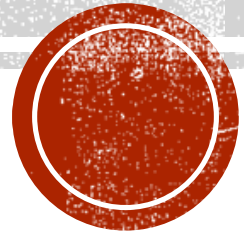
人工智能实践：TENSORFLOW2（二）



神经网络优化

- 学习率
- 激活函数
- 损失函数
- 欠拟合和过拟合
- 优化器

学习率

$$W_{t+1} = W_t - lr * \frac{\partial loss}{\partial W_t}$$

学习率

指数衰减学习率：

可以先用较大的学习率，快速得到较优解，然后逐步减小学习率，使模型在训练后期稳定。公式为

指数衰减学习率=初始学习率*学习率衰减率（当前轮数 / 多少轮衰减一次）

[1]:
```
import tensorflow as tf
import numpy as np
```

[2]:
```
w = tf.Variable(tf.constant(5, dtype=tf.float32))
```

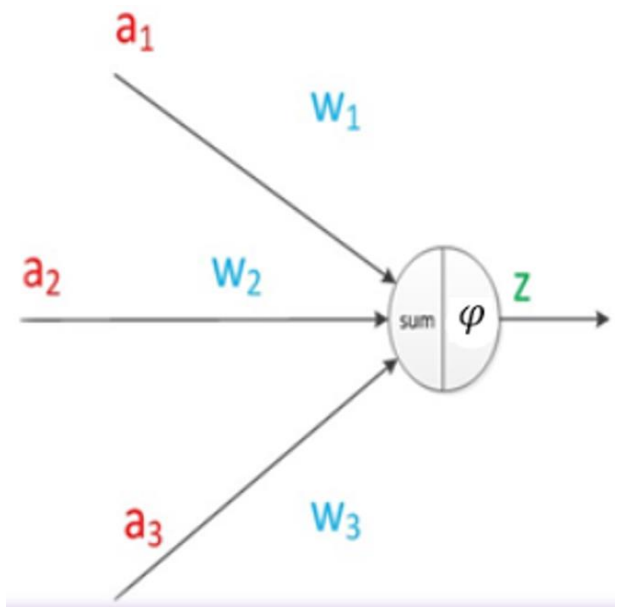
[3]:
```
epoch = 40
LR_BASE = 0.2  # 最初学习率
LR_DECAY = 0.99  # 学习率衰减率
LR_STEP = 1  # 喂入多少轮BATCH_SIZE后，更新一次学习率
```

[4]:
```
for epoch in range(epoch):  # for epoch 定义顶层循环，表示对数据集循环epoch次
    lr = LR_BASE * LR_DECAY ** (epoch / LR_STEP)
    with tf.GradientTape() as tape:  # with结构到grads框起了梯度的计算过程。
        loss = tf.square(w + 1)
    grads = tape.gradient(loss, w)  # .gradient函数告知谁对谁求导

    w.assign_sub(lr * grads)  # .assign_sub 对变量做自减 即：w -= lr*grads 即 w = w - lr*grads
    print("After %s epoch,w is %f,loss is %f,lr is %f" % (epoch, w.numpy(), loss, lr))
```

```
After 0 epoch,w is 2.600000,loss is 36.000000,lr is 0.200000
After 1 epoch,w is 1.174400,loss is 12.959999,lr is 0.198000
After 26 epoch,w is -0.999952,loss is 0.000000,lr is 0.154009
After 27 epoch,w is -0.999967,loss is 0.000000,lr is 0.152469
After 28 epoch,w is -0.999977,loss is 0.000000,lr is 0.150944
After 29 epoch,w is -0.999984,loss is 0.000000,lr is 0.149434
After 30 epoch,w is -0.999989,loss is 0.000000,lr is 0.147940
After 31 epoch,w is -0.999992,loss is 0.000000,lr is 0.146461
After 32 epoch,w is -0.999994,loss is 0.000000,lr is 0.144996
After 33 epoch,w is -0.999996,loss is 0.000000,lr is 0.143546
After 34 epoch,w is -0.999997,loss is 0.000000,lr is 0.142111
After 35 epoch,w is -0.999998,loss is 0.000000,lr is 0.140690
After 36 epoch,w is -0.999999,loss is 0.000000,lr is 0.139283
After 37 epoch,w is -0.999999,loss is 0.000000,lr is 0.137890
After 38 epoch,w is -0.999999,loss is 0.000000,lr is 0.136511
After 39 epoch,w is -0.999999,loss is 0.000000,lr is 0.135146
```

激活函数



优秀的激活函数：

- 非线性：激活函数非线性时，多层神经网络可逼近所有函数
- 可微性：优化器大多用梯度下降更新参数
- 单调性：当激活函数是单调的，能保证单层网络的损失函数是凸函数
- 近似恒等性：f(x)≈x当参数初始化为随机小值时，神经网络更稳定

$$Z=\varphi(a_1w_1+a_2w_2+a_3w_3+b)$$

φ称为激活函数或传递函数

激活函数输出值的范围：

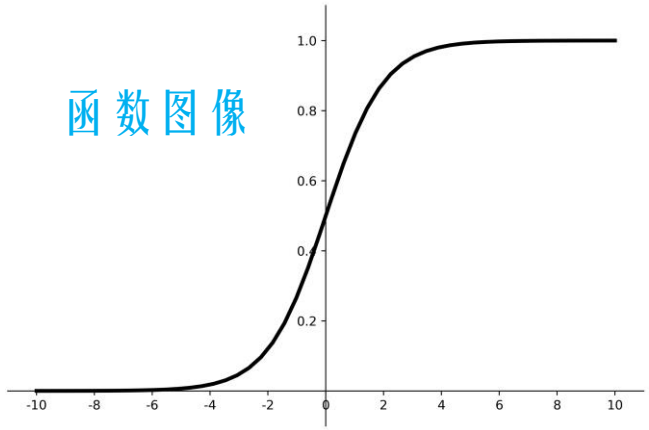
- 激活函数输出为有限值时，基于梯度的优化方法更稳定
- 激活函数输出为无限值时，建议调小学习率

