Machine Learning: Assignment #3
Fall 2020

Due: November 30th, 23:59:59 CST (UTC +8).

1. Neural Networks

The codes of this section are in the *neural_networks* folder.

In this problem, we will implement the entire process of the neural networks training, such as feedforward, backpropagation and optimizer. We will use *mnist_all.mat* as our experiment data. MNIST is a famous hand-written digit recognition dataset.

We will start by introducing the basic structure and building blocks of neural networks. Neural networks are made up of layers that have learnable parameters including weights and bias. Each layer gets the output from the previous layer, performs some operations and produces output. The final layer is typically a softmax function, which outputs the probability of an image being in different classes. We optimize a objective function over the parameters of every layer and then use stochastic gradient descent (SGD) to update the parameters to train a model.

(a) **Layers**

Depending on the operation in the layers, we can divide the layers into the following types:

(i) **Affine Layer (fully connected layer or dense layer)**

As the name suggests, every output neuron of the inner product layer has full connection to the input neurons. See here for a detailed explanation. The output is the multiplication of the input with a weight matrix plus a bias offset, i.e.:

$$f(x) = Wx + b$$

This is simply a linear transformation of the input. The weight parameter W and bias parameter b are learnable in this layer. The input x is a d dimensional vector, and W is an $n \times d$ matrix and b is n dimensional vector. You should already be familiar with this type through lectures.

(ii) **ReLU layer**

We add nonlinear functions after the inner product layers to model the nonlinearity of real data. One of the activation functions found to work well in image classification is the rectified linear unit (ReLU):

$$f(x) = max(0, x)$$

There are also many other activation functions such as *sigmoid* and *tanh* function. See here for a detailed comparison among them. Note that there is no learnable parameter in the ReLU layer.

(iii) **Convolution Layer**

See here for a detailed explanation of the convolution layer.

The convolution layer is the core building block of CNNs. Different from the fully connected layer, each output neuron of a convolution layer is only connected to some input neurons. As the name suggests, in the convolution layer, we apply a convolution operation with filters on input feature maps (or images). Recall that in image processing, there are many types of kernels (filters) that can be used to blur, sharpen an image or to detect edges in an image. In a convolution layer, the filter (or kernel) parameters are learnable and we want to adapt the filters to data. There is also more than one filter at each convolution layer. The input is a three dimensional tensor, rather than a vector as in the inner product layer. We represent the input feature maps (it can be the output from a previous layer, or an image from the original data) as a three dimensional tensor with height h, width w and channel c (for a color image, it has three channels).

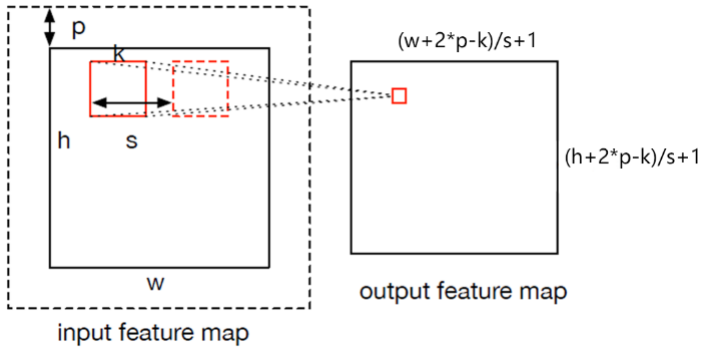

Figure 1: convolution layer

Figure 1 shows the detailed convolution operation. The input is a feature map, i.e., a three dimensional tensor with size $h \times w \times c$. Assume the (square) window size is k, then each filter is of shape $k \times k \times c$ since we use the filter across all input channels. We use n filters in a convolution layer, then the dimension of the filter parameter is $k \times k \times c \times n$. Another two hyper-parameters in the convolution operation, are the padding size p and stride step s. Zero padding is typically used; after padding, the first two dimensions of input feature maps are $(h + 2p) \times (w + 2p)$. The stride s controls the step size of the convolution operation. As Figure 1 shows, the red square on the left is a filter applied locally on the input feature map. We multiply the filter weights (of size $k \times k \times c$) with a local region of the input filter map and then sum the product to get the output feature map. Hence, the first two dimensions of the output feature map are $[(h+2p-k)/s+1][(w+2p-k)/s+1]$. Since we have n filters in a convolution layer, the output feature map is of size $[(h+2p-k)/s+1] \times [(w+2p-k)/s+1] \times n$. Besides the filter weight parameters, we also have the filter bias parameters which is a vector of size n, that is, we add one scalar to each channel of the output feature map.

