16-385 Computer Vision, Fall 2020

Programming Assignment 6
Neural Networks for Recognition

Due Date: Wed December 2, 2020 23:59

Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. Include the names of your collaborators in your write up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is prohibited and may lead to failure of this course.

2. **Start early!** Training a network takes time. If you start late, you will be pressed for time while debugging. Also, speed read the entire assignment before you start so you get the gist of the assignment.

3. **Extra credit:** This assignment contains two extra credit components. Don't feel pressured if you are not able to complete all of them, but if you start early, you can maximize your credit.

4. **Questions:** If you have any question, please look at piazza first. Other students may have encountered the same problem, and is solved already. If not, post your question on the discussion board. TAs will respond as soon as possible.

5. **Write-up:** Please note that we DO NOT accept handwritten scans for your write-up in this assignment. Please type your answers to theory questions and discussions for experiments electronically. Any word processor is allowed.

6. To get the data, we have included some scripts in `scripts/`.

7. For your code submission, **do not** use any libraries other than *numpy, scipy, scikit-image, matplotlib* and (in the appropriate section) *pytorch*. Including other libraries (for example, cv2, ipdb, etc.) **may lead to loss of credit** on the assignment. In your imple-mentation, feel free to change *plt.show()* to *plt.savefig('xyz.png')* to save the figure.

8. **Submission:** Your submission for this assignment should be a zip file, `<andrew-id.zip>`, composed of your write-up, your Python implementations (including helper functions), and your implementations, results for extra credit (optional). Do not submit anything from the `data/` folder in your submission. Additionally, make sure to include your implementations and results for extra credit (optional).

Your final upload should have the files arranged in this layout:

- `<AndrewID>.zip`
  - `<AndrewId>/`
    - `<AndrewId>.pdf`
    - `python/`
      - `nn.py` (*provided*)
      - `q4.py` (*provided*)
      - `run_q2.py` (*provided*)
      - `run_q3.py` (*provided*)
      - `run_q4.py` (*provided*)
      - `run_q5.py` (*optional*)
      - *Any other helper functions you need*

1 Theory - 10 points

Q1.1 Theory [**2 points**] Prove that softmax is invariant to translation, that is

$$softmax(x) = softmax(x + c) \quad \forall c \in \mathbb{R}$$

Softmax is defined as below, for each index i in a vector x.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Often we use $c = -\max x_i$. Why is that a good idea? (Tip: consider the range of values that numerator will have with $c = 0$ and $c = -\max x_i$)

Q1.2 Theory [**2 points**] Softmax can be written as a three step processes, with $s_i = e^{x_i}$, $S = \sum s_i$ and $softmax(x_i) = \frac{1}{S} s_i$.

- As $x \in \mathbb{R}^d$, what are the properties of $softmax(x)$, namely what is the range of each element? What is the sum over all elements?
- One could say that *"softmax takes an arbitrary real valued vector x and turns it into a _____"*.
- Can you see the role of each step in the multi-step process now? Explain them.

Q1.3 Theory [**3 points**] Given $y = W^T x + b$ (or $y_j = \sum_{i=1}^{d} x_i W_{ij} + b_j$), and the gradient of some loss J with respect y, show how to get $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ and $\frac{\partial J}{\partial b}$. Be sure to do the derivatives with scalars and re-form the matrix form afterwards. Here are some notional suggestions.

$$\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{k \times 1} \quad W \in \mathbb{R}^{d \times k} \quad x \in \mathbb{R}^{d \times 1} \quad b \in \mathbb{R}^{k \times 1}$$

Q1.4 Theory [**3 points**] Convolutional operations are usually implemented as matrix multiplication in practice. For example, we consider a 3×3 convolutional kernel W:

$$\begin{bmatrix} w_0 & w_1 & w_2 \\ w_3 & w_4 & w_5 \\ w_6 & w_7 & w_8 \end{bmatrix}$$

and a 3×3 single-channel image I:

$$\begin{bmatrix} x_0 & x_1 & x_2 \\ x_3 & x_4 & x_5 \\ x_6 & x_7 & x_8 \end{bmatrix}$$

When we apply the filter W to the image I with stride 1 and **no paddings** on the borders, we will obtain a 1×1 output feature map. This is because without any paddings, there is

only 1 valid location in the image to apply the filter W. This feature map can be obtained by the following matrix multiplication (followed by appropriate reshaping):

$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix}$$

Here we assume the convolutional filter is already flipped. The first matrix is obtained by flattening the convolutional filter W. The second matrix is obtained rearranging all the 3×3 blocks in image I into columns.

