9 过拟合

数据集的划分

为了挑选模型超参数和检测过拟合现象，一般需要将原来的训练集切分为新的训练集(Train set)和验证集(Validation set)，然后剩下的部分用做验证集(Test set)。

一般而言，将整个数据集划分为训练集和测试集。训练集占 80%，测试集占 20%。为了选择模型的超参数，因此需要将训练集再次划分为新的训练集和验证集，所以最终的数据集划分为，训练集 60%，验证集 20%以及测试集 20%。

```
(x, y), (x_test, y_test) = datasets.mnist.load_data()   继续划分训练集

x_train, x_val = tf.split(x, num_or_size_splits=[50000, 10000])
y_train, y_val = tf.split(y, num_or_size_splits=[50000, 10000])
db_train = tf.data.Dataset.from_tensor_slices((x_train,y_train))
db_train = db_train.map(preprocess).shuffle(50000).batch(batchsz)

db_val = tf.data.Dataset.from_tensor_slices((x_val,y_val))
db_val = db_val.map(preprocess).shuffle(10000).batch(batchsz)

db_test = tf.data.Dataset.from_tensor_slices((x_test, y_test))
db_test = db_test.map(preprocess).batch(batchsz)
```

在训练的时候进行验证集评估

```
network.compile(optimizer=optimizers.Adam(lr=0.01),
        loss=tf.losses.CategoricalCrossentropy(from_logits=True),
        metrics=['accuracy']
    )

network.fit(db_train, epochs=5, validation_data=db_val, validation_freq=2)
```

训练完成之后进行测试集评估

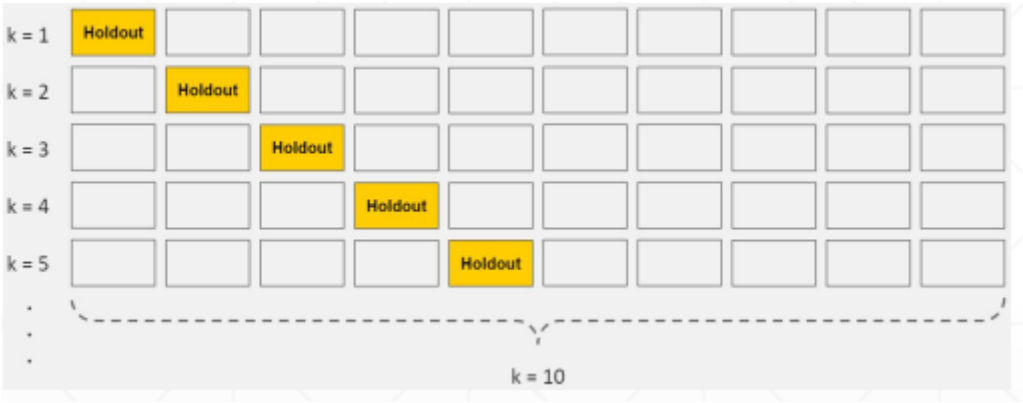
```
network.fit(db_train, epochs=5, validation_data=db_val, validation_freq=2)

print('Test performance:')
network.evaluate(db_test)
```

K-fold cross-validation

K 折叠交叉验证分为两步：

1、 将训练集和验证集合并

2、 随机选择 1/k 的样本用作验证集




```
for epoch in range(500):
  idx = tf.range(60000)
  idx = tf.random.shuffle(idx)
  x_train, y_train = tf.gather(x, idx[:50000]), tf.gather(y, idx[:50000])
  x_val, y_val = tf.gather(x, idx[-10000:]) , tf.gather(y, idx[-10000:])

  db_train = tf.data.Dataset.from_tensor_slices((x_train,y_train))
  db_train = db_train.map(preprocess).shuffle(50000).batch(batchsz)

  db_val = tf.data.Dataset.from_tensor_slices((x_val,y_val))
  db_val = db_val.map(preprocess).shuffle(10000).batch(batchsz)

  # training...

  # evalutation...
```

```
network.fit(db_train_val, epochs=6, validation_split=0.1, validation_freq=2)
```

9.1 减少过拟合

- ✓ More data
- ✓ Constraint model cmplexity
  - ✓ shallow
  - ✓ regularization
- ✓ Dropout
- ✓ Data argumentation
- ✓ Early Stopping

正则化

什么是奥卡姆剃刀原理？

切勿浪费较多东西，去做'用较少的东西就同样可以做好的事情！'

