7 反向传播算法

7.1 梯度下降

- ✓ 导数，derivative
- ✓ 偏微分，partial derivative
- ✓ 梯度，gradient

导数和偏微分都是标量，偏微分是导数的特列。梯度则是偏微分的的向量形式。

$$\nabla\mathcal{L} = (\frac{\mathcal{L}}{\partial\theta_1}, \frac{\mathcal{L}}{\partial\theta_2}, \frac{\mathcal{L}}{\partial\theta_3}, \dots, \frac{\mathcal{L}}{\partial\theta_n})$$

梯度下降法一般是寻找函数的最小值，

$$\theta' = \theta - \eta * \nabla\mathcal{L}$$

梯度上升法则是沿着梯度的方向更新，

$$\theta' = \theta + \eta * \nabla\mathcal{L}$$

梯度是由所有的偏导数组成，表征方向，梯度的方向表示函数值上升最快的方向，梯度的反方向表示函数值下降最快的方向。梯度的模表示函数增大的速度。

AutoGrad

- With Tf.GradientTape() as tape:
  - Build computation graph
  - $loss = f_\theta(x)$
- [w_grad] = tape.gradient(loss, [w])

```
In [15]: w = tf.constant(1.)

In [16]: x = tf.constant(2.)

In [17]: with tf.GradientTape() as tape:
    ...:     tape.watch([w])
    ...:     y2 = x*w
    ...:

In [18]: grad1 = tape.gradient(y, [w])

In [19]: grad1
Out[19]: [None]

In [20]: with tf.GradientTape() as tape:
    ...:     tape.watch([w])
    ...:     y2 = x*w
    ...:

In [21]: grad2 = tape.gradient(y2, [w])

In [22]: grad2
Out[22]: [<tf.Tensor: id=17, shape=(), dtype=float32, numpy=2.0>]
```

```
In [23]: w = tf.constant(1.)

In [24]: x = tf.constant(2.)

In [25]: y = x*w

In [26]: with tf.GradientTape() as tape:
    ...:     tape.watch([w])
    ...:     y2 = x*w
    ...:

In [27]: tape.gradient(y2, [w])
Out[27]: [<tf.Tensor: id=24, shape=(), dtype=float32, numpy=2.0>]

In [28]: # tape.gradient(y2, [w])

In [29]: # RuntimeError: GradientTape.gradient can only be called once on non-persistent tapes.

In [30]: with tf.GradientTape(persistent=True) as tape:
    ...:     tape.watch([w])
    ...:
```

