SVM

Support Vector Machine (SVM) is designed to find the optimal hyperplane that separates data into different classes in the feature space. Given that:

- $\mathbf{X}$: a matrix with the size of $N \times D$, representing the features of training set.
- $\mathbf{W}$: a matrix with the size of $D \times C$, representing the features of each categories.
- b : a vector with the size of C, representing the bias.

We consider the dot product of $\vec{x}^{(i)}$ and $\vec{w}^{(j)}$ as the score between the point i and the category j, which is also the distance from the point to the hyperplane.

So we can compute $\mathbf{Z} = \mathbf{X} \cdot \mathbf{W}$, where $\mathbf{Z}$ is called the score matrix. For point i, find a category j which has the highest score and predict that i belongs to category j.

The cost function of SVM is:

$$J(\mathbf{W}, b) = \frac{1}{n} \sum_{i=1}^{N} \sum_{j=1}^{C} \max(0, z^{(i,j)} - z^{(i,y_i)} + 1)$$

where $z^{(i,j)} - z^{(i,y_i)} + 1$ is defined as **margin**. The cost function encourage the negative margin so that we can make better prediction while optimizing the cost function.

```
def svm_loss_vectorized(W, X, y, reg):
    loss = 0.0
    dW = np.zeros(W.shape)  # initialize the
gradient as zero
    num_train = X.shape[0]
```

```
    score = np.matmul(X, W)
    target = score[np.arange(num_train), y] # The
score of the target category
    score = score - target.reshape(-1, 1) + 1 #
compute margin
    loss = np.sum(np.maximum(score, 0)) /
num_train  + reg * np.sum(W * W) # compute loss
    loss -= 1 # skip if y[i] == j, but in the
vectorized version we haven't skipped

    # only when margin>0 contributes gradient
    score = (score > 0).astype(int)
    # The target category contributes negative
gradient
    # attention that the score of target category
ought to be 1
    score[np.arange(num_train), y] -=
np.sum(score, axis = 1)
    dW = np.matmul(X.T, score) / num_train + 2 *
reg * W

    return loss, dW
```

Softmax

Firstly, compute the score matrix and the probability of each categories is defined as follow:

$$\hat{p}_i = \frac{e^{z_i}}{\sum e^{z_j}}$$

Cost function of softmax is defined as follow:

$$J(\mathbf{W}, b) = \frac{1}{n} \sum_{i=1}^{n} -\log(\hat{p}_{y_i})$$

The cost function is trying to increase $\hat{p}_{y_i}$ to 1 so that we can have a higher probability to predict y_i. Now, let's try to compute the gradient of J with respect to $\mathbf{W}$:

$$\frac{\partial J}{\partial \hat{p}_{y_i}} = -\frac{1}{\hat{p}_{y_i}}$$

$$\frac{\partial J}{\partial z_j} = \frac{\partial J}{\partial \hat{p}_{y_i}} \cdot \frac{\partial \hat{p}_{y_i}}{\partial z_j}$$

When $j = y_i$:

$$\frac{\partial \hat{p}_{y_i}}{\partial z_j} = (1 - \frac{\sum_{k \neq y_i} e^{z_k}}{\sum_k e^{z_k}})' = \frac{\sum_{k \neq y_i} e^{z_k} \cdot e^{z_{y_i}}}{(\sum_k e^{z_k})^2} = \hat{p}_{y_i} \cdot (1 - \hat{p}_{y_i})$$

$$\frac{\partial J}{\partial w_{k,j}} = -\frac{1}{\hat{p}_{y_i}} \cdot \hat{p}_{y_i} \cdot (1 - \hat{p}_{y_i}) \cdot x_{i,k} = (\hat{p}_{y_i} - 1) \cdot x_{i,k}$$

When $j \neq y_i$:

$$\frac{\partial \hat{p}_{y_i}}{\partial z_j} = -\frac{e^{z_{y_i}} \cdot e^{z_j}}{(\sum_k e^{z_k})^2} = -\hat{p}_j \cdot \hat{p}_{y_i}$$

$$\frac{\partial J}{\partial w_{k,j}} = -\frac{1}{\hat{p}_{y_i}} \cdot \hat{p}_{y_i} \cdot (-\hat{p}_j) \cdot x_{i,k} = \hat{p}_j \cdot x_{i,k}$$

In conclusion, the gradient can be described as:

$$\frac{\partial J}{\partial w_{k,j}} = (\hat{p}_j - [j = y_i]) \cdot x_{i,k}$$

```
def softmax_loss_vectorized(W, X, y, reg):
    loss = 0.0
    dW = np.zeros_like(W)

    z = np.exp(np.matmul(X, W)) # compute the
score
    sum = np.sum(z, axis = 1)
    z = z / sum.reshape(-1, 1) # transform into
probability

    N = X.shape[0]
```

```
    loss = - np.log(np.sum(z[np.arange(N), y]))
    dw = np.matmul(X.T, z) # w_k,j = sum p_i,j *
x_i,k
    for i in range(N):
        dw[:, y[i]] -= X[i]
    dw /= N
    loss /= N
    loss += reg * np.sum(W**2)
    dw += 2 * reg * W

    return loss, dw
```