Sigmoid函数

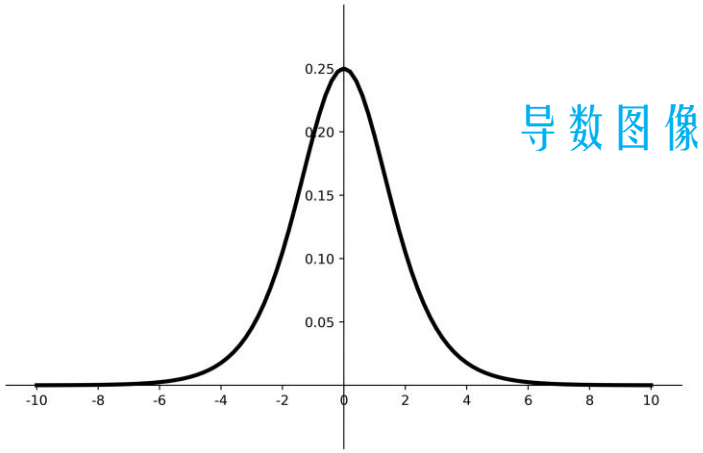
$$\varphi(x) = \frac{1}{1 + e^{-x}}$$

`tf.nn.sigmoid(x)`

函数图像



导数图像



可用于2分类

优点：

1. 输出映射在(0, 1)之间，单调连续，输出范围有限，优化稳定，可用作输出层；
2. 求导容易

缺点

（1）易造成梯度消失

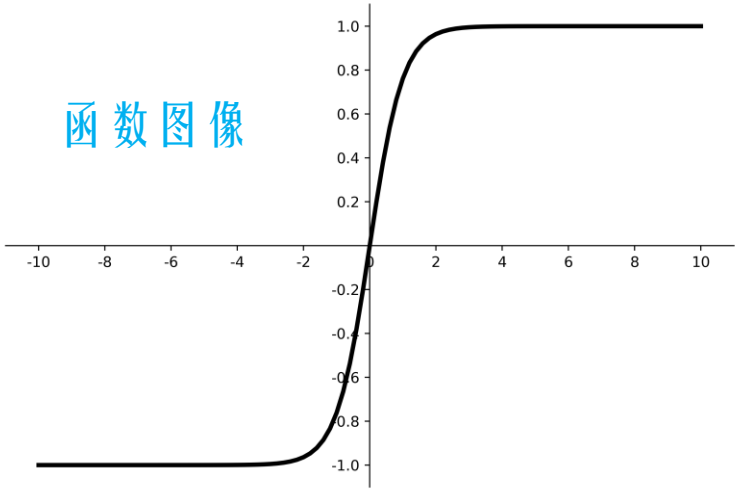
（2）输出非0均值，收敛慢

（3）幂运算复杂，训练时间长

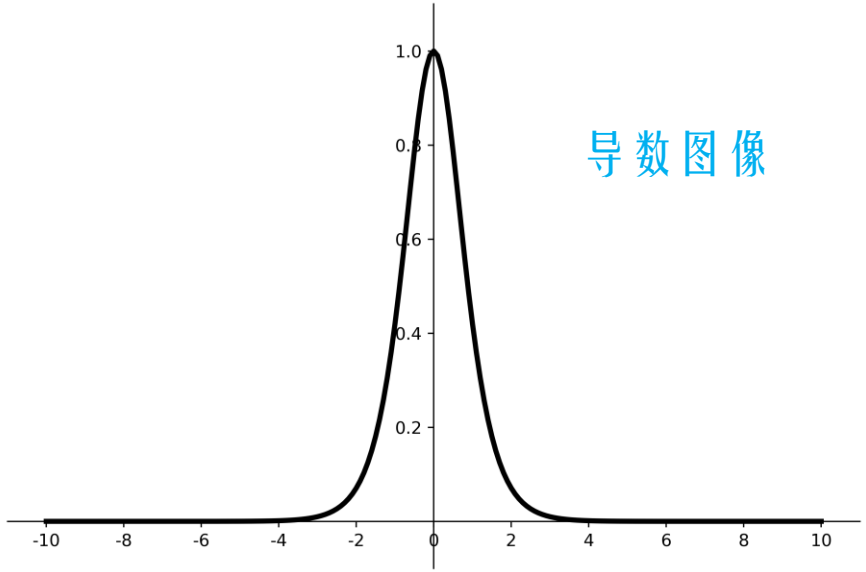
Tanh函数： $$\varphi(x)=\frac{1-e^{-2x}}{1+e^{-2x}}$$

tf.math.tanh(x)

