Carnegie Mellon University
16-720: Computer Vision
Homework 5: Neural Networks for Recognition

- **Due date.** Please refer to course schedule for the due date for **HW5**.
- **Gradescope submission.** You will need to submit both (1) a `YourAndrewName.pdf` and (2) a `YourAndrewName.zip` file containing your code, either as standalone python (`.py` or colab notebooks `.ipynb`). Remember you *must* use Gradescope's functionality to mark pages in your PDF that answer individual questions, and please make sure *all* text is legible and large enough to read; if not, you will risk loosing points!
- **Suggestions for creating a PDF.** We suggest you create a PDF by `print`ing your colab notebook from your browser. However, you are responsible for making sure all your code is visible and not cut off. Long lines of code can print poorly; we suggest you add a backslash to break a single long line into multiple lines. You also may wish to look this video for alternate pathways to convert notebooks to PDFs: https://youtu.be/-Ti9Mm21uVc?si=bo4kHfp2BoJvPpZI. In some cases, you may wish to download image or screengrabs and explicitly append them to your PDF using online tools such as https://combinepdf.com/, or insert empty cells to create adequate spacing so that you don't need to zoom out too much to prevent content from being clipped. You may also find it useful to look at this post: https://askubuntu.com/questions/2799/how-to-merge-several-pdf-files.

The starter code can be found at the course gdrive folder (you will need your andrew account to access): https://drive.google.com/drive/u/1/folders/1xu9Vp9dewSDhqgvcwQWuDnQH4-2wjMLA

We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

Please remember to list your collaborators in your report.

1 Theory

Q1.1 (3 points): Softmax is defined as below, for each index i in a vector $x \in \mathbb{R}^d$.

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

Prove that softmax is invariant to translation, that is

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

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

Q1.2.1 (1 point): 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?

Q1.2.2 (1 point): One could say that *"softmax takes an arbitrary real valued vector x and turns it into a _____"*.

Q1.2.3 (1 point): Now explain the role of each step in the multi-step process.

Q1.3 (3 points): Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

Q1.4 (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 (12 points): Given $y = Wx + b$ (or $y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$), and the gradient of some loss J (a scalar) with respect to y, show how to get the gradients $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ 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.

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

Q1.6: 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, $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$.

Q1.6.1 (1 point): 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 the gradient you derived in Q1.4)?

Q1.6.2 (1 point): 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?

Q1.6.3 (1 point): Why does $\tanh(x)$ have less of a vanishing gradient problem? (plotting the gradients helps! for reference: $\tanh'(x) = 1 - \tanh(x)^2$)

Q1.6.4 (1 point): tanh is a scaled and shifted version of the sigmoid. Show how $\tanh(x)$ can be written in terms of $\sigma(x)$.

2 Implement a Fully Connected Network

2.1 Network Initialization

Q2.1.1 (3 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 be after training?

Q2.1.2 (3 points): Implement the `initialize_weights()` function to initialize the weights for a single layer 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 [Glorot et al]).

Q2.1.3 (2 points): Why do we scale the initialization depending on layer size (see Fig 6 in the [Glorot et al])?

2.2 Forward Propagation

The appendix (sec 6) has the math for forward propagation, we will implement it here.

Q2.2.1 (4 points): Implement the `sigmoid()` function, which computes the elementwise sigmoid activation of entries in an input array. Then implement the `forward()` function which computes forward propagation for **a single layer**, namely $y = \sigma(XW + b)$.

Q2.2.2 (3 points): Implement the `softmax()` function. Be sure to use the numerical stability trick you derived in Q1.1 softmax.

Q2.2.3 (3 points): Implement the `compute_loss_and_acc()` function to compute the accuracy given 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 N data samples $\mathbf{x}$ (which are $D\times 1$ vectors, D is the dimensionality of data) and labels $\mathbf{y}$ (which are $C\times 1$ one-hot vectors, C is the number of classes), and $\mathbf{f}:\mathbb{R}^D \rightarrow [0,1]^C$ is the classifier which outputs the probabilities for the classes. The log is the natural log.

2.3 Backwards Propagation

Q2.3 (7 points): Implement the `backwards()` function to compute backpropagation for **a single layer**, given the original weights, the appropriate intermediate results, and the gradient with respect to the loss. You should return the gradient with respect to the inputs (`grad_X`) so that it can be used in the backpropagation for the previous layer. As a size check, your gradients should have the same dimensions as the original objects.

2.4