

# 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：

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

## 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:

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

$$q = W \cdot x$$

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

## 2.7 NN

假设我们将神经网络的计算图表示为:

$$f = \text{Softmax}(W_2 \text{Relu}(W_1 x))$$

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、车头或者车尾。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  $32 \times 32 \times 3$  的图片, 卷积核  $w \ 5 \times 5 \times 3$ , 以及偏置  $b$ , 卷积后我们获得  $28 \times 28 \times 1$  的矩阵, 其中 1 是卷积核的数量。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  $7 \times 7$  的输入,  $3 \times 3$  的卷积核, 2 的步长, 最后的输出为  $3 \times 3$ 。  
此时  $\text{outputsize} = \frac{(N-F)}{\text{stride}} + 1$

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,  $\text{outputsize} = \frac{(N-F+2P)}{\text{stride}} + 1$ 。

Example:

input volume  $32 \times 32 \times 3$ , 10  $5 \times 5$  filters (include 3 depth), stride 1, pad 2, we have 760 parameters ( $10 * (5 * 5 * 3 + 1 \text{ bias}) = 760$ )

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,

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

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

6	8
3	4

## 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

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

Elu

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

SELU

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

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，接近最优值放缓。

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

```
1 # ZERO-CENTER
2 X -= np.mean(X,axis = 0)
3 # normalize
4 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:

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

Listing 2: Xavier

原因:

we want  $\text{Var}(y) = \text{Var}(x_i)$  , and we have

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

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

$$\text{var}(y) = D_{in} \times \text{var}(x) \times \text{var}(w_i)$$

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

Kaiming/MSRA:

```
1 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  $N \times D$

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

so that we have:

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

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

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

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  $(1 \times 1 \times C)$

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

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

group norm , 对于所有 batch, 我们将 channel 分为  $k$  个组, 在每个组上取平均值得到  $(H \times W \times k)$

# 3 Assignment1

## 3.1 KNN

双循环实现距离矩阵:

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

```

7         t2 = self.X_train[j]
8         t = t1 - t2
9         t = t*t
10        t = t.sum()
11        t = t**0.5
12        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：

```

1    num_test = X.shape[0]
2    num_train = self.X_train.shape[0]
3    dists = np.zeros((num_test, num_train))
4    for i in range(num_test):
5        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）：

```

1    M = np.dot(X, self.X_train.T)
2    nrow=M.shape[0]
3    ncol=M.shape[1]
4    te = np.diag(np.dot(X,X.T))
5    tr = np.diag(np.dot(self.X_train,self.X_train.T))
6    te= np.reshape(np.repeat(te,ncol),M.shape)
7    tr = np.reshape(np.repeat(tr, nrow), M.T.shape)
8    sq=-2 * M +te+tr.T
9    dists = np.sqrt(sq)

```

Listing 6: KNN 向量操作实现距离矩阵

KNN 预测：

```

1    num_test = dists.shape[0]
2    y_pred = np.zeros(num_test)
3    labelnum = len(np.unique(self.y_train))
4    for i in range(num_test):
5        # A list of length k storing the labels of the k nearest neighbors
6        # to
7        # the ith test point.
8        closest_y = []
9        closest_y = np.argsort(dists[i,:])[:k]
10       t = np.zeros(labelnum,dtype=int)
11       for j in closest_y:
12           t[self.y_train[j]]+=1
13       y_pred[i] = t.argmax()

```

Listing 7: KNN 预测

## 3.2 SVM

计算 SVM loss、dW(naive):

```
1 dW = np.zeros(W.shape) # initialize the gradient as zero
2
3 # compute the loss and the gradient
4 num_classes = W.shape[1]
5 num_train = X.shape[0]
6 loss = 0.0
7 for i in range(num_train):
8     scores = X[i].dot(W)
9     correct_class_score = scores[y[i]]
10    for j in range(num_classes):
11        if j == y[i]:
12            continue
13        margin = scores[j] - correct_class_score + 1 # note delta = 1
14        if margin > 0:
15            loss += margin
16    loss /= num_train
17
18 # Add regularization to the loss.
19 loss += reg * np.sum(W * W)
20
21 for i in range(num_train):
22     scores = X[i].dot(W)
23     correct_class_score = scores[y[i]]
24     for j in range(num_classes):
25         if j == y[i]:
26             continue
27         margin = scores[j] - correct_class_score + 1 # note delta = 1
28         if margin > 0:
29             dW[:,j] += X[i]
30             dW[:,y[i]] -= X[i]
31 dW /= num_train
32 dW += reg * W * 2
```

Listing 8: SVM(naive)

计算 SVM loss、dW(vectorized):

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



```

12
13 condition[range(N), y] = - np.sum(condition, axis = 1)
14 dW += np.dot(X.T, condition)/N + 2 * reg * W

```

Listing 9: SVM(vectorized)

