AI5213/EC4213/ET5402/CT5303: Machine learning and deep learning **Fall 2020**

Coding Assignment #2

(due on Saturday Dec. 19, 2020. 11 :59PM)

Instructor: Jonghyun Choi
TA: Hyunseo Koh, Jaeyoung An *Student name (GIST ID#):*

Please specify your name and your student ID in the top heading. Push your answer sheet to github classroom.
Note : make sure that you justify your answers.

1 Implement a Fully Connected Network - 40 Points

All the functions should be implemented in `python/nn.py` and `python/q1.py`. Please include a screenshot of the running results of `python/run q1.py` after all the functions have been implemented.

1.1 Network Initialization

1.1.1 Theory [2 points]

Why is it not a good idea to initialize a network with all zeros ? If you imagine that every layer has weights and biases, what can a zero-initialized network output after training ?

1.1.2 Code [2 points]

Implement a function to initialize neural network weights with Xavier initialization [1], where $Var[w] = \frac{2}{n_i n + n_{out}}$ where n is the dimensionality of the vectors. This can be implemented by using uniform distribution to sample random numbers (see eq 16 in the paper [1]).

1.1.3 Theory [1 points]

Why do we initialize with random numbers ? Why do we scale the initialization depending on layer size (see near Fig 6 in the paper [1]) ?

1.2 Forward Propagation

1.2.1 Code [4 points]

Implement sigmoid, along with forward propagation for a single layer with an activation function, namely $y = \sigma(XW + b)$, returning the output and intermediate results for an $N \times D$ dimension input X, with examples along the rows, data dimensions along the columns.

1.2.2 Theory [2 points]

Prove that softmax is invariant to translation, which means $softmax(x_i + c) = softmax(x_i)$. Why using c = -max$x_i$ is a good idea ?

1.2.3 Theory [3 points]

Softmax can be written as a three step processes : $s_i = e^{x_i}$, $S = \sum s_i$ and $softmax(x_i) = \frac{1}{S} s_i$.

1. As $x = [x_1, x_2, ..., x_d] \in \mathbb{R}^d$, what is the range of each element ? What is the sum over all elements ?

2. One could say that *"softmax takes an arbitrary real valued vector x and turns it into a _____."*

3. What is the role of each three steps ?

1.2.4 Code [3 points]

Implement the softmax function. (Hint : Use the numerical stability trick(softmax is invariant to translation) you derived.)

1.2.5 Code [3 points]

Write a function to compute the accuracy of a set of labels, along with the scalar loss across the data. The loss function generally used for classification is the cross-entropy loss.

1.3 Back Propagation

1.3.1 Code [10 points]

Compute backpropogation for a single layer, given the original weights, the appropriate intermediate results, and given gradient with respect to the loss. You should return the gradient with respect to X so you can feed it into the next layer. As a size check, your gradients should be the same dimensions as the original objects.

1.4 Training Loop

You will tend to see gradient descent in three forms : "normal", "stochastic" and "batch". "Normal" gradient descent aggregates the updates for the entire dataset before changing the 5weights. Stochastic gradient descent applies updates after every single data example. Batch gradient descent is a compromise, where random subsets of the full dataset are evaluated before applying the gradient update.

1.4.1 Code [5 points]

Write a training loop that generates random batches, iterates over them for many iterations, does forward and backward propagation, and applies a gradient update step.

1.5 Numerical Gradient Checker

1.5.1 Code/Writeup [5 points]

Implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add offset to each element in the weights, and compute the numerical gradient of the loss with central differences. Central differences is just $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$. Remember, this needs to be done for each scalar dimension in all of your weights independently.

2 Training Models - 25 Points

In this section, all the functions should be implemented in `python/q2.py`. The initial lines of `python/q2.py` starts with loading the dataset you will use in this section.

Since our input images are 32 × 32 images, unrolled into one 1024 dimensional vector, that gets multiplied by $W^{(1)}$, each row of $W^{(1)}$ can be seen as a weight image. Reshaping each row into a 32 × 32 image can give us an idea of what types of images each unit in the hidden layer has a high response to.

We have provided you three data .mat files to use for this section. The training data in `train.mat` contains samples for each of the 26 upper-case letters of the alphabet and the 10 digits. This is the set you should use for training your network. The cross-validation set in `valid.mat` contains samples from each class, and should be used in the training loop to see how the network is performing on data that it is not training on. This will help to spot over fitting. Finally, the test data in `test.mat` contains testing data, and should be used for the final evaluation on your best model to see how well it will generalize to new unseen data.

2.1 Training with implementation

2.1.1 Code/Writeup [3 points]

Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 30 epochs. Pick the best batch size and learning rate. Check accuracy on validation set. With these settings, you should see an accuracy on the validation set of at least 75%.

