

2020_lab_softmax

October 15, 2020

1 Softmax Regression

```
[1]: import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import jax.numpy as jnp
from jax import grad
from jax import random
```

Assignment Project Exam Help

1.0.1 Linear function for score

For data point \mathbf{x} , the score is $\mathbf{w}_k^\top \mathbf{x}$. The predicted probability is

$$\hat{y}_k = \exp(a_k) /$$

Below we give different implementations of the probability. First, in **soft** all the components appearing explicitly. Next, in **softmax** vectorised form. Note that exponentiation can cause over/under flow problems. There is a fix that I have introduced that relies on

$$\hat{y}_k = \exp(a_k - A) / \sum_i \exp(a_i - A)$$

for any A .

```
[2]: def softmax_prob_forloop(W, b, inputs): # output is datalen-by-C (NumPy, no JAX)
    # inputs is dim-by-datalen
    # b is C-dimensional vector W is (C-by-dim)
    dim, datalen = np.shape(inputs) # how many dimensions, points
    c = len(b) # number of classes, C, each class has a bias
    score = np.zeros((c, datalen))
    for ci in range(c):
        for lj in range(datalen):
            score[ci, lj] = b[ci]
            for dk in range(dim):
                score[ci, lj] += W[ci, dk]*inputs[dk, lj]
    maxes = np.zeros(datalen)
```

```

for lj in range(datalen):
    maxes[lj] = np.max(score[:, lj])
for ci in range(c):
    for lj in range(datalen):
        score[ci, lj] = score[ci, lj] - maxes[lj]
# subtract off the largest score from the bias of each class
# This is for stability to underflow/overflow when exponentiating
expscore = np.exp(score)
norm_factor = np.diag(1/np.sum(expscore, axis=0))
return np.dot(expscore, norm_factor).T

```

below we convert the same steps into vector form, hence no for loops

```

def softmax_prob1(W, b, inputs): # output is datalen-by-C
    # inputs is dim-by-datalen
    # b is C-dimensional vector W is (C-by-dim)
    # Make sure all numerical operations are from JAX, so 'jnp', not 'np'
    datalen = jnp.shape(inputs)[1] # how many points
    c = len(b) # number of classes, C, each class has a bias
    linear_part = jnp.dot(W, inputs) # (C-by-dim)*(dim-by-datalen) = C-by-datalen
    large = jnp.m
    # subtract off the la
    # to underflow/overflow
    large_offset = jnp.dot(np.ones(c, datalen)
    # (C-by-L)*(L-by-L)
    expscore = jnp.exp(linear_part + bias_offset - large_offset)
    norm_factor = jnp.diag(1/jnp.sum(expscore, axis=0))
    return jnp.dot(expscore, norm_factor).T

```

In what follows, the trick of setting the zeroth feature to be 1 is used to absorb the constant w_0 into the dot product. Redefine the input data to be

$$\mathbf{x} = (x_1, \dots, x_p) \longrightarrow \mathbf{x} = (1, x_1, \dots, x_p).$$

Correspondingly redefining the weight vectors to be $\mathbf{w} = (w_0, w_1, \dots, w_p)$, we have:

$$w_{k0} + \mathbf{w}_k^T \mathbf{x} \longrightarrow \mathbf{w}_k^T \mathbf{x}.$$

Thus the **softmax_prob** below has all the weights packaged into a matrix W as in the lecture slides.

```

[3]: def softmax_prob(W, inputs):
    # output is datalen-by-C
    # inputs is (dim)-by-datalen
    # W is C-by-(dim+1)
    # Make sure all numerical operations are from JAX, so 'jnp', not 'np'