1. When we apply the filter W to the image I with stride 1 and 1-dim **zero padding** on each image border, we will obtain a feature map of the same size as the input image. Please write down the matrix multiplication for this convolutional operation.

2. When we apply 64 convolutional filters of size 5×5 to a single-channel input image of size 225×225, with stride 4 and no paddings, what are the dimensions of the two matrices to be multiplied to obtain the output feature map? *Hints: Consider the size of the output feature map.*

2 Implement a Fully Connected Network - 35 Points

All of these functions should be implemented in `python/nn.py` and `python/run_q2.py`. Please include a screenshot of the running results of `python/run_q2.py` after all functions have been implemented.

2.1 Network Initialization

Q2.1.1 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?

Q2.1.2 Code [2 points] Implement a function to initialize neural network weights with Xavier initialization [1], where $Var[w] = \frac{2}{n_{in}+n_{out}}$ where n is the dimensionality of the vectors. This can be implemented by using **uniform distribution** to sample random numbers (see Equation 16 in the paper [1]).

Q2.1.3 Theory [1 points] Why do we initialize with random numbers? Why do we scale the initialization depending on layer size (see near Figure 6 in the paper)?

2.2 Forward Propagation

The appendix (Section 7) has the math for forward propagation, we will implement it here.

Q2.2.1 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.

Q2.2.2 Code [3 points] Implement the softmax function. Be sure to use the numerical stability trick you derived in Q1.1 softmax.

Q2.2.3 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.

$$L_{\mathbf{f}}(\mathbf{D}) = -\sum_{(\mathbf{x},\mathbf{y})\in\mathbf{D}} \mathbf{y}\cdot\log(\mathbf{f}(\mathbf{x}))$$

Here $\mathbf{D}$ is the full training dataset of data samples $\mathbf{x}$ ($N \times 1$ vectors, N = dimensionality of data) and labels $\mathbf{y}$ ($C \times 1$ one-hot vectors, C = number of classes).

2.3 Backwards Propagation

Q2.3.1 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.

2.4 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 weights. 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.

Q2.4.1 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.

2.5 Numerical Gradient Checker

Q2.5.1 [**5 points**] Implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add ϵ 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.

3 Training Models - 15 points

First, be sure to run the script, from inside the scripts folder, `get_data.sh`. This will use `unzip` to extract files to `data/` and `image/` folders.

Since our input images are 32×32 images, unrolled into one 1024 dimensional vector, that gets multiplied by $\mathbf{W}^{(1)}$, each row of $\mathbf{W}^{(1)}$ 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 `nist36_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 `nist36_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 `nist36_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.

Q3.1 Code [5 points] Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 30 epochs. Modify the script to 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. With these settings, you should see an accuracy on the validation set of at least 75%.

Q3.2 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 4 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.

Q3.3 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?

Q3.4 Writeup [3 points] Visualize the confusion matrix for your best model. Comment on the top few pairs of classes that are most commonly confused.

4 Extract Text from Images - 15 points

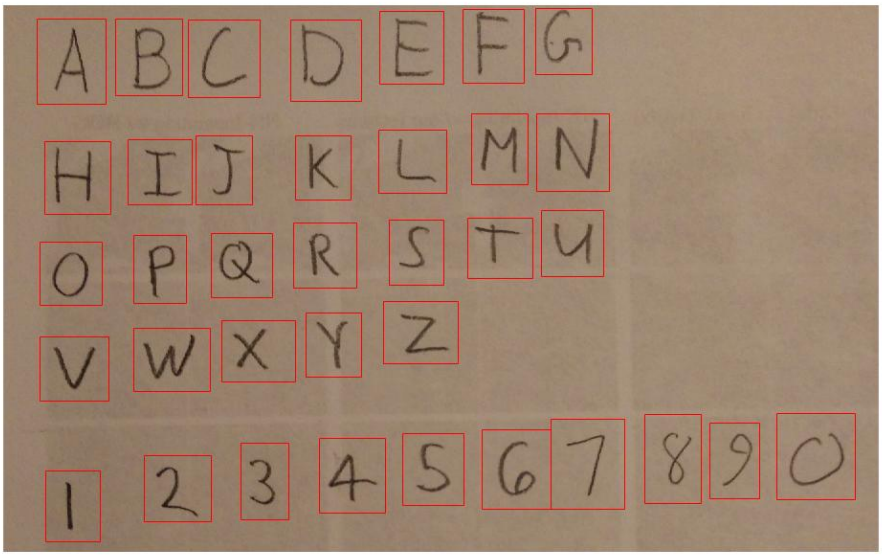

Figure 1: Sample image with handwritten characters annotated with boxes around each character.

