深度学习中的正则化

朱明超
deityrayleigh@gmail.com

正则化的目标是减少模型泛化误差，为此提出了各种方法。本篇提出的正则化方法主要是考虑**当训练误差较小，但泛化误差较大**的情况下。

1 参数范数惩罚

许多正则化方法 (如神经网络、线性回归、逻辑回归) 通过对目标函数 J 添加一个参数范数惩罚 $\Omega(\theta)$ ，限制模型的学习能力。将正则化后的目标函数记为 $\tilde{J}$:

$$\tilde{J}(\theta;\boldsymbol{X},\boldsymbol{y}) = J(\theta;\boldsymbol{X},\boldsymbol{y}) + \alpha\Omega(\theta) \tag{1}$$

其中 $\alpha \in [0,+\infty)$ 是衡量参数范数惩罚程度的超参数。$\alpha = 0$ 表示没有正则化，α 越大对应正则化惩罚越大。

在神经网络中，参数包括每层线性变换的权重和偏置，我们**通常只对权重做惩罚而不对偏置做正则惩罚**；使用向量 $\boldsymbol{w}$ 表示应受惩罚影响的权重，用向量 θ 表示所有参数。

1.1 L^2 正则化

L^2 参数正则化 (也称为岭回归、Tikhonov 正则) 通常被称为权重衰减 (`weight decay`)，是通过向目标函数添加一个正则项 $\Omega(\theta) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ ，使权重更加接近原点。

目标函数:

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X},y) + \frac{\alpha}{2}\boldsymbol{w}^\top\boldsymbol{w} \tag{2}$$

计算梯度:

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha\boldsymbol{w} \tag{3}$$

更新权重:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})) = (1-\epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) \tag{4}$$

从上式可以看出，加入权重衰减后会导致学习规则的修改，即在每步执行梯度更新前先收缩权重 (乘以 $(1-\epsilon\alpha)$)。

以第 6 章介绍的代价函数 $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log(1-\hat{\boldsymbol{y}}^{(i)})\right)$ 为例，在增加 L^2 正则化后，代价函数变为:

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\left(\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log\left(1-\hat{\boldsymbol{y}}^{(i)}\right)\right)\right.}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}} \tag{5}$$

而在反向传播的时候，须加上正则化项的梯度:

$$\frac{d}{d\boldsymbol{W}}(\frac{1}{2}\frac{\lambda}{m}\boldsymbol{W}^2) = \frac{\lambda}{m}\boldsymbol{W} \tag{6}$$

1.2 L^1 正则化

将参数惩罚项 $\Omega(\theta)$ 由权重衰减项修改为各个参数的绝对值之和，可以得到 L^1 正则化:

$$\Omega(\theta) = \|\boldsymbol{w}\|_1 = \sum_i |w_i| \tag{7}$$

目标函数:

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha\|\boldsymbol{w}\|_1 \tag{8}$$

计算梯度:

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha sgn(\boldsymbol{w}) \tag{9}$$

其中 $sgn(x)$ 为符号函数，取各个元素的正负号。

更新权重:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha sgn(\boldsymbol{w}) + \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})) \tag{10}$$

同样，在增加 L^1 正则化后，代价函数变为：

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log(1-\hat{\boldsymbol{y}}^{(i)})\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}|W_{k,j}^{[l]}|}_{\text{L1 regularization cost}} \tag{11}$$

而在反向传播的时候，须加上正则化项的梯度：

$$\frac{d}{d\boldsymbol{W}}(\frac{\lambda}{m}\|\boldsymbol{W}\|) = \frac{\lambda}{m}sgn(\boldsymbol{W}) \tag{12}$$

[1]:
```
from abc import ABC, abstractmethod
import numpy as np
from PIL import Image
%matplotlib inline
import matplotlib.pyplot as plt
import math
import re
import time
```

[2]:
```
class RegularizerBase(ABC):

    def __init__(self, **kwargs):
        super().__init__()

    @abstractmethod
    def loss(self, **kwargs):
        raise NotImplementedError

    @abstractmethod
    def grad(self, **kwargs):
        raise NotImplementedError

class L1Regularizer(RegularizerBase):

    def __init__(self, lambd=0.001):
        super().__init__()
        self.lambd = lambd

    def loss(self, params):
        loss = 0
        pattern = re.compile(r'^W\d+')
        for key, val in params.items():
            if pattern.match(key):
                loss +=  0.5 * np.sum(np.abs(val)) * self.lambd
        return loss

    def grad(self, params):
        for key, val in params.items():
            grad = self.lambd * np.sign(val)
        return grad

class L2Regularizer(RegularizerBase):

    def __init__(self, lambd=0.001):
        super().__init__()
        self.lambd = lambd

    def loss(self, params):
        loss = 0
        for key, val in params.items():
            loss +=  0.5 * np.sum(np.square(val)) * self.lambd
        return loss
```

```
    def grad(self, params):
        for key, val in params.items():
            grad = self.lambd * val
        return grad

class RegularizerInitializer(object):

    def __init__(self, regular_name="l2"):
        self.regular_name = regular_name

    def __call__(self):
        r = r"([a-zA-Z]*)=([^,)]*)"
        regular_str = self.regular_name.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, regular_str)])
        if  "l1" in regular_str.lower():
            regular = L1Regularizer(**kwargs)
        elif "l2" in regular_str.lower():
            regular = L2Regularizer(**kwargs)
        else:
            raise ValueError("Unrecognized regular: {}".format(regular_str))
        return regular
```

我们对第六章介绍的 DFN 引入正则项，我们将第六章中介绍的函数存储在 chapter6.py 中。

[3]:
```
from chapter6 import WeightInitializer, ActivationInitializer, LayerBase, CrossEntropy, OrderedDict, softmax
```

