CS231n: Deep Learning for Computer Vision (2024 Spring) Assignment 1

> Created by: *Haoming Cen*
>
> Date: 2024-08-31 (Week 1)
>
> Completed assignments will be uploaded to *https://github.com/CENHM/CS231n-Notes-and-Assignments*

Q1: k-Nearest Neighbor Classifier

> Use `np.linalg.norm()` to calculate the norm:
>
> ```
> np.linalg.norm(x, ord=None, axis=None, keepdims=False)
> ```
>
> - `ord=1` : $\sum_i |x_i|$
> - `ord=2` : $\sqrt{\sum_i (x_i)^2}$

Broadcasting two arrays together follows these rules:

- If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
- The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
- The arrays can be broadcast together if they are compatible in all dimensions.
- After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
- In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension1

> Use `np.newaxis` to expand array dimension:
>
> ```
> a = np.random.randint(0, 10, size=(5,))
> print(a[:, np.newaxis].shape) # (5,1)
> print(a[np.newaxis, :].shape) # (1,5)
> ```

Q2: Training a Support Vector Machine

> It is vital to understand matrix calculus. Here are some supporting learning materials:
>
> 1. *The Nature of Matrix Derivation and the Nature of Numerator Layout and Denominator Layout*: https://zhuanlan.zhihu.com/p/263777564
> 2. *Mathematical derivation of the matrix derivation formula*: https://zhuanlan.zhihu.com/p/273729929
>
> **In summary,** Matrix derivation is essentially for **each** f in the function of **each** element of the variant of the partial derivatives.

e.g. in `cs231n/classifiers/linear_svm.py/svm_loss_naive()`

$$L = \frac{1}{N}\sum_i \sum_{j\neq y_i}[\max(0, f_j(\boldsymbol{x}_i;\mathbf{W}) - f_{y_i}(\boldsymbol{x}_i;\mathbf{W}) + \Delta)] + \lambda \sum_k \sum_l \mathbf{W}_{k,l}^2$$

The gradient with respect to weights $\mathbf{W}$ of the second term of L:

$$\begin{aligned}
\frac{\partial \lambda \sum_k \sum_l \mathbf{W}_{kl}^2}{\partial \mathbf{W}} &= \lambda \frac{\partial \sum_k \sum_l \mathbf{W}_{kl}^2}{\partial \mathbf{W}} \\
&= \lambda \frac{\partial \mathbf{W}_{11}^2 + \mathbf{W}_{12}^2 + \cdots + \mathbf{W}_{DC}^2}{\partial \mathbf{W}} \\
&= \lambda \begin{bmatrix} \frac{\partial \mathbf{W}_{11}^2}{\partial \mathbf{W}_{11}} & \cdots & \frac{\partial \mathbf{W}_{1C}^2}{\partial \mathbf{W}_{1C}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{W}_{D1}^2}{\mathbf{W}_{D1}} & \cdots & \frac{\partial \mathbf{W}_{DC}^2}{\partial \mathbf{W}_{DC}} \end{bmatrix} = \lambda \times 2 \times \mathbf{W}
\end{aligned}$$

The gradient with respect to weights $\mathbf{W}$ of the first term (ignore $\frac{1}{N}$) of L:

$$L_i^{(1)} = \sum_{j\neq y_i} \max(0, f_j(\boldsymbol{x}_i;\mathbf{W}) - f_{y_i}(\boldsymbol{x}_i;\mathbf{W}) + \Delta)$$

$$\begin{aligned}
\frac{\partial L_i^{(1)}}{\partial \mathbf{W}} &= \frac{\partial \sum_{j\neq y_i} \max(0, f_j(\boldsymbol{x}_i;\mathbf{W}) - f_{y_i}(\boldsymbol{x}_i;\mathbf{W}) + \Delta)}{\partial \mathbf{W}} \\
&= \frac{\partial x_{i1}\mathbf{W}_{1j_1} + x_{i2}\mathbf{W}_{2j_1} + \cdots + x_{iD}\mathbf{W}_{Dj_1} - x_{i1}\mathbf{W}_{1y_i} - x_{i2}\mathbf{W}_{2y_i} - \cdots - x_{iD}\mathbf{W}_{Dy_i}}{\partial \mathbf{W}} \\
&+ \frac{\partial x_{i1}\mathbf{W}_{1j_2} + x_{i2}\mathbf{W}_{2j_2} + \cdots + x_{iD}\mathbf{W}_{Dj_2} - x_{i1}\mathbf{W}_{1y_i} - x_{i2}\mathbf{W}_{2y_i} - \cdots - x_{iD}\mathbf{W}_{Dy_i}}{\partial \mathbf{W}} + \cdots \\
&= \sum_{j\neq y_i \text{ and } \boldsymbol{s}_j - \boldsymbol{s}_{y_i} + \Delta} \left(\begin{bmatrix} 0 & \cdots & x_{i1\,(1,j_1)} & \cdots & 0 \\ 0 & \cdots & x_{i1\,(2,j_2)} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & x_{i1\,(D,j_D)} & \cdots & 0 \end{bmatrix} - \begin{bmatrix} 0 & \cdots & x_{i1\,(1,y_i)} & \cdots & 0 \\ 0 & \cdots & x_{i1\,(2,y_i)} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & x_{i1\,(D,y_i)} & \cdots & 0 \end{bmatrix} \right)
\end{aligned}$$