训练线性分类器 (batch):

```

1 num_train, dim = X.shape
2 num_classes = (
3     np.max(y) + 1
4 ) # assume y takes values 0...K-1 where K is number of classes
5 if self.W is None:
6     # lazily initialize W
7     self.W = 0.001 * np.random.randn(dim, num_classes)
8
9 # Run stochastic gradient descent to optimize W
10 loss_history = []
11 for it in range(num_iters):
12     X_batch = None
13     y_batch = None
14     indices = np.random.choice(num_train, batch_size)
15     X_batch = X[indices]
16     y_batch = y[indices]
17
18     # evaluate loss and gradient
19     loss, grad = self.loss(X_batch, y_batch, reg)
20     loss_history.append(loss)
21
22     # perform parameter update
23     self.W -= learning_rate*grad

```

Listing 10: 训练线性分类器

SVM 预测:

```

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

```

Listing 11: SVM 预测

SVM grid search:

```

1 for i in learning_rates:
2     for j in regularization_strengths:
3         svm = LinearSVM()
4         svm.train(X_train, y_train, learning_rate=i, reg=j,
5                 num_iters=1500, verbose=True)
6         y_train_pred = svm.predict(X_train)
7         y_val_pred = svm.predict(X_val)
8         y_train_acc = np.mean(y_train == y_train_pred)
9         y_val_acc = np.mean(y_val == y_val_pred)
10        results[(i,j)] = (y_train_acc, y_val_acc)
11        if y_val_acc > best_val:

```

```

12         best_val = y_val_acc
13         best_svm = svm

```

Listing 12: SVM grid search

### 3.3 softmax

Softmax loss, dW(Naive):

```

1  loss = 0.0
2  dW = np.zeros_like(W)
3
4  N , D = X.shape
5  C = W.shape[1]
6
7  for i in range(N):
8      f = np.dot(X[i],W)
9      f -= np.max(f) # avoid overflow
10     loss = loss + np.log(np.sum(np.exp(f))) - f[y[i]]
11     dW[:, y[i]] -= X[i]
12     s = np.exp(f).sum()
13     for j in range(C):
14         dW[:, j] += np.exp(f[j]) / s * X[i]
15 loss = loss / N + reg * np.sum(W * W)
16 dW = dW / N + 2*reg * W

```

Listing 13: Softmax(Naive)

Softmax loss, dW(vectorized):

```

1  N , D = X.shape
2  C = W.shape[1]
3  score = np.dot(X,W)
4  t = np.max(score,axis = 1).reshape(N,1)
5  score -= t
6  s = np.exp(score).sum(axis = 1)
7  loss = - score[range(N),y].sum() + np.log(s).sum()
8  counts = np.exp(score) / s.reshape(N, 1)
9  counts[range(N),y] -= 1
10 dW = np.dot(X.T,counts)
11
12 loss = loss/N + reg * np.sum(W * W)
13
14 dW = dW/N + reg *W

```

Listing 14: Softmax(vectorized)

$$Loss = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) = -s_{y_i} + \log \sum_j e^{s_j}$$

$$\frac{dL}{dW} = -\frac{ds_{y_i}}{dW} + \frac{1}{\sum_j e^{s_j}} \cdot \left(\sum_j (e^{s_j} \frac{dS_j}{dW})\right)$$

softmax grid search:

```
1 from cs231n.classifiers.linear_classifier import Softmax
2
3 for lr in learning_rates:
4     for rs in regularization_strengths:
5         softmax = Softmax()
6         softmax.train(X_train, y_train, learning_rate = lr, reg=rs,
7             num_iters = 1500,
8                 verbose = True)
9
10        y_pred_train = softmax.predict(X_train)
11        acc_train = np.mean(y_pred_train == y_train)
12
13        y_pred_val = softmax.predict(X_val)
14        acc_val = np.mean(y_pred_val == y_val)
15        results[(lr, rs)] = (acc_train, acc_val)
16
17        if acc_val > best_val:
18            best_val = acc_val
19            best_softmax = softmax
```

Listing 15: softmax grid search

### 3.4 FC Net

FC 单层向后传播:

```
1 N = x.shape[0]
2 D = x[0].reshape(-1,1).shape[0]
3 M = b.shape[0]
4 # print(N,D,M)
5 out = np.zeros((N,M))
6 for i in range(N):
7     t = x[i].reshape(-1,1)
8     # print(w.shape,t.shape)
9     t = np.dot(w.T,t) + b.reshape(-1,1)
10    out[i,:] = t.T
```

Listing 16: FC 单层前向传播

FC 单层向后传播:

```
1 N = x.shape[0]
2 t = x.shape
3 D = x[0].reshape(-1,1).shape[0]
4 M = b.shape[0]
5
6 x = x.reshape(N,D)
7 dx = np.dot(dout,w.T).reshape(t)
8 dw = np.dot(x.T,dout)
```

```

9      db = dout.sum(axis=0)