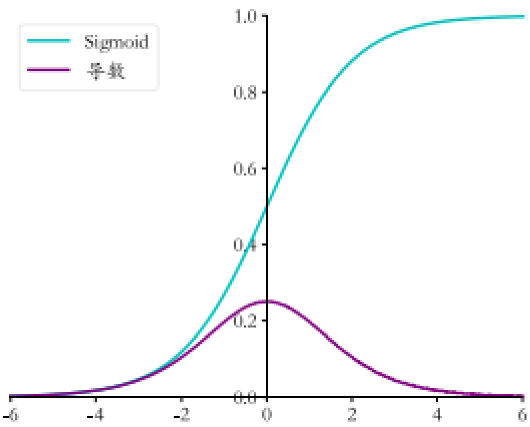
第一次求导完了之后，资源被释放了，再次求导就会报错

7.2 激活函数及其梯度

- ✓ sigmoid
- ✓ ReLU
- ✓ LeakyReLU
- ✓ Tanh

sigmoid 函数 →→ tf.sigmoid

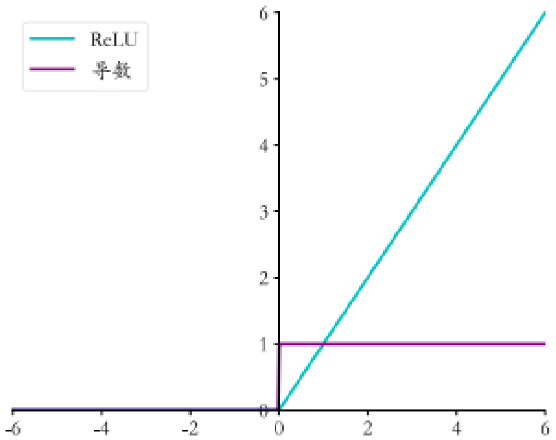
$$\sigma(x) = \frac{1}{1+e^{-x}} \longrightarrow \sigma'(x) = \sigma(x) * (1 - \sigma(x))$$




```
import numpy as np
def sigmoid(x):          # sigmoid 函数
    return 1 / (1 + np.exp(-x))

def derivative(x):       # sigmoid 函数的导数
    return sigmoid(x) * (1 - sigmoid(x))
```

ReLU 函数 →→ tf.nn.relu

$$ReLU(x) := \max(0, x) \longrightarrow \frac{d}{dx}ReLU = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

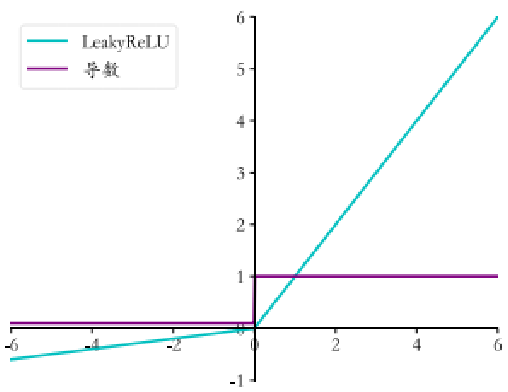



```
def derivative(x):        # ReLU 函数的导数
    d = np.array(x, copy=True)   # 用于保存梯度的张量
    d[x<0] = 0            # 元素为负的导数为 0
    d[x>0] = 1            # 元素为正的导数为 1
    return d
```

LeakyReLU 函数

$$LeakyReLU(x) = \begin{cases} x, & x \geq 0 \\ p * x, & x < 0 \end{cases} \qquad \frac{d}{dx}LeakyReLU = \begin{cases} 1, & x \geq 0 \\ p, & x < 0 \end{cases}$$

该函数与 ReLU 函数不同之处在于，当 x 小于零时，LeakyReLU 函数的导数不为 0，而是 p，一般 p 设置为一个较小的数值，如 0.01 或者 0.02 等。




```
def derivative(x,p):     # 其中 p 为 LeakyReLU 的负半段斜率
    dx = np.ones_like(x)       # 创建梯度张量
    dx[x<0] = p                # 元素为负的导数为 p
    return dx
```

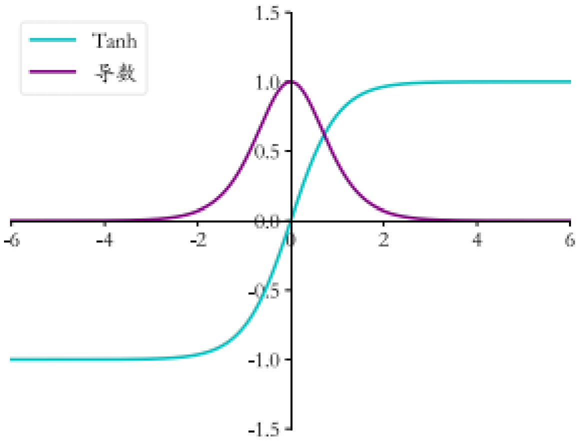
Tanh 函数 →→ **tf.nn.relu**

$$tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} = 2 * sigmoid(2x) - 1$$

对其求导：

$$\frac{d}{dx}tanh(x) = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - tanh^2(x)$$




```
def sigmoid(x):          # sigmoid 函数
    return 1 / (1 + np.exp(-x))

def tanh(x):             # tanh 函数
    return 2*sigmoid(2*x) - 1

def derivative(x):       # tanh 函数的导数
    return 1-tanh(x)**2
```

