Aditya N Bhatt

> 231057015

> DL Assignment-1

> 05-01-2024

> github link: https://github.com/adityab24840/Deep-Learning/blob/Assignments/Assignments/Softmax_Classifier.ipynb

```
## Load libraries
import numpy as np
import sys
import matplotlib.pyplot as plt
import matplotlib.cm as cm
plt.style.use('dark_background')
%matplotlib inline
```

```
import tensorflow as tf
```

```
tf.__version__
```

```
'2.15.0'
```

```
# Generate artificial data with 5 samples, 4 features per sample
# and 3 output classes
num_samples = 5 # number of samples
num_features = 4 # number of features (a.k.a. dimensionality)
num_labels = 3 # number of output labels
# Data matrix (each column = single sample)
X = np.random.choice(np.arange(3, 10), size = (num_features, num_samples), replace = True)
# Class labels
y = np.random.choice([0, 1, 2], size = num_samples, replace = True)
print(X)
print('------')
print(y)
print('------')
# One-hot encode class labels
y = tf.keras.utils.to_categorical(y)
print(y)
```

```
[[5 7 7 7 9]
 [6 9 3 5 3]
 [6 4 7 7 4]
 [8 6 6 3 3]]
------
[2 2 0 2 0]
------
[[0. 0. 1.]
 [0. 0. 1.]
 [1. 0. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
```

A generic layer class with forward and backward methods

```
class Layer:
  def __init__(self):
    self.input = None
    self.output = None

  def forward(self, input):
    pass

  def backward(self, output_gradient, learning_rate):
    pass
```

The softmax classifier steps for a generic sample $\mathbf{x}$ with (one-hot encoded) true label $\mathbf{y}$ (3 possible categories) using a randomly initialized weights matrix (with bias abosrbed as its last last column):

1. Calculate raw scores vector for a generic sample $\mathbf{x}$ (bias feature added):

$$\mathbf{z} = \mathbf{W}\mathbf{x}.$$

2. Calculate softmax probabilities (that is, softmax-activate the raw scores)

$$\mathbf{a} = \mathrm{softmax}(\mathbf{z}) \Rightarrow \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \mathrm{softmax}\left(\begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{z_0}}{e^{z_0}+e^{z_1}+e^{z_2}} \\ \frac{e^{z_1}}{e^{z_0}+e^{z_1}+e^{z_2}} \\ \frac{e^{z_2}}{e^{z_0}+e^{z_1}+e^{z_2}} \end{bmatrix}$$

3. Softmax loss for this sample is (where output label y is not yet one-hot encoded)

$$\begin{aligned} L &= -\log([a]_y) \\ &= -\log\Big([\mathrm{softmax}(\mathbf{z})]_y\Big) \\ &= -\log\Big([\mathrm{softmax}(\mathbf{W}\mathbf{x})]_y\Big). \end{aligned}$$

4. Predicted probability vector that the sample belongs to each one of the output categories is given a new name

$$\hat{\mathbf{y}} = \mathbf{a}.$$

5. One-hot encoding the output label

$$\underbrace{y \rightarrow \mathbf{y}}_{\text{e.g. } 2 \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}$$

results in the following representation for the softmax loss for the sample which is also referred to as the categorical crossentropy (CCE) loss:

$$L = L\left(\mathbf{y}, \hat{\mathbf{y}}\right) = \sum_{k=0}^{2} -y_k \log\left(\hat{y}_k\right).$$

6. Calculate the gradient of the loss for the sample w.r.t. weights by following the computation graph from top to bottom (that is, backward):

$$\begin{gathered} L \\ \downarrow \\ \hat{\mathbf{y}} = \mathbf{a} \\ \downarrow \\ \mathbf{z} \\ \downarrow \\ \mathbf{W} \end{gathered}$$

$$\begin{aligned} \Rightarrow \nabla_{\mathbf{W}}(L) &= \nabla_{\mathbf{W}}(\mathbf{z}) \times \nabla_{\mathbf{z}}(\mathbf{a}) \times \nabla_{\mathbf{a}}(L) \\ &= \underbrace{\nabla_{\mathbf{W}}(\mathbf{z})}_{\text{first term}} \times \underbrace{\nabla_{\mathbf{z}}(\mathbf{a})}_{\text{second to last term}} \times \underbrace{\nabla_{\hat{\mathbf{y}}}(L)}_{\text{last term}}. \end{aligned}$$

7. Now focus on the last term $\nabla_{\hat{\mathbf{y}}}(L)$:

$$\nabla_{\hat{\mathbf{y}}}(L) = \begin{bmatrix} \nabla_{\hat{y}_0}(L) \\ \nabla_{\hat{y}_1}(L) \\ \nabla_{\hat{y}_2}(L) \end{bmatrix} = \begin{bmatrix} -y_0/\hat{y}_0 \\ -y_1/\hat{y}_2 \\ -y_0/\hat{y}_2. \end{bmatrix}$$

