Homework Set 4: Convolutional Neural Networks for Image Classification

Instructions

- This homework contains two parts, i.e., a set of questions (30 pts, in Section 1) and a programming project (70 pts, in Section 2).
- Upload a zipped file named as `id_yourname.zip` via this link, which includes
  (1) an electronic version (saved as `id_name.pdf`, can be printed or handwritten) of your report, it includes the answers of the questions in Section 1; and describes the algorithm process, shows the results and discussions (if required) in Section 2.
  (2) a folder (named as `id_name`) includes the codes and results. You may use a Jupyter Notebook file (saved as `id_name.ipynb`) instead.
  For more details about python and Jupyter Notebook, as well as some useful packages (such as `numpy` *and* `matplotlib`*), please check the following link.*
- The deadline is 23:59 pm, June 30th, 2022.

1 Questions (30 pts)

1.1 (4 pts) Many traditional computer vision algorithms use convolutional filters to extract feature representations, e.g., in SIFT, to which we then often apply machine learning classification techniques. Convolutional neural networks also use filters within a machine learning algorithm.

(a) What is different about the construction of the filters in each of these approaches?

(b) Please declare and explain the advantages and disadvantages of these two approaches.

1.2 (6 pts) Given a neural network and the stochastic gradient descent training approach for that classifier, discuss how the *learning rate*, *batch size*, and *training time* hyperparameters might affect the training process and outcome.

1.3 (20 pts) Let us consider using a neural network (non-convolutional) to perform classification on the MNIST dataset of handwritten digits, with 10 classes covering the digits $0 \sim 9$. Each image is 28×28 pixels, and so the network input is a 784-dimensional vector $\mathbf{x} = (x_1, x_2, \dots, x_i, \dots, x_{784})$. The output of the neural network is a probability distribution $\mathbf{p} = (p_1, \dots, p_j, \dots, p_{10})$ over the 10 classes. Suppose our network has one fully-connected layer with 10 neurons — one for each class. Each neuron has a weight for each input $\mathbf{w} = (w_1, \dots, w_i, \dots, w_{784})$, plus a bias b. As we only have one layer with no multi-layer composition, there's no need to use an activation function.

When we pass in a vector $\mathbf{x}$ to this layer, we will compute a 10-dimensional output vector $\mathbf{l} = (l_1, l_2, ..., l_j, ..., l_{10})$ as:

$$l_j = \mathbf{w}_j \cdot \mathbf{x} + b_j = \sum_{i=1}^{784} w_{ij} x_i + b_j, \tag{1}$$

These distances from the hyperplane are sometimes called 'logits' (hence l) when they are the output of the last layer of a network. In our case, we only have *one* layer, so our single layer is the last layer.

Often we want to talk about the confidence of a classification. So, to turn our logits into a probability distribution $\mathbf{p}$ for our ten classes, we apply the `softmax` function:

$$p_j = \frac{e^{l_j}}{\sum_j e^{l_j}} \tag{2}$$

Each p_j will be positive, and $\sum_j p_j = 1$, and so `softmax` is guaranteed to output a probability distribution. Picking the most probable class provides our network's prediction.

If our weights and biases were trained, Eq. (2) would classify a new test example.

We have two probability distributions: the true distribution of answers from our training labels $\mathbf{y}$, and the predicted distribution produced by our current classifier $\mathbf{p}$. To train our network, our goal is to define a loss to reduce the distance between these distributions.

Let $y_j = 1$ if class j is the true label for x, and $y_j = 0$ if j is not the true label. Then, we define the *cross-entropy loss*:

$$L(w, b, x) = -\sum_{j=1}^{10} y_j \ln(p_j), \tag{3}$$

which, after substitution of Eqs. (2) and (1), lets us compute an error between the labeled ground truth distribution and our predicted distribution.

