# Note2

代码连接：

https://github.com/Buzzy0423/2022_Winter_ML

# Optimization

## Gradient Descent

Core idea: $x_1 = x_0 - \mu f^{'}(x_0)$

With Backtracking:
$$
\begin{aligned}
&\textbf{while } \underline{f'(x_n) > \epsilon} \textbf{ do} \\
&\quad \textbf{while } \underline{f(x_n) - \mu * f'(x_n) > f(x_n) - \alpha * f'(x_n)} \textbf{ do} \\
&\quad\quad \mu = \mu * \beta; \\
&\quad \textbf{end} \\
&\quad x_{n+1} = x_n - \mu f'(x_n); \\
&\textbf{end}
\end{aligned}
$$

在训练网络的时候最好使用衰减的学习率

**Stochastic Gradient Decent**:
$$
\begin{aligned}
&\textbf{while } \underline{||W_{i+1} - W_i||_2^2 > \epsilon} \textbf{ do} \\
&\quad B \leftarrow \text{random subset of D}; \\
&\quad L(W) = \textstyle\sum_{x_j \in B} l_W(y_j, \hat{y}(x_j)); \\
&\quad W_{n+1} = W_n - \mu \nabla_W L(W); \\
&\textbf{end}
\end{aligned}
$$

即不对整个数据集求Loss而是随机选一部分子集求Loss

## Adagrad

Core idea:$\mu_i = \frac{\mu_0}{\sqrt{s(i,t)+c}}$

$s(i+1,t) = s(i,t) + (\partial_i f(x))^2$

即对每个特征的学习率进行近似的单独调整

```
optim = torch.optim.Adagrad(net.parameters(), lr=0.005, lr_decay=0, weight_decay=0)
```

# Seed

```
def seed_torch(seed=1029):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed) # 为了禁止hash随机化，使得实验可复现
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed) # if you are using multi-GPU.
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
```

为了让实验结果稳定，需要使用固定种子

# Cuda

在GPU上跑网络

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

```
x=x.cuda()#把tensor推到显卡上，网络要用的所有的tensor，lossfunc和网络自身需要在同一个硬件上CPU/GPU
```

# CNN

代码如下

```
class ConvNet(nn.Module):

  def __init__(self):
    super(ConvNet,self).__init__()

    self.feature=nn.Sequential(#Sequential相当于将操作打包，但是要注意没有异常处理
        nn.Conv2d(in_channels=1,out_channels=32,kernel_size=3,stride=1,padding=1),#size=28+2-2=28#步幅1，填充1
        nn.BatchNorm2d(num_features=32),#BatchNorm即变量归一化，使得变量分布更加均匀(接近标准正态)，利于训练
        nn.ReLU(inplace=True),

 nn.Conv2d(in_channels=32,out_channels=32,kernel_size=3,stride=1,padding=1),#size=28+2-2=28
        nn.BatchNorm2d(num_features=32),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(stride=2,kernel_size=2)#size=28*28/4=14*14
    )

    self.linear=nn.Sequential(
        nn.Linear(32*14*14,256),
        nn.BatchNorm1d(256),
        nn.ReLU(inplace=True),
        nn.Linear(256,10),
        nn.Softmax()
    )

  def forward(self, x):
    x=self.feature(x)
    x=x.view(x.size(0),-1)#压扁再输入全链接层
    x=self.linear(x)
    print(x)
    return x
```

# SVM

空间中点到超平面的距离为$r=\frac{|w^Tx+b|}{||w||}$

若有：$\begin{cases} \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b \geqslant +1, & y_i = +1\,; \\ \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b \leqslant -1, & y_i = -1\,. \end{cases}$

则两个异类支持向量之间的距离为$margin = \frac{2}{||w||}$

要使间隔最大化，即找到参数$w$和$b$使间隔最大化，即最小化$||w||^2$

若是样本在原始维度不是线性可分的，那就将样本通过核函数映射到更高维度中，直到他们线性可分

**表 6.1 常用核函数**

| 名称 | 表达式 | 参数 |
| --- | --- | --- |
| 线性核 | $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j$ | |
| 多项式核 | $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j)^d$ | $d \geqslant 1$ 为多项式的次数 |
| 高斯核 | $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}{2\sigma^2}\right)$ | $\sigma > 0$ 为高斯核的带宽(width) |
| 拉普拉斯核 | $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|}{\sigma}\right)$ | $\sigma > 0$ |
| Sigmoid 核 | $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(\beta\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + \theta)$ | tanh 为双曲正切函数, $\beta > 0$, $\theta < 0$ |