7.3 损失函数及其梯度

- ✓ Mean Squared Error
- ✓ Cross Entropy Loss

MSE

$$\text{loss} = \sum[y - f_\theta(x)]^2$$

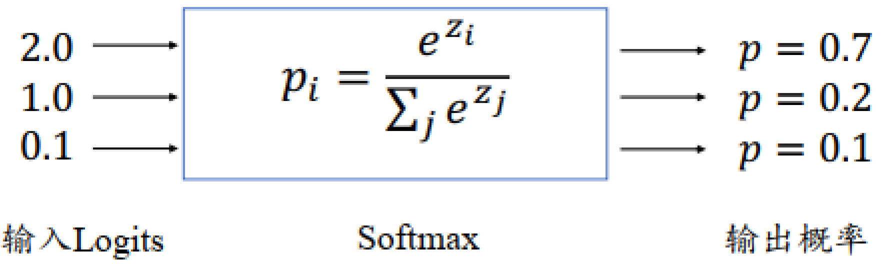
$$\frac{\nabla loss}{\nabla \theta} = 2\sum[y - f_\theta(x)] * \frac{\nabla f_\theta(x)}{\nabla \theta}$$

tensorflow 中的实现

```
In [34]: x = tf.random.normal([2,4])

In [35]: w = tf.random.normal([4,3])

In [36]: b = tf.zeros([3])

In [37]: y = tf.constant([2,0])

In [38]: with tf.GradientTape() as tape:
    ...:     tape.watch([w,b])
    ...:     prob = tf.nn.softmax(x@w+b, axis=1)
    ...:     loss = tf.reduce_mean(tf.losses.MSE(tf.one_hot(y, depth=3), prob))
    ...:

In [39]: grads = tape.gradient(loss, [w,b])
```

没有声明w和b为tf.Variable，则需要用这一句

Softmax 梯度



softmax 求导推导

Derivation

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}}$$

$$\frac{\partial p_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}}}{\partial a_j}$$

$$f(x) = \frac{g(x)}{h(x)}$$

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$$

$$g(x) = e^{a_i}$$

$$h(x) = \sum_{k=1}^N e^{a_k}$$

when $i = j$

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}}}{\partial a_j} = \frac{e^{a_i}\sum_{k=1}^N e^{a_k} - e^{a_j}e^{a_i}}{\left(\sum_{k=1}^N e^{a_k}\right)^2}$$

$$= \frac{e^{a_i}\left(\sum_{k=1}^N e^{a_k} - e^{a_j}\right)}{\left(\sum_{k=1}^N e^{a_k}\right)^2}$$

$$= \frac{e^{a_j}}{\sum_{k=1}^N e^{a_k}} \times \frac{\left(\sum_{k=1}^N e^{a_k} - e^{a_j}\right)}{\sum_{k=1}^N e^{a_k}}$$

$$= p_i(1 - p_j)$$

Derivative

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}$$

$$\frac{\partial p_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j}$$

$$f(x) = \frac{g(x)}{h(x)}$$

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$$

$$g(x) = e^{a_i}$$

$$h(x) = \sum_{k=1}^{N} e^{a_k}$$

when $i \neq j$

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{0 - e^{a_j} e^{a_i}}{\left(\sum_{k=1}^{N} e^{a_k}\right)^2}$$

$$= \frac{-e^{a_j}}{\sum_{k=1}^{N} e^{a_k}} \times \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}$$

$$= -p_j . p_i$$

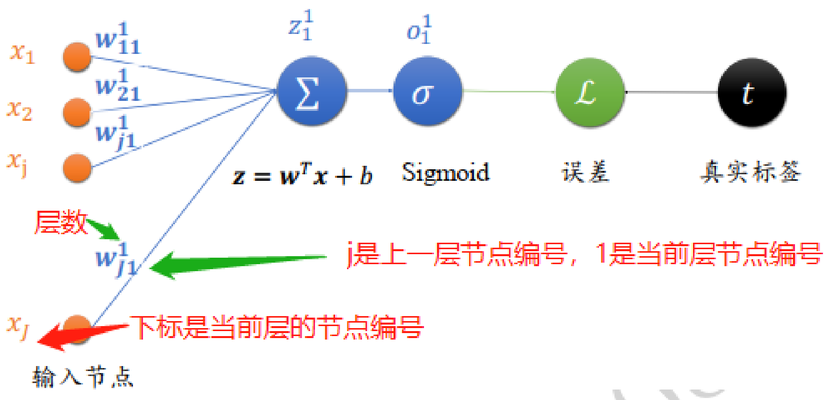
最终的形式：

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1 - p_j) & if \quad i = j \\ -p_j . p_i & if \quad i \neq j \end{cases}$$