[4]:
```
class FullyConnected(LayerBase):
    """
    定义全连接层，实现 a=g(x*W+b)，前向传播输入 x，返回 a；反向传播输入
    """
    def __init__(self, n_out, acti_fn, init_w, optimizer=None):
        """
        参数说明：
        acti_fn：激活函数， str 型
        init_w：权重初始化方法， str 型
        n_out：隐藏层输出维数
        optimizer：优化方法
        """
        super().__init__(optimizer)
        self.n_in = None  # 隐藏层输入维数， int 型
        self.n_out = n_out  # 隐藏层输出维数， int 型
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.is_initialized = False  # 是否初始化， bool 型变量

    def _init_params(self):
        b = np.zeros((1, self.n_out))
        W = self.init_weights((self.n_in, self.n_out))
        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        全连接网络的前向传播，原理见上文 反向传播算法 部分。

        参数说明：
        X：输入数组，为 (n_samples, n_in)，float 型
```

```
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if not self.is_initialized:  # 如果参数未初始化，先初始化参数
            self.n_in = X.shape[1]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        z = X @ W + b
        a = self.acti_fn.forward(z)
        if retain_derived:
            self.X.append(X)
        return a

    def backward(self, dLda, retain_grads=True, regular=None):
        """
        全连接网络的反向传播，原理见上文 反向传播算法 部分。

        参数说明：
        dLda: 关于损失的梯度，为 (n_samples, n_out)，float 型
        retain_grads: 是否计算中间变量的参数梯度，bool 型
        regular: 正则化项
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        for da, x in zip(dLda, X):
            dx, dw, db = self._bwd(da, x, regular)
            dX.append(dx)
            if retain_grads:
                self.gradients["W"] += dw
                self.gradients["b"] += db
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X, regular):
        W = self.params["W"]
        b = self.params["b"]
        Z = X @ W + b
        dZ = dLda * self.acti_fn.grad(Z)
        dX = dZ @ W.T
        dW = X.T @ dZ
        db = dZ.sum(axis=0, keepdims=True)
        if regular is not None:
            n = X.shape[0]
            dW_norm = regular.grad(self.params) / n
            dW += dW_norm
        return dX, dW, db

    @property
    def hyperparams(self):
        return {
            "layer": "FullyConnected",
            "init_w": self.init_w,
            "n_in": self.n_in,
            "n_out": self.n_out,
            "acti_fn": str(self.acti_fn),
            "optimizer": {
                "hyperparams": self.optimizer.hyperparams,
            },
```

```
            "components": {
                k: v for k, v in self.params.items()
            }
        }
```

[5]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练，具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]

    return mb_generator(), n_batches

class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        regular_act=None,
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.regular_act = regular_act
        self.regular = None
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        if self.regular_act is not None:
```

```
            self.regular = RegularizerInitializer(self.regular_act)()
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out, regular=self.regular)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()

        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch)
                y_pred_batch = softmax(out)
```

```
            batch_loss = self.loss(y_batch, y_pred_batch)
            # 正则化损失
            if self.regular is not None:
                for _, layerparams in self.hyperparams['components'].items():
                    assert type(layerparams) is dict
                    batch_loss += self.regular.loss(layerparams)
            grad = self.loss.grad(y_batch, y_pred_batch)
            _, _ = self.backward(grad)
            self.update()
            loss += batch_loss
            if self.verbose:
                fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))

        loss /= n_batch
        fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
        print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
        prev_loss = loss
```

```
    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "regular": str(self.regular_act),
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[6]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
```

```
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[7]:
```
"""
不引入正则化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 2.286  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 2.209  Delta: 0.078 (0.01m/epoch)
[Epoch 3] Avg. loss: 1.993  Delta: 0.215 (0.01m/epoch)
[Epoch 4] Avg. loss: 1.640  Delta: 0.353 (0.01m/epoch)
[Epoch 5] Avg. loss: 1.305  Delta: 0.335 (0.01m/epoch)
[Epoch 6] Avg. loss: 1.063  Delta: 0.242 (0.01m/epoch)
[Epoch 7] Avg. loss: 0.898  Delta: 0.166 (0.01m/epoch)
[Epoch 8] Avg. loss: 0.781  Delta: 0.117 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.696  Delta: 0.085 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.634  Delta: 0.062 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.586  Delta: 0.048 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.549  Delta: 0.037 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.518  Delta: 0.031 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.493  Delta: 0.025 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.473  Delta: 0.021 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.454  Delta: 0.018 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.439  Delta: 0.015 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.425  Delta: 0.014 (0.01m/epoch)
[Epoch 19] Avg. loss: 0.414  Delta: 0.012 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.404  Delta: 0.010 (0.01m/epoch)
```

[8]:
```
print("without regularization -- accuracy:{}".format(model.evaluate(X_test, y_test)))


###### if show params ######
# print("regular", model.hyperparams["regular"], "\nparams:", model.hyperparams["components"])
```

```
without regularization -- accuracy:0.8961
```

[9]:
```
"""
引入 l2 正则化
"""
model_re = DFN(hidden_dims_1=200, hidden_dims_2=10, regular_act="l2(lambd=0.01)")
model_re.fit(X_train, y_train, n_epochs=20)
```

```
[Epoch 1] Avg. loss: 2.363  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.284  Delta: 0.079 (0.02m/epoch)
[Epoch 3] Avg. loss: 2.068  Delta: 0.216 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.729  Delta: 0.339 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.428  Delta: 0.301 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.226  Delta: 0.202 (0.02m/epoch)
[Epoch 7] Avg. loss: 1.096  Delta: 0.130 (0.02m/epoch)
[Epoch 8] Avg. loss: 1.013  Delta: 0.083 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.958  Delta: 0.055 (0.02m/epoch)
[Epoch 10] Avg. loss: 0.923  Delta: 0.035 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.899  Delta: 0.024 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.883  Delta: 0.016 (0.01m/epoch)
```

```
[Epoch 13] Avg. loss: 0.872  Delta: 0.011 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.865  Delta: 0.007 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.860  Delta: 0.004 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.858  Delta: 0.002 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.858  Delta: 0.001 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.858  Delta: -0.000 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.859  Delta: -0.001 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.860  Delta: -0.001 (0.01m/epoch)
```

