Tensor (aka multidimensional array) manipulation

Lecture 05 — CS 577 Deep Learning

Instructor: Yutong Wang

Computer Science
Illinois Institute of Technology

September 18, 2024

Multiclass classification

Let $i = 1, \ldots, N$ (the sample index)

- Training samples $\mathbf{x}^{(i)} \in \mathcal{X} \subseteq \mathbb{R}^d$
- Labels $y^{(i)} \in \mathcal{Y} = \{1, \ldots, K\}$
- $f(\cdot; \boldsymbol{\theta}) : \mathcal{X} \to \mathcal{Y}$

Multiclass classification

```python
import numpy as np
from sklearn.datasets import load_iris

n_classes = 3  # Number of classes

X, y = load_iris(return_X_y = True)
n = X.shape[0]
X = X[:,:2]                     # Keep only the first two features
X = X - np.mean(X, axis=0)  # Center the data
```

Note: Shape of data

1-layer neural network

To define $f(\cdot;\boldsymbol{\theta}) : \mathcal{X} \to \mathcal{Y}$

$$\mathbf{h}^{(i)} = \mathbf{W}^{(1)\top}\mathbf{x}^{(i)} + \mathbf{b}^{(1)} \quad \text{and} \quad \mathbf{z}^{(i)} = \mathbf{W}^{(2)\top} g(\mathbf{h}^{(i)}) + \mathbf{b}^{(2)}$$

```python
def init_params(input_dim, hidden_dim, output_dim):
    np.random.seed(0)  # Ensure reproducibility
    W1 = np.random.randn(input_dim, hidden_dim)
    b1 = np.random.randn(hidden_dim)
    W2 = np.random.randn(hidden_dim,output_dim)
    b2 = np.random.randn(output_dim)
    return W1, b1, W2, b2
```

1-layer neural network

$$\mathbf{h}^{(i)} = \mathbf{W}^{(1)\top}\mathbf{x}^{(i)} + \mathbf{b}^{(1)} \quad \text{and} \quad \mathbf{z}^{(i)} = \mathbf{W}^{(2)\top} g(\mathbf{h}^{(i)}) + \mathbf{b}^{(2)}$$

```
input_dim = 2          # Number of input dimension/features
hidden_dim = 10        # Number of hidden neurons
output_dim = 3         # Number of classes

W1, b1, W2, b2 = init_params(input_dim, hidden_dim, output_dim)

theta = {
    "W1": W1,
    "b1": b1,
    "W2": W2,
    "b2": b2
}
```

Relu

$$\mathbf{h}^{(i)} = \mathbf{W}^{(1)\top}\mathbf{x}^{(i)} + \mathbf{b}^{(1)} \quad \text{and} \quad \mathbf{z}^{(i)} = \mathbf{W}^{(2)\top} g(\mathbf{h}^{(i)}) + \mathbf{b}^{(2)}$$

```python
def relu(z):
    return np.maximum(0, z)
```

Note: What is the difference between `np.max` vs `np.maximum`?

Pairwise vs Reduction operations

- `np.maximum` is a type of pairwise operation like `+` and `*`

```
a = 12*np.random.randn(3,4)
b = np.arange(12).reshape((3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
np.maximum(a,b)
# array([[ 0.        ,  3.88881425,  2.        ,  3.        ],
#        [ 4.        , 23.30701124,  7.05094358,  7.        ],
#        [ 8.        ,  9.        , 10.        , 12.53350356]])
```

- `np.max` is a type of reduction operation like `np.sum` and `np.mean`

```
np.max(a,b) # will throw an error
```

Pairwise operations between tensors

- `np.maximum` is a type of pairwise operation like `+` and `*`

```
a = 5
b = np.arange(12).reshape((3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
np.maximum(a,b)
# array([[ 5,  5,  5,  5],
#        [ 5,  5,  6,  7],
#        [ 8,  9, 10, 11]])
```

What is a tensor (aka multidimensional array) anyways?

```
a = np.arange(3*2*4).reshape([3,2,4])


# array([[[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7]],
#
#        [[ 8,  9, 10, 11],
#         [12, 13, 14, 15]],
#
#        [[16, 17, 18, 19],
#         [20, 21, 22, 23]]])
```