8. Now focus on the second to last term $\nabla_{\mathbf{z}}(\mathbf{a})$:

$$\begin{aligned} \nabla_{\mathbf{z}}(\mathbf{a}) &= \nabla_{\mathbf{z}}\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}\right) \\ &= \begin{bmatrix} \nabla_{\mathbf{z}}(a_0) & \nabla_{\mathbf{z}}(a_1) & \nabla_{\mathbf{z}}(a_2) \end{bmatrix} \\ &= \begin{bmatrix} \nabla_{z_0}(a_0) & \nabla_{z_0}(a_1) & \nabla_{z_0}(a_2) \\ \nabla_{z_1}(a_0) & \nabla_{z_1}(a_1) & \nabla_{z_1}(a_2) \\ \nabla_{z_2}(a_0) & \nabla_{z_2}(a_1) & \nabla_{z_2}(a_2) \end{bmatrix} \\ &= \begin{bmatrix} a_0(1-a_0) & -a_1 a_0 & -a_2 a_0 \\ -a_0 a_1 & a_1(1-a_1) & -a_1 a_1 \\ -a_0 a_2 & -a_1 a_2 & a_2(1-a_2) \end{bmatrix}. \end{aligned}$$

9. On Monday, we will focus on the first term to complete the gradient calculation using the computation graph.

```
## Softmax activation class
class Softmax(Layer):
  def forward(self, input):
    self.output = np.array(tf.nn.softmax(input))

  def backward(self, output_gradient, learning_rate):
    return(np.dot((np.identity(np.size(self.output))-self.output.T) * self.output, output_gradient))
```

```
## Define the loss function and its gradient
def cce(y, yhat):
  return(-np.sum(y*np.log(yhat)))

def cce_gradient(y, yhat):
  return(-y/yhat)

# TensorFlow in-built function for categorical crossentropy loss
cce = tf.keras.losses.CategoricalCrossentropy()
cce
```

```
<keras.src.losses.CategoricalCrossentropy at 0x7ac0abf573a0>
```

```
## Train the 0-layer neural network using batch training with batch size = 1

# Steps: run over each sample, calculate loss, gradient of loss,
# and update weights.

# Step-1: add the bias feature to all the samples
# Step-2: initialize the entries of the weights matrix randomly
# Step-3: create softmax layer object softmax

# Step-4: run over each sample
for i in range(X.shape[1]):
  # Step-5: forward step
  # (a) Raw scores z = Wx = np.dot(W, x[:, i])
  # (b) Softmax activation: softmax.forward(z)
  # (c) Calculate cce loss for sample: cce(y[i, :], softmax.output)
  # (d) Print cce loss

  # Step-6: backward step
  # (a) Calculate the gradient of the sample loss w.r.t. input of the
  # softmax layer: softmax.backward(output_gradient = cce_gradient(y[i, :], softmax.output))
  # (d) Print gradient
```

```
  File "<ipython-input-8-7ceb0d13911d>", line 21
    # (d) Print gradient
                       ^
SyntaxError: incomplete input
```

SUGGEST FIX

1. Calculate raw scores vector for a generic sample $\mathbf{x}$ (bias feature added):

$$\mathbf{z} = \mathbf{W}\mathbf{x}.$$

```
x = tf.constant([[1, 2, 3], [4, 5, 6]])
```

```
W = tf.Variable(tf.random.normal([3, 2]))
```

```
import numpy as np
z = np.dot(W, x)
z
```

```
array([[ -9.08701134, -11.99186206, -14.89671278],
       [ -6.39343089,  -7.85449612,  -9.31556135],
       [  4.78612366,   6.11788398,   7.4496443 ]])
```

2. Calculate softmax probabilities (that is, softmax-activate the raw scores)

$$\mathbf{a} = \mathrm{softmax}(\mathbf{z}) \Rightarrow \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \mathrm{softmax}\left(\begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{z_0}}{e^{z_0}+e^{z_1}+e^{z_2}} \\ \frac{e^{z_1}}{e^{z_0}+e^{z_1}+e^{z_2}} \\ \frac{e^{z_2}}{e^{z_0}+e^{z_1}+e^{z_2}} \end{bmatrix}$$

```
z = tf.matmul(W, tf.cast(x, tf.float32))
z
```

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[ -9.087011 , -11.991862 , -14.896712 ],
       [ -6.3934307,  -7.8544965,  -9.315561 ],
       [  4.7861238,   6.117884 ,   7.449644 ]], dtype=float32)>
```

