Weekly Report(Apr 30 - May 6)

Liu Junnan

Abstract

1 Work Done

This week I finished assignment 1 of cs231n, which I will introduce the details regarding the implementation in this report.

1.1 Assignment1

1.1.1 Linear SVM

Recall that for the i-th example we are given the pixels of image x_i and the label y_i that specifies the index of the correct class. The score function takes the pixels and computes the vector $f(x_i, W)$ of class scores, which we will abbreviate to s (short for scores). For example, the score for the j-th class is the j-th element: $s_j = f(x_i, W)_j$. The Multiclass SVM loss for the i-th example is then formalized as follows:

$$L_i = \sum_{j \neq y} \max(0, s_j - s_{y_i} + \Delta) \tag{1}$$

where Δ is set to 1 in this experiment, and since we only experiment linear SVM, the activation function f is also linear.

The gradient with respect to W_{y_i} is

$$\frac{\partial L_i}{\partial W_{y_i}} = -(\sum_{j \neq y_i} \mathbf{1}(w_j^T x_i - w_{y_t}^T x_i + 1 > 0))x_i$$

where **1** is the indicator function that is one if the condition inside is true or zero otherwise. For the other rows where $j \neq y_i$ the gradient is:

$$\frac{\partial L_i}{\partial W_j} = \mathbf{1}(w_j^T x_i - w_{y_i}^T x_i + 1 > 0)x_i$$

First look at the trivial version of this:

```
loss = 0.0
for i in range(num_train):
  scores = X[i].dot(W)
  correct_class_score = scores[y[i]]
  for j in range(num_classes):
    if j == y[i]:
      continue
    margin = scores[j] - correct_class_score + 1 # note delta = 1
    if margin > 0:
      dW[:, j] += X[i]
      dW[:, y[i]] -= X[i]
```

```
        loss += margin

# Right now the loss is a sum over all training examples, but we want it
# to be an average instead so we divide by num_train.
loss /= num_train
dW /= num_train

# Add regularization to the loss.
loss += reg * np.sum(W * W)
dW += reg * W
```

This part of code just accomplishes forward pass of computing loss and backward pass of computing gradient, with loops. But vectorization is always preferred.

```
1 scores = X.dot(W)
2 scores = scores - scores[range(num_train), y].reshape((-1,1)) + 1
3 scores[scores<0] = 0
4 scores[range(num_train), y] = 0
5 loss = np.sum(scores) / num_train
6 loss += 0.5 * reg * np.sum(W**2)
7 scores[scores>0] = 1
8 scores[range(num_train), y] = -np.sum(scores, axis=1)
9 dW = X.T.dot(scores) / num_train + reg * W
```

Line 1 computes $W^T X$. But line 2 is quite tricky. Notice that the loss function of SVM(1) requires each row of the score matrix to subtract the y_i-th element of that row, except the y_i-th one. In line 2, "scores[range(num_train), y]" just indexes y_i-th element of each row. For example, num_train is 3, and y=[3,1,2]. "scores[range(num_train), y]" will retrieve scores[0][3], scores[1][1] and scores[2][2]. I have to say with numpy the code will be concise and elegant. To match the dimension constraint of broadcasting, we have to reshape the result of "scores[range(num_train), y]" to make it a "column vector". Line 3 is the $\max$ operation that makes all the elements less than zero zero.

There is also experiment to compare the time cost between them. The time of naive version that computes loss and gradients is 0.090697s, and the time of vectorized version is 0.004672s. We can see that vectorized one is about 19.4 times faster that naive one.

The assignment also needs us to tune the hyperparameters of SVMs like learning rate and regularization strength. Tuning hyperparameter is a tricky task, which needs a lot of engineering practices. The course provides a note where lists many useful rules of thumb as follows.

- Search hyperparameters on log scale. For example, a typical sampling of the learning rate would look as follows: "learning_rate = 10 ** np.random.uniform(-6,1)".

- Prefer random search to grid search.

Then we can shrink the range to search for better results. After tuning hyperparameters, I get 0.392 accuracy on validation set and 0.377 on test set.

1.1.2 Softmax Classifier

The softmax exercise is analogous to SVM exercise, so I will only introduce how to implement loss and gradients.

Cross-entropy loss is defined as follows:

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

And the gradient with respect to W

$$\frac{\partial L_i}{\partial W_j} = \begin{cases} \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} X_i^T & \text{,if } i \neq j \\ \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} - 1 \right) X_i^T & \text{,if } i = j \end{cases}$$

In practice we usually subtract max of the scores from the scores to guarantee numerical stability.

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i}+\log C}}{\sum_j e^{f_j+\log C}}$$

where $C = -\max_j f_j$.

```
scores = X.dot(W)
shift_scores = scores - scores.max(axis=1).reshape((-1,1))

softmax_output = np.exp(shift_scores)
softmax_sum = np.sum(softmax_output, axis=1)

loss = - np.sum(shift_scores[range(num_train), y]) + np.sum(np.log(
    softmax_sum))
loss = loss / num_train + 0.5 * reg * np.sum(W**2)

prob = softmax_output / softmax_sum.reshape((-1,1))
prob[range(num_train), y] -= 1
dW = np.dot(X.T, prob)
dW = dW / num_train + reg * W
```

After tuned, the classifier yields 0.357 accuracy on validation set and also 0.357 on test set.

1.1.3 Two layer Neural Network

The network in the exercise consists of an input layer, a hidden layer with ReLU activation, another hidden layer with linear activation and an output layer with softmax loss. The forward pass is easy:

```
fc1 = np.maximum(X.dot(W1)+ b1 ,0)
scores = fc1.dot(W2) + b2
shift_scores = scores - np.max(scores, axis=1).reshape((-1,1))
softmax_output = np.exp(shift_scores)
softmax_sum = np.sum(softmax_output, axis=1)
softmax_loss = - np.sum(shift_scores[range(N), y]) + np.sum(np.log(
    softmax_sum))
regularization = 0.5 * reg * (np.sum(W1**2) + np.sum(W2**2))
loss = softmax_loss / N + regularization
```

1.2 Numpy exercise

2 Plans