```

```

datalen = jnp.shape(inputs)[1] # how many points
c = len(W) # number of classes, C, each class has a bias
inputs = jnp.concatenate((jnp.ones((1,datalen)), inputs), axis=0)
# create inputs (dim+1)-by-datalen
score = jnp.dot(W,inputs)
# (C-by-(1+dim))*((1+dim)-by-datalen) = C-by-datalen
large = jnp.max(score, axis=0) # largest of the class scores for each data
→point
# subtract off the largest score from the bias of each class for stability
→to underflow/overflow
large_offset = jnp.dot(np.ones((c, datalen)),jnp.diag(large)) #
→(C-by-L)*(L-by-L)
expscore = jnp.exp(score - large_offset)
norm_factor = jnp.diag(1/jnp.sum(expscore, axis=0))
return jnp.dot(expscore, norm_factor).T

```

```

[4]: def softmax_xentropy(Wb, inputs, targets, num_classes):
    epsilon = 1e-8
    ys = get_one_hot(targets, num_classes)
    logprobs = jnp.log(softmax_prob(Wb, inputs)+epsilon)
    return jnp.mean(ys*logprobs)

```

```

[5]: def get_one_hot(targets, num_classes):
    res = jnp.eye(num_classes)[targets]
    return res.re

```

```

[6]: Wb = jnp.array([[[-1., 1.3, 2.0, -0.], [1.5, 2.0, 1.0, 2.
→5], [3., 4.0, 4.0, -2.5]])]
# Build a toy dataset: 6 3-dim points with C=4 targets dim-by-datalen
inputs = jnp.array([[0.52, 1.12, 0.77],
                    [3.82, -6.11, 3.15],
                    [0.88, -1.08, 0.15],
                    [0.52, 0.06, -1.30],
                    [0.74, -2.49, 1.39],
                    [0.14, -0.43, -1.69]]) # transpose to make it a
→dim-by-datalen array
targets = jnp.array([0, 1, 3, 2, 1, 2])

```

```

/opt/anaconda3/lib/python3.8/site-packages/jax/lib/xla_bridge.py:130:
UserWarning: No GPU/TPU found, falling back to CPU.
warnings.warn('No GPU/TPU found, falling back to CPU.')

```

```

[7]: # Initialize random model coefficients
key = random.PRNGKey(0)
key, W_key= random.split(key, 2)
[classes, dim] = 4, 3
Winit = random.normal(W_key, (classes, dim+1))

```

```
print(Winit)
```

```
[[ 0.20820066 -1.0580499 -0.29374585 -0.4411725 ]
 [ 0.2366984 -0.03426386 -1.002556  1.1560111 ]
 [-0.5381381 -0.48968917  0.24939033 -1.4128866 ]
 [ 1.8543104  0.22756499  0.49751544 -2.089685  ]]
```

1.1 Automatic Differentiation used here

Here, we will not explicitly define what the exact form of the gradient of the cross entropy loss function is. Recall, for linear regression, we computed the gradient and used it to reduce the loss. In this next code block, we will invoke

```
grad(softmax_xentropy, (0))(W1, inputs, targets, num_classes)
```

where the (0) is shorthand for `argnums=0` which indicates that we take the gradient with respect to the first (using python's indexing convention of starting from 0) of the arguments of **softmax_entropy**. How this is done will be explored in another lab sheet.

