8.23-9.5 周报

赵晓辉

2021 年 9 月 3 日

1 写在前面

这个双周主要完成了 cs231n 课程的 Lecture 1 至 Lecture 6，主要内容包括距离函数、KNN、SVM、损失函数及优化、BP 算法、CNN 架构、非线性激活函数以及神经网络的参数优化等，并完成 cs231n assignment1。课程概要笔记及 assignment 将在https://github.com/zxh991103/cs231NOTE持续跟踪。

2 Lec 1-6 课程概要

2.1 距离函数

L_1 Distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L_2 Distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

2.2 KNN

计算测试样本与所有训练集样本之间的距离值，并根据 K 值投票选举出最相似的标签。

2.3 SVM

计算能够划分训练集样本且距离最大的超平面。

$$w \cdot x + b = 0$$

2.4 损失函数

损失函数评估模型预测值与模型真实值之间的差异性，我们要将其最小化。对于给定的训练集 $(x_i, y_i)_{i=1}^N$，我们有损失函数：

$$L = \frac{1}{N}\sum_i L_i(f(x_i, W), y_i)$$

对于 Multi-SVM，我们有损失函数，即 hinge loss：

$$s = f(x_i, W)$$

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

对于 softmax loss:

$$L_i = -log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

2.5 正则化

根据奥卡姆剃刀原则，模型越简单越符合实际，所以我们将正则惩罚项加在损失函数上。

$$L = \frac{1}{N}\sum_i L_i(f(x_i, W), y_i) + R(W)$$

L1

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

L2

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Elastic

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

2.6 BP 算法

链式法则：

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

故在计算损失函数对于参数的梯度值时，我们应当将本地梯度值与上游回传梯度值相乘。

此时，我们也可以发现 Relu 函数，即 max gate 中只有前向传播计算中的正值能影响下游。

A vectorized example：

$$f(x,W)=||W\ x||^2=\sum_{i=1}^{n}(W\ x)_i^2$$

$$q=W\cdot x$$

$$\nabla_W f=2q\cdot x^T$$

2.7 NN

假设我们将神经网络的计算图表示为：

$$f=Softmax(W_2Relu(W_1x))$$

神经元 A 拥有 W_1，能够具有识别出来 100 种特征的功能，比如识别出马的左脸或者右脸、车头或者车位。而神经元 B 拥有 W_2, 其功能就在于将马的左脸或右脸合并为马的特征，将车头或车尾合并成车的特征，从而进行识别。

2.8 CNN

卷积层：

假设有 32*32*3 的图片，卷积核 w 5*5*3 ，以及偏置 b，卷积后我们获得 28*28*1 的矩阵，其中 1 时卷积核的数量。卷积公式为（每位相乘再求和）：

$$f[x,y]*g[x,y]=\sum_{n_1=-\infty}^{\infty}\sum_{n_2=-\infty}^{\infty}f[x,y]*g[x-n_1,y-n_2]$$

步长（stride）：

假设我们有 7*7 的输入，3*3 的卷积核，2 的步长，最后的输出为 3*3。此时 outputsize $=\frac{(N-F)}{stride}+1$

填充（Pad）：

图像四周补充 0，来防止在深层卷积时张量过小。此时，outputsize $=\frac{(N-F+2P)}{stride}+1$。

Example:

input volume 32*32*3,10 5*5 filters (include 3 depth), stride 1 ,pad 2 ,wo have 760 parameters (10 *(5 * 5 *3 +1 bias)=760)

pooling layer: 相当于下采样。

maxpooling:

一般，每一个池化 filter 具有和步长相同的大小以避免 overlap.

例如，

$$\begin{matrix} 1 & 1 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{matrix}$$