函数图像



导数图像



优点：

1. 输出映射在(−1, 1)之间，单调连续，输出范围有限，优化稳定，可用作输出层；
2. 比sigmoid函数收敛速度快

特点

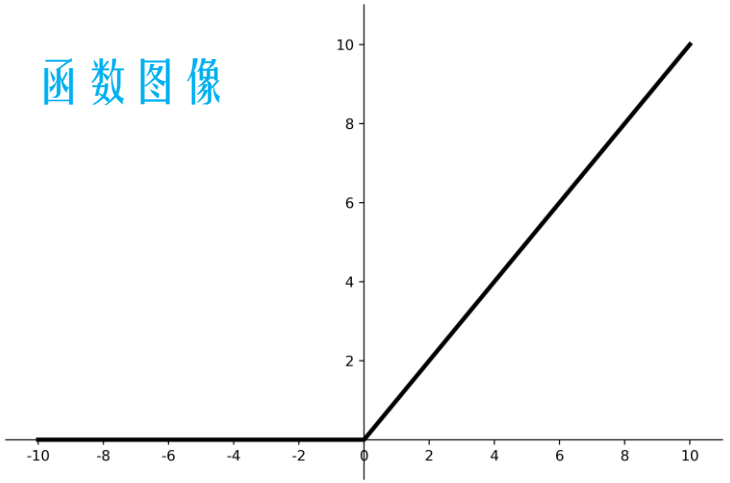
（1）易造成梯度消失

（2）幂运算复杂，训练时间长

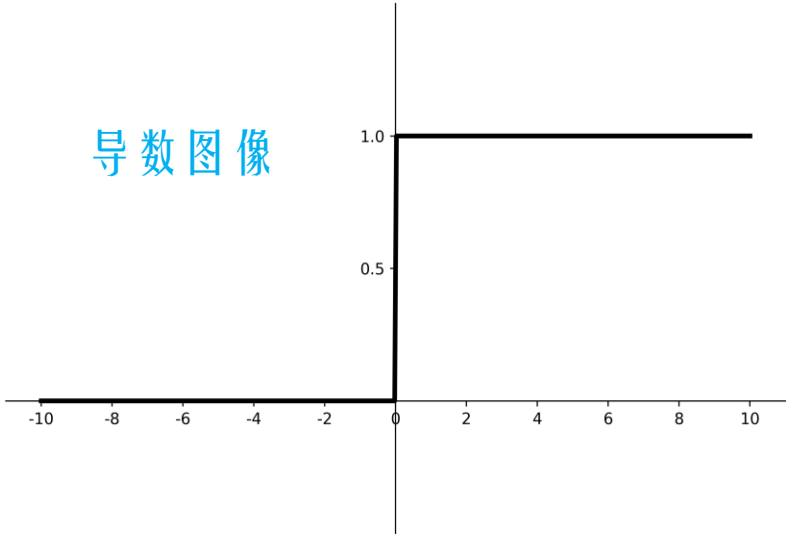
ReLU函数： $\varphi(x) = max(0, x) = \begin{cases} x, x > 0, \\ 0, x \leq 0. \end{cases}$

tf.nn.relu(x)

函数图像



导数图像



优点：

1. 解决了梯度消失问题(在正区间)；
2. 只需判断输入是否大于0，计算速度快；
3. 收敛速度远快于sigmoid和tanh，因为sigmoid和tanh涉及很多expensive的操作；
4. 提供了神经网络的稀疏表达能力

缺点

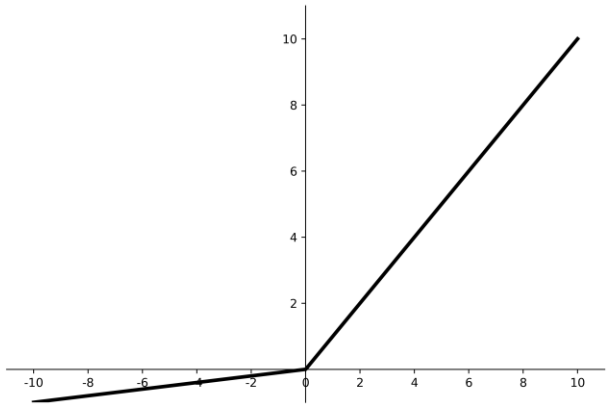
1. 输出非0均值，收敛慢；
2. Dead ReLU问题：某些神经元可能永远不会被激活，导致相应的参数永远不能被更新.

Leaky ReLU函数：

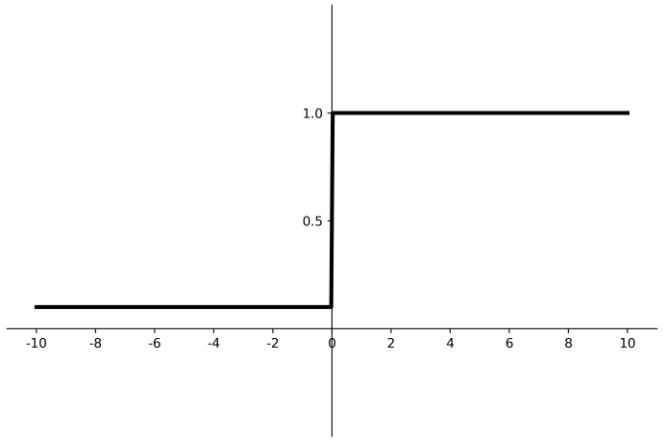
$$\varphi(x) = max(0, x) = \begin{cases} x, & x > 0, \\ \alpha x, & x \leq 0. \end{cases}$$

tf.nn.leaky_relu(x)

函数图像



导数图像



理论上来讲，Leaky ReLU有ReLU的所有优点，外加不会有Dead ReLU问题，但是在实际操作当中，并没有完全证明Leaky ReLU总是好于ReLU。

softmax函数：

$$\varphi(x_k) = \frac{e^{x_k}}{\sum_{k=1}^{K} e^{x_k}}, k = 1,2,\cdots,K$$

tf.nn.softmax(x)

对神经网络全连接层输出进行变换，使其服从概率分布，即每个值都位于[0, 1]区间且和为1.

对于初学者的建议：

- 首选relu激活函数；
- 学习率设置较小值；
- 输入特征标准化，即让输入特征满足以0为均值，1为标准差的正态分布；
- 初始参数中心化，即让随机生成的参数满足以0为均值，

$\sqrt{\frac{2}{\text{当前层输入特征个数}}}$为标准差的正态分布。

损失函数

损失函数（loss）：预测值（y）与真实值（y_）的差距. 神经网络模型的效果及优化的目标是通过损失函数来定义的。回归和分类是监督学习中的两个大类。

1. 均方误差

$$\mathrm{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y - y\_)^2$$

loss_mse = tf.reduce_mean(tf.square(y-y_))

tf.losses.mean_squared_error(y, y_)

2. 交叉熵损失函数CE（Cross Entropy）：表征两个概率分布之间的距离，交叉熵越小说明二者分布越接近，是分类问题中使用较广泛的损失函数.

$$H(y,y\_)=-\sum y\_ * lny$$

tf.losses.categorical_crossentropy(y_ , y)

例：二分类 已知答案$y_=(1,0)$ 预测$y_1=(0.6,0.4)$ $y_2=(0.8,0.2)$哪个更接近标准答案？

$H_1((1,0),(0.6,0.4))=-(1*ln0.6+0*ln0.4)\approx-(-0.511+0)=0.511$

$H_2((1,0),(0.8,0.2))=-(1*ln0.8+0*ln0.2)\approx-(-0.223+0)=0.223$

因为$H_1>H_2$，所以y2预测更准。

3. Softmax与交叉熵结合

对于多分类问题，神经网络的输出y一般不是概率分布，因此需要引入softmax层，使得输出服从概率分布，再计算它与y_的交叉熵损失函数.