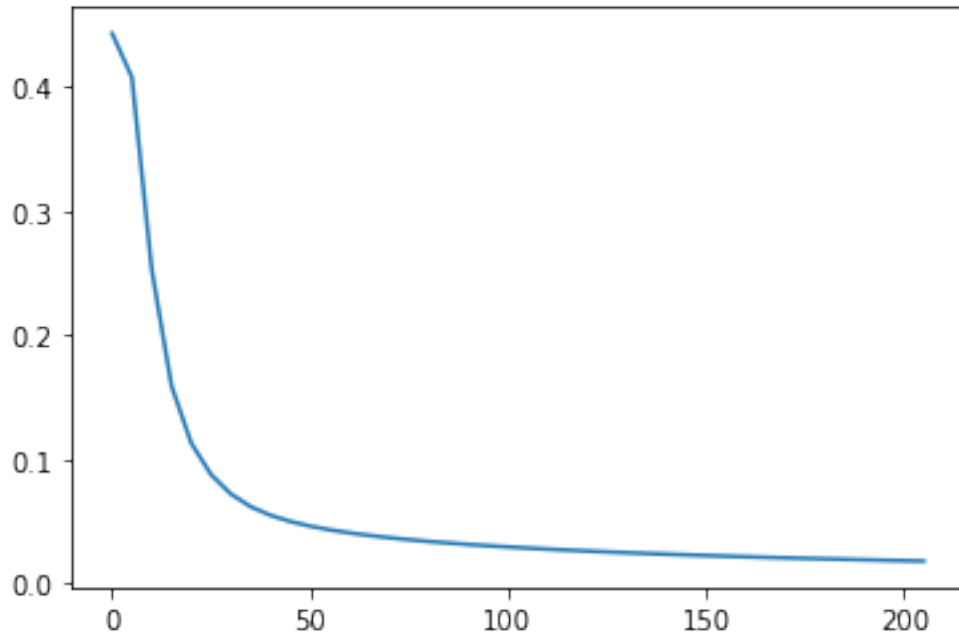
```
[8]: def grad_descent(Wb, inputs, targets, num_classes, lrate, nsteps):
      W1 = Wb
      Whist = [W1]
      losshist = [softmax_xentropy(W1, inputs, targets, num_classes)]
      eta = lrate
      for i in range(nsteps):
          gWb = grad(softmax_xentropy, (0))(W1, inputs, targets, num_classes)
          W1 = W1 - eta*gWb
          if (i%5 == 0):
              Whist.append(W1)
              losshist.append(softmax_xentropy(W1, inputs, targets, num_classes))
      Whist.append(W1)
      losshist.append(softmax_xentropy(W1, inputs, targets, num_classes))
      return W1, Whist, losshist
```

```
[9]: W2, Whist, losshist = grad_descent(Winit, inputs, targets, 4, 0.75, 200)
```

Loss history Now that we have the initial weights `Winit`, the history of weights and the history of losses, we can see how the loss function reduces as a function of iteration step. You should experiment with different learning rates and iteration steps, etc.

```
[10]: plt.plot([5*i for i in range(len(losshist))], losshist)
```

```
[10]: [ <matplotlib.lines.Line2D at 0x7ff0adbe69a0> ]
```



Assignment Project Exam Help

Compare the predictions w
produced as the predicted p
iterations) prediction an

<https://eduassistpro.github.io/>

```
[11]: print('From:\n',np.around(softmax_prob(Winit
print('To:\n',np.around(softmax_prob(W2,input
print('Target:\n',get_one_hot(targets,'4'))
```

Add WeChat edu_assist_pro

From:

```
[[0.09  0.243 0.05  0.618]
 [0.    1.    0.    0.   ]
 [0.073 0.507 0.028 0.392]
 [0.011 0.002 0.025 0.962]
 [0.008 0.99  0.    0.002]
 [0.013 0.001 0.028 0.957]]
```

To:

```
[[0.951 0.009 0.006 0.034]
 [0.    0.996 0.    0.004]
 [0.025 0.13  0.043 0.802]
 [0.004 0.    0.949 0.047]
 [0.001 0.926 0.    0.074]
 [0.    0.    0.979 0.021]]
```

Target:

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
```

```
[0. 0. 1. 0.]  
[0. 1. 0. 0.]  
[0. 0. 1. 0.]]
```

1.1.1 Your turn:

Create your own input data and targets. You may choose them to be random. For instance, in numpy `np.random.normal(mean, std_dev, (dim, datalen))` will create a set of datalen inputs of dimension dim, drawn from a normal distribution of a chosen mean and standard deviation. You must generate the Winit from `jax.numpy` in order to be able to use the gradient. Experiment with different learning rates, and see what you find.

[]:

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu_assist_pro