What is broadcasting?

```python
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(4)
a + b
# array([[[ 0,  2,  4,  6],
#         [ 4,  6,  8, 10]],
#
#        [[ 8, 10, 12, 14],
#         [12, 14, 16, 18]],
#
#        [[16, 18, 20, 22],
#         [20, 22, 24, 26]]])
```

What is a tensor (aka multidimensional array) anyways?

```
a = np.arange(3*2*4).reshape([3,2,4])


# array([[[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7]],
#
#        [[ 8,  9, 10, 11],
#         [12, 13, 14, 15]],
#
#        [[16, 17, 18, 19],
#         [20, 21, 22, 23]]])
```

What is broadcasting?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(2*4).reshape([2,4])
a + b
# array([[[ 0,  2,  4,  6],
#         [ 8, 10, 12, 14]],
#
#        [[ 8, 10, 12, 14],
#         [16, 18, 20, 22]],
#
#        [[16, 18, 20, 22],
#         [24, 26, 28, 30]]])
```

What is broadcasting?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(3*4).reshape([3,4])
a + b
# error
#
#
#
#
#
#
#
```

What is broadcasting?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(3*4).reshape([3,1,4])
a + b
# array([[[ 0,  2,  4,  6],
#         [ 4,  6,  8, 10]],
#
#        [[12, 14, 16, 18],
#         [16, 18, 20, 22]],
#
#        [[24, 26, 28, 30],
#         [28, 30, 32, 34]]])
```

What is a tensor (aka multidimensional array) anyways?

```
a = np.arange(3*2*4).reshape([3,2,4])


# array([[[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7]],
#
#        [[ 8,  9, 10, 11],
#         [12, 13, 14, 15]],
#
#        [[16, 17, 18, 19],
#         [20, 21, 22, 23]]])
```

What is broadcasting?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(3).reshape([3])
a + b
# ?
#
#
#
#
#
#
#
```

What is broadcasting?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(3).reshape([3,1])
a + b
# ?
#
#
#
#
#
#
#
```

What is broadcasting?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(3).reshape([3,1,1])
a + b
# ?
#
#
#
#
#
#
#
```

What is broadcasting?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(3).reshape([3,1,1])
a + b
# array([[[ 0,  1,  2,  3],
#          [ 4,  5,  6,  7]],
#
#         [[ 9, 10, 11, 12],
#          [13, 14, 15, 16]],
#
#         [[18, 19, 20, 21],
#          [22, 23, 24, 25]]])
```

Which of these two works?

```
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(2).reshape([2,1])
a + b
# ?
#
a = np.arange(3*2*4).reshape([3,2,4])
b = np.arange(2).reshape([1,2,1])
a + b
# ?
#
```

Broadcasting rule applies

Note: Back to `np.maximum`...

- `np.maximum` is a type of pairwise operation like `+` and `*`

```
a = np.array([5,5,6,6])
b = np.arange(12).reshape((3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
np.maximum(a,b)
# array([[ 5,  5,  6,  6],
#        [ 5,  5,  6,  7],
#        [ 8,  9, 10, 11]])
```

Broadcasting rule applies

Note: Back to `np.maximum`...

- `np.maximum` is a type of pairwise operation like `+` and `*`

```
a = np.array([2,6,9])
b = np.arange(12).reshape((3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
np.maximum(a,b)
# ?
#
#
```

Broadcasting rule applies

Note: Back to `np.maximum`...

- `np.maximum` is a type of pairwise operation like `+` and `*`

```
a = np.array([2,6,9]).reshape((3,1))
b = np.arange(12).reshape((3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
np.maximum(a,b)
# array([[ 2,  2,  2,  3],
#        [ 6,  6,  6,  7],
#        [ 9,  9, 10, 11]])
```

Broadcasting rule applies

Note: Back to `np.maximum`...

- `np.maximum` is a type of pairwise operation like `+` and `*`

```
a = np.expand_dims(np.array([2,6,9]), axis=1)
b = np.arange(12).reshape((3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
np.maximum(a,b)
# array([[ 2,  2,  2,  3],
#        [ 6,  6,  6,  7],
#        [ 9,  9, 10, 11]])
```

Broadcasting rule applies

- `np.maximum` is a type of pairwise operation like `+` and `*`