Note that for the fast version of convolution layer, we employ a trick called *im2col*. This trick essentially avoids the potential loops for convolution layers, and convert the convolution operations to matrix multiplications, while the latter has been highly optimized for fast computation. The following two figures provide a graphical description of the *im2col* and *col2im*. In Figure 2, the feature window moves over the input image in a row major fashion (red, green, brown, etc). The content of the feature window will be reshaped to a column and put into a matrix M with $k \times k \times c$ rows and $h\_out \times w\_out$ columns in the same order as they are retrieved from the image.

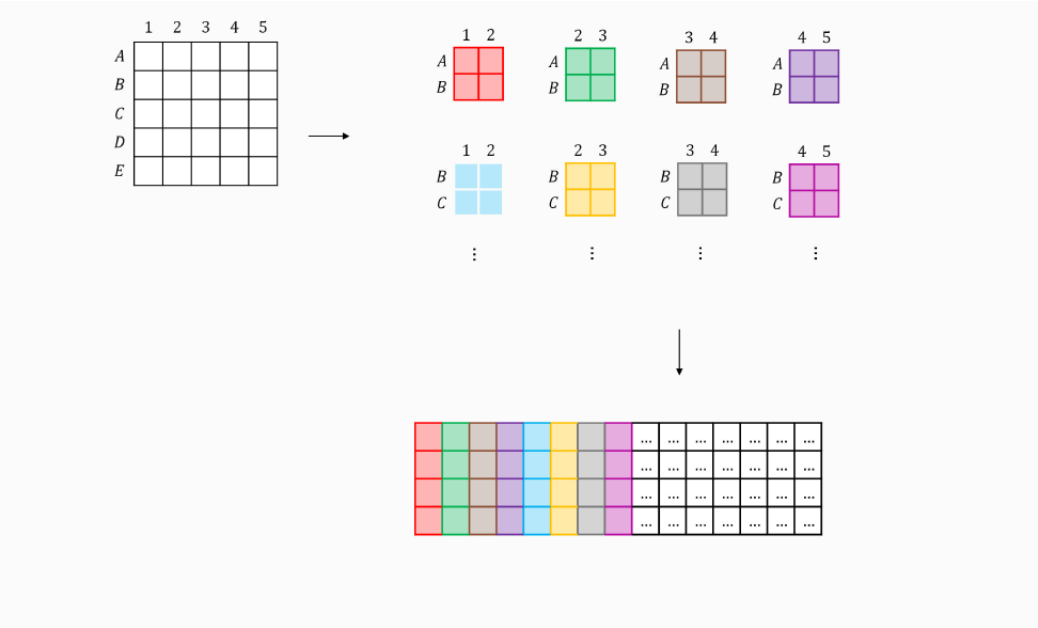

Figure 2: $im2col : s = 1, k = 2, c = 1, h\_out = w\_out = 5$

In Figure 3, a large computed feature map has been given. Then the input column is reshaped to a matrix M with $k \times k \times c$ rows and $h\_out \times w\_out$ columns. Each column in the matrix is first reshaped to a feature window of $k \times k \times c$, and added to the image in the row major order. Based on these two operations, we can convert convolution operations to the following steps:

i. Apply *im2col* to a feature map.
ii. Do matrix multiplication to get the output matrix.
iii. Apply *col2im* to get back the resulted feature map.

You do not need to implement the fast version of convolution layer and the *im2col*/*col2im* operations as we have provided you with them. However, it is beneficial to know how it works.

(iv) **Pooling Layer**

It is common to use pooling layers after convolutional layers to reduce the spatial size of feature maps. Pooling layers are also called down-sample layers. With a

