CSCI 5561: Assignment #5

Convolutional Neural Network

1 Submission

- Assignment due: **Dec 6 (11:59:59 pm)**
- Individual assignment
- Submission through Canvas.
- NOTE: This assignment is optional. We will take the highest three scores from HW2, HW3, HW4 and HW5. You can use this assignment to make up for earlier missing or low-score assignments.
- You will complete hw5.py that contains the following functions:
  - `get_mini_batch`
  - `fc`
  - `fc_backward`
  - `loss_cross_entropy_softmax`
  - `relu`
  - `relu_backward`
  - `conv`
  - `conv_backward`
  - `pool2x2`
  - `pool2x2_backward`

  The code can be downloaded from Canvas.
- You will also submit two .npz files that contain the following trained weights:
  - `mlp.npz`: `w1`, `b1`, `w2`, `b2`
  - `cnn.npz`: `w_conv`, `b_conv`, `w_fc`, `b_fc`
- Do not change function interface (e.g. arguments and return values) in your code submission. Besides, please also keep their shapes consistent with that specified in this handout. If you really want to customize the functions for your convenience, you can add keyword arguments with default values at the back.
- Comment out visualization code snippets inside those functions before submission.
- Please submit a write-up (in pdf format) including resulting visualizations and brief descriptions.
- Please place the code (i.e. hw5.py, DO NOT RENAME) and write-up into THE SAME FOLDER, compress it, and submit the compressed file.

CSCI 5561: Assignment #5

Convolutional Neural Network

- DO NOT SUBMIT THE PROVIDED IMAGE AND DATA
- The code must be run with Python 3 interpreter.
- Use numpy array to represent matrices and vectors for this homework.
- You are not allowed to use high-level computer vision and machine learning related package functions unless explicitly mentioned. Please consult with TAs if you are not sure about the list of allowed functions.

CSCI 5561: Assignment #5

Convolutional Neural Network

2 Overview

Figure 1: You will implement (1) a multi-layer perceptron (neural network) and (2) convolutional neural network to recognize hand-written digits using the MNIST dataset.

The goal of this assignment is to implement neural networks to recognize hand-written digits in the MNIST data.

MNIST Data We have reduced the image size ($28 \times 28 \rightarrow 14 \times 14$) and subsampled the data. You can download the training and testing data from Canvas.

Description: The zip file includes two .npz files (`mnist_train.npz` and `mnist_test.npz`). Each file includes `img` and `label` variables:

- `img` is a matrix ($N \times 14 \times 14$) storing image data
- `label` is $N \times 1$ vector storing the label for each image data.

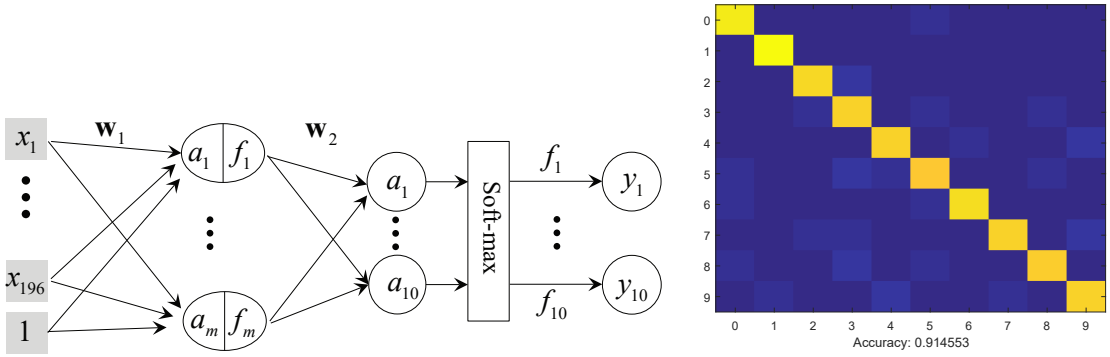
N is the number of images. You can visualize the i^{th} image, e.g.,
`plt.imshow(mnist_train['im_train'][i]), cmap='gray')`.