tf.nn.softmax_cross_entropy_with_logits(y_, y)

```
import tensorflow as tf
import numpy as np
```

```
y_=np.array([[1,0,0],[0,1,0],[0,0,1],[1,0,0],[0,1,0]])
y=np.array([[12,3,2],[3,10,3],[2,1,5],[4,6,1.2],[3,6,1]])
y_sm=tf.nn.softmax(y)
loss_1=tf.losses.categorical_crossentropy(y_,y_sm)
loss_2=tf.nn.softmax_cross_entropy_with_logits(y_,y)
print("分步计算的结果：",loss_1)
print("和步计算的结果：",loss_2)
```

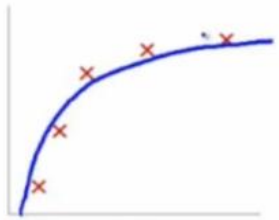
损失函数的功能主要就是让网络更加逼近真实函数，识别准确率更高，一般需要针对特定的背景、具体的任务设计损失函数.

```
分步计算的结果： tf.Tensor(
[1.68795487e-04 1.82210289e-03 6.58839038e-02 2.13415060e+00
 5.49852354e-02], shape=(5,), dtype=float64)
和步计算的结果： tf.Tensor(
[1.68795487e-04 1.82210289e-03 6.58839038e-02 2.13415060e+00
 5.49852354e-02], shape=(5,), dtype=float64)
```

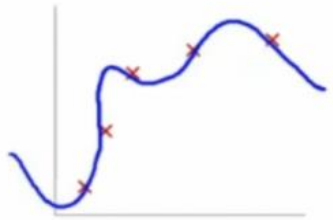
欠拟合与过拟合

回归（Regression）问题中三种拟合状态

欠拟合

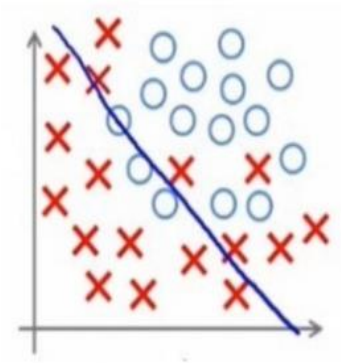


好的拟合

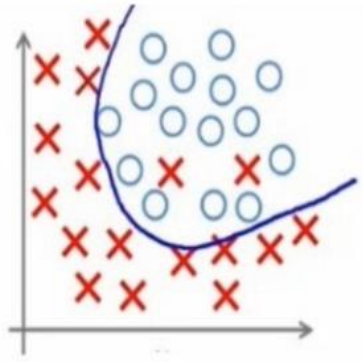


过拟合

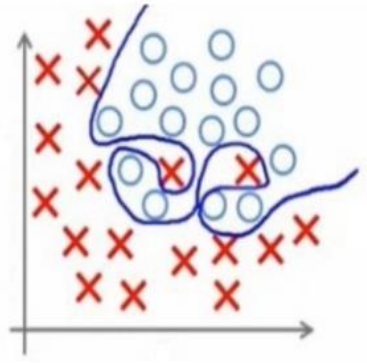
分类（Classification）问题中三种拟合状态



欠拟合



好的拟合



过拟合

- 欠拟合的解决方法：
  增加输入特征项
  增加网络参数
  减少正则化参数
- 过拟合的解决方法：
  数据清洗
  增大训练集
  采用正则化
  增大正则化参数

  Dropout（在训练过程中随机删除一些隐藏层的神经元，同时保证输入层和输出层的神经元不变。）

正则化缓解过拟合

正则化是在损失函数中引入模型复杂度指标，利用给W加权值，弱化了训练数据的噪声（一般不正则化b）

$$loss = loss(y, y\_) + \lambda * J(\mathbf{W})$$

$$J(\mathbf{W}) = \|W\|_{l_1}$$

$$J(\mathbf{W}) = \|W\|_{l_2}$$

模型中所有参数的损失函数如：交叉熵、均方误差

用超参数λ给出参数w在总loss中的比例，即正则化的权重

需要正则化的参数

正则化的选择：

- L1正则化大概率会使很多参数变为零，因此该方法可通过稀疏参数，即减少参数的数量，降低复杂度。
- L2正则化会使参数很接近零但不为零，因此该方法可通过减小参数值的大小降低复杂度。

神经网络优化器

优化算法可以分成一阶优化和二阶优化算法，其中一阶优化就是指的梯度算法及其变种，而二阶优化一般是用二阶导数（Hessian 矩阵）来计算，如牛顿法，由于需要计算Hessian阵和其逆矩阵，计算量较大，因此没有流行开来。这里主要总结一阶优化的各种梯度下降方法。

深度学习优化算法经历了SGD -> SGDM -> NAG ->AdaGrad -> AdaDelta -> Adam -> Nadam这样的发展历程。

待优化参数W，损失函数$loss$，初始学习率lr. 每次迭代一个batch，t表示当前batch迭代的总次数.

1. 计算损失函数关于当前参数的梯度：$g_t = \nabla loss$
2. 根据历史梯度计算**一阶动量**m_t和**二阶动量**V_t：

$$m_t = \phi(g_1, g_2, \cdots, g_t), V_t = \varphi(g_1, g_2, \cdots, g_t),$$

3. 计算t时刻的下降梯度：$\eta_t = lr * \frac{m_t}{\sqrt{V_t}}$
4. 计算t+1时刻参数：$W_{t+1} = W_{t+1} - \eta_t$

一阶动量：与梯度相关的函数
二阶动量：与梯度平方相关的函数

1. SGD (Stochastic Gradient Descent)（无momentum），常用的梯度下降法

$$m_t = g_t, \quad V_t = 1, \qquad \eta_t = lr * g_t, \qquad W_{t+1} = W_{t+1} - lr * g_t$$

