cs231n Notes

pjhades

April 9, 2016

https://cs231n.github.io/

Softmax classifier

Multiclass logistic regression.

Individual loss (softmax function):

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

where f is the score vector, f_j is the j-th element.

Interpretation

The exponential term $e^f_{y_i}$ can be viewed as **unnormalized probability**, and the division simply normalizes them. With this interpretation, we can regard **minimizing the full loss** (just the summation of the individual losses) as performing **MLE**, and regard the **regularization term in the full loss** as **a Gaussian prior**, and instead of MLE we are doing **MAP**(Maximum A Posterior) estimation.

Numeric Issues

Exponentials make results large, and dividing large numbers can be numerically unstable. So it's common to multiply a constant C to the individual loss:

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

Data Preprocessing

Common tricks:

- Mean subtraction
- Normalization
- PCA & whitening

For PCA, we do the following:

1. subtract the mean from each data point
2. compute variance-covariance matrix $\frac{1}{N}\mathbf{X^T X}$
3. SVD the covariance matrix to get the orthonormal eigenvectors as a basis
4. project the data to those basis
5. keep the top k dimensions, which are the features on which the data has the largest variance

All the preprocessing statistics (e.g. sample mean) must be computed **only on the training data**, and applied to validation/test data, since we cannot peek the validation/test set in any way.

Optimization

Momemtum update

To soothe the bad situation that normal GD/SGD may run into where the loss surface has very steep cliffs but the local minimum should be reached through a long ravine. Directly following the gradient may result in very slow convergence.

The idea is from the physics, in which we treat the loss value at each position as the potential there, so the optimization process then becomes the motion of a ball placed randomly somewhere on the loss surface with initial velocity 0. The force felt by the ball is then the negative gradient at some position.

The update first reduces the velocity of the ball by a factor μ, and calculate the new momentum incurred by the force. Finally this new velocity is used to update the position of the ball. The updated position of the ball is affected by both the gradient and its old velocity.

```
v = mu * v - learning_rate * dx
x += v # integrate position
```

Nesterov Momentum

It's almost like the momemtum update, but instead of calculating the gradient at the old position, it calculates the gradient at a **lookahead** position which is just `x + mu * v`. Then we do the rest of things as we do in normal momemtum update.

```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
```

```
x += v

# Another version. Here x always holds the parameter vector at the
# lookahead position.
v_prev = v # the old velocity
# update the velocity using the gradient at the lookahead position
v = mu * v - learning_rate * dx
# x - mu * v_prev calculates the original parameter vector
# then + v is the updated original parameter vector
# then + mu * v is the updated lookahead position
x += -mu * v_prev + (1 + mu) * v
```

Convnets

Input: width-height-depth $W \times H \times D$-sized images.

Layers: convolutional layer, poooling layer, full-connected layer.

Typical structure: Input-Conv-ReLU-Pool-FC

Convolutional layers

A convolutional layer maps the input $W \times H \times D$ volume to another $W_1 \times H_1 \times D_1$ volume.

For each feature we're interested in, we have a **receptive field** of size $X \times Y \times D$ that goes over the input volume, compute the cross product of the scanned area (convolution), and move a **stride** to the next scanning position. This will find all the positions that may contain some feature.

Zeros may be added on the left/right ends of the volume to keep the equavalent size in the output volumes.

The number of "fit" neurons can be compute by:

$$\frac{W - F + 2P}{S} + 1$$

where W the input width, F is the size of the receptive field size, P is the number of zero paddings on each end, and S is the stride length.

K neurons for different features "stacked up" to form the convolutional layer. Thus each "depth slice" in the output contains a "feature map", and there are K slices (maps) in total.

You may think of this process as "scanning" but actually each scanning position has a neuron with the same weight (called **filter** or **kernel**), i.e., looking for the same feature at different places.

So the output size is

- $W_1 = \frac{W-F+2P}{S} + 1$
- $H_1 = \frac{H-F+2P}{S} + 1$

- $D_1 = K$

Pooling layers

The target is to progressively reduce the spatial size of the representation thus reduce the amount of parameters and computation in the network.

Just like convolutional layers, but no zero padding. The filter moves by a **stride** over the input, and takes the maximum within the area.

The output size

- $W_2 = \frac{W_1 - F}{S} + 1$
- $H_2 = \frac{H_1 - F}{S} + 1$
- $D_2 = D1$

Converting from FC to Conv

我理解的是原 FC 网络每次只能处理一个“局部”，处理完整个输入需要遍历所有局部。而处理一次局部需要 forward 过整个网络。而改为卷积层之后，每个 depth 可以处理各自的特征，forward 一次即可得到所有的结果。