Note: We have provided a function called `get_MNIST_data` to read image and label data from disk.

CSCI 5561: Assignment #5

Convolutional Neural Network

3 Multi-layer Perceptron



(a) Multi-layer perceptron (b) Confusion

Figure 2: You will implement a multi-layer perceptron (30 hidden units) trained with the stochastic gradient descent method. The goal is to reach accuracy more than 85% on testing data.

You will implement a multi-layer perceptron with a single hidden layer (30 hidden units) as shown in Figure 2(a). You will train it using a stochastic gradient descent method where a pseudo-code can be found below. Through training, you are expected to see reduction of loss. As a result of training, the network should produce more than 85% accuracy on the testing data (Figure 2(b)).

Note: We PROVIDED the skeleton code that outlines the pipeline already (See also Algorithm 1 for psuedo-code). It mainly involves two functions `train_mlp` and `infer_mlp` to train and test a multi-layer perceptron. What you need to do is to implement functions for randomly sampling mini-batches, forward and backward propagation of each specific type of layer, and loss computation.

Note: In the forward propagation, your network should take an image as input, go through each layer (FC1 $\rightarrow$ ReLU $\rightarrow$ FC2 $\rightarrow$ Softmax) and finally compute a loss value. Then in the backward propagation phase, the loss derivative with respect to the weight and bias of each layer is computed in the reverse order (Softmax $\rightarrow$ FC2 $\rightarrow$ ReLU $\rightarrow$ FC1). Specifically, for a function that implements a backward propagation of a layer, you need to compute the loss derivative with respect the function inputs, given (1) loss derivative with respect the function output, and (2) function inputs. For example, for the backward propagation function of FC2 layer (`y = wx + b`), you need to compute the loss derivative with respect to the function inputs (i.e., `dl_dx, dl_dw, dl_db`), given loss derivative with respect the function output (i.e., `dl_dy`), and function inputs (i.e., `x, w, b`). Note that `dl_dx` here is then the loss derivative with respect to the output of the ReLu layer (i.e., `dl_dy` for function `relu_backward`). This is essentially an application of the chain-rule and matrix calculus. You are allowed to review online

CSCI 5561: Assignment #5

Convolutional Neural Network

Algorithm 1 Stochastic Gradient Descent based Training

Set the learning rate γ
Set the decay rate $\lambda \in (0, 1]$
Initialize the weights with a Gaussian noise $\mathbf{w} \in \mathcal{N}(0, 1)$
$k = 1$
for iIter = 1 : nIters **do**
    At every 1000^{th} iteration, $\gamma \leftarrow \lambda\gamma$
    $\frac{\partial L}{\partial \mathbf{w}} \leftarrow 0$ and $\frac{\partial L}{\partial \mathbf{b}} \leftarrow 0$
    for Each image $\mathbf{x}_i$ in k^{th} mini-batch (R samples) **do**
        Label prediction of $\mathbf{x}_i$
        Loss computation l
        Gradient back-propagation of $\mathbf{x}_i$, $\frac{\partial l}{\partial \mathbf{w}}$ using back-propagation.
        $\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{w}} + \frac{\partial l}{\partial \mathbf{w}}$ and $\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{b}} + \frac{\partial l}{\partial \mathbf{b}}$
    end for
    k++ (Set $k = 1$ if k is greater than the number of mini-batches.)
    Update the weights, $\mathbf{w} \leftarrow \mathbf{w} - \frac{\gamma}{R}\frac{\partial L}{\partial \mathbf{w}}$, and bias $\mathbf{b} \leftarrow \mathbf{b} - \frac{\gamma}{R}\frac{\partial L}{\partial \mathbf{b}}$
end for

material but must implement your own functions.

```
def get_mini_batch(im_train, label_train, batch_size)
  ...
  return mini_batch_x, mini_batch_y
```