为了防止过拟合，通过限制网络参数的稀疏性，可以用来约束网络的实际容量。这种约束一般通过在损失函数上添加额外的参数稀疏性惩罚项来实现，在未加约束之前的优化目标是

$$Minimize\ \mathcal{L}(f_\theta(x), y), \qquad (x, y) \in D^{train}$$

对模型的参数添加额外的约束后，优化的目标变为：

$$Minimize\ \mathcal{L}(f_\theta(x), y) + \lambda * \Omega(\theta), \qquad (x, y) \in D^{train}$$

其中，$\Omega(\theta)$表示对网络参数Θ的稀疏性约束函数，一般地，参数Θ的稀疏性约束通过约束参数Θ的 L 范数实现，即，

$$\Omega(\theta) = \sum_{\theta_i} \|\theta_i\|_l$$

新的优化目标除了要最小化原来的损失函数$\mathcal{L}(x, y)$之外，还需要约束网络参数的稀疏性，优化算法会在降低$\mathcal{L}(x, y)$的同时， 尽可能地迫使网络参数θ_i变得稀疏，他们之间的权重关系通过超参数λ来平衡，较大的λ意味着网络的稀疏性更重要；较小的λ则意味着网络的训练误差更重要。通过选择合适的λ超参数可以获得较好的训练性能，同时保证网络的稀疏性，从而获得不错的泛化能力。

常用的正则化方式有 L0,L1(Lasso Regularization),L2 正则化(Ridge Regularization)

```
l2_model = keras.models.Sequential([
    keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001),
                       activation=tf.nn.relu, input_shape=(NUM_WORDS,)),
    keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l2(0.001),
                       activation=tf.nn.relu),
    keras.layers.Dense(1, activation=tf.nn.sigmoid)
])
```

```
for step, (x,y) in enumerate(db):
    with tf.GradientTape() as tape:
        # ...
        loss = tf.reduce_mean(tf.losses.categorical_crossentropy(y_onehot, out,
from_logits=True))

        loss_regularization = []
        for p in network.trainable_variables:
            loss_regularization.append(tf.nn.l2_loss(p))
        loss_regularization = tf.reduce_sum(tf.stack(loss_regularization))

        loss = loss + 0.0001 * loss_regularization

    grads = tape.gradient(loss, network.trainable_variables)
    optimizer.apply_gradients(zip(grads, network.trainable_variables))
```

动量和学习率衰减

- ✓ momentum
- ✓ learning rate decay

Momentum

$$w^{k+1} = w^k - \alpha \nabla f(w^k).$$

$$z^{k+1} = \beta z^k + \nabla f(w^k)$$
$$w^{k+1} = w^k - \alpha z^{k+1}$$

```
optimizer = SGD(learning_rate=0.02, momentum=0.9)
optimizer = RMSprop(learning_rate=0.02, momentum=0.9)

optimizer = SGD(learning_rate=0.02,
    beta_1=0.9,
    beta_2=0.999)
```

Learning rate tunning

```
optimizer = SGD(learning_rate=0.2)

for epoch in range(100):
    # get loss

    # change learning rate
    optimizer.learning_rate = 0.2 * (100-epoch)/100

    # update weights
```

Early Stopping, Dropout

- ✓ layer.Dropout(rate)
- ✓ tf.nn.dropout(x,rate)

```
# 添加 Dropout 参数
x = tf.nn.dropout(x, rate=0.5)
#也可以将 Dropout 作为网络层使用，在网络中间插入一个 Dropout 层
# 添加 Dropout 层
model.add(layers.Dropout(rate=0.5))
```

在测试的时候，不需要 dropout，网络还是全连接的，所以在 Tensorflow 中，在训练和测试的时候需要切换过来。

```
for step, (x,y) in enumerate(db):

    with tf.GradientTape() as tape:
        # [b, 28, 28] => [b, 784]
        x = tf.reshape(x, (-1, 28*28))
        # [b, 784] => [b, 10]
        out = network(x, training=True)


    # test
    out = network(x, training=False)
```

提前停止

随机初始化参数θ

repeat

  for $step = 1, \ldots, N$ **do**

    随机采样训练 batch $\{(\boldsymbol{x}, y)\}$

    $\theta \leftarrow \theta - \eta \nabla \mathcal{L}(f(x), y)$

  end

  数个 epoch 后验证一次

  if 验证性能连续数次不提升 **do**

    提取停止训练

  end

until 训练回合数 epoch 达到要求

Stochastic Gradient Descent

随机梯度下降并不是随机的意思，而是符合某一个分布的。比如你有 60k 个样本，那么我随机地从中选出一个 batch，比如 128 个样本出来，这个就叫做随机

梯度下降。最原始的随机梯度下降是取一个样本，现在也指取一个 batch 的样本。然后损失就是这一个 batch 的均值损失。这样能够解决一次加载全部样本所带来的内存不足的情况。