```
softmax = lambda z: np.exp(z) / np.sum(np.exp(z))
a = softmax(z)
a
```

```
array([[4.9335377e-08, 2.7014548e-09, 1.4792358e-10],
       [7.2939986e-07, 1.6921265e-07, 3.9255497e-08],
       [5.2261811e-02, 1.9795233e-01, 7.4978489e-01]], dtype=float32)
```

3. Softmax loss for this sample is (where output label y is not yet one-hot encoded)

$$\begin{aligned} L &= -\log([a]_y) \\ &= -\log\Big([\mathrm{softmax}(\mathbf{z})]_y\Big) \\ &= -\log\Big([\mathrm{softmax}(\mathbf{W}\mathbf{x})]_y\Big). \end{aligned}$$

```
def softmax_loss(y, a):
    return -np.log(a[y])

y = 1
a = softmax(z)
loss = softmax_loss(y, a)
print(loss)

```

```
[14.131043 15.59211  17.053175]
```

4. Predicted probability vector that the sample belongs to each one of the output categories is given a new name

$$\hat{\mathbf{y}} = \mathbf{a}.$$

```
y_hat = a
y_hat
```

```
array([[4.9335377e-08, 2.7014548e-09, 1.4792358e-10],
       [7.2939986e-07, 1.6921265e-07, 3.9255497e-08],
       [5.2261811e-02, 1.9795233e-01, 7.4978489e-01]], dtype=float32)
```

5. One-hot encoding the output label

$$\underbrace{y \to \mathbf{y}}_{\text{e.g. } 2 \to \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}$$

results in the following representation for the softmax loss for the sample which is also referred to as the categorical crossentropy (CCE) loss:

$$L = L\left(\mathbf{y}, \hat{\mathbf{y}}\right) = \sum_{k=0}^{2} -y_k \log\left(\hat{y}_k\right).$$

```
y_encoded = tf.keras.utils.to_categorical(y, num_labels)
y_encoded
```

```
array([0., 1., 0.], dtype=float32)
```

6. Calculate the gradient of the loss for the sample w.r.t. weights by following the computation graph from top to bottom (that is, backward):

$$\begin{gathered} L \\ \downarrow \\ \hat{\mathbf{y}} = \mathbf{a} \\ \downarrow \\ \mathbf{z} \\ \downarrow \\ \mathbf{W} \end{gathered}$$

$$\begin{aligned} \Rightarrow \nabla_{\mathbf{W}}(L) &= \nabla_{\mathbf{W}}(\mathbf{z}) \times \nabla_{\mathbf{z}}(\mathbf{a}) \times \nabla_{\mathbf{a}}(L) \\ &= \underbrace{\nabla_{\mathbf{W}}(\mathbf{z})}_{\text{first term}} \times \underbrace{\nabla_{\mathbf{z}}(\mathbf{a})}_{\text{second to last term}} \times \underbrace{\nabla_{\hat{\mathbf{y}}}(L)}_{\text{last term}}. \end{aligned}$$

```
import numpy as np

def softmax(z):
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z)

def softmax_loss(y, a):
    return -np.log(a[y])

def gradient_W(X, y, a):
    # Compute the gradient of the loss with respect to z
    grad_z = a
    grad_z[y] -= 1

    # Compute the gradient of z with respect to W
    grad_W = np.outer(grad_z, X)

    return grad_W

x_0 = np.random.rand()
x_1 = np.random.rand()
x_2 = np.random.rand()
x_3 = np.random.rand()

X = np.array([x_0, x_1, x_2, x_3])

print(X)
print('------')
grad_W = gradient_W(X, y, a)
print('------')
print(grad_W)
```

```
[0.95981135 0.52359943 0.68329462 0.59092701]
------
------
[[ 4.73526547e-08  2.58319749e-08  3.37105973e-08  2.91536066e-08]
 [ 2.59288703e-09  1.41448020e-09  1.84588955e-09  1.59636263e-09]
 [ 1.41978728e-10  7.74526996e-11  1.01075383e-10  8.74120365e-11]
 [-9.59810668e-01 -5.23599052e-01 -6.83294129e-01 -5.90926587e-01]
 [-9.59811182e-01 -5.23599332e-01 -6.83294495e-01 -5.90926904e-01]
 [-9.59811297e-01 -5.23599395e-01 -6.83294577e-01 -5.90926974e-01]
 [ 5.01614793e-02  2.73642541e-02  3.57102140e-02  3.08829155e-02]
 [ 1.89996894e-01  1.03647726e-01  1.35259762e-01  1.16975378e-01]
 [ 7.19652048e-01  3.92586936e-01  5.12323977e-01  4.43068141e-01]]
```

7. Now focus on the last term $\nabla_{\hat{\mathbf{y}}}(L)$:

$$\nabla_{\hat{\mathbf{y}}}(L) = \begin{bmatrix} \nabla_{\hat{y}_0}(L) \\ \nabla_{\hat{y}_1}(L) \\ \nabla_{\hat{y}_2}(L) \end{bmatrix} = \begin{bmatrix} -y_0/\hat{y}_0 \\ -y_1/\hat{y}_2 \\ -y_0/\hat{y}_2. \end{bmatrix}$$