Input: `im_train`$\in \mathbb{R}^{196\times N}$ and `label_train`$\in \mathbb{R}^{1\times N}$ are a set of vectorized images and corresponding labels, and `batch_size` is the size of the mini-batch for stochastic gradient descent.
Output: `mini_batch_x` and `mini_batch_y` are python lists that contain a set of batches (images and labels, respectively). Each batch of images is a matrix with size 196×`batch_size`, and each batch of labels is a matrix with size 10×`batch_size` (one-hot encoding). Note that the number of images in the last batch may be smaller than `batch_size`.
Description: You should randomly permute the the order of images when building the batch, and whole sets of `mini_batch_*` must span all training data.

```
def fc(x, w, b)
  ...
  return y
```

Input: `x`$\in \mathbb{R}^{m\times 1}$ is the input to the fully connected layer, and `w`$\in \mathbb{R}^{n\times m}$ and `b`$\in \mathbb{R}^{n\times 1}$ are the weights and bias.
Output: `y`$\in \mathbb{R}^{n\times 1}$ is the output of the linear transform (fully connected layer).
Description: `FC` is a linear transform of `x`, i.e., `y = wx + b`.

```
def fc_backward(dl_dy, x, w, b)
  ...
  return dl_dx, dl_dw, dl_db
```

Input: `dl_dy` $\in \mathbb{R}^{n\times 1}$ is the loss derivative with respect to the output $\mathbf{y}$ of fully connected layer.
Output: `dl_dx` $\in \mathbb{R}^{m\times 1}$ is the loss derivative with respect the input $\mathbf{x}$, `dl_dw` $\in \mathbb{R}^{n\times m}$ is the loss derivative with respect to the weights, and `dl_db` $\in \mathbb{R}^{n\times 1}$ is the loss derivative with respect to the bias.
Description: The partial derivatives w.r.t. input, weights, and bias will be computed. `dl_dx` will be back-propagated, and `dl_dw` and `dl_db` will be used to update the weights and bias.

```
def relu(x)
  ...
  return y
```

Input: `x` is a general tensor, matrix, or vector.
Output: `y` is the output of the Rectified Linear Unit (ReLu) with the same shape as input.
Description: `ReLu` is an activation unit ($\mathbf{y}_i = \max(0, \mathbf{x}_i)$).

```
def relu_backward(dl_dy, x)
  ...
  return dl_dx
```

Input: `dl_dy` is the loss derivative with respect to the output `y` of ReLu layer. It has the same shape as `y` (it can be a tensor, matrix, or vector).
Output: `dl_dx` is the loss derivative with respect to the input $\mathbf{x}$. It has the same shape as `dl_dy`.

```
def loss_cross_entropy_softmax(x, y)
  ...
  return l, dl_dx
```

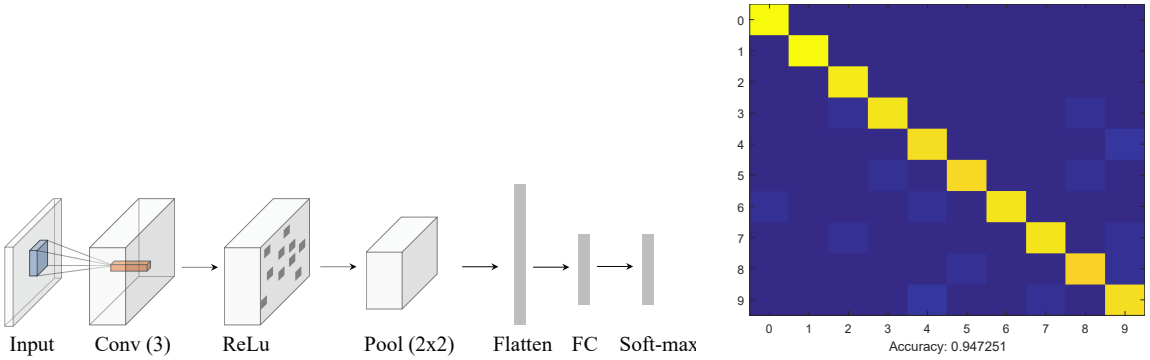
Input: `x` $\in \mathbb{R}^{m\times 1}$ is the input to the soft-max, and `y`$\in \{0,1\}^{m\times 1}$ is the ground truth label.
Output: `l`$\in \mathbb{R}$ is the loss, and `dl_dx` $\in \mathbb{R}^{m\times 1}$ is the loss derivative with respect to $\mathbf{x}$.
Description: `Loss_cross_entropy_softmax` measure cross-entropy between two distributions $l = -\sum_i^m \mathbf{y}_i \log \widetilde{\mathbf{y}}_i$ where $\widetilde{\mathbf{y}}_i$ is the soft-max output that approximates the