Use two loops:

```
for i in range(num_train):
    for j in range(num_classes):
        if j == y[i]:
```

```
            continue
        # margin = s_j - s_{y[i]} + \Delta
        if margin > 0:
            dW[:, y[i]] -= X[i].T
            dW[:, j] += X[i].T
dW /= num_train
dW += 2 * reg * W
```

How to extract elements from two-dimensional matrix with an index array?

```
# - W: A numpy array of shape (N, D)
# - y: A numpy array of shape (N,)

_W = W[range(W.shape[0]), y]
```

As for the vectorized implementation, The gradient with respect to weights $\mathbf{W}$ of the first term (ignore $\frac{1}{N}$) of L without loop on subject index:

$$
\begin{aligned}
\left(\frac{\partial L^{(1)}}{\partial \mathbf{W}}\right)_{(i,j)} &= \frac{\partial \sum_{j\neq y_1} \max(0, f_j(\boldsymbol{x}_1;\mathbf{W}) - f_{y_1}(\boldsymbol{x}_1;\mathbf{W}) + \Delta)}{\partial \mathbf{W}} \\
&+ \frac{\partial \sum_{j\neq y_2} \max(0, f_j(\boldsymbol{x}_2;\mathbf{W}) - f_{y_2}(\boldsymbol{x}_2;\mathbf{W}) + \Delta)}{\partial \mathbf{W}} + \cdots \\
&= \frac{\partial x_{11}\mathbf{W}_{1j_1} + x_{12}\mathbf{W}_{2j_1} + \cdots + x_{1D}\mathbf{W}_{Dj_1} - x_{11}\mathbf{W}_{1y_1} - x_{12}\mathbf{W}_{2y_1} - \cdots - x_{1D}\mathbf{W}_{Dy_1}}{\partial \mathbf{W}} \\
&+ \frac{\partial x_{11}\mathbf{W}_{1j_2} + x_{12}\mathbf{W}_{2j_2} + \cdots + x_{1D}\mathbf{W}_{Dj_2} - x_{11}\mathbf{W}_{1y_1} - x_{12}\mathbf{W}_{2y_1} - \cdots - x_{1D}\mathbf{W}_{Dy_1}}{\partial \mathbf{W}} + \cdots \\
&+ \frac{\partial x_{21}\mathbf{W}_{1j_1} + x_{22}\mathbf{W}_{2j_1} + \cdots + x_{2D}\mathbf{W}_{Dj_1} - x_{21}\mathbf{W}_{1y_2} - x_{22}\mathbf{W}_{2y_2} - \cdots - x_{2D}\mathbf{W}_{Dy_2}}{\partial \mathbf{W}} \\
&+ \frac{\partial x_{21}\mathbf{W}_{1j_2} + x_{22}\mathbf{W}_{2j_2} + \cdots + x_{2D}\mathbf{W}_{Dj_2} - x_{21}\mathbf{W}_{1y_2} - x_{22}\mathbf{W}_{2y_2} - \cdots - x_{2D}\mathbf{W}_{Dy_2}}{\partial \mathbf{W}} + \cdots \\
&= \sum_{k=1}^{N} x_{ki}\mathbf{1}(\boldsymbol{s}_j - \boldsymbol{s}_{y_k} + \Delta > 0 \text{ and } j \neq y_k) - \sum_{k=1}^{N}\sum_{l=1}^{C} x_{ki}\mathbf{1}(\boldsymbol{s}_l - \boldsymbol{s}_{y_k} + \Delta > 0 \text{ and } j \neq y_k)
\end{aligned}
$$

code:

```
scores = X.dot(W)
correct_class_score = scores[range(num_train), y].reshape(num_train, 1)
threshold = np.maximum(0, scores - correct_class_score + 1)
threshold[range(num_train), y] = 0
threshold[threshold > 0] = 1

# Now, threshold[k,j] = 1 when $s_j-s_{y_k}+\Delta > 0$ and $j\neq y_k$. the result of the
first term of $\left(\frac{\partial L^{(1)}}{\partial \mathbf{W}}\right)_{(i,j)}$ is
```

```
`np.dot(X.T, threshold)`.
# Meanwhile, `np.dot(X.T, threshold)[k,j] = 0` when $j\neq y_k$ (threshold[k,y_k] = 0)
# Concerning the second term:
# of each k when $j\neq y_k$, the number of considerable $\mathbf{x}_{ki}$ equals the
number of elements satisfy `threshold[k,:] = 1` (use `np.sum(threshold, axis=1)`). Because
`threshold[k,y_k] = 0`, we can place the result at `threshold[k,y_k]` for dot product.

row_sum = np.sum(threshold, axis=1)
threshold[range(num_train), y] = -row_sum
dW += np.dot(X.T, threshold) / num_train
dW += 2 * reg * W
```