```
a = np.expand_dims(np.array([2,6,9]), axis=[1,2])
# a.shape == (3,1,1)
b = np.arange(12).reshape((3,4))
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11]])
np.maximum(a,b)
# ???
```

Note: Breakout session: explain the output.

Reduction

- `np.max` is a type of reduction operation like `np.sum` and `np.mean`

```
a = np.arange(3*2*4).reshape([3,2,4])
# array([[[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7]],
#
#        [[ 8,  9, 10, 11],
#         [12, 13, 14, 15]],
#
#        [[16, 17, 18, 19],
#         [20, 21, 22, 23]]])
np.max(a)
# 23
```

Reduction

- `np.max` is a type of reduction operation like `np.sum` and `np.mean`

```
a = np.arange(3*2*4).reshape([3,2,4])
# array([[[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7]],
#
#        [[ 8,  9, 10, 11],
#         [12, 13, 14, 15]],
#
#        [[16, 17, 18, 19],
#         [20, 21, 22, 23]]])
np.max(a)
# 23
```

Reduction

- `np.max` is a type of reduction operation like `np.sum` and `np.mean`

```python
a = np.arange(3*2*4).reshape([3,2,4])
# array([[[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7]],
#
#        [[ 8,  9, 10, 11],
#         [12, 13, 14, 15]],
#
#        [[16, 17, 18, 19],
#         [20, 21, 22, 23]]])
np.max(a, axis=0)
# a.shape == (2,4)
# array([[16, 17, 18, 19],
#        [20, 21, 22, 23]])
```

Reduction

- `np.max` is a type of reduction operation like `np.sum` and `np.mean`

```
a = np.arange(3*2*4).reshape([3,2,4])
# array([[[ 0,  1,  2,  3],
#          [ 4,  5,  6,  7]],
#
#         [[ 8,  9, 10, 11],
#          [12, 13, 14, 15]],
#
#         [[16, 17, 18, 19],
#          [20, 21, 22, 23]]])
np.max(a, axis=1)
# a.shape == (3,4)
# array([[ 4,  5,  6,  7],
#         [12, 13, 14, 15],
#         [20, 21, 22, 23]])
```

Reduction

- `np.max` is a type of reduction operation like `np.sum` and `np.mean`

```
a = np.arange(3*2*4).reshape([3,2,4])
# array([[[ 0,  1,  2,  3],
#          [ 4,  5,  6,  7]],
#
#         [[ 8,  9, 10, 11],
#          [12, 13, 14, 15]],
#
#         [[16, 17, 18, 19],
#          [20, 21, 22, 23]]])
np.max(a, axis=-1)
# a.shape == (3,2)
# array([[ 3,  7],
#         [11, 15],
#         [19, 23]])
```

Forward

$$\mathbf{h}^{(i)} = \mathbf{W}^{(1)\top}\mathbf{x}^{(i)} + \mathbf{b}^{(1)} \quad \text{and} \quad \mathbf{z}^{(i)} = \mathbf{W}^{(2)\top} g(\mathbf{h}^{(i)}) + \mathbf{b}^{(2)}$$

```python
def relu(z):
    return np.maximum(0, z)

def forward(X,theta):
    W1, b1, W2, b2 = theta["W1"], theta["b1"], theta["W2"], theta["b2"]
    h = relu(np.dot(X, W1) + b1)
    z = np.dot(h, W2) + b2
    return h,z
```

Note: Batched or vectorized operation

Forward (Non-vectorized)

$$\mathbf{h}^{(i)} = \mathbf{W}^{(1)\top}\mathbf{x}^{(i)} + \mathbf{b}^{(1)} \quad \text{and} \quad \mathbf{z}^{(i)} = \mathbf{W}^{(2)\top}g(\mathbf{h}^{(i)}) + \mathbf{b}^{(2)}$$

```python
def forward_nonvectorized(X, theta):
    W1, b1, W2, b2 = theta["W1"], theta["b1"], theta["W2"], theta["b2"]
    h = np.zeros((X.shape[0], W1.shape[1]))  # Initialize h
    z = np.zeros((X.shape[0], W2.shape[1]))  # Initialize z

    for i in range(X.shape[0]):
        h[i, :] = relu(np.dot(X[i, :], W1) + b1)
        z[i, :] = np.dot(h[i, :], W2) + b2
    return h, z
```

Note: Non-vectorized or row-wise operation

Timing the Forward Functions