2.1.2 Code/Writeup [3 points]

Generate two plots : one showing the accuracy on both the training and validation set over the epochs, and the other showing the cross-entropy loss averaged over the data. The x-axis should represent the epoch number, while the y-axis represents the accuracy or loss.

2.1.3 Code/Writeup [3 points]

Use your modified training script to train three networks, one with your best learning rate, one with 10 times that learning rate and one with one tenth that learning rate. Include all 3 plots in your writeup. Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set.

2.1.4 Code/Writeup [3 points]

Visualize the first layer weights that your network learned (using `reshape` and `ImageGrid`). Compare these to the network weights immediately after initialization. Include both visualizations in your writeup. Comment on the learned weights. Do you notice any patterns ?

2.1.5 Code/Writeup [4 points]

Visualize the confusion matrix for your best model. To make us to see the results more clearly, make it in two versions : with and without correct cases. Comment on the top few pairs of classes that are most commonly confused.

2.2 Training with Pytorch

While you were able to derive manual backpropogation rules for sigmoid and fully-connected layers, wouldn't it be nice if someone did that for lots of useful primatives and made it fast and easy to use for general computation ?

For extra credit, we will use PyTorch as a framework. Many computer vision projects use neural networks as a basic building block, so familiarity with one of these frameworks is a good skill to develop. Here, we basically replicate and slightly expand our handwritten character recognition networks, but do it in PyTorch instead of doing it ourselves. Feel free to use any tutorial you like.

For this section, you're free to implement these however you like. All of the tasks required here are fairly small and don't require a GPU if you use small networks.

2.2.1 Code/Writeup [5 points]

Re-write and re-train your fully-connected network on the included dataset in PyTorch. Plot training accuracy and loss over time. Include a discussion about your result in the writeup.

2.2.2 Code/Writeup [5 points]

Train a convolutional neural network with PyTorch on the included dataset. You're free to implement theses however you like. Plot training accuracy and loss over time. Include a discussion about your result in the writeup.

3 Image Compression with Autoencoders - 15 Points

In this section, all the functions should be implemented in `python/run q3.py`.

An autoencoder is a neural network that is trained to attempt to copy its input to its output, but it usually allows copying only approximately. This is typically achieved by restricting the number of hidden nodes inside the autoencoder ; in other words, the autoencoder would be forced to learn to represent data with this limited number of hidden nodes. This is a useful way of learning compressed representations. In this section, we will continue using the dataset you have from the previous questions.

3.1 Building the Autoencoder

3.1.1 Code [5 points]

Due to the difficulty in training auto-encoders, we have to move to the $relu(x) = max(x, 0)$ activation function. It is provided for you in `util.py`. Implement a 2 hidden layer autoencoder where the layers are

— 1024 to 32 dimensions, followed by a ReLU

— 32 to 32 dimensions, followed by a ReLU

— 32 to 32 dimensions, followed by a ReLU

— 32 to 1024 dimensions, followed by a sigmoid (this normalizes the image output for us)

The loss function that you're using is total squared error for the output image compared to the input image (they should be the same !).

3.1.2 Code [3 points]

To help even more with convergence speed, we will implement momentum. Now, instead of updating $W = W - \alpha \frac{\partial J}{\partial W}$, we will use the update rules $M_W = 0.9M_W - \alpha \frac{\partial J}{\partial W}$ and $W = W + W_M$. To implement this, populate the parameters dictionary with zero-initialized momentum accumulators, one for each parameter. Then simply perform both update equations for every batch.

3.2 Training and Evaluating the Autoencoder

3.2.1 Code/Writeup [2 points]

Using provided default settings, train the network for 100 epochs. What do you observe in the plotted training loss curve as it progresses ? What differences do you observe using new update rules ? What do you think is the reason ?

3.2.2 Code/Writeup [3 points]

Now let's evaluate how well the autoencoder has been trained. Select 5 classes from the total 36 classes in your dataset and for each selected class include in your report 2 validation images and their reconstruction. What differences do you observe that exist in the reconstructed validation images, compared to the original ones ?

3.2.3 Code/Writeup [2 points]

Let's evaluate the reconstruction quality using Peak Signal-to-noise Ratio (PSNR). PSNR is defined as

$$PSNR = 20 \times \log_{10}(MAX_I) - 10 \times \log_{10}(MSE)$$

where MAX_I is the maximum possible pixel value of the image, and MSE(mean squared error) is computed across all pixels. You may check you answer using skimage.measure.compare psnr for convenience. Report the average PSNR you get from the autoencoder across all validation images.

Reference

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010.