[10]:
```
print("with L2 regularization -- accuracy:{}".format(model_re.evaluate(X_test, y_test)))


###### if show params ######
# print("regular", model_re.hyperparams["regular"], "\nparams:", model_re.hyperparams["components"])
```

```
with L2 regularization -- accuracy:0.8958
```

1.3 总结

相比 L^2 正则化，L^1 正则化会产生更稀疏的解。

假设 $\boldsymbol{w}^*$ 为未正则化的目标函数取得最优时的权重向量，可以得到 L^2 正则化后的最优解为 $\tilde{\boldsymbol{w}} = (\mathrm{H} + \alpha I)^{-1}\mathrm{H}\boldsymbol{w}^*$。如果 Hessian 矩阵是对角正定矩阵，我们得到 L^2 正则化的最优解是 $\tilde{w}_i = \frac{\mathrm{H}_{i,i}}{\mathrm{H}_{i,i}+\alpha}\boldsymbol{w}_i^*$ 。如果 $w_i^* \neq 0$ ，则 $\tilde{w}_i \neq 0$ ，这说明 L^2 正则化不会使参数变得稀疏。

L^1 正则化的最优解为 $\tilde{w}_i = \mathrm{sign}(w_i^*)\max\left\{|w_i^*| - \frac{\alpha}{H_{i,i}}, 0\right\}$ ，可以看出，L^1 正则化有可能通过足够大的 α 实现稀疏。

- 正则化策略可以被解释为最大后验 (MAP) 贝叶斯推断。(详细内容见第五章)
  - L^2 正则化相当于权重是高斯先验的 MAP 贝叶斯推断；
  - L^1 正则化相当于权重是 `Laplace` 先验的 MAP 贝叶斯推断。

1.4 作为约束的范数惩罚

考虑参数范数正则化的代价函数：

$$\tilde{J}(\theta; \boldsymbol{X}, \boldsymbol{y}) = J(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\theta) \tag{13}$$

如果想约束 $\Omega(\theta) < k$，k 是某个常数，可以构造广义 Lagrange 函数：

$$\mathcal{L}(\theta, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\theta) - k) \tag{14}$$

该约束问题的解是：

$$\theta^* = \arg\min_{\theta}\max_{\alpha,\alpha\geq 0}\mathcal{L}(\theta, \alpha) \tag{15}$$

对于该问题，可以通过调节 α 与 k 的值来扩大或缩小权重的约束区域。较大的 α 将得到一个较小的约束区域；而较小的 α 将得到一个较大的约束区域。

另一方面，重新考虑 `1.1` 和 `1.2` 中的正则化，正则化式等价于带约束的目标函数中的约束项。例如以平方损失函数和 L^2 正则化为例，优化模型如下:

$$\begin{aligned} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) &= \sum_{i=1}^{n}(y_i - \theta^{\top}\mathrm{x_i})^2 \\ s.t. \|\boldsymbol{\theta}\|_2^2 &<= \mathrm{C} \end{aligned} \tag{16}$$

采用拉格朗日乘积算子法可以转化为无约束优化问题，即：

$$J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \sum_{i=1}^{n}(y_i - \boldsymbol{\theta}^{\top}\boldsymbol{x}_i)^2 + \lambda(\|\boldsymbol{\theta}\|_2^2 - \mathrm{C}) \tag{17}$$

1.5 欠约束问题

机器学习中许多线性模型，如线性回归和 PCA，都依赖于矩阵 $\boldsymbol{X}^{\top}\boldsymbol{X}$ 求逆。如果 $\boldsymbol{X}^{\top}\boldsymbol{X}$ 不可逆，这些方法就会失效。这种情况下，**正则化的许多形式对应求逆 $\boldsymbol{X}^{\top}\boldsymbol{X} + \alpha\boldsymbol{I}$，且这个正则化矩阵是可逆的**。大多数正则化方法能够保证应用于欠定问题的迭代方法收敛。

例如线性回归的损失函数是平方误差之和：$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\top}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$。

我们添加 L^2 正则项后，目标函数变为 $(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\top}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\alpha\boldsymbol{w}^{\top}\boldsymbol{w}$。

这将普通方程的解从 $\boldsymbol{w} = (\boldsymbol{X}^{\top}\boldsymbol{X})^{-1}\boldsymbol{X}^{\top}\boldsymbol{y}$ 变为 $\boldsymbol{w} = (\boldsymbol{X}^{\top}\boldsymbol{X} + \alpha\boldsymbol{I})^{-1}\boldsymbol{X}^{\top}\boldsymbol{y}$。

2 2 数据增强

2.1 数据集增强

数据集增强是解决数据量有限的问题，**让机器学习模型泛化得更好的最好办法是使用更多的数据进行训练** (这在对象识别 (`object detection`) 问题很有效)。

方法:

- 类别不改变，平移不变性。比如对图像的平移，旋转，缩放。
- 注入噪声，可以使模型对噪声更健壮 (如去噪自编码器)。

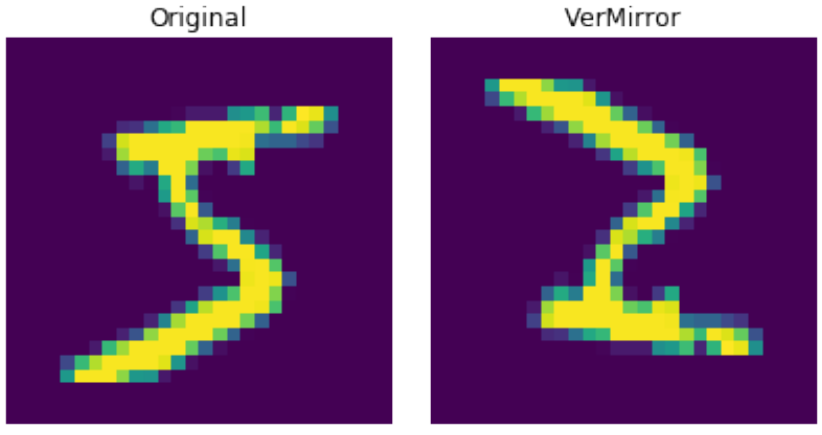
[11]:
```
class Image(object):

    def __init__(self, image):
        self._set_params(image)

    def _set_params(self, image):
        self.img = image
        self.row = image.shape[0] # 图像高度
        self.col = image.shape[1] # 图像宽度
        self.transform = None

    def Translation(self, delta_x, delta_y):
        """
        平移。

        参数说明:
        delta_x: 控制左右平移，若大于 0 左移，小于 0 右移
        delta_y: 控制上下平移，若大于 0 上移，小于 0 下移
        """
        self.transform = np.array([[1, 0, delta_x],
                                   [0, 1, delta_y],
                                   [0,  0,  1]])

    def Resize(self, alpha):
        """
        缩放。

        参数说明:
        alpha: 缩放因子，不进行缩放设置为 1
        """
        self.transform = np.array([[alpha, 0, 0],
                                   [0, alpha, 0],
                                   [0,  0,  1]])

    def HorMirror(self):
        """
        水平镜像。
        """
        self.transform = np.array([[1,  0,  0],
                                   [0, -1, self.col-1],
                                   [0,  0,  1]])

    def VerMirror(self):
        """
        垂直镜像。
        """
        self.transform = np.array([[-1, 0, self.row-1],
                                   [0,  1,  0],
                                   [0,  0,  1]])
```

```
def Rotate(self, angle):
    """
    旋转。

    参数说明:
    angle: 旋转角度
    """
    self.transform = np.array([[math.cos(angle),-math.sin(angle),0],
                               [math.sin(angle), math.cos(angle),0],
                               [    0,              0,         1]])

def operate(self):
    temp = np.zeros(self.img.shape, dtype=self.img.dtype)
    for i in range(self.row):
        for j in range(self.col):
            temp_pos = np.array([i, j, 1])
            [x,y,z] = np.dot(self.transform, temp_pos)
            x = int(x)
            y = int(y)
            if x>=self.row or y>=self.col or x<0 or y<0:
                temp[i,j,:] = 0
            else:
                temp[i,j,:] = self.img[x,y]
    return temp

def __call__(self, act):
    r = r"([a-zA-Z]*)=([^,)]*)"
    act_str = act.lower()
    kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, act_str)])
    if "translation" in act_str:
        self.Translation(**kwargs)
    elif "resize" in act_str:
        self.Resize(**kwargs)
    elif "hormirror" in act_str:
        self.HorMirror(**kwargs)
    elif "vermirror" in act_str:
        self.VerMirror(**kwargs)
    elif "rotate" in act_str:
        self.Rotate(**kwargs)
    return self.operate()
```

[12]:
```
"""
导入数据， 手写体数字
"""
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
```

[13]:
```
img = X_train[0].reshape(28, 28, 1)
Img = Image(img)
ax = plt.subplot(121)
plt.tight_layout()
plt.imshow(img.reshape(28,28))
```

```
plt.title('Original')
plt.axis('off')
ax = plt.subplot(122)
plt.imshow(Img('vermirror').reshape(28,28))
plt.title('VerMirror')
plt.axis('off')
plt.show()
```



2.2 噪声鲁棒性

- 将噪声**加入到输入**，等同于**数据集增强**。
- 将噪声**加入到权重**，能够**表现权重的不确定性**，这项技术主要用于循环神经网络。这可以被解释为关于权重的贝叶斯推断的随机实现，贝叶斯学习过程将权重视为不确定的，并且可以通过概率分布表示这种不确定性，向权重添加噪声是反映这种不确定性的一种实用的随机方法。
- 将噪声**加入到输出**，对噪声建模 (滤波)，**标签平滑**。由于大多数数据集的 $\boldsymbol{y}$ 标签都有一定错误，错误的 $\boldsymbol{y}$ 不利于最大化 $\log p(\boldsymbol{y} \mid \boldsymbol{x})$ ，避免这种情况的一种方法是显式地对标签上的噪声进行建模。

3 训练方案

3.1 半监督学习

监督学习指训练样本都是带标记的。然而在现实中，获取数据是容易的，但是收集到带标记的数据却是非常昂贵的。**半监督学习指的是既包含部分带标记的样本也有不带标记的样本**，通过这些数据来进行学习。在半监督学习的框架下，$P(\boldsymbol{x})$ 产生的未标记样本和 $P(\boldsymbol{x}, \boldsymbol{y})$ 中的标记样本都用于估计 $P(\boldsymbol{y} \mid \boldsymbol{x})$ 。在深度学习的背景下，半监督学习通常指的是学习一个表示 $h = f(\boldsymbol{x})$ ，**学习表示的目的是使相同类中的样本有类似的表示**。

我们可以构建这样一个模型，其中生成模型 $P(\boldsymbol{x})$ 或 $P(\boldsymbol{x}, \boldsymbol{y})$ 与判别模型 $P(\boldsymbol{y} \mid \boldsymbol{x})$ 共享参数，而不用分离无监督和监督部分。

3.2 多任务学习

多任务学习是基于共享表示，**把多个相关的任务放在一起学习**的一种机器学习方法。当模型的一部分被多个额外的任务共享时，这部分将被约束为良好的值，通常会带来**更好的泛化能力**。

