103590450 四資四 馬茂源

```
[18]  import numpy as np
      from cvxopt import matrix, solvers
```

1 . We mentioned that the parameters γ and β in batch normalization are trained through backprop. Explain why we don't want to directly compute these two parameters from the training samples and also give the details how the training is carried out.

γ and β是根據model training時在經過batch norm後所學習的數值跟input沒有關係，γ, β目的是為了反轉換來學習μ, σ的影響力，如果γ, β跟μ, σ一樣(抵銷)就是代表μ, σ沒有影響力，γ, β是經由model學習出來的，我們在訓練之前無從得知這兩參數實際的數值。

學習時候神經元輸出經過batch norm轉換再經過γ and β反轉換一次最後丟入activation func，backprop修正時先從activation func開始接著γ and β之後W參數。

2 . We use the "Reparameterization Trick" in training Variational AutoEncoder (VAE). Explain why this trick is necessary and how it is accomplished.

因為在VAE，如果要讓model可以無中生有，中間的latent space的分布很重要，但一般來說train set mapping不到完整的space，因此一個單純的AE在被訓練完成後，通常會一直產生之前train set的東西，無法無中生有是因為train set很少涵蓋整個feature space因此latent space不會是continuous, 這對分類是好事但對generation可不是好現象。我們希望decoder是吃 random sampling出來的東西，但神經網路**backpropogation cannot flow through random node**，若要克服此問題可以使用reparameterization trick，我們以μ, σ來代替我們學習normal distribution特性，μ, σ 由encoder部分學習，decoder負責解開sampling，μ代表dim的方向而σ分布大小，所以decoder不再是從單一point來學習而是從機率分布來看latent feature，即使decoder只看過某些點，但這些點周圍的區域依然可以refer出來差不多的東西。

以keras為例

```
# build your encoder upto here. It can simply be a series of dense layers,
a convolutional network
# or even an LSTM decoder. Once made, flatten out the final layer of the
encoder, call it hidden.

# we use Keras to build the graph

latent_size = 5
mean = Dense(latent_size)(hidden)
```

```
# we usually don't directly compute the stddev σ
# but the log of the stddev instead, which is log(σ)
# the reasoning is similar to why we use softmax, instead of directly
outputting
# numbers in fixed range [0, 1], the network can output a wider range of
numbers which we can later compress down
log_stddev = Dense(latent_size)(hidden)

def sampler(mean, log_stddev):
    # we sample from the standard normal a matrix of batch_size *
latent_size (taking into account minibatches)
    std_norm = K.random_normal(shape=(K.shape(mean)[0], latent_size),
mean=0, stddev=1)
    # sampling from Z~N(μ, σ^2) is the same as sampling from μ + σX,
X~N(0,1)
    return mean + K.exp(log_stddev) * std_norm

latent_vector = Lambda(sampler)([mean, log_stddev])
# pass latent_vector as input to decoder layers
```

3 . Use the equations of optimal margin (linear) SVM (in pp. 12) to find w given $X_1 = [1 \; -1]^T \in C_{+1}$ and $X_2 = [-1 \; -1]^T \in C_{-1}$.

[19]
```
P = matrix([[2., 0], [0., 2.]])
q = matrix([-1., -1.])
```

[20]
```
sol = solvers.qp(P, q)
```

[21]
```
print(sol['x'])
```

```
[ 5.00e-01]
[ 5.00e-01]
```

4 . Implement a discrete HMM training program. Use the three-urn example (in pp. 12 of the PPT file) to test your program and produce the training results after 100 iterations. Use the red and blue balls in each urn to compute the initial emission probability. The initial transition probability and .