Or using Kronecker delta $\delta ij = \begin{cases} 1 & if \quad i = j \\ 0 & if \quad i \neq j \end{cases}$

$$\frac{\partial p_i}{\partial a_j} = p_i(\delta_{ij} - p_j)$$

7.4 单输出感知机梯度



神经元只有一个输出，因此损失可以表达为：

$$\mathcal{L} = \frac{1}{2}(o_1^1 - t)^2$$

下面对该单层感知机进行推导，以第 j 号节点的权值w_{j1}为例，考虑损失函数$\mathcal{L}$对其的偏导数$\frac{\partial\mathcal{L}}{\partial w_{j1}}$

$$\frac{\partial\mathcal{L}}{\partial w_{j1}} = (o_1 - t)\frac{\partial o_1}{\partial w_{j1}}$$

将$o_1 = \sigma(z_1)$分解，考虑到 sigmoid 函数的导数为$\sigma'(x) = \sigma(x) * (1 - \sigma(x))$，因此

$$\frac{\partial\mathcal{L}}{\partial w_{j1}} = (o_1 - t)\frac{\partial\sigma(z_1)}{\partial w_{j1}} = (o_1 - t)\sigma(z_1)(1 - \sigma(z_1))\frac{\partial z_1^1}{\partial w_{j1}}$$

将$\sigma(z_1)$写成o_1，继续推导$\frac{\partial z_1^1}{\partial w_{j1}}$：

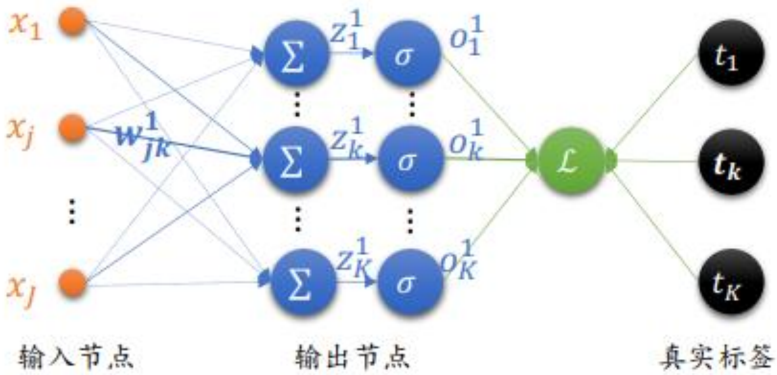
$$\frac{\partial\mathcal{L}}{\partial w_{j1}} = (o_1 - t)o_1(1 - o_1)\frac{\partial z_1^1}{\partial w_{j1}}$$