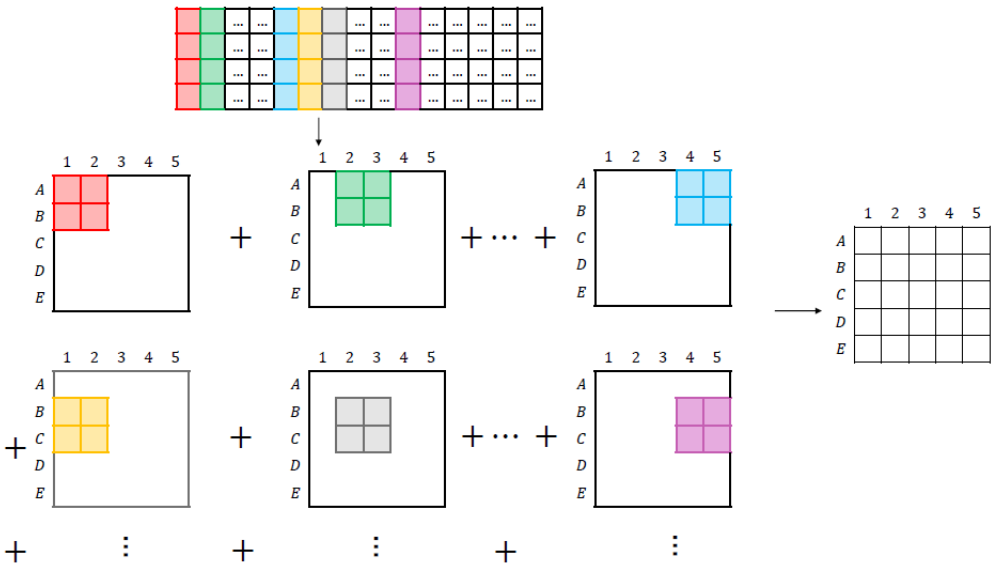

Figure 3: $col2im : s = 1, k = 2, c = 1, h\_out = w\_out = 5$

pooling layer, we can extract more salient feature maps and reduce the number of parameters of CNNs to reduce over-fitting. Like a convolution layer, the pooling operation also acts locally on the feature maps, and there are also several hyper parameters that controls the pooling operation including the windows size k and stride s. The pooling operation is typically applied independently within each channel of the input feature map. There are two types of pooling operations: max pooling and average pooling. For max pooling, for each window of size $k \times k$ on the input feature map, we take the max value of the window. For average pooling, we take the average of the window. We can also use zero padding on the input feature maps. If the padding size is p, the first two dimension of output feature map are $[(h + 2p - k)/s + 1] \times [(w + 2p - k)/s + 1]$. This is the same as in the convolutional layer. Since pooling operation is channel-wise independent, the output feature map channel size is the same as the input feature map channel size.

Refer to here for a more detailed explanation of the pooling layer.

(v) **Loss Layer**

For classification, instead of using MSE loss function, we adopt softmax loss function here. Recall that in Assignment #2, we used logistic regression to classify two classes. And softmax regression model is a model that extends logistic regression to classify more clases than two[1]. Another feasible loss function is the multi-class SVM[2]. Here provides a comparison of these two different loss layers. The implementation for these two loss layers have been provided.

[1] http://cs231n.github.io/linear-classify/#softmax

[2] http://cs231n.github.io/linear-classify/#svm

(b) **Optimizers**

(i) **SGD**

The simplest form of update is to change the parameters along the negative gradient direction (since the gradient indicates the direction of increase, but we usually wish to minimize a loss function). Assuming a vector of parameters x and the gradient dx, the simplest update has the form:

$$\boldsymbol{x} = \boldsymbol{x} - learning\_rate * d\boldsymbol{x}$$

where $learning\_rate$ is a hyperparameter - a fixed constant. When evaluated on the full dataset, and when the learning rate is low enough, this is guaranteed to make non-negative progress on the loss function.

(ii) **SGD + Momentum**

Momentum update is another approach that almost always enjoys better converge rates on deep networks. This update can be motivated from a physical perspective of the optimization problem. It has the form:

$$\boldsymbol{v} = mu * \boldsymbol{v} - learning\_rate * d\boldsymbol{x}$$
$$\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{v}$$

Here we see an introduction of a v variable that is initialized at zero, and an additional hyperparameter mu. This variable v damps the velocity and reduces the kinetic energy of the system, or otherwise the particle would never come to a stop at the bottom of a hill. Refer to here for a more detailed explanation of the SGD + Momentum.

Following the instructions in *run.ipynb* and the comments in the code files, you should be able to complete this assignment. **Please report the critical figures and critical results in your homework report, and answer the inline questions of *run.ipynb* in the report.** You also need to give the derivation of backpropagation.

2. Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train. One idea along these lines is batch normalization which was proposed in 2015.

The idea is relatively straightforward. Machine learning methods tend to work better when their input data consists of uncorrelated features with zero mean and unit variance. When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features; this will ensure that the first layer of the network sees data that follows a nice distribution. However, even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will no longer have zero mean or unit variance since they are output from earlier layers in the

network. Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.

The authors hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, they proposes to insert batch normalization layers into the network. At training time, a batch normalization layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance. To this end, the batch normalization layer includes learnable shift β and scale γ parameters for each feature dimension.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 4: Algorithm of Batch Normalization copied from the Paper by Ioffe and Szegedy mentioned above.

Now you need to derive the backpropagation of batch normalization, i.e. $\frac{\partial l}{\partial \gamma}$, $\frac{\partial l}{\partial \beta}$ and $\frac{\partial l}{\partial x_i}$., where l means loss.

Please submit your homework report to at http://courses.zju.edu.cn:8060/course/18843 in pdf format, with all your code in a zip archive.