缺点是下降速度慢，而且可能会在沟壑的两边持续震荡，停留在一个局部最优点。

2. SGDM （带momentum的SGD）

为了抑制SGD的震荡，SGDM认为梯度下降过程可以加入惯性。下坡的时候，如果发现是陡坡，那就利用惯性跑的快一些。SGDM在SGD基础上引入了一阶动量：

$$m_t = \beta \cdot m_{t-1} + (1-\beta) \cdot g_t, \quad V_t = 1,$$

$$\eta_t = lr * m_t, \qquad W_{t+1} = W_{t+1} - lr * m_t$$

t时刻的下降方向，不仅由当前点的梯度方向决定，而且由此前累积的下降方向决定。β的经验值为0.9，这就意味着下降方向主要偏向此前累积的下降方向，并略微偏向当前时刻的下降方向。

3. AdaGrad （在SGD基础上增加二阶动量）

AdaGrad算法引入二阶动量给学习率一个缩放比例，从而达到了自适应学习率的效果（Ada = Adaptive）。其思想是：对于频繁更新的参数，不希望被单个样本影响太大，给它们很小的学习率；对于偶尔出现的参数，希望能多得到一些信息，给它较大的学习率。

$$m_t = g_t, \quad V_t = \sum_{\tau=1}^{t} g_\tau^2,$$

$$\eta_t = lr * g_t \Big/ \sqrt{\sum_{\tau=1}^{t} g_\tau^2},$$

$$W_{t+1} = W_{t+1} - lr * g_t \Big/ \sqrt{\sum_{\tau=1}^{t} g_\tau^2}$$

AdaGrad 在稀疏数据场景下表现最好。因为对于频繁出现的参数，学习率衰减得快；对于稀疏的参数，学习率衰减得更慢。然而在实际很多情况下，二阶动量呈单调递增，累计从训练开始的梯度，学习率会很快减至0，导致参数不再更新，训练过程提前结束.

4. RMSProp （Root Mean Square Prop）

为了解决 Adagrad 学习率急剧下降问题引入RMSProp，思想是使用指数加权平均，旨在消除梯度下降中的摆动，与Momentum的效果一样，某一维度的导数比较大，则指数加权平均就大，某一维度的导数比较小，则其指数加权平均就小，这样就保证了各维度导数都在一个量级，进而减少了摆动。允许使用一个更大的学习率η）

$$m_t = g_t, \quad V_t = \beta V_{t-1} + (1-\beta) g_\tau^2,$$

$$\eta_t = lr * g_t / \sqrt{V_t},$$

$$W_{t+1} = W_{t+1} - lr * g_t / \sqrt{V_t}$$

5. Adam （adaptive moment estimation）

SGDM在SGD基础上增加了一阶动量，AdaGrad和RMSProp在SGD基础上增加了二阶动量。把一阶动量和二阶动量结合起来，再修正偏差，就是Adam了。它也是计算每个参数的自适应学习率的方法，**相当于 RMSprop + Momentum**。

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$$

修正一阶动量的偏差：$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$

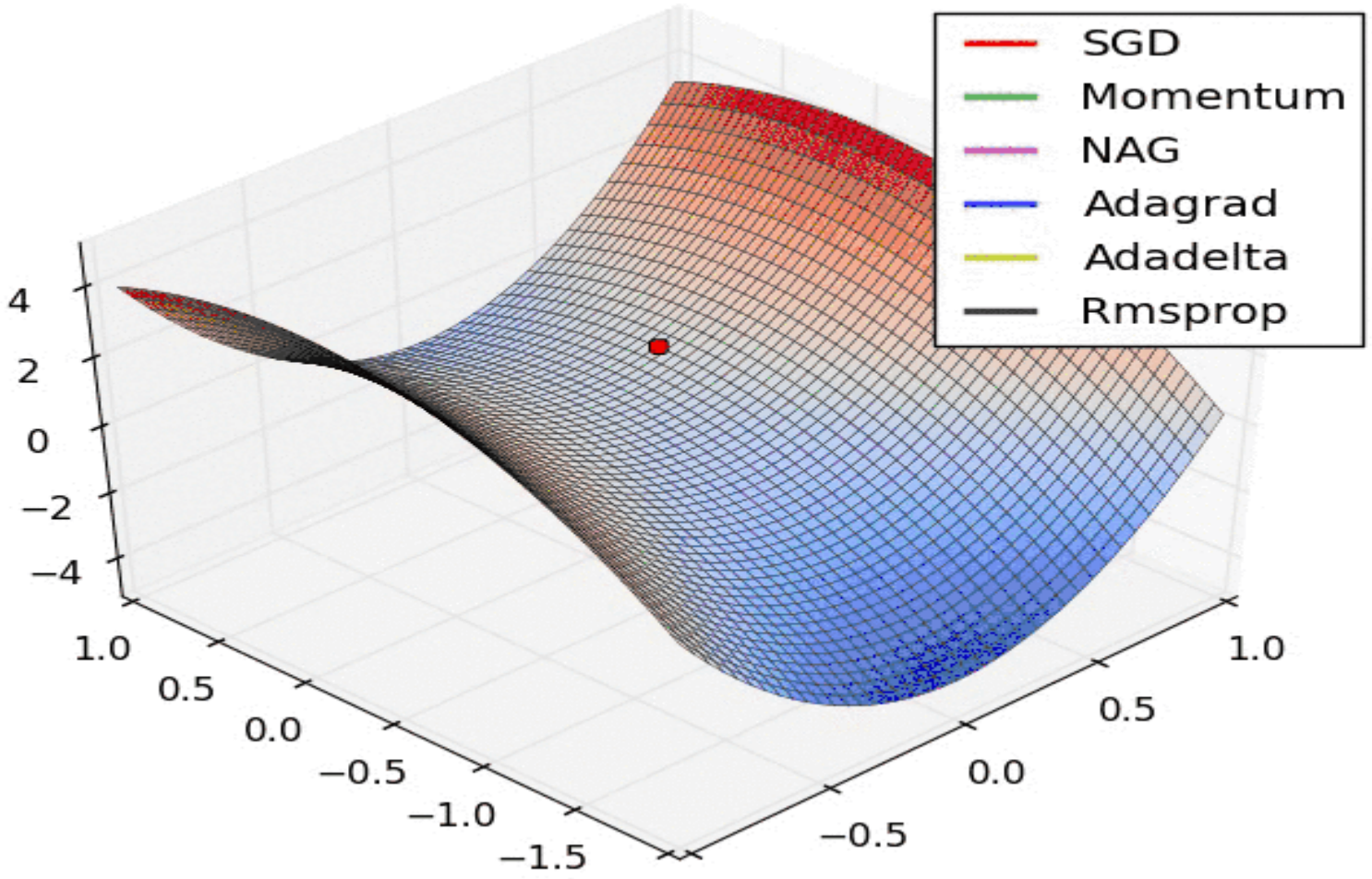
$$V_t = \beta_2 \cdot V_{t-1} + (1-\beta_2) \cdot g_t^2,$$