Now that you have a network that can recognize handwritten letters with reasonable accuracy, you can now use it to parse text in an image. Given an image with some text on it, our goal is to have a function that returns the actual text in the image. However, since your neural network expects a a binary image with a single character, you will need to process the input image to extract each character. There are various approaches that can be done so feel free to use any strategy you like.

Here we outline one possible method, another is that given in a tutorial

1. Process the image (blur, threshold, opening morphology, etc. (perhaps in that order)) to classify all pixels as being part of a character or background.

2. Find connected groups of character pixels (see skimage.measure.label). Place a bounding box around each connected component.

3. Group the letters based on which line of the text they are a part of, and sort each group so that the letters are in the order they appear on the page.

4. Take each bounding box one at a time and resize it to 32×32, classify it with your network, and report the characters in order (inserting spaces when it makes sense).

Since the network you trained likely does not have perfect accuracy, you can expect there to be some errors in your final text parsing. Whichever method you choose to implement

for the character detection, you should be able to place a box on most of there characters in the image. We have provided you with `01_list.jpg`, `02_letters.jpg`, `03_haiku.jpg` and `04_deep.jpg` to test your implementation on.

Q4.1 Theory [2 points] The method outlined above is pretty simplistic, and makes several assumptions. What are two big assumptions that the sample method makes. In your writeup, include two example images where you expect the character detection to fail (either miss valid letters, or respond to non-letters).

Q4.2 Code [5 points] Find letters in the image. Given an RGB image, this function should return bounding boxes for all of the located handwritten characters in the image, as well as a binary black-and-white version of the image `im`. Each row of the matrix should contain `[y1,x1,y2,x2]` the positions of the top-left and bottom-right corners of the box. The black and white image should be floating point, 0 to 1, with the characters in black and background in white.

Q4.3 Writeup [3 points] Run `findLetters(..)` on all of the provided sample images in `images/`. Plot all of the located boxes on top of the image to show the accuracy of your `findLetters(..)` function. Include all the result images in your writeup.

Q4.4 Code/Writeup [5 points] Now you will load the image, find the character locations, classify each one with the network you trained in **Q3.1**, and return the text contained in the image. Be sure you try to make your detected images look like the images from the training set. Visualize them and act accordingly.

Run your `run_q4` on all of the provided sample images in `images/`. Include the extracted text in your writeup.

5 PyTorch [Extra Credit] - 25 points

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? Meet automatic differentiation. Since we have high-dimensional inputs (images) and low-dimensional outputs (a scalar loss), it turns out **forward mode AD** is very efficient. Popular autodiff packages include pytorch (Facebook), tensorflow (Google), autograd (Boston-area academics). Autograd provides its own replacement for numpy operators and is a drop-in replacement for numpy, except you can ask for gradients now. The other two are able to utilize GPUs to perform highly optimized and parallel computations, and are very popular for researchers who train large networks. Tensorflow asks you to build a computational graph using its API, and then is able to pass data through that graph. PyTorch builds a dynamic graph and allows you to mix autograd functions with normal python code much more smoothly, so it is currently more popular among CMU students.

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, but we like the offical one or this tutorial (in a jupyter notebook) or these slides (starting from number 35).

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.

5.1 Train a neural network in PyTorch

Q5.1.1 Code/Writeup [5 points] Re-write and re-train your fully-connected network on the included NIST36 in PyTorch. Plot training accuracy and loss over time.

Q5.1.2 Code/Writeup [2 points] Train a convolutional neural network with PyTorch on the included NIST36 dataset. Compare its performance with the previous fully-connected network.

Q5.1.3 Code/Writeup [3 points] Train a convolutional neural network with PyTorch on CIFAR-10 (`torchvision.datasets.CIFAR10`). Plot training accuracy and loss over time.

Q5.1.4 Code/Writeup [10 points] In Homework 1, we tried scene classification with the bag-of-words (BoW) approach on a subset of the SUN database. Use the same dataset in HW1, and implement a convolutional neural network with PyTorch for scene classification. Compare your result with the one you got in HW1, and briefly comment on it.

5.2 Fine Tuning

When training from scratch, a lot of epochs and data are often needed to learn anything meaningful. One way to avoid this is to instead initialize the weights more intelligently.

These days, it is most common to initialize a network with weights from another deep network that was trained for a different purpose. This is because, whether we are doing image classification, segmentation, recognition etc..., most real images share common properties. Simply copying the weights from the other network to yours gives your network a head start, so your network does not need to learn these common weights from scratch all over again. This is also referred to as fine tuning.

