16720-A S19 Neural Networks for Recognition

Instructor: Srinivasa Narasimhan

TAs: Adam Harley, Peiyun Hu, Rishi Madhok,Talha Siddiqui, Anuj Pahuja and Nitin Singh

Total Points: 110

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. **Questions:** If you have any question, please look at piazza first.
3. **Submission:** The submission is on Canvas and Gradescope. **You will be submitting the writeup on Gradescope and a zip file on Canvas**. The zip file, `<andrew-id.zip>` contains your code and any results files we ask you to save. **Note: You have to submit your writeup separately to Gradescope, and include results in the write-up**.
4. **Do not** submit anything from the `data/` folder in your submission.
5. 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.
6. To get the data, we have included `get_data.sh` file in `scripts`.

Contents

1 Theory

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 [**2 points**] Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

Q1.4 Theory [**3 points**] Given the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$, derive the gradient of the sigmoid function and show that it can be written as a function of $\sigma(x)$ (without having access to x directly)

Q1.5 Theory [**12 points**] Given $y = x^T W + b$ (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 is some notional suggestions.

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

We won't grade the derivative with respect to b but you should do it anyways, you will need it later in the assignment.

Q1.6 Theory [**4 points**] When the neural network applies the elementwise activation function (such as sigmoid), the gradient of the activation function scales the backpropogation update. This is directly from the chain rule, $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$.

1. Consider the sigmoid activation function for deep neural networks. Why might it lead to a "vanishing gradient" problem if it is used for many layers (consider plotting Q1.3)?

2. Often it is replaced with $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. What are the output ranges of both tanh and sigmoid? Why might we prefer tanh ?

3. Why does $\tanh(x)$ have less of a vanishing gradient problem? (plotting the derivatives helps! for reference: $\tanh'(x) = 1 - \tanh(x)^2$)

4. tanh is a scaled and shifted version of the sigmoid. Show how $\tanh(x)$ can be written in terms of $\sigma(x)$. (*Hint: consider how to make it have the same range*)

2 Implement a Fully Connected Network

All of these functions should be implemented in `python/nn.py`

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 and you use a **uniform distribution** to sample random numbers (see eq 16 in the paper).

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 Fig 6 in the paper)?

2.2 Forward Propagation

The appendix (sec 6) 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 the 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

First, be sure to run the script, from inside the scripts folder, `get_data.sh`. This will use `wget` and `unzip` to download

http://www.cs.cmu.edu/~lkeselma/16720a_data/data.zip

http://www.cs.cmu.edu/~lkeselma/16720a_data/images.zip

and extract them 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.

Q3.1.1 Code [10 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 plot 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%. **Submit** the trained weights file q3_weights.pickle for evaluation.

Q3.1.2 Writeup [5 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.

Q3.1.3 Writeup [5 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?

Q3.1.4 Writeup [5 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

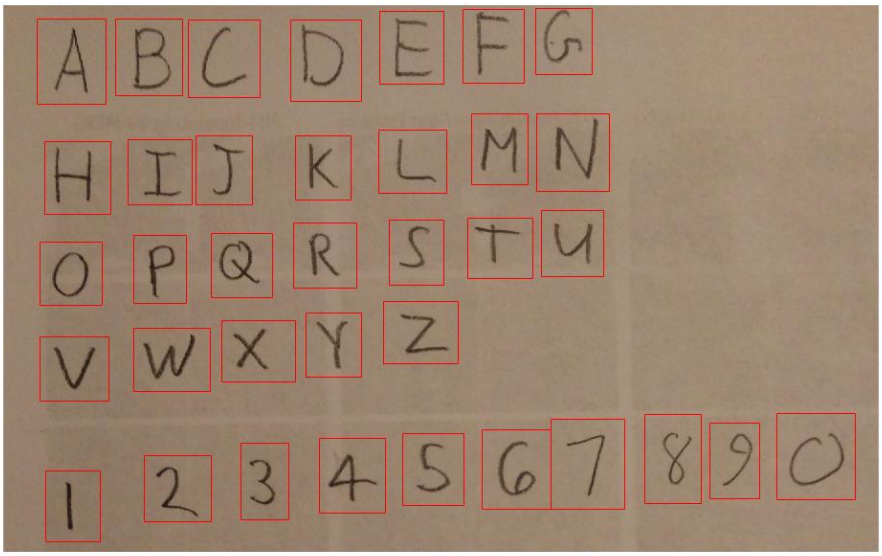

Figure 1: Sample image with handwritten characters annotated with boxes around each character.

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 [10 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 [5 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 [8 points] Now you will load the image, find the character locations, classify each one with the network you trained in **Q3.1.1**, 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 Extra Credits: 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? 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 act as shim layers for cuDNN, an implementation of auto-diff made by Nvidia for use on their GPUs. Since GPUs are able to perform large amounts of math much faster than CPUs, this makes the former two packages 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.The tasks required here are fairly small and don't require a GPU if you use small networks.

Train a neural network in PyTorch

Q5 Code/Writeup [10 points] Train a convolutional neural network with PyTorch on the EMNIST Balanced dataset and evaluate it on the findLetters bounded boxes from the images folder. Mention the extracted text in the writeup.

You should **submit** the file **q5_train.py**, **trained weights** and a script **q5_evaluate.py** which evaluates the findLetter bounded boxes using the trained network.

A network with two convolution layers and two fully connected layers along with maxpool and dropout layers should suffice for the task.

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

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

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:

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

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

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).