Why does this loss L work? Using the cross-entropy loss exploits concepts from information theory — Aurélien Géron has produced a video with a succinct explanation of this loss. Briefly, the loss minimizes the difference in the amounts of information needed to represent the two distributions. Other losses are also applicable, with their own interpretations.

Onto the training algorithm. The loss is computed once for every different training example. When every training example has been presented to the training process, we call this an *epoch*. Typically we will train for many epochs until our loss over all training examples is minimized.

Neural networks are usually optimized using gradient descent. For each training example in each epoch, we compute gradients via backpropagation (an application of the chain rule in differentiation) to update the classifier parameters via a learning rate λ:

$$w_{ij} = w_{ij} - \lambda \frac{\partial L}{\partial w_{ij}}, \tag{4}$$

$$b_j = b_j - \lambda \frac{\partial L}{\partial b_j}. \tag{5}$$

We must deduce $\frac{\partial L}{\partial w_{ij}}$ and $\frac{\partial L}{\partial b_j}$ as expressions in terms of x_i and p_j.

Intuition: Let's just consider the weights. Recall that our network has one layer of neurons followed by a softmax function. To compute the change in the cross-entropy loss with respect to neuron weights $\frac{\partial L}{\partial w_{ij}}$, we will need to compute and chain together three different terms:

1. the change in the loss with respect to the softmax output $\frac{\delta L}{\delta p_j}$,
2. the change in the softmax output with respect to the neuron output $\frac{\delta p_j}{\delta l_j}$, and
3. the change in the neuron output with respect to the neuron weights $\frac{\delta l_j}{w_{ij}}$.

We must derive each individually, and then formulate the final term via the chain rule. The biases follow in a similar fashion.

The derivation is beyond the scope of this class, and so we provide them here:

$$\frac{\delta L}{\delta w_{ij}} = \frac{\delta L}{\delta p_a}\frac{\delta p_a}{\delta l_j}\frac{\delta l_j}{w_{ij}} = \begin{cases} x_i(p_j - 1), a = j \\ x_i p_j, a \neq j \end{cases} \tag{6}$$

$$\frac{\delta L}{\delta b_j} = \frac{\delta L}{\delta p_a}\frac{\delta p_a}{\delta l_j}\frac{\delta l_j}{b_j} = \begin{cases} (p_j - 1), a = j \\ p_j, a \neq j \end{cases} \tag{7}$$

Here, a is the predicted class label and j is the true class label. An alternative form you might see shows $\frac{\delta L}{\delta w_{ij}} = x_i(p_j - y_j)$ and $\frac{\delta L}{\delta b_j} = p_j - y_j$ where $y_j = 1$ if class j is the true label for x, and $y_j = 0$ if j is not the true label.

We will implement these steps in code using `numpy`. We provide a code stencil `main.py` which loads one of two datasets: MINST and the scene recognition dataset. We also provide two models: a neural network, and then a neural network whose logits are used as input to an SVM classifier. Please look at the comments in `main.py` for the arguments to pass in to the program for each condition. The neural network model is defined in `model.py`, and the parts we must implement are in function `train_nn()`.

Tasks: Please follow the steps to implement the forward model evaluation and backward gradient update steps. Then, run your model on all the four conditions (NN on MNIST, and NN+SVM on MNIST, NN on SceneRec and NN+SVM on SceneRec) and report training loss and accuracy as the number of training epochs increases.

What do these numbers tell us about the capacity of the network, the value of training, and the value of the two different classification approaches?

Please also include your `train_nn()` code in your report.

```
def train_nn(self):
    indices = list(range(self.train_images.shape[0]))
    delta_W = np.zeros((self.input_size, self.num_classes))
    delta_b = np.zeros((1, self.num_classes))

    for epoch in range(hp.num_epochs):
       loss_sum = 0
        random.shuffle(indices)

        for index in range(len(indices)):
            i = indices[index]
            img = self.train_images[i]
            gt_label = self.train_labels[i]

            ################
            # FORWARD PASS:
            # Step 1:

            # Step 2:

            # Step 3:

            #loss_sum = loss_sum + your_loss

            ################
            # BACKWARD PASS (BACK PROPAGATION):
            # Step 4:

            # Step 5:

        print( "Epoch "+str(epoch)+": Total loss: "+str(loss_sum) )
```

2 Programming Project (70 pts)

2.1 Overview

We will design and train convolutional neural networks (CNNs) for scene recognition using the TensorFlow system[1]. We are going to complete the same task on the 15 scenes database with deep learning.