```python
import time

# Vectorized forward function
start_time = time.time()
h_vec, z_vec = forward(X, theta)
end_time = time.time()
print(f"Vectorized time: {end_time - start_time:.5f} seconds")

# Non-vectorized forward function
start_time = time.time()
h_nonvec, z_nonvec = forward_nonvectorized(X, theta)
end_time = time.time()
print(f"Non-vectorized time: {end_time - start_time:.5f} seconds")
```

Note: Measure the time for vectorized vs. non-vectorized forward pass

Training loss

1. $$J(\boldsymbol{\theta}) := \frac{1}{N}\sum_{i=1}^{N} J_i(\boldsymbol{\theta})$$

   and

   $$J_i(\boldsymbol{\theta}) := L(f(\mathbf{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

Backward

$$
\begin{aligned}
\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(2)}} &= \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}} \qquad\qquad \Longleftarrow \\
\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(2)}} &= g(\mathbf{h}^{(i)}) \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}^{\top} \\
\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(1)}} &= (\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}) \\
\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} &= ((\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}))\mathbf{x}^{(i)\top}
\end{aligned}
$$

Cross entropy

Cross entropy

$$L(\mathbf{z}, y) = -\log(\mathsf{softmax}(\mathbf{z})_y)$$

Derivative

$$\frac{\partial L}{\partial \mathbf{z}}(\mathbf{z}, y) = \underbrace{\mathbf{p}}_{\mathsf{softmax}(\mathbf{z})} - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \\ \\ \\ \leftarrow \quad y\text{-th position} \\ \\ \\ \\ \end{array}$$

Softmax

$$\text{softmax}(\mathbf{z}^{(i)}) = \frac{1}{\sum_{j=1}^{K} \exp(z_j^{(i)})} \begin{bmatrix} \exp(z_1^{(i)}) \\ \vdots \\ \exp(z_K^{(i)}) \end{bmatrix} = \begin{bmatrix} \text{softmax}(\mathbf{z}^{(i)})_1 \\ \vdots \\ \text{softmax}(\mathbf{z}^{(i)})_K \end{bmatrix} = \begin{bmatrix} p_1 \\ \vdots \\ p_K \end{bmatrix} = \mathbf{p}$$

```python
p = np.zeros((n,n_classes))
h,z = forward(X,theta)
for i in range(n):
    p[i,:] = np.exp(z[i,:])/np.sum(np.exp(z[i,:]))
```

Note: Breakout session: compute this without the for-loop

Loss derivative

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}}_{\text{pick } y\text{-th column}} = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \quad y\text{-th position}$$

Identity matrix of size `n_classes`

```
E = np.eye(n_classes)
E[y[i], :]
```

Cross entropy

Derivative

$$\frac{\partial L}{\partial \mathbf{z}}(\mathbf{z}, y) = \underbrace{\mathbf{p}}_{\mathsf{softmax}(\mathbf{z})} - \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \quad y\text{-th position} = \quad \mathbf{p} - \mathbf{e}^{(y)}$$

```python
loss_der = np.zeros((n,n_classes))
E = np.eye(n_classes)
h,z = forward(X,theta)
for i in range(n):
    loss_der[i,:] = p[i,:] - E[y[i], :]
```

Note: Vectorize this

1-layer neural network (multicategory data)

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(2)}} = \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}} \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(2)}} = g(\mathbf{h}^{(i)}) \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}^\top \qquad \Longleftarrow$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(1)}} = (\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)})$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} = ((\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}))\mathbf{x}^{(i)\top}$$

Matrix multiplication

```
np.random.seed(42)
A = np.random.randint(0,9,(2,3))
B = np.random.randint(0,9,(3,4))
A@B
# array([[85, 70, 79, 68],
#        [74, 72, 42, 52]])
```

Matrix multiplication

```python
C = np.zeros((2, 4))

for i in range(A.shape[0]):
    for j in range(B.shape[1]):
        for k in range(A.shape[1]):
            C[i, j] += A[i, k] * B[k, j]
# array([[85., 70., 79., 68.],
#          [74., 72., 42., 52.]])
```

Matrix multiplication

```
D = np.zeros((2, 4))

for k in range(A.shape[1]):  # iterate over the elements to multiply
    D += np.outer(A[:, k], B[k, :])
# array([[85., 70., 79., 68.],
#        [74., 72., 42., 52.]])
```

Cross entropy

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(2)}} = g(\mathbf{h}^{(i)}) \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}^{\top}$$