从深度学习的观点看，**底层的先验知识**为：**能解释数据变化的因素中，某些因素是跨多个任务共享的**。

可以考虑对复杂的问题，分解为简单且相互独立的子问题来单独解决。做单任务学习时，各个任务之间的模型空间是相互独立的，但这忽略了问题之间所富含的丰富的关联信息；而多任务学习便把多个相关的任务放在一起学习，学习过程中通过一个在浅层的共享表示来互相分享、互相补充学习到的领域相关的信息，互相促进学习，提升泛化的效果。

多任务学习是正则化的一种方法，对于**与主任务相关的任务**，可以看作是添加额外信息，**数据增强**；与**主任务不相关的任务**，可以看作是**引入噪音**，从而提高泛化。

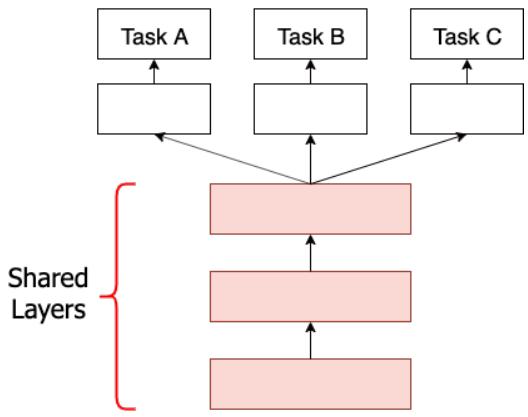


图 1. 多任务学习框架示意图，A, B, C 三个任务共享底层。

3.3 提前终止

当训练次数过多时会经常遭遇过拟合，此时训练误差会随时间推移减少，而验证集误差会再次上升。提前终止 (`Early Stopping`) 是一种交叉验证策略，我们将一部分训练集作为验证集 (`Validation Set`)。当我们看到验证集的性能越来越差时，我们立即停止对该模型的训练。

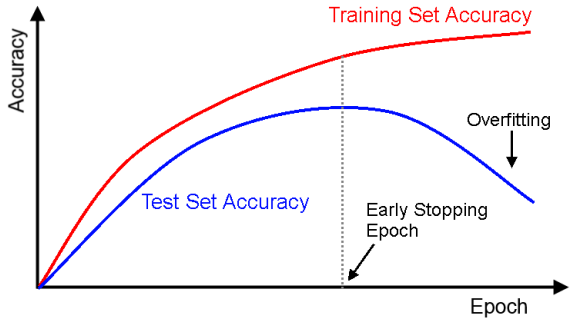


图 2. 学习曲线与提前终止。当测试集准确率下降时，可以提前终止。

提前终止的步骤如下：

- 将训练数据划分为训练集和测试集。
- 在训练集上训练，并在一段时间间隔内在测试集上预测。
- 当验证集上的误差高于上次时立即停止训练。
- 使用上一时刻中所得的权重来作为最终权重。

具体终止实现的策略可以多种：

- 策略一：当泛化损失大于某个阈值时立即停止。
- 策略二：假定过拟合仅仅发生在训练误差变化缓慢时。
- 策略三：当 s 个连续时刻的泛化误差增大时停止。

一般而言，除非网络性能的小改进比训练时间更重要，否则选择第一个策略。

将测试集重新融入训练集 为了更好的利用所有数据，我们需要在完成提前终止的首次训练之后进行第二轮的训练，在第二轮中，所有的数据都被包括在内。对此我们有两个基本策略：

- 再次初始化模型，然后使用所有数据再次训练，在第二轮训练中，我们采用第一轮提前终止训练确定的最佳步数。
- 保持从第一轮训练获得的参数，然后使用验证集的数据继续训练，直到验证集的平均损失函数低于提前终止过程终止时的目标值。

提前终止具有正则化效果，其真正机制可理解**为将优化过程的参数空间限制在初始参数值 θ_0 的小邻域内**。考虑平方误差的简单线性模型，采用梯度下降法，可以证明假如学习率为 ϵ ，进行 τ 次训练迭代，则 $\frac{1}{\epsilon\tau}$ 等价于权重衰减系数 α 。

[14]:
```
"""
策略：连续 4 个时刻验证集正确率没有增加
"""
def early_stopping(valid):
    """
    参数说明：
    valid：验证集正确率列表
    """
    if len(valid) > 5:
        if valid[-1] < valid[-5] and valid[-2] < valid[-5] and valid[-3] < valid[-5] and valid[-4] < valid[-5]:
```

```
            return True
    return False
```

4 模型表示

4.1 参数绑定与共享

参数范数惩罚或约束是相对于固定区域或点，如 L^2 正则化是对参数偏离 0 进行惩罚。有时我们需要对模型参数之间的相关性进行惩罚，使模型参数尽量接近或者相等：

参数共享：强迫模型某些参数相等：

- 主要应用：卷积神经网络 (CNN)
- 优点：显著降低了 CNN 模型的参数数量 (CNN 模型参数数量经常是千万量级以上)，减少模型所占用的内存，并且显著提高了网络大小而不需要相应的增加训练数据。

4.2 稀疏表示

稀疏表示也是卷积神经网络经常用到的正则化方法。L^1 正则化会诱导稀疏的参数，使得许多参数为 0；而稀疏表示是惩罚神经网络的激活单元，稀疏化激活单元。换言之，稀疏表示是使得每个神经元的输入单元变得稀疏，很多输入是 0。如下图所示，相比于全连接层，隐藏层的 h 接受到稀疏的输入 x。