我们使用 2*2 的 filter 和 2 的 stride,maxpooling 后变为：

$$\begin{matrix} 6 & 8 \\ 3 & 4 \end{matrix}$$

2.9 激活函数

若全部线性连接则等同于一个线性连接，所以网络中需要非线性的激活函数变换。

sigmod

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

tanh

$$tanh(x)$$

Relu

$$max(0, x)$$

LeakyRelu

$$max(0.1x, x)$$

Maxout

$$\max\left(w_1^T x + b_1, w_2^T x + b_2\right)$$

Elu

$$\begin{cases} x & x \geq 0 \\ \alpha\left(e^x - 1\right) & x < 0 \end{cases}$$

SELU

$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha\left(e^x - 1\right) & \text{otherwise} \end{cases}$$

softmax 的问题：

- x 过大或过小，本地梯度接近 0，使得与上游梯度乘积也接近于 0，更新缓慢。
- x 的值恒正或恒负，本地梯度总是大于 0 的，造成 w 的移动时锯齿状的，接近最优点放缓。

relu 的问题：

- 若 $w \cdot x + b$ 总是负的，则本地梯度为 0，造成参数不更新。

2.10 数据处理

```
# ZERO-CENTER
X -= np.mean(X,axis = 0)
# normalize
X /= np.std(X,axis = 0)
```

Listing 1: normalization

2.11 参数初始化

Naive: 为参数初始化小随机数。但是随着网络深度的增加，本地梯度与上游梯度相乘之后接近零，学习十分缓慢。

Xavier：

```
W = np.random.randn(dim_in,dim_out)/np.sqrt(dim_in)
```

Listing 2: Xavier

原因：

we want Var(y) = Var(x_i) , and we have

$$y = \sum_{i=1}^{Din} x_i w_i$$

and we assume that every x has same var. so we have

$$var(y) = Din \times var(x) \times var(w_i)$$

and obviously initial w_i $N(0, 1)$, we make $\frac{w_i}{\sqrt{Din}}$ to achieve the var is $\frac{1}{Din}$

Kaiming/MSRA:

```
   W = np.random.randn(dim_in,dim_out)*np.sqrt(2/dim_in)
```

Listing 3: MSRA

2.12 Batch Normalization

we have the input x like N×D

$$\widehat{x}^{(k)}=\frac{x^{(k)}-\mathrm{E}\left[x^{(k)}\right]}{\sqrt{\operatorname{Var}\left[x^{(k)}\right]}}$$

so that we have:

$$\mu_j=\frac{1}{N}\sum_{i=1}^{N}x_{i,j}$$

$$\sigma_j^2=\frac{1}{N}\sum_{i=1}^{N}\left(x_{i,j}-\mu_j\right)^2$$

$$\hat{x}_{i,j}=\frac{x_{i,j}-\mu_j}{\sqrt{\sigma_j^2+\varepsilon}}$$

the net is supposed to learn $\gamma\in R^D$ and $\beta\in R^D$

$$y_{i,j}=\gamma_j\hat{x}_{i,j}+\beta_j$$

BN 层经常用于全连接或卷积层后。

2.13 Norm For Conv.

batch norm , 对于每一个 batch 中的每个 channel 取平均得到 (1× 1 × C)

layer norm , 对于所有的 batch 我们取一个平均的图片 (H × W× C)

instance norm , 对于所有的 batch 我们将取一个平均的单通道图片 (H × W ×1)

group norm , 对于所有 batch，我们将 channel 分为 k 个组，在每个组山上取平均值得到 (H × W × k)

3 Assignment1

3.1 KNN

双循环实现距离矩阵:

```
num_test = X.shape[0]
num_train = self.X_train.shape[0]
dists = np.zeros((num_test, num_train))
for i in range(num_test):
    for j in range(num_train):
        t1 = X[i]
```

```python
            t2 = self.X_train[j]
            t = t1 - t2
            t = t*t
            t = t.sum()
            t = t**0.5
            dists[i][j] = t
```

Listing 4: KNN 双循环实现距离矩阵

单循环实现距离矩阵:

```python
num_test = X.shape[0]
num_train = self.X_train.shape[0]
dists = np.zeros((num_test, num_train))
for i in range(num_test):
    dists[i,:] = np.sum((X[i,:]-self.X_train)**2,axis = 1)
```

Listing 5: KNN 单循环实现距离矩阵

向量操作实现距离矩阵（矩阵下的完全平方公式）:

```python
M = np.dot(X, self.X_train.T)
nrow=M.shape[0]
ncol=M.shape[1]
te = np.diag(np.dot(X,X.T))
tr = np.diag(np.dot(self.X_train,self.X_train.T))
te= np.reshape(np.repeat(te,ncol),M.shape)
tr = np.reshape(np.repeat(tr, nrow), M.T.shape)
sq=-2 * M +te+tr.T
dists = np.sqrt(sq)
```