考虑到$\frac{\partial z_1^1}{\partial w_{j1}} = x_j$，于是，

$$\frac{\partial\mathcal{L}}{\partial w_{j1}} = (o_1 - t)o_1(1 - o_1)x_j$$

从上面可以看出，误差对权值w_{j1}的偏导数只与输出值o_1、真实值 t 以及当前权值链接的输入x_j有关。

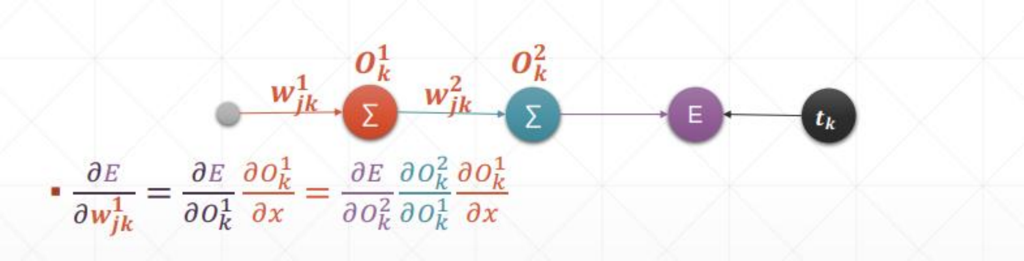
7.5 多输出感知机梯度



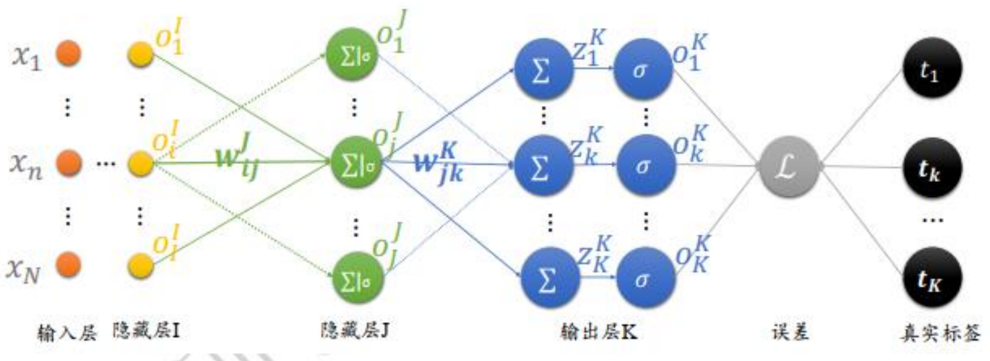
最终结果：$\frac{\sigma\mathcal{L}}{\partial w_{jk}} = \delta_k x_j$，其中，$\delta_k = (o_k - t_k)o_k(1 - o_k)$，即$\frac{\sigma\mathcal{L}}{\partial w_{jk}}$的偏导数只与当前链接的起始节点$x_j$，终止节点$\delta_k$有关。

7.6 链式法则

Rules	Function	Derivative
Multiplication by constant	cf	cf′
Power Rule	x^n	nx^{n-1}
Sum Rule	f + g	f′ + g′
Difference Rule	f - g	f′ − g′
Product Rule	fg	f g′ + f′ g
Quotient Rule	f/g	$(f' g - g' f)/g^2$
Reciprocal Rule	1/f	$-f'/f^2$
Chain Rule (as "Composition of Functions")	f ∘ g	(f′ ∘ g) × g′
Chain Rule (using ′)	f(g(x))	f′(g(x))g′(x)
Chain Rule (using $\frac{d}{dx}$)	$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$	



7.7 反向传播算法



最终结果：$\frac{\sigma\mathcal{L}}{\partial w_{jk}} = o_j(1-o_j)o_i\sum_k \delta_k^K w_{jk}$，令$\delta_j^J = o_j(1-o_j)\sum_k \delta_k^K w_{jk}$，此时，$\frac{\sigma\mathcal{L}}{\partial w_{jk}}$可以写为当前连接的起始节点的输出值$o_i$与终止节点$j$的梯度信息$\delta_j^J$的简单相乘