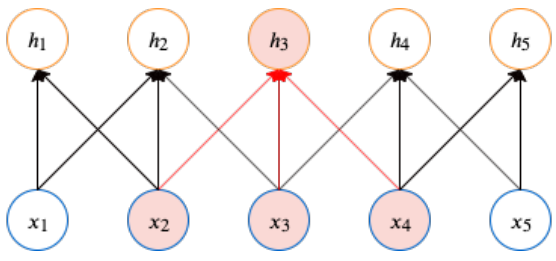


图 3. 稀疏表示示意图，每个隐藏层神经元最多连接三个输入单元。

4.3 Bagging 及其他集成方法

集成学习主要有 Boosting 和 Bagging 两种思路。在深度学习中，也可以考虑用 Bagging 的思路来正则化。Bagging (`bootstrap aggregating`) 是通过结合几个模型降低泛化误差的技术，具体来说 Bagging 步骤：

- **构造 k 个不同的数据集**，每个数据集是从原始数据集中**重复采样**构成，和原始数据集具有相同数量的样本；
- 分别用这 k 个数据集去训练网络，**得到 k 个网络模型**；
- **最终输出的结果**可以用对 k 个网络模型的输出用**加权平均法**或者**投票法**来决定。

这种策略在机器学习中被称为模型平均 (`Model Averaging`)。模型平均是一个减少泛化误差的非常强大可靠的方法，例如我们假设有 k 个回归模型, 每个模型误差是 ϵ_i ，误差服从零均值、方差为 v 、协方差为 c 的多维正态分布，则模型平均预测的误差为 $\frac{1}{k}\sum_i \epsilon_i$，均方误差的期望为

$$\mathbb{E}\left[(\frac{1}{k}\sum_i \epsilon_i)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i(\epsilon_i^2 + \sum_{i\neq j}\epsilon_i\epsilon_j)\right] = \frac{1}{k}v + \frac{k-1}{k}c \tag{18}$$

可见，在误差完全相关即 $c = v$ 的情况下，均方误差为 v，模型平均没有帮助。在误差完全不相关即 $c = 0$ 时，模型平均的均方误差的期望仅为 $\frac{1}{k}v$ ，这说明集成平方误差的期望随集成规模的增大而线性减少。

不过用集成学习 Bagging 的方法有一个问题，就是我们的网络模型本来就比较复杂，参数很多，现在参数又增加了 k 倍，从而导致训练这样的网络要花更加多的时间和空间。因此一般 k 的个数不能太多，比如 5 - 10 个就可以了。

[15]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
```

```
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[16]:
```
def bootstrap_sample(X, Y):
    N, M = X.shape
    idxs = np.random.choice(N, N, replace=True)
    return X[idxs], Y[idxs]

class BaggingModel(object):

    def __init__(self, n_models):
        """
        参数说明:
        n_models: 网络模型数目
        """
        self.models = []
        self.n_models = n_models

    def fit(self, X, Y):
        self.models = []
        for i in range(self.n_models):
            print("training {} base model:".format(i))
            X_samp, Y_samp = bootstrap_sample(X, Y)
            model = DFN(hidden_dims_1=200, hidden_dims_2=10)
            model.fit(X_samp, Y_samp)
            self.models.append(model)

    def predict(self, X):
        model_preds = np.array([[np.argmax(t.forward(x)[0]) for x in X] for t in self.models])
        return self._vote(model_preds)

    def _vote(self, predictions):
        out = [np.bincount(x).argmax() for x in predictions.T]
        return np.array(out)

    def evaluate(self, X_test, y_test):
        acc = 0.0
        y_pred = self.predict(X_test)
        y_true = np.argmax(y_test, axis=1)
        acc += np.sum(y_pred == y_true)
        return acc / X_test.shape[0]
```

[17]:
```
n_models = 3
```

[18]:
```
model = BaggingModel(n_models)
model.fit(X_train, y_train)
```

```
training 0 base model:
[Epoch 1] Avg. loss: 2.283  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 2.200  Delta: 0.083 (0.01m/epoch)
[Epoch 3] Avg. loss: 1.972  Delta: 0.229 (0.01m/epoch)
```

```
[Epoch 4] Avg. loss: 1.607  Delta: 0.365 (0.01m/epoch)
[Epoch 5] Avg. loss: 1.272  Delta: 0.335 (0.01m/epoch)
[Epoch 6] Avg. loss: 1.034  Delta: 0.238 (0.01m/epoch)
[Epoch 7] Avg. loss: 0.873  Delta: 0.161 (0.01m/epoch)
[Epoch 8] Avg. loss: 0.761  Delta: 0.112 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.681  Delta: 0.080 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.622  Delta: 0.058 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.578  Delta: 0.045 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.542  Delta: 0.036 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.513  Delta: 0.029 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.489  Delta: 0.024 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.469  Delta: 0.020 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.452  Delta: 0.017 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.437  Delta: 0.016 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.423  Delta: 0.013 (0.01m/epoch)
[Epoch 19] Avg. loss: 0.412  Delta: 0.012 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.402  Delta: 0.010 (0.01m/epoch)
training 1 base model:
[Epoch 1] Avg. loss: 2.282  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 2.201  Delta: 0.081 (0.01m/epoch)
[Epoch 3] Avg. loss: 1.974  Delta: 0.227 (0.01m/epoch)
[Epoch 4] Avg. loss: 1.606  Delta: 0.368 (0.01m/epoch)
[Epoch 5] Avg. loss: 1.269  Delta: 0.337 (0.01m/epoch)
[Epoch 6] Avg. loss: 1.033  Delta: 0.235 (0.01m/epoch)
[Epoch 7] Avg. loss: 0.873  Delta: 0.160 (0.01m/epoch)
[Epoch 8] Avg. loss: 0.762  Delta: 0.112 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.681  Delta: 0.081 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.620  Delta: 0.060 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.575  Delta: 0.046 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.538  Delta: 0.037 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.508  Delta: 0.030 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.483  Delta: 0.025 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.462  Delta: 0.021 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.445  Delta: 0.017 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.429  Delta: 0.016 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.416  Delta: 0.013 (0.01m/epoch)
[Epoch 19] Avg. loss: 0.404  Delta: 0.012 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.393  Delta: 0.011 (0.01m/epoch)
training 2 base model:
[Epoch 1] Avg. loss: 2.282  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 2.198  Delta: 0.084 (0.01m/epoch)
[Epoch 3] Avg. loss: 1.967  Delta: 0.231 (0.01m/epoch)
[Epoch 4] Avg. loss: 1.601  Delta: 0.366 (0.01m/epoch)
[Epoch 5] Avg. loss: 1.268  Delta: 0.333 (0.01m/epoch)
[Epoch 6] Avg. loss: 1.035  Delta: 0.233 (0.01m/epoch)
[Epoch 7] Avg. loss: 0.877  Delta: 0.158 (0.01m/epoch)
[Epoch 8] Avg. loss: 0.766  Delta: 0.111 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.686  Delta: 0.080 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.626  Delta: 0.060 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.579  Delta: 0.046 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.543  Delta: 0.036 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.513  Delta: 0.030 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.488  Delta: 0.025 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.467  Delta: 0.021 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.450  Delta: 0.017 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.434  Delta: 0.016 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.420  Delta: 0.014 (0.01m/epoch)
[Epoch 19] Avg. loss: 0.408  Delta: 0.012 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.398  Delta: 0.010 (0.01m/epoch)
```