代码很简单

```
svm=SVC(kernel='rbf')
svm=svm.fit(X, y)
```

# Cluster

原理见人工智能导论笔记

代码实现(非api):

```
N=4

center=np.random.randint(0,255,N)

for i in range(5):
  center=center[np.newaxis,np.newaxis,:]
  diff=(im[:,:,np.newaxis]-center)**2
  arg=np.argmin(diff,axis=2)
  new_center=[]
  for num in np.unique(arg):
    t=im*(arg==num)
    new_center.append(np.sum(t)/np.sum(arg==num))
  center=np.array(new_center)

print(center)
cent=center[np.newaxis,np.newaxis,:]
diff=(im[:,:,np.newaxis]-cent)**2
arg=np.argmin(diff,axis=2)
new_im=center[arg]
```

```
plt.imshow(new_im)
```

代码实现(api):

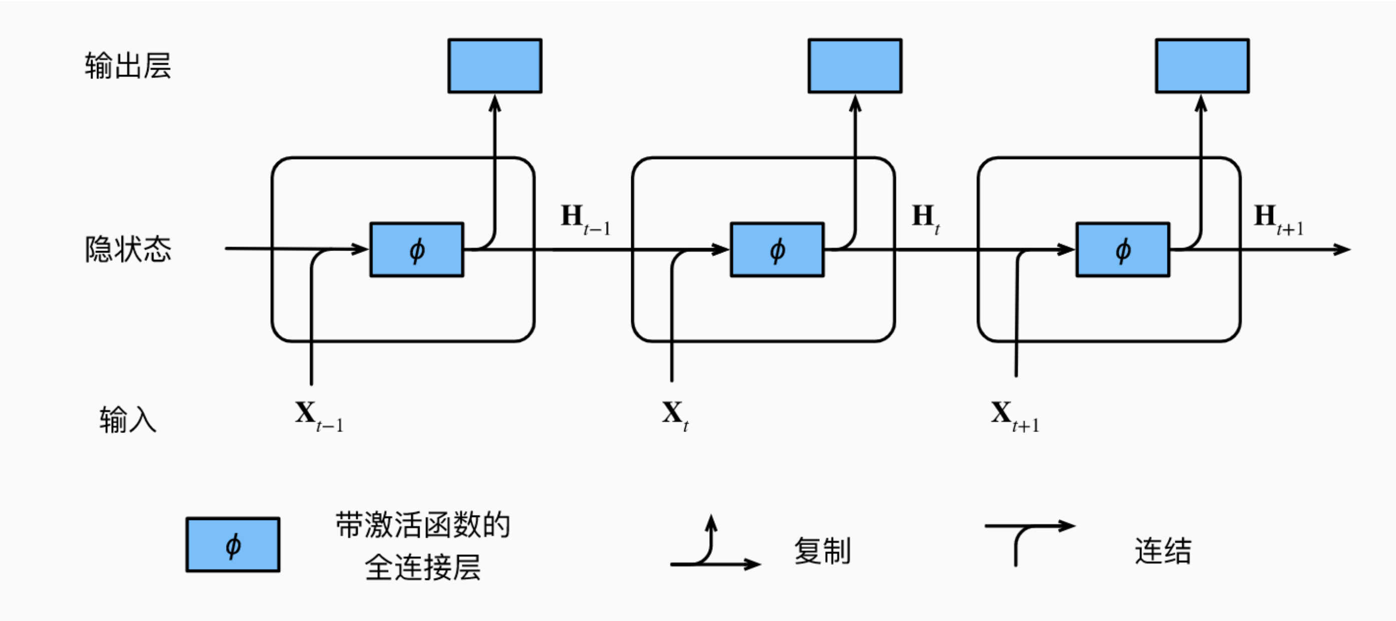
```
N=4
kmeans=cluster.KMeans(n_clusters=N)
kmeans.fit(im)
```

# NLP

马尔可夫模型：$P(x_1,\ldots,x_T)=\prod_{t=1}^{T}P(x_t|x_{t-1})$ 当$P(x_1|x_0)=P(x_1)$

NLP的原理即是条件概率，例如“树上有一只”这段话后面接“猴子”的可能性远比“房子”高

Core idea:$H_t=\phi(X_tW_{xh}+H_{t-1}W_{hh}+b_h)$




困惑度：$exp(-\frac{1}{n}\sum_{t=1}^{n}logP(x_t|x_{t-1},\ldots,x_1))$ ,用来表示下一个词元的实际选择数的调和平均数，最好为1，最坏为0

后面现代循环神经网络那章没怎么看懂。。