```
def gradient_y(y, y_hat):
    grad_y = -y / y_hat
    return grad_y

grad_y = gradient_y(y_encoded, y_hat)
grad_y
```

```
array([[-0.0000000e+00, -3.7017091e+08, -0.0000000e+00],
       [ 0.0000000e+00,  1.0000002e+00,  0.0000000e+00],
       [-0.0000000e+00, -5.0517211e+00, -0.0000000e+00]], dtype=float32)
```

8. Now focus on the second to last term $\nabla_{\mathbf{z}}(\mathbf{a})$:

$$\begin{aligned}
\nabla_{\mathbf{z}}(\mathbf{a}) &= \nabla_{\mathbf{z}}\left(\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}\right) \\
&= \begin{bmatrix} \nabla_{\mathbf{z}}(a_0) & \nabla_{\mathbf{z}}(a_1) & \nabla_{\mathbf{z}}(a_2) \end{bmatrix} \\
&= \begin{bmatrix} \nabla_{z_0}(a_0) & \nabla_{z_0}(a_1) & \nabla_{z_0}(a_2) \\ \nabla_{z_1}(a_0) & \nabla_{z_1}(a_1) & \nabla_{z_1}(a_2) \\ \nabla_{z_2}(a_0) & \nabla_{z_2}(a_1) & \nabla_{z_2}(a_2) \end{bmatrix} \\
&= \begin{bmatrix} a_0(1-a_0) & -a_1 a_0 & -a_2 a_0 \\ -a_0 a_1 & a_1(1-a_1) & -a_1 a_1 \\ -a_0 a_2 & -a_1 a_2 & a_2(1-a_2) \end{bmatrix}.
\end{aligned}$$

```
import numpy as np

def gradient_z(a):
    diag_a = np.diag(a)
    return diag_a - np.outer(a, a)

a_0 = np.random.rand()
a_1 = np.random.rand()
a_2 = np.random.rand()

a = np.array([a_0, a_1, a_2])
grad_z = gradient_z(a)
grad_z
```

```
array([[ 0.15138129, -0.04468256, -0.09806931],
       [-0.04468256,  0.18254319, -0.12671096],
       [-0.09806931, -0.12671096,  0.2492516 ]])
```

Assignment 1

```
## Step-1: add the bias feature to all the samples

X = np.reshape(X, (X.shape[0], 1))
X_bias = np.hstack((np.ones((X.shape[0], 1)), X))
```

```
import numpy as np

# Step-2: initialize the entries of the weights matrix randomly
initialize_weights = lambda num_features, num_labels: np.random.randn(num_labels, num_features)

num_features = 4
num_labels = 3
W = initialize_weights(num_features, num_labels)
print(W)
```

```
[[-0.13677493 -0.89663344 -0.4914765  -0.30098478]
 [ 1.99771577 -0.32930935 -1.2273706   0.53561762]
 [ 0.40719042  0.71531368  1.0341364  -1.16897045]]
```

```
# Step-3: create softmax layer object softmax
class SoftmaxLayer:
    def __init__(self):
        pass

    def forward(self, z):
        exp_z = np.exp(z)
        self.output = exp_z / np.sum(exp_z)

    def backward(self, y):
        return self.output - y

softmax = SoftmaxLayer()
```

```
# Step-4: run over each sample

def cross_entropy_loss(y_true, y_pred):
  return -np.sum(y_true * np.log(y_pred))

# Start the loop over each sample
for i in range(X.shape[1]):
  # Calculate the raw scores by multiplying the weights with the sample
  z = np.dot(W, X[:, i])

  # Apply the softmax activation function
  softmax.forward(z)

  # Calculate the cross-entropy loss for the sample
  def cross_entropy_loss(y_true, y_pred):
    return -np.sum(y_true * np.log(y_pred))
  # Print the loss for the sample
  print(f"Loss for sample {i}: {loss}")

  # Calculate the gradient of the loss with respect to the input of the softmax layer
  grad_z = softmax.backward(y)

  # Print the gradient for the sample
```

```
  # Print the gradient for the sample
  print(f"Gradient for sample {i}: {grad_z}")
```

```
Loss for sample 0: [14.131043 15.59211  17.053175]
Gradient for sample 0: [-0.94447549 -0.42514257 -0.63038194]
```