Task 1: Design a CNN architecture with less than 15 million parameters, and train it on a small dataset of 1,500 training examples. This isn't really enough data, so we will use:

- Standardization (a type of normalization)
- Data augmentation
- Regularization via dropout

You will be implementing standardization and augmentation in `preprocess.py`. Regularization via dropout layers will be in `YourModel`. It's a good idea to have an (at least) preliminary preprocessing routine set up before building your model that you can fine-tune later. You can see some of the results of your preprocessing function visualized after/during training under the "IMAGES" tab in Tensorboard.

Task 2: Write and train a classification head for the VGG-F pre-trained CNN to recognize scenes, where the CNN was pre-trained on ImageNet. With the weights of the pre-trained network frozen, there should be no more than 15 million trainable parameters in this model. The pretrained VGG16 model is stored in your project's code directory.

These are the two most common approach to recognition problems in computer vision today: either train a deep network from scratch — if you have enough data — or fine tune a pre-trained network.

[1]using PyTorch is okay, please try the system you are more familiar with.

In your submission, you will include your best performing weights for `YourModel` (you will not have to include weights for `VGGModel` in your submission).

The only files you need to edit for the assignment are `preprocess.py`, `your_model.py`, `vgg_model.py`, and possibly `hyperparameters.py`. The locations in these files that need editing are marked by TODO comments.

Each time the program is run, a summary of the network will be printed, including the number of trainable and non-trainable parameters. Make sure to pay attention to this so that you don't exceed the limit enforced on each network.

2.2 Rubric

(a) **(30 pts)** Task 1: Build a convolutional network with standardization, data augmentation, and regularization via dropout that achieves at least 65% test accuracy (for any training epoch) on the 15 scene database. No points will be awarded for architectures with more than 15 million parameters or extremely low test accuracy.

(b) **(20 pts)** Task 2: Train VGG-F to achieve at least 85% test accuracy (for any training epoch) on the 15 scene database. No points will be awarded for architectures with more than 15 million trainable parameters or extremely low test accuracy.

(c) **(10 pts)** Report with design decisions and evaluation.

  i) Describe your process and algorithm, show your results, describe any extra credit, and tell us any other information you feel is relevant.

  ii) Report your classification performance for each step in Task 1, plus for Task 2.

  iii) Include graphs of your loss function over time during training. You can use Tensorboard to view these graphs.

  iv) Include screenshots of the model summaries. Make sure to capture both the architecture and the number of parameters used (trainable and non-trainable).

(d) **(10 pts)** Extra credit. Each item is "up to" some amount of points because trivial implementations may not be worthy of full extra credit. Some ideas:

  i) **(up to 10 pts)** Gather additional scene training data (e.g., from the SUN database or the Places database) and train a network from scratch. Report performance on those datasets and then use the learned networks for the 15 scene database with fine tuning.

  ii) **(up to 10 pts)** Produce visualizations using your own code or methods such as mNeuron, Understanding Deep Image Representations by Inverting Them, DeepVis, or DeepDream.

Setting up TensorFlow on local machine. We will use TensorFlow through Python. For a personal machine, please visit the TensorFlow Website for installation instructions. Usually this can be achieved in your computer's terminal via the command:

```
pip3 install --upgrade tensorflow
```

If you have an NVIDIA GPU and want to use it, it may be a little more complicated to set up; please venture for yourself.

Feedback? (Optional)

Please help us make the course better. If you have any feedback for this assignment, we'd love to hear it!