Q3: Implement a Softmax classifier

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) = -f_{y_i} + \log\sum_j e^{f_j}$$

$$\frac{\partial L_i}{\partial \mathbf{W}} = -\frac{\partial(\mathbf{x}_i\mathbf{W})_{y_i}}{\partial \mathbf{W}} + \frac{1}{\sum_j e^{f_j}}\left(\sum_j \frac{\partial e^{f_j}}{\partial \mathbf{W}}\right)$$

$$= -\begin{bmatrix} 0 & \cdots & x_{i1\,(1,y_i)} & \cdots & 0 \\ 0 & \cdots & x_{i2\,(2,y_i)} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & x_{iD\,(D,y_i)} & \cdots & 0 \end{bmatrix} + \frac{1}{\sum_j e^{f_j}}\left(\sum_j e^{ij}\begin{bmatrix} 0 & \cdots & x_{i1\,(1,j)} & \cdots & 0 \\ 0 & \cdots & x_{i2\,(2,j)} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & x_{iD\,(D,j)} & \cdots & 0 \end{bmatrix}\right)$$

Use two loops:

```
num_train = X.shape[0]
num_classes = W.shape[1]

for i in range(num_train):
    exp_val = np.exp(X[i].dot(W))
    softmax = exp_val / np.sum(exp_val)

    dW[:, y[i]] -= X[i]
    dW += X[i].reshape(X[i].shape[0], 1).dot(softmax.reshape(1, softmax.shape[0]))

dW /= num_train
dW += 2 * reg * W
```

$$\frac{\partial L}{\partial \mathbf{W}} = -f_y + \log\sum_j e^{f_j}$$

vectorized implementation:

```
num_train = X.shape[0]
num_classes = W.shape[1]
```

```
exp_val = np.exp(X.dot(W))
softmax = exp_val / np.sum(exp_val, axis=1)

tmp = np.zeros((num_train, num_classes))
tmp[range(num_train), y] = 1
dW -= (X.T).dot(tmp)

dW -= (X.T).dot(softmax)

dW /= num_train
dW += 2 * reg * W
```

Q3: Two-Layer Neural Network

Remember basic patterns in backward flow:

- "gradient distributor" Add gate: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial g}$
- "swap multiplier" Multiply gate: $\frac{\partial f}{\partial x} = y\frac{\partial f}{\partial g}, \frac{\partial f}{\partial y} = x\frac{\partial f}{\partial g}$
- "gradient adder" Copy gate: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} + \frac{\partial f}{\partial z}$
- "gradient router" Max gate: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}, (x > y)$

Affine layer is consisted of one Multiply gate and one Add gate:

$$\mathbf{Y} = \mathbf{x}\mathbf{W} + \mathbf{b}\frac{\partial L}{\partial \mathbf{x}} = \mathbf{W}^T\frac{\partial L}{\partial \mathbf{Y}}, \frac{\partial L}{\partial \mathbf{W}} = \mathbf{x}^T\frac{\partial L}{\partial \mathbf{Y}}, \frac{\partial L}{\partial \mathbf{b}} = [1, 1, \cdots, 1]_n\frac{\partial L}{\partial \mathbf{Y}}$$

```
dx = (dout.dot(w.T)).reshape(x.shape)
dw = (x.reshape(x.shape[0], -1).T).dot(dout)
db = np.sum(dout, axis=0)
```

Gradients for vectorized operations Matrix-Matrix multiply gradient is possibly the most tricky operation is the matrix-matrix multiplication operations:

```
# forward pass
W = np.random.randn(5, 10)
X = np.random.randn(10, 3)
D = W.dot(X)

# now suppose we had the gradient on D from above in the circuit
dD = np.random.randn(*D.shape) # same shape as D
dW = dD.dot(X.T) #.T gives the transpose of the matrix
dX = W.T.dot(dD)
```

Tip: use dimension analysis! Note that you do not need to remember the expressions for `dW` and `dX` because they are easy to re-derive based on dimensions. For instance, we know that the gradient on the weights `dW` must be of the same size as `W` after it is computed, and that it must depend on matrix multiplication of `X` and `dD` (as is the case when both `X` , `W` are single numbers and not matrices). There is always exactly one way of achieving this so that the dimensions work out. For example, `X` is of size `[10 x 3]` and `dD` of size `[5 x 3]` , so if we want `dW` and `W` has shape `[5 x 10]` , then the only way of achieving this is with `dD.dot(X.T)` , as shown above.

Recommendation: explicitly write out a minimal vectorized example, derive the gradient on paper and then generalize the pattern to its efficient, vectorized form.