```

Listing 17: FC 单层向后传播

Relu 向前传播:

```

1      condition = (x>0).astype(int)
2      out = condition * x

```

Listing 18: Relu 向前传播

Relu 向后传播:

```

1      dx = (x > 0 ).astype(int) * dout

```

Listing 19: Relu 向后传播

SVM loss、dx

```

1      N,C = x.shape
2      scores = x
3      score_yi = scores[range(N),y].reshape(-1,1)
4
5      t = scores - score_yi + 1
6      t[range(N),y] = 0
7      condition = (t>0).astype(int)
8      t = condition*t
9      t = t.sum() / N
10     loss = t
11
12     condition[range(N), y] = - np.sum(condition, axis = 1)
13     # print(condition.shape)
14     dx = condition/N

```

Listing 20: SVM loss、dx

Softmax loss、dx

```

1      N,C = x.shape
2      score = x - np.max(x,axis = 1).reshape(N,1)
3      s = np.exp(score).sum(axis = 1).reshape(N,1)
4      score_yi = score[range(N),y].reshape(N,1)
5      loss = (-score_yi + np.log(s)).sum() /N
6
7
8      expscore = np.exp(score)
9
10     dx = (expscore / s).reshape(N,C)
11
12     dx[range(N),y] -=1
13
14     dx /= N

```

Listing 21: Softmax loss、dx

## Two Layer Net Loss:

```
1  scores = None
2  N = X.shape[0]
3  D = X.reshape(N,-1).shape[1]
4  C = self.C
5
6  X_new = X.reshape(N,D)
7
8  # layer 1
9  cp = X.reshape(N,D)
10 cp = np.dot(cp,self.params['W1'])
11 cp = cp + self.params['b1']
12 h = cp
13
14 # relu
15 condition1 = (cp>0).astype(int)
16 cp = condition1 * cp
17 h1 = cp
18 # layer 2
19 cp = np.dot(cp,self.params['W2'])
20 cp = cp + self.params['b2']
21 # softmax
22 scores = cp
23
24
25
26 if y is None:
27     return scores
28
29 loss, grads = 0, {}
30
31 scoresmax = np.max(scores,axis = 1).reshape(N,1)
32 scores = scores - scoresmax
33 expscores = np.exp(scores)
34 t = expscores.sum(axis =1).reshape(N,1)
35 expscores = expscores / t
36 scores_yi = expscores[range(0,N),y]
37 loss = -np.log(scores_yi).sum() /N+ 0.5*self.reg * (self.params['W1'] *
38 self.params['W1']).sum() + 0.5*self.reg * (self.params['W2'] *self.
39 params['W2']).sum()
40
41 #         print(loss)
42
43 '''gradient'''
44 # Loss
45 dscore = (expscores).reshape(N,C)
46 dscore[range(N),y] -= 1
47 dscore /= N
48
49 # f2
```

```

48     dw2 = np.dot(h1.T , dscore)
49     db2 = np.sum(dscore,axis = 0)
50     dh1 = np.dot(dscore,self.params['W2'].T)
51
52     # relu
53
54     dh = (h>0).astype(int)
55     dh = dh * dh1
56
57     # f1
58
59     dw1 = np.dot(X_new.T,dh)
60     db1 = np.sum(dh,axis = 0)
61
62     #         dw1 = dw1.reshape(self.params['W1'].shape)
63     #         dw2 = dw2.reshape(self.params['W2'].shape)
64
65     dw1 += self.reg * self.params['W1']
66     dw2 += self.reg * self.params['W2']
67
68     grads['W1']=dw1
69     grads['b1']=db1
70     grads['W2']=dw2
71     grads['b2']=db2

```

Listing 22: Two Layer Net Loss

sgd

```

1     w -= config['learning_rate'] * dw

```

Listing 23: sgd

FC Net grid search:

```

1     input_size = 32 * 32 * 3
2     num_classes = 10
3     best_acc = -1
4     for bs in [200, 400]:
5         for lr in [1e-3, 1e-4, 1e-5]:
6             for hidden_size in [50, 100, 200]:
7                 net = TwoLayerNet(input_size, hidden_size, num_classes)
8                 solver = Solver(net, data,
9                                 num_train_samples=100,
10                                lr_decay=0.9,
11                                num_epochs=20,
12                                print_every=50000,
13                                batch_size = bs,
14                                optim_config={
15                                    'learning_rate': lr,
16                                })
17

```

```

18         solver.train()
19
20         # Predict on the validation set
21         val_acc = solver.check_accuracy(data['X_val'],data['y_val'])
22         print ('batch_size = %d, lr = %f, hidden size = %f,
Valid_accuracy: %f' %(bs, lr, hidden_size, val_acc))
23         if val_acc > best_acc:
24             best_acc = val_acc
25             best_model = net

```

Listing 24: FC Net grid search