[19]:
```
model.evaluate(X_test, y_test)
```

[19]: 0.8934

4.4 Dropout

Dropout 的原理为：在每个迭代过程中，**随机选择某些神经元**，并且删除它们在网络中的前向和后向连接，相当于是**“去掉”这些神经元**。如图 4 所示，在每批样本训练时，将原始网络中部分隐藏层单元“去掉”。当然，Dropout 并不意味着这些神经元永远的消失了，在下一批数据迭代前，我们会把网络恢复成最初的全连接网络，然后再用随机的方法去掉部分隐藏层的神经元，接着去迭代更新 $\boldsymbol{W}$, $\boldsymbol{b}$。Dropout 思想可以理解为每次训练时放弃部分神经元对剩下的神经元加重训练，使剩下的神经元具有更强的能力。

每个迭代过程都会有不同的神经元节点的组合，从而导致不同的输出。这可以看成机器学习中的集成方法 (`Ensemble Technique`)。集成模型一般优于单一模型，因为它们可以捕获更多的随机性；相似地，Dropout 使得神经网络模型优于正常的模型。

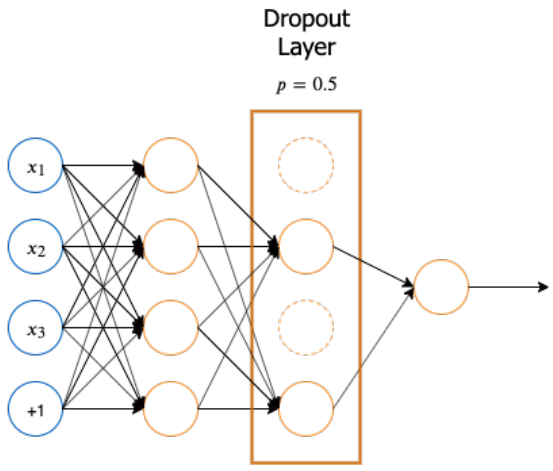


图 4. Dropout 示意图，此处 p 为 0.5，训练阶段随机“去掉”一半神经元。

Dropout 的实现步骤：

- 每次加载小批量样本，然后随机采样二值掩码，对于每个单元，掩码是独立采样的。**(将一些单元的输出乘零就能有效的删除一个单元)**；
- 传统的 Dropout，在训练阶段，用了 Dropout 的层，每个神经元以 p 的概率保留（或以 $1-p$ 的概率关闭），然后在测试阶段，不执行 Dropout，也就是所有神经元都不关闭，但是对训练阶段应用了 Dropout 的层上的神经元，为保证强度一致其输出激活值要乘以 p。**原理：为保证训练阶段与测试阶段强度一致，预测时对于每个隐层的输出，都乘以概率** p。可以从数学期望的角度去理解，我们考虑一个神经元的输出为 x (没有 Dropout 的情况下)，则它输出的数学期望为 $px+(1-p)0$，于是在测试阶段，我们直接把每个输出 x 都变换为 px，是可以保持一样的数学期望。而现在我们在训练阶段应用 Dropout 时并没有让神经元 a 的输出激活值除以 p，因此其期望值为 pa，在测试阶段如果不用 Dropout，所有神经元都保留，则输出期望值为 x，为了让测试阶段神经元的输出期望值和训练阶段保持一致 (这样才能正确评估训练出的模型)，就要给测试阶段的输出激活值乘上 p，使其输出期望值保持为 px；
- 现在主流的方法是 Inverted Dropout，和传统的 Dropout 方法有两点不同：在训练阶段，对执行了 Dropout 操作的层，其输出激活值要除以 p；测试阶段则不执行任何操作，既不执行 Dropout，也不用对神经元的输出乘 p。
- 其余部分，与之前一样，运行前向传播、反向传播和学习更新。

Dropout 不仅可以应用在隐含层，也可以应用在输入层。选择保留多少单元的概率值 p 是一个超参数。通常输入单元被保留的概率为 0.8，隐藏单元被保留的概率为 0.5。

Dropout 优点：

- 计算方便，训练过程中使用 Dropout 产生 n 个 (神经单元数目) 随机二进制数与状态相乘即可。
- 适用广 (几乎在所有使用分布式表示且可以用随机梯度下降训练的模型上都表现很好，如前馈神经网络、概率模型、受限波尔兹曼机、循环神经网络等)。
- 相比其他正则化方法 (如权重衰减、过滤器约束和稀疏激活) 更有效，也可与其他形式的正则化合并，得到进一步提升。