[22]
```
N = 3
O = [0,0,1,0,1]
A = [[1/3, 1/3, 1/3], [1/3, 1/3, 1/3], [1/3, 1/3, 1/3]]
pi = [1/3, 1/3, 1/3]
B = [[4/6, 2/6], [2/6, 4/6], [3/6, 3/6]]
symbols = [0, 1]

def forward(N, O, pi, A, B):
    fwd = [{}]
    for i in range(N):
```

```
            fwd[0][i] = pi[i] * B[i][O[0]]
        for t in range(1, len(O)):
            fwd.append({})
            for j in range(N):
                fwd[t][j] = sum((fwd[t-1][i] *
                                 A[i][j] *
                                 B[j][O[t]]) for i in range(N))

        prob = sum((fwd[len(O) - 1][i]) for i in range(N))
        return prob, fwd

    def backward(N, O, A, B):
        T = len(O)
        bwk = [{} for t in range(T)]
        for i in range(N):
            bwk[T-1][i] = 1
        for t in reversed(range(T-1)):
            for i in range(N):
                bwk[t][i] = sum((bwk[t+1][j] * A[i][j] *
                                 B[j][O[t+1]]) for j in range(N))
        prob = sum((pi[i]* B[i][O[0]] *
                    bwk[0][i]) for i in range(N))
        return prob, bwk

    def train(N, O, pi, A, B, symbols):
        T = len(O)
        gamma = [{} for t in range(T)]
        zi = [{} for t in range(T - 1)]
        p_o, fwd = forward(N, O, pi, A, B)
        _, bwk = backward(N, O, A, B)

        for t in range(T):
            for i in range(N):
                gamma[t][i] = (fwd[t][i] * bwk[t][i]) / p_o
                if t == 0:
                    pi[i] = gamma[t][i]
                if t == T - 1:
                    continue
                zi[t][i] = {}
                for j in range(N):
                    zi[t][i][j] = (fwd[t][i] * A[i][j] *
                    B[j][O[t + 1]] * bwk[t + 1][j] / p_o)


        for i in range(N):
            for j in range(N):
                val = sum([zi[t][i][j] for t in range(T - 1)])
                val /= sum([gamma[t][i] for t in range(T - 1)])
                A[i][j] = val

        for i in range(N):
            for k in symbols:
```

```
                val = 0.0
                for t in range(T):
                    if O[t] == k:
                        val += gamma[t][i]
                val /= sum([gamma[t][i] for t in range(T)])
                B[i][k] = val
    return pi, A, B

for iter_ in range(100):
    pi, A, B = train(N, O, pi, A, B, symbols)

print('π:', pi)
print('A:', A)
print('B:', B)
```

```
π: [1.0, 0.0, 0.0]
A: [[0.0, 0.0, 1.0], [0.0, 0.0, 1.0], [0.0, 1.0, 0.0]]
B: [[1.0, 0.0], [0.0, 1.0], [1.0, 0.0]]
```

5 . Assuming that the following is a part of subpixel convolution networks with stride . Compute the resultant values with the ReLU activation function.

[23]
```
h_range, h_range = 11, 11
```

[24]
```
k = np.array([[[3], [-1], [2]],
              [[-2], [1], [-3]],
              [[-2], [0], [3]]])
p = np.zeros((13, 13, 1))
```

[25]
```
p[2:-2, 2:-2, [0]] = np.array([[[6], [0], [0],  [0],  [-4], [0], [0],  [0
                                [[0],  [0], [0],  [0],  [0],  [0], [0],  [0
                                [[4],  [0], [4],  [0],  [0],  [0], [2],  [0
                                [[0],  [0], [0],  [0],  [0],  [0], [0],  [0
                                [[3],  [0], [-7], [0],  [1],  [0], [4],  [0
                                [[0],  [0], [0],  [0],  [0],  [0], [0],  [0
                                [[-2], [0], [2],  [0],  [1],  [0], [-4], [0
                                [[0],  [0], [0],  [0],  [0],  [0], [0],  [0
                                [[5],  [0], [1],  [0],  [2],  [0], [4],  [0
```

[26]
```
out = np.zeros((h_range, h_range))
```

[27]
```
nf = 1  # number of filters
rf = 3  # filter size
s = 1
```

[28]
```
for z in range(nf):
    h_range = int((p.shape[1] - rf) / s) + 1  # (W - F + 2P) / S
    for _h in range(h_range):
        w_range = int((p.shape[0] - rf) / s) + 1  # (W - F + 2P) / S
```

```
            for _w in range(w_range):
                # print(_h, _w)
                # print(np.sum((p[_h:_h+rf, _w:_w+rf, :]*k[:, :, :])))
                out[_h:, _w] = np.maximum((np.sum((p[_h:_h+rf, _w:_w+rf, :]*k
```

[29]
```
out
```

```
array([[18.,  0.,  0.,  0.,  0.,  0.,  8.,  0.,  3.,  0.,  0.],
       [ 0.,  6.,  0.,  0., 12.,  0.,  8.,  0.,  0.,  1.,  0.],
       [24.,  0., 22.,  0.,  0.,  4.,  0.,  0.,  1.,  0.,  1.],
       [ 0.,  4.,  0.,  4.,  0.,  0.,  0.,  2.,  0.,  1.,  0.],
       [17.,  0.,  0.,  0., 29.,  0., 14.,  0.,  6.,  0.,  0.],
       [ 0.,  3., 15.,  0., 11.,  1.,  0.,  4.,  0.,  2.,  0.],
       [ 0.,  0.,  5.,  7.,  0.,  0.,  0.,  0., 30.,  0.,  2.],
       [ 6.,  0.,  0.,  2.,  0.,  1., 10.,  0.,  2.,  2.,  0.],
       [11.,  2.,  0.,  0., 12.,  0.,  3.,  4.,  0.,  0.,  8.],
       [ 0.,  5.,  0.,  1.,  0.,  2.,  0.,  4.,  0.,  0.,  2.],
       [10.,  0., 17.,  0.,  7.,  0., 14.,  0., 10.,  1.,  0.]])
```

[ ]