```
dJdW2 = np.zeros_like(W2)
# h,z = forward(X,theta) # already computed previously

for i in range(n):
    dJdW2 += np.outer(relu(h[i,:]), loss_der[i,:])

dJdW2/n
```

Note: "vectorize" this

1-layer neural network (multicategory data)

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(2)}} = \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}} \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(2)}} = g(\mathbf{h}^{(i)}) \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}^{\top} \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(1)}} = (\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}) \qquad \Longleftarrow$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} = ((\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}))\mathbf{x}^{(i)\top}$$

Cross Entropy

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(1)}} = (\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)})$$

```
def relu_derivative(z):
    return 1.0*(z > 0)

dJdb1 = np.zeros_like(b1)

# h,z = forward(X,theta)
for i in range(n):
    dJdb1 += (W2@(loss_der[i,:])) * relu_derivative(h[i,:])
dJdb1 /=n
```

Note: Vectorize this

1-layer neural network (multicategory data)

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(2)}} = \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}} \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(2)}} = g(\mathbf{h}^{(i)}) \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}^{\top} \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(1)}} = (\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}) \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} = ((\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}))\mathbf{x}^{(i)\top} \qquad \Longleftarrow$$

Cross Entropy

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} = ((\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}))\mathbf{x}^{(i)\top}$$

```
dJdW1 = np.zeros_like(W1)

# h,z = forward(X,theta)
for i in range(n):
    dJdW1 += np.outer((W2@loss_der[i,:])  *
                            relu_derivative(h[i,:]), X[i,:]).T
dJdW1 /=n
```

Note: Vectorize this

Cross Entropy

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} = ((\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}))\mathbf{x}^{(i)\top}$$

```
dJdW1 = np.zeros_like(W1)

# h,z = forward(X,theta)
for i in range(n):
    dJdW1 += np.outer((W2@loss_der[i,:])  *
                         relu_derivative(h[i,:]), X[i,:]).T
dJdW1 /=n
```

Note: Vectorize this

1-layer neural network (multicategory data)

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(2)}} = \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}} \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(2)}} = g(\mathbf{h}^{(i)}) \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}^\top \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{b}^{(1)}} = (\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}) \qquad (\checkmark)$$

$$\frac{\partial J_i(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} = ((\mathbf{W}^{(2)} \frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}) \odot g'(\mathbf{h}^{(i)}))\mathbf{x}^{(i)\top} \qquad (\checkmark)$$

How to implement the loss derivative

$$\frac{\partial L(\mathbf{z}^{(i)}, y)}{\partial \mathbf{z}}^{\top}$$

```
    h,z = forward(X,theta)
    p = np.exp(z[i,:])/np.sum(np.exp(z[i,:]))
    loss_der = p - E[y[i], :]
    dJdW1 += np.outer((W2@loss_der) * relu_derivative(h[i,:]), X[i,:]).T
    dJdb1 += (W2@(loss_der)) * relu_derivative(h[i,:])
    dJdW2 += np.outer(relu(h[i,:]), loss_der)
    dJdb2 += loss_der
```

Stochastic gradient descent (SGD)

Let $\eta_t > 0$ be learning rates, $t = 1, 2, \ldots$
Let $m \geq 1$ be an integer

- Initialize $\boldsymbol{\theta}$
- While not converged (t = iteration counter):
  - Select m samples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ and matching labels $\{y^{(1)}, \ldots, y^{(m)}\}$
  - Compute gradient $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \boldsymbol{\theta}), y^{(i)})$
  - Compute update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_t \mathbf{g}$

Stochastic gradient descent (SGD)

Let $\eta_t > 0$ be learning rates, $t = 1, 2, \ldots$
Let $m \geq 1$ be an integer

- Initialize $\boldsymbol{\theta}$
- While not converged ($t =$ iteration counter):
  - Select 1 sample $\{\mathbf{x}^{(t)}\}$ and its label $\{y^{(t)}\}$
  - Compute gradient $\mathbf{g} \leftarrow \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^{(t)}, \boldsymbol{\theta}), y^{(t)})$
  - Compute update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_t \mathbf{g}$

References I