max operation by clamping $\mathbf{x}$ to $[0, 1]$ range:

$$\widetilde{\mathbf{y}}_i = \frac{e^{\mathbf{x}_i}}{\sum_i e^{\mathbf{x}_i}},$$

where $\mathbf{x}_i$ is the i^{th} element of $\mathbf{x}$.

4 Convolutional Neural Network



(a) CNN (b) Confusion

Figure 3: You will implement a convolutional neural network that produces accuracy more than 90% on testing data.

You will implement a convolutional neural network (CNN) as shown in Figure 3(a). The network is composed of: a single channel input ($14 \times 14 \times 1$) $\rightarrow$ Conv layer (3×3 convolution with 3 channel output and stride 1) $\rightarrow$ ReLu layer $\rightarrow$ Max-pooling layer (2×2 with stride 2) $\rightarrow$ Flattening layer (147 units) $\rightarrow$ FC layer (10 units) $\rightarrow$ Soft-max.

You will train it using the same stochastic gradient descent method as detailed in Algorithm 1. Through training, you are expected to see reduction of loss. As a result of training, the network should produce more than 90% accuracy on the testing data (Figure 3(b)).

Note: We PROVIDED the skeleton code that outlines the pipeline already. It mainly involves two functions `train_cnn` and `infer_cnn` to train and test a convolutional neural network. You need to implement functions for forward and backward propagation of convolutional and pooling layer.

```
def conv(x, w_conv, b_conv)
  ...
  return y
```

CSCI 5561: Assignment #5

Convolutional Neural Network

Input: `x` $\in \mathbb{R}^{H\times W\times C_1}$ is an input to the convolutional operation, `w_conv` $\in \mathbb{R}^{h\times w\times C_1\times C_2}$ and `b_conv` $\in \mathbb{R}^{C_2\times 1}$ are weights and bias of the convolutional operation.
Output: `y` $\in \mathbb{R}^{H\times W\times C_2}$ is the output of the convolutional operation. Note that to get the same size with the input, you may pad zero at the boundary of the input image.
Description: You can use np.pad for padding 0s at the boundary. Optionally, you may use `im2col`[1] to simplify the convolution operation.

```
def conv_backward(dl_dy, x, w_conv, b_conv)
  ...
  return dl_dw, dl_db
```

Input: `dl_dy`$\in \mathbb{R}^{H\times W\times C_2}$ is the loss derivative with respect to the output of the convolutional layer. The rest inputs are the same as defined in function `conv`.
Output: `dl_dw`$\in \mathbb{R}^{h\times w\times C_1\times C_2}$ and `dl_db`$\in \mathbb{R}^{C_2\times 1}$ are the loss derivatives with respect to convolutional weights `w_conv` and bias `b_conv`, respectively.
Description: Note that for the network with a single convolutional layer applied on input image, $\frac{\partial L}{\partial \mathbf{x}}$ is not needed.

```
def pool2x2(x)
  ...
  return y
```

Input: `x` $\in \mathbb{R}^{H\times W\times C}$ is a general tensor or matrix.
Output: `y` $\in \mathbb{R}^{\frac{H}{2}\times\frac{W}{2}\times C}$ is the output of the 2×2 max-pooling operation with stride 2.

```
def pool2x2_backward(dl_dy, x)
  ...
  return dl_dx
```

Input: `dl_dy` is the loss derivative with respect to the output `y`. `x` $\in \mathbb{R}^{H\times W\times C}$ is the input to the pooling layer.
Output: `dl_dx` $\in \mathbb{R}^{H\times W\times C}$ is the loss derivative with respect to the input `x`.

[1] `https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/convolution_layer/making_faster`