: 1.2746 - val_accuracy: 0.5500
Epoch 2/20
1563/1563 [==============================] - 30s 19ms/step - loss: 1.1326 - accuracy: 0.6003 - val_loss: 1.0321 - val_accuracy: 0.6407
Epoch 3/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.9718 - accuracy: 0.6588 - val_loss: 0.9851 - val_accuracy: 0.6552
Epoch 4/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.8711 - accuracy: 0.6963 - val_loss: 0.8739 - val_accuracy: 0.6914
Epoch 5/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.7935 - accuracy: 0.7212 - val_loss: 0.8558 - val_accuracy: 0.6989
Epoch 6/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.7331 - accuracy: 0.7446 - val_loss: 0.8511 - val_accuracy: 0.7075
Epoch 7/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.6840 - accuracy: 0.7605 - val_loss: 0.8394 - val_accuracy: 0.7132
Epoch 8/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.6333 - accuracy: 0.7795 - val_loss: 0.8677 - val_accuracy: 0.7092
Epoch 9/20
1563/1563 [==============================] - 29s 19ms/step - loss: 0.5956 - accuracy: 0.7899 - val_loss: 0.8244 - val_accuracy: 0.7265
Epoch 10/20
1563/1563 [==============================] - 29s 18ms/step - loss: 0.5515 - accuracy: 0.8061 - val_loss: 0.8743 - val_accuracy: 0.7096
Epoch 11/20
1563/1563 [==============================] - 26s 16ms/step - loss: 0.5144 - accuracy: 0.8195 - val_loss: 0.8837 - val_accuracy: 0.7178
Epoch 12/20
1563/1563 [==============================] - 27s 17ms/step - loss: 0.4790 - accuracy: 0.8295 - val_loss: 0.8924 - val_accuracy: 0.7245
Epoch 13/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.4468 - accuracy: 0.8413 - val_loss: 0.9427 - val_accuracy: 0.7141
Epoch 14/20
1563/1563 [==============================] - 29s 18ms/step - loss: 0.4187 - accuracy: 0.8518 - val_loss: 1.0013 - val_accuracy: 0.7084
Epoch 15/20
1563/1563 [==============================] - 28s 18ms/step - loss: 0.3855 - accuracy: 0.8630 - val_loss: 1.0178 - val_accuracy: 0.7116
Epoch 16/20
1563/1563 [==============================] - 29s 18ms/step - loss: 0.3622 - accuracy: 0.8701 - val_loss: 1.0380 - val_accuracy: 0.7117
Epoch 17/20
1563/1563 [==============================] - 29s 18ms/step - loss: 0.3342 - accuracy: 0.8782 - val_loss: 1.1559 - val_accuracy: 0.7056
Epoch 18/20
1563/1563 [==============================] - 29s 18ms/step - loss: 0.3179 - accuracy: 0.8861 - val_loss: 1.1379 - val_accuracy: 0.7118
Epoch 19/20
1563/1563 [==============================] - 29s 18ms/step - loss: 0.2898 - accuracy: 0.8971 - val_loss: 1.2466 - val_accuracy: 0.7013
Epoch 20/20
1563/1563 [==============================] - 26s 17ms/step - loss: 0.2726 - accuracy: 0.9016 - val_loss: 1.2275 - val_accuracy: 0.7095
313/313 - 1s - loss: 1.2275 - accuracy: 0.7095
0.7095000147819519
```

4.编程体会

这次实验的要求的是实现基本的CNN并通过Mnist和Cifar-10数据进行训练。通过tensorflow提供的api我可以很方便的构建起CNN，设置损失函数和优化器，并通过数据集进行训练拟合和验证。最后在LeNet模型上能够达到98%的精度，可以看出模型的性能是相当不错的，较好的完成了手写数字识别的任务。并在简单改进后，对更复杂的Cifar-10的数据集也能达到70%。

大概率是我目前知识有限，这次实验让我开始怀疑机器学习。我们不断更改网络的层数和每一层的参数，更换激活函数，更换优化函数和损失函数，并以大批数据的训练，以期待能得到更好的结果。我们惊讶于计算机能够轻松完成各种“智能的任务。我们知道what，这是一大堆参数对特定数据的拟合，拟合效果越好意味着它越“智能”。我们也明白训练过程的不可预知性：在结果出来之前，我们不知道这一次是会更好还是更坏，好是好多少。我们仍然无法解释why。我们没办法接受所谓的智能只不过是一堆参数。我们期待“人工智能能够“懂”什么是手写字体，而不是仅仅告诉我们“这是数字几”。

5.个人照片