Q5.2.1 Code/Writeup [5 points] Fine-tune a single layer classifier using pytorch on the flowers 17 (or flowers 102!) dataset using squeezenet1_1, as well as an architecture you've designed yourself (*3 conv layers, followed by 2 fully-connected layers, is standard slide 6*) and trained from scratch. How do they compare?

We include a script in `scripts/` to fetch the flowers dataset and extract it in a way that it can be consumed by PyTorch ImageFolder from `data/oxford-flowers17` (see an example). You should look at how SqueezeNet is defined, and just replace the classifier layer. There exists a pretty good example for fine-tuning in PyTorch.

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[2] P. J. Grother. Nist special database 19 – handprinted forms and characters database. https://www.nist.gov/srd/nist-special-database-19, 1995.

6 Appendix: Neural Network Overview

Deep learning has quickly become one of the most applied machine learning techniques in computer vision. Convolutional neural networks have been applied to many different computer vision problems such as image classification, recognition, and segmentation with great success. In this assignment, you will first implement a fully connected feed forward neural network for hand written character classification. Then in the second part, you will implement a system to locate characters in an image, which you can then classify with your deep network. The end result will be a system that, given an image of hand written text, will output the text contained in the image.

6.1 Basic Use

Here we will give a brief overview of the math for a single hidden layer feed forward network. For a more detailed look at the math and derivation, please see the class slides.

A fully-connected network $\mathbf{f}$, for classification, applies a series of linear and non-linear functions to an input data vector $\mathbf{x}$ of size $N \times 1$ to produce an output vector $\mathbf{f}(\mathbf{x})$ of size $C \times 1$, where each element i of the output vector represents the probability of $\mathbf{x}$ belonging to the class i. Since the data samples are of dimensionality N, this means the input layer has N input units. To compute the value of the output units, we must first compute the values of all the hidden layers. The first hidden layer *pre-activation* $\mathbf{a}^{(1)}(\mathbf{x})$ is given by

$$\mathbf{a}^{(1)}(\mathbf{x}) = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

Then the *post-activation* values of the first hidden layer $\mathbf{h}^{(1)}(\mathbf{x})$ are computed by applying a non-linear activation function $\mathbf{g}$ to the *pre-activation* values

$$\mathbf{h}^{(1)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(1)}(\mathbf{x})) = \mathbf{g}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

Subsequent hidden layer $(1 < t \leq T)$ pre- and post activations are given by:

$$\mathbf{a}^{(t)}(\mathbf{x}) = \mathbf{W}^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)}$$
$$\mathbf{h}^{(t)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(t)}(\mathbf{x}))$$

The output layer *pre-activations* $\mathbf{a}^{(T)}(\mathbf{x})$ are computed in a similar way

$$\mathbf{a}^{(T)}(\mathbf{x}) = \mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)}$$

and finally the *post-activation* values of the output layer are computed with

$$\mathbf{f}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(T)}(\mathbf{x})) = \mathbf{o}(\mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)})$$

Figure 2: Samples from NIST Special 19 dataset [2]

where $\mathbf{o}$ is the output activation function. Please note the difference between $\mathbf{g}$ and $\mathbf{o}$! For this assignment, we will be using the sigmoid activation function for the hidden layer, so:

$$\mathbf{g}(y) = \frac{1}{1 + \exp(-y)}$$

where when $\mathbf{g}$ is applied to a vector, it is applied element wise across the vector.

Since we are using this deep network for classification, a common output activation function to use is the softmax function. This will allow us to turn the real value, possibly negative values of $\mathbf{a}^{(T)}(\mathbf{x})$ into a set of probabilities (vector of positive numbers that sum to 1). Letting $\mathbf{x}_i$ denote the i^{th} element of the vector $\mathbf{x}$, the softmax function is defined as:

$$\mathbf{o}_i(\mathbf{y}) = \frac{\exp(\mathbf{y}_i)}{\sum_j \exp(\mathbf{y}_j)}$$

Gradient descent is an iterative optimisation algorithm, used to find the local optima. To find the local minima, we start at a point on the function and move in the direction of negative gradient (steepest descent) till some stopping criteria is met.

6.2 Backprop

The update equation for a general weight $W_{ij}^{(t)}$ and bias $b_i^{(t)}$ is

$$W_{ij}^{(t)} = W_{ij}^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial W_{ij}^{(t)}}(\mathbf{x}) \qquad b_i^{(t)} = b_i^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial b_i^{(t)}}(\mathbf{x})$$

α is the learning rate. Please refer to the backpropagation slides for more details on how to derive the gradients. Note that here we are using softmax loss (which is different from the least square loss in the slides).