Dropout 缺点：

- 不适合宽度太窄的网络，否则大部分网络没有输入到输出的路径。
- 不适合训练数据太小（如小于 5000）的网络，训练数据太小时，Dropout 没有其他方法表现好。
- 不适合非常大的数据集，数据集大的时候正则化效果有限 (大数据集本身的泛化误差就很小)，使用 Dropout 的代价可能超过正则化的好处。

Dropout 与 Bagging 的比较

Dropout 模型中的参数 $\boldsymbol{W}$，$\boldsymbol{b}$ 是共享的，Dropout 下网络迭代时，更新的是同一组 $\boldsymbol{W}$，$\boldsymbol{b}$；而 Bagging 正则化中每个模型有自己的一套参数，相互之间是独立的。当然两种策略都是每次使用基于原始数据集得到的分批的数据集来训练模型。

[20]:
```
class Dropout(ABC):


    def __init__(self, wrapped_layer, p):
        """
```

```
        参数说明:
        wrapped_layer: 被 dropout 的层
        p: 神经元保留率
        """
        super().__init__()
        self._base_layer = wrapped_layer
        self.p = p
        self._init_wrapper_params()

    def _init_wrapper_params(self):
        self._wrapper_derived_variables = {"dropout_mask": None}
        self._wrapper_hyperparams = {"wrapper": "Dropout", "p": self.p}

    def flush_gradients(self):
        """
        函数作用: 调用 base layer 重置更新参数列表
        """
        self._base_layer.flush_gradients()

    def update(self):
        """
        函数作用: 调用 base layer 更新参数
        """
        self._base_layer.update()

    def forward(self, X, is_train=True):
        """
        参数说明:
        X: 输入数组;
        is_train: 是否为训练阶段, bool 型;
        """
        mask = np.ones(X.shape).astype(bool)
        if is_train:
            mask = (np.random.rand(*X.shape) < self.p) / self.p
            X = mask * X
        self._wrapper_derived_variables["dropout_mask"] = mask
        return self._base_layer.forward(X)

    def backward(self, dLda):
        return self._base_layer.backward(dLda)

    @property
    def hyperparams(self):
        hp = self._base_layer.hyperparams
        hpw = self._wrapper_hyperparams
        if "wrappers" in hp:
            hp["wrappers"].append(hpw)
        else:
            hp["wrappers"] = [hpw]
        return hp
```

[21]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用: 将数据集分割成 batch, 基于 mini batch 训练, 具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
```

```
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]
    return mb_generator(), n_batches

class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        p=1.0,
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.p = p
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = Dropout(  # 这里引入 dropout
            FullyConnected(
                n_out=self.hidden_dims_1,
                acti_fn="sigmoid",
                init_w=self.init_w,
                optimizer=self.optimizer
            ), self.p
        )
        self.layers["FC2"] = FullyConnected(
                n_out=self.hidden_dims_2,
                acti_fn="affine(slope=1, intercept=0)",
                init_w=self.init_w,
                optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train, is_train=True):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            try:  # 考虑 dropout
                out = v.forward(out, is_train=is_train)
            except:
                out = v.forward(out)
        return out, Xs

    def backward(self, grad):
```

```
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch, is_train=True)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss
```

```
    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch, is_train=False)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "dropout keep ratio": self.p,
            "components": {k: v.hyperparams for k, v in self.layers.items()}
        }
```

[22]:
```
"""
引入 dropout
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, p=0.5)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 2.286  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.215  Delta: 0.071 (0.02m/epoch)
[Epoch 3] Avg. loss: 2.017  Delta: 0.198 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.681  Delta: 0.335 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.356  Delta: 0.326 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.124  Delta: 0.231 (0.02m/epoch)
[Epoch 7] Avg. loss: 0.965  Delta: 0.160 (0.02m/epoch)
[Epoch 8] Avg. loss: 0.851  Delta: 0.114 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.779  Delta: 0.072 (0.02m/epoch)
[Epoch 10] Avg. loss: 0.718  Delta: 0.061 (0.02m/epoch)
[Epoch 11] Avg. loss: 0.677  Delta: 0.041 (0.02m/epoch)
[Epoch 12] Avg. loss: 0.643  Delta: 0.034 (0.02m/epoch)
[Epoch 13] Avg. loss: 0.615  Delta: 0.027 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.591  Delta: 0.024 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.573  Delta: 0.018 (0.02m/epoch)
[Epoch 16] Avg. loss: 0.554  Delta: 0.020 (0.02m/epoch)
[Epoch 17] Avg. loss: 0.541  Delta: 0.013 (0.02m/epoch)
[Epoch 18] Avg. loss: 0.530  Delta: 0.011 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.525  Delta: 0.005 (0.02m/epoch)
[Epoch 20] Avg. loss: 0.516  Delta: 0.009 (0.02m/epoch)
```

[23]:
```
print("accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
accuracy:0.8889
```

5 样本测试

在正则化背景下，通过对抗训练可以减少原有独立同分布的测试集的错误率——在对抗扰动的训练集样本上训练网络。

主要原因之一是高度线性，神经网络主要是基于线性模块构建的。输入改变 ϵ ，则权重为 $\boldsymbol{w}$ 的线性函数将改变 $\epsilon\|\boldsymbol{w}\|_1$ ，对于高维的 $\boldsymbol{w}$ 这是一个

非常大的数。对抗训练通过鼓励网络在训练数据附近的局部区域恒定来限制这一个高度敏感的局部线性行为。

[24]:
```
import numpy
import PIL
import matplotlib
import re

print("numpy:", numpy.__version__)
print("PIL:", PIL.__version__)
print("matplotlib:", matplotlib.__version__)
print("re:", re.__version__)
```

```
numpy: 1.14.5
PIL: 6.2.1
matplotlib: 3.1.1
re: 2.2.1
```