运算：

$$\frac{\sigma\mathcal{L}}{\partial w_{jk}} = \delta_j^J o_i^I$$

通过定义δ变量，每一层的梯度表达式变得更加清晰简洁，其中δ可以简单理解为当前连接w_{ij}对误差函数的贡献值。

For an output layer node $k \in K$

$$\frac{\partial E}{\partial W_{jk}} = \mathcal{O}_j \delta_k$$

where

$$\delta_k = \mathcal{O}_k(1-\mathcal{O}_k)(\mathcal{O}_k - t_k)$$

For a hidden layer node $j \in J$

$$\frac{\partial E}{\partial W_{ij}} = \mathcal{O}_i \delta_j$$

where

$$\delta_j = \mathcal{O}_j(1-\mathcal{O}_j)\sum_{k \in K} \delta_k W_{jk}$$

输出层

$$\frac{\partial \mathcal{L}}{\partial w_{jk}} = \delta_k^K o_j$$

$$\delta_k^K = o_k(1-o_k)(o_k - t_k)$$

倒数第二层：

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \delta_j^J o_i$$

$$\delta_j^J = o_j(1-o_j)\sum_k \delta_k^K w_{jk}$$

倒数第三层：

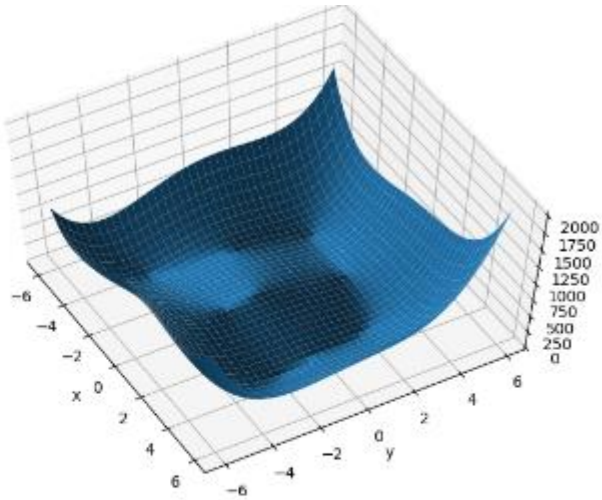
$$\frac{\partial \mathcal{L}}{\partial w_{ni}} = \delta_i^I o_n$$

$$\delta_i^I = o_i(1-o_i)\sum_j \delta_j^J w_{ij}$$

其中o_n为倒数第三层的输入，即倒数第四层的输出。

7.8 Himmelblau 函数优化

函数表达式为： $f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$




```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def himmelblau(x):
    return (x[0]**2 + x[1] - 11) ** 2 + (x[0] + x[1] ** 2 - 7) ** 2

x = np.arange(-6, 6, 0.1)
y = np.arange(-6, 6, 0.1)
print("x, y range:", x.shape, y.shape)
X, Y = np.meshgrid(x, y)
print("X,Y mpas:", X.shape, Y.shape)
Z = himmelblau([X, Y])

fig = plt.figure('himmelblau')
ax = fig.gca(projection='3d')
ax.plot_surface(X, Y, Z)
ax.view_init(60, -30)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```

Gradient Descent

```
# [1., 0.], [-4, 0.], [4, 0.]    改变初始点
x = tf.constant([-4., 0.])

for step in range(200):
```

```
    with tf.GradientTape() as tape:
        tape.watch([x])
        y = himmelblau(x)

    grads = tape.gradient(y, [x])[0]
    x -= 0.01 * grads

    if step % 20 == 0:
        print('step {}: x = {}, f(x) = {}'.format(step, x.numpy(), y.
numpy()))
```

7.9 可视化

- ✓ installation
- ✓ curves
- ✓ image visualization

Installation

pip install tensorboard

分为三步实现 tensorboard 可视化：

- ✓ listen logdir
- ✓ build summary instance
- ✓ fed data into summary instance

Step 1 run listener :

```
(tf2.0) C:\Users\Leowen>tensorboard --logdir logs
2019-11-25 19:20:55.091514: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successful
brary cudart64_100.dll
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.0.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

tensorboard --logdir logs

open URL: http://localhost:6006

Step 2 build summary

```
current_time = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
log_dir = 'logs/' + current_time
summary_writer = tf.summary.create_file_writer(log_dir)
```

Step 3 fed scalar

```
with summary_writer.as_default():
    tf.summary.scalar('loss', float(loss), step=epoch)
    tf.summary.scalar('accuracy', float(train_accuracy), step=epoch)
```

Step 3 fed single image

```
# get x from (x,y)
sample_img = next(iter(db))[0]
# get first image instance
sample_img = sample_img[0]
sample_img = tf.reshape(sample_img, [1,28,28,1])
with summary_writer.as_default():
    tf.summary.image('Training sample', sample_img, step=0)
```

Step 3 fed multi- images

```
val_images = x[:25]
val_images = tf.reshpae(val_images, [-1,28,28,1])
with summary_writer.as_default():
    tf.summary.image('val-onebyone-
images: ', val_images, max_output=25,step=step)
```