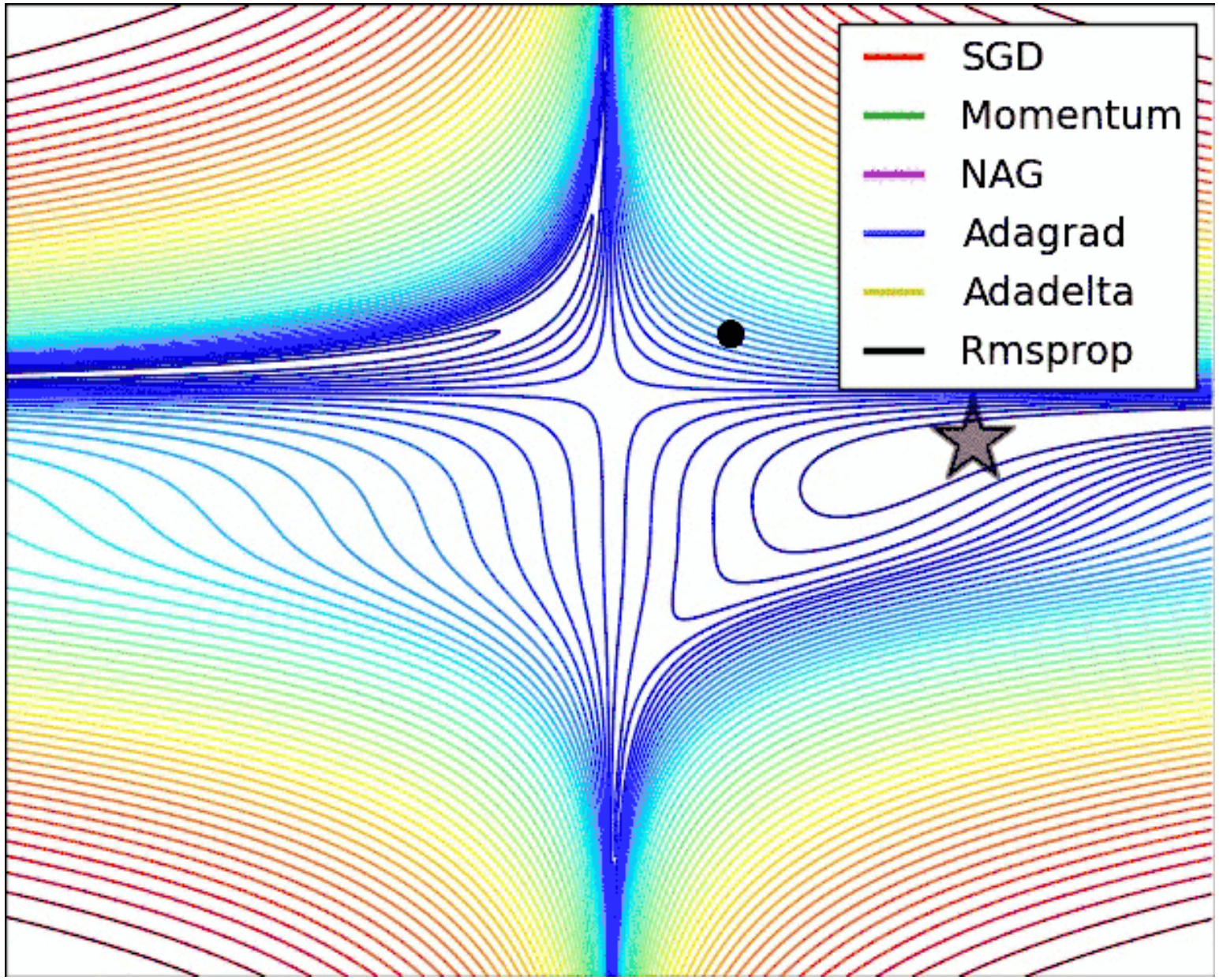
修正一阶动量的偏差：$\widehat{V}_t = \frac{V_t}{1-\beta_2^t}$

$$\eta_t = lr * \widehat{m}_t / \sqrt{\widehat{V}_t},$$

$$W_{t+1} = W_{t+1} - lr * \widehat{m}_t / \sqrt{\widehat{V}_t}$$

Adagrad, Adadelta, RMSprop 几乎很快就找到了正确的方向并前进，收敛速度也相当快，而其它方法要么很慢，要么走了很多弯路才找到。即：自适应学习率方法即 Adagrad, Adadelta, RMSprop, Adam 收敛性更好。




SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop

使用八股搭建神经网络

- 神经网络搭建八股
- iris代码复现
- MNIST数据集
- 训练MNIST数据集
- Fashion数据集

用Tensorflow API： tf.keras搭建网络八股

1. **输入相关模块：Import**
2. 输入网络的训练集和测试集： **train, test**
3. 逐层搭建网络结构：**model =tf.keras.models.Sequential**
4. 配置训练方法： **model.compile**
5. 执行训练过程：**model.fit**
6. 打印网络结构，统计参数数目： **model.summary**

API(Application Programming Interface,应用程序编程接口)

- Tensorflow是比较底层的深度学习模型开发语言，Keras是基于tensorflow的高级API，通常直接调用一些封装好的函数就可以实现某些功能，而tensorflow虽然比较底层，但是可以比较灵活地定义模型结构.
- keras的关键计算依托于tensorflow或者theano. theano不更新了，keras封装的比较好。初学和入门的话建议用keras。但是想要深入或者做自己的APP的话建议用tensorflow。
- tensorflow好比是木头，Keras好比是拿tensorflow做好的木板。如果你盖的房子简单，形状大众，Keras调用起来会很方便。但如果想设计特殊的房子，那就要从木料开始。