Listing 6: KNN 向量操作实现距离矩阵

KNN 预测:

```python
num_test = dists.shape[0]
y_pred = np.zeros(num_test)
labelnum = len(np.unique(self.y_train))
for i in range(num_test):
    # A list of length k storing the labels of the k nearest neighbors
to
    # the ith test point.
    closest_y = []
    closest_y = np.argsort(dists[i,:])[:k]
    t = np.zeros(labelnum,dtype=int)
    for j in closest_y:
        t[self.y_train[j]]+=1
    y_pred[i] = t.argmax()
```

Listing 7: KNN 预测

3.2 SVM

计算 SVM loss、dW(naive):

```
dW = np.zeros(W.shape)  # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
    scores = X[i].dot(W)
    correct_class_score = scores[y[i]]
    for j in range(num_classes):
        if j == y[i]:
            continue
        margin = scores[j] - correct_class_score + 1  # note delta = 1
        if margin > 0:
            loss += margin
loss /= num_train

# Add regularization to the loss.
loss += reg * np.sum(W * W)

for i in range(num_train):
    scores = X[i].dot(W)
    correct_class_score = scores[y[i]]
    for j in range(num_classes):
        if j == y[i]:
            continue
        margin = scores[j] - correct_class_score + 1  # note delta = 1
        if margin > 0:
            dW[:,j] += X[i]
            dW[:,y[i]] -= X[i]
dW /= num_train
dW += reg * W * 2
```

Listing 8: SVM(naive)

计算 SVM loss、dW(vectorized):

```
loss = 0.0
dW = np.zeros(W.shape)  # initialize the gradient as zero
N = X.shape[0]
scores = X.dot(W)
score_yi = scores[range(N),y].reshape(-1,1)
t = scores - score_yi + 1
t[range(N),y] = 0
condition = (t>0).astype(int)
t = condition*t
t = t.sum() / N
loss = t + 2 * reg * np.sum(W * W)
```

```

condition[range(N), y] = - np.sum(condition, axis = 1)
dW += np.dot(X.T,condition)/N + 2 * reg * W
```

Listing 9: SVM(vectorized)

训练线性分类器（batch）：

```
num_train, dim = X.shape
num_classes = (
    np.max(y) + 1
)  # assume y takes values 0...K-1 where K is number of classes
if self.W is None:
    # lazily initialize W
    self.W = 0.001 * np.random.randn(dim, num_classes)

# Run stochastic gradient descent to optimize W
loss_history = []
for it in range(num_iters):
    X_batch = None
    y_batch = None
    indices = np.random.choice(num_train,batch_size)
    X_batch = X[indices]
    y_batch = y[indices]

    # evaluate loss and gradient
    loss, grad = self.loss(X_batch, y_batch, reg)
    loss_history.append(loss)

    # perform parameter update
    self.W -= learning_rate*grad
```

Listing 10: 训练线性分类器

SVM 预测：

```
y_pred = np.argmax(np.dot(X,self.W),axis = 1)
```

Listing 11: SVM 预测

SVM grid search：

```
for i in learning_rates:
for j in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train, y_train, learning_rate=i, reg=j,
                  num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    y_val_pred = svm.predict(X_val)
    y_train_acc = np.mean(y_train == y_train_pred)
    y_val_acc = np.mean(y_val == y_val_pred)
    results[(i,j)] = (y_train_acc,y_val_acc)
    if y_val_acc > best_val:
```

```
            best_val = y_val_acc
            best_svm = svm
```

Listing 12: SVM grid search

3.3 softmax

Softmax loss、dW(Naive):

```
loss = 0.0
dW = np.zeros_like(W)

N , D = X.shape
C = W.shape[1]

for i in range(N):
    f = np.dot(X[i],W)
    f -= np.max(f) # avoid overflow
    loss = loss + np.log(np.sum(np.exp(f))) - f[y[i]]
    dW[:, y[i]] -= X[i]
    s = np.exp(f).sum()
    for j in range(C):
        dW[:, j] += np.exp(f[j]) / s * X[i]
loss = loss / N +  reg * np.sum(W * W)
dW = dW / N + 2*reg * W
```

Listing 13: Softmax(Naive)