Keras 官方文档:
https://tensorflow.google.cn/api_docs/python/tf

model= tf.keras.models.Sequential([网络结构]) #
Sequential 函数是一个容器， 描述了神经网络的网络结构，在Sequential函数的输入参数中描述从输入层到输出层的网络结构

网络结构举例：1.拉直层： **tf.keras.layers.Flatten( )**

2.全连接层：**tf.keras.layers.Dense(神经元个数,activation="激活函数“,**
kernel_regularizer=哪种正则化)

activation（字符串给出）可选: relu、 softmax、 sigmoid 、 tanh

kernel_regularizer可选: tf.keras.regularizers.l1()、 tf.keras.regularizers.l2()

3.卷积层： **tf.keras.layers.Conv2D(filters = 卷积核个数, kernel_size = 卷积核尺寸,** strides = 卷积步长， padding = " valid" or "same")

4. **LSTM层： tf.keras.layers.LSTM()**

model.compile(optimizer = 优化器，loss = 损失函数， metrics = [“准确率”]) # Compile用于配置神经网络的训练方法，告知训练时使用的优化器、损失函数和准确率评测标准.

Optimizer 可以是字符串形式给出的优化器名字，也可以是函数形式，使用函数形式可以设置学习率、动量和超参数。可选：

‘sgd’ or tf.keras.optimizers.SGD (lr=学习率,momentum=动量参数)

‘adagrad’ or tf.keras.optimizers.Adagrad (lr=学习率)

‘adadelta’ or tf.keras.optimizers.Adadelta (lr=学习率)

‘adam’ or tf.keras.optimizers.Adam (lr=学习率, beta_1=0.9, beta_2=0.999)

Loss可以是字符串形式给出的损失函数的名字，也可以是函数形式。
可选项包括：

‘mse’ or tf.keras.losses.MeanSquaredError()

‘sparse_categorical_crossentropy’ or
tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)

#损失函数常需要经过 softmax 等函数将输出转化为概率分布的形式。from_logits 则用来标注该损失函数是否需要转换为概率的形式， 取 False 时表示转化为概率分布，取 True 时表示没有转化为概率分布，直接输出。

Metrics 标注网络评测指标。可选项包括：

‘accuracy’：y_和y都是数值，如y_=[1] y=[1]

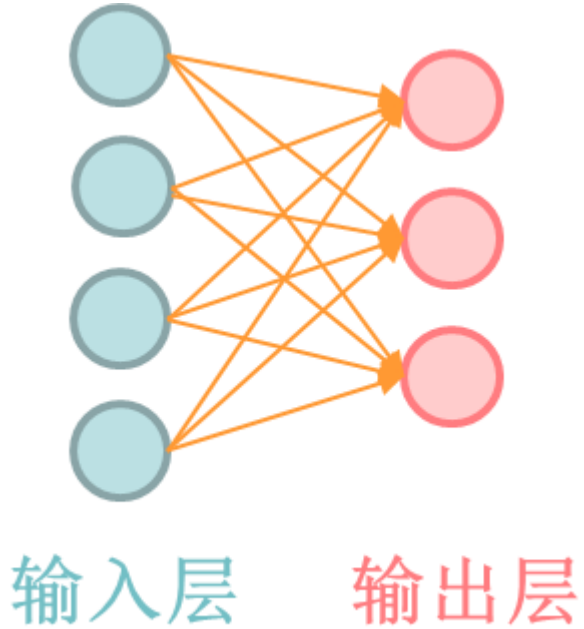
‘categorical_accuracy’：y_和y都是以独热码和概率分布表示。如 y_=[0, 1, 0], y=[0.256, 0.695, 0.048]

‘sparse_ categorical_accuracy’：y_是以数值形式给出， y是以独热码形式给出。如 y_=[1],y=[0.256, 0.695, 0.048]。

model.fit (训练集的输入特征, 训练集的标签, batch_size= , epochs= , validation_data=(测试集的输入特征, 测试集的标签),validation_split=从训练集划分多少比例给测试集，validation_freq = 多少次epoch测试一次) # `fit` 函数用于执行训练过程。

model.summary（）#summary 函数用于打印网络结构和参数统计

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                multiple                  15
=================================================================
Total params: 15
Trainable params: 15
Non-trainable params: 0
_________________________________________________________________
```



上图是 model.summary()对鸢尾花分类网络的网络结构和参数统计，对于一个输入为4输出为3的全连接网络，共有15个参数

iris 数据集代码复现

Jupyter notebook代码

[1]:
```
import tensorflow as tf
from sklearn import datasets
import numpy as np
```

[2]:
```
x_train = datasets.load_iris().data
y_train = datasets.load_iris().target
```

[3]:
```
np.random.seed(116)
np.random.shuffle(x_train)
np.random.seed(116)
np.random.shuffle(y_train)
```

[4]:
```
model = tf.keras.models.Sequential(
        [tf.keras.layers.Dense(3, activation='softmax',
        kernel_regularizer=tf.keras.regularizers.l2())])
```

```
[6]: model.fit(x_train, y_train, batch_size=32, epochs=500, validation_split=0.2, validation_freq=20)
```

```
Epoch 65/500
120/120 [==============================] - 0s 42us/sample - loss: 0.3681 - sparse_categorical_accuracy: 0.9500
Epoch 66/500
120/120 [==============================] - 0s 50us/sample - loss: 0.3688 - sparse_categorical_accuracy: 0.9417
Epoch 67/500
120/120 [==============================] - 0s 42us/sample - loss: 0.4027 - sparse_categorical_accuracy: 0.9333
Epoch 68/500
120/120 [==============================] - 0s 50us/sample - loss: 0.4419 - sparse_categorical_accuracy: 0.8833
Epoch 69/500
120/120 [==============================] - 0s 42us/sample - loss: 0.5956 - sparse_categorical_accuracy: 0.7083
Epoch 70/500
120/120 [==============================] - 0s 42us/sample - loss: 0.4194 - sparse_categorical_accuracy: 0.9167
Epoch 71/500
120/120 [==============================] - 0s 50us/sample - loss: 0.3993 - sparse_categorical_accuracy: 0.9083
Epoch 72/500
120/120 [==============================] - 0s 33us/sample - loss: 0.3899 - sparse_categorical_accuracy: 0.9083
Epoch 73/500
120/120 [==============================] - 0s 50us/sample - loss: 0.4301 - sparse_categorical_accuracy: 0.8250
Epoch 74/500
```

[7]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                multiple                  15
=================================================================
Total params: 15
Trainable params: 15
Non-trainable params: 0
_________________________________________________________________
```

VScode代码

```
import tensorflow as tf
from sklearn import datasets
import numpy as np

x_train = datasets.load_iris().data
y_train = datasets.load_iris().target

np.random.seed(116)
np.random.shuffle(x_train)
np.random.seed(116)
np.random.shuffle(y_train)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(3, activation='softmax', kernel_regularizer=tf.keras.regularizers.l2())
])

model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.1),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
              metrics=['sparse_categorical_accuracy'])

model.fit(x_train, y_train, batch_size=32, epochs=500, validation_split=0.2, validation_freq=20)

model.summary()
```

用Tensorflow API： tf.keras搭建网络八股

1. **输入相关模块：Import**
2. 输入网络的训练集和测试集： **train, test**
3. 逐层搭建网络结构：**class MyModel(Model) model=MyModel**
4. 配置训练方法： **model.compile**
5. 执行训练过程：**model.fit**
6. 打印网络结构，统计参数数目： **model.summary**

使用Sequential可以快速搭建网络结构，但是如果网络包含跳连等其他复杂网络结构，Sequential就无法表示了。这就需要使用class来声明网络结构。

```
class MyModel(Model):
    def __init__(self):
     super(MyModel, self).__init__()
        定义网络结构快
    def call(self,x):
          调用网络结构块，实现前向传播
      return y
Model=MyModel()
```

```
class IrisModel(Model):
    def __init__(self):
        super(IrisModel, self).__init__()
        self.d1 = Dense(3)

    def call(self, x):
        y = self.d1(x)
        return y

model = IrisModel()
```

__init__( )定义所需网络结构块
call( )写出前向传播

VScode代码 iris

```
1   import tensorflow as tf
2   from tensorflow.keras.layers import Dense
3   from tensorflow.keras import Model
4   from sklearn import datasets
5   import numpy as np
6
7   x_train = datasets.load_iris().data
8   y_train = datasets.load_iris().target
9
10  np.random.seed(116)
11  np.random.shuffle(x_train)
12  np.random.seed(116)
13  np.random.shuffle(y_train)
```

```
14
15  class IrisModel(Model):
16      def __init__(self):
17          super(IrisModel, self).__init__()
18          self.d1 = Dense(3, activation='softmax', kernel_regularizer=tf.keras.regularizers.l2())
19
20      def call(self, x):
21          y = self.d1(x)
22          return y
23
24  model = IrisModel()
25
26  model.compile(optimizer=tf.keras.optimizers.SGD(lr=0.1),
27                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
28                metrics=['sparse_categorical_accuracy'])
29
30  model.fit(x_train, y_train, batch_size=32, epochs=500, validation_split=0.2, validation_freq=20)
31  model.summary()
```

MNIST数据集：

提供6万张28*28 像素点的0~9手写数字图片和标签，用于训练。

提供1万张28*28 像素点的0~9手写数字图片和标签，用于测试。

- 导入MNIST数据集：

  mnist = tf.keras.datasets.mnist

  (x_train, y_train) , (x_test, y_test) = mnist.load_data()

- 作为输入特征，输入神经网络时，将数据拉伸为一维数组：

  tf.keras.layers.Flatten( )

  [0 0 0 48 238 252 252 …… …… …… 253 186 12 0 0 0 0 0]

```
plt.imshow(x_train[0],cmap='gray’) #绘制灰度图
plt.show()
```

```
print("x_train[0]:\n" , x_train[0])
```

x_train[0]：

```
print("y_train[0]:", y_train[0])
```

y_train[0]： 5

```
print("x_test.shape:", x_test.shape)
```

x_test.shape： (10000, 28, 28)

VScode代码 mnist

```
1  import tensorflow as tf
2
3  mnist = tf.keras.datasets.mnist
4  (x_train, y_train), (x_test, y_test) = mnist.load_data()
5  x_train, x_test = x_train / 255.0, x_test / 255.0
6
7  model = tf.keras.models.Sequential([
8      tf.keras.layers.Flatten(),
9      tf.keras.layers.Dense(128, activation='relu'),
10     tf.keras.layers.Dense(10, activation='softmax')
11 ])
12
13 model.compile(optimizer='adam',
14               loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
15               metrics=['sparse_categorical_accuracy'])
16
17 model.fit(x_train, y_train, batch_size=32, epochs=5, validation_data=(x_test, y_test), validation_freq=1)
18 model.summary()
19
```

VScode代码 mnist class

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras import Model

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

class MnistModel(Model):
    def __init__(self):
        super(MnistModel, self).__init__()
        self.flatten = Flatten()
        self.d1 = Dense(128, activation='relu')
        self.d2 = Dense(10, activation='softmax')

    def call(self, x):
        x = self.flatten(x)
        x = self.d1(x)
        y = self.d2(x)
        return y

model = MnistModel()

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
              metrics=['sparse_categorical_accuracy'])

model.fit(x_train, y_train, batch_size=32, epochs=5, validation_data=(x_test, y_test), validation_freq=1)
model.summary()
```

上机练习（3）

FASHION数据集：

提供 6万张 28*28 像素点的衣裤等图片和标签，用于训练。

提供 1万张 28*28 像素点的衣裤等图片和标签，用于测试。

Label	Description
0	T恤（T-shirt/top）
1	裤子（Trouser）
2	套头衫（Pullover）
3	连衣裙（Dress）
4	外套（Coat）
5	凉鞋（Sandal）
6	衬衫（Shirt）
7	运动鞋（Sneaker）
8	包（Bag）
9	靴子（Ankle boot）

导入FASHION数据集：

```
fashion = tf.keras.datasets.fashion_mnist
(x_train, y_train),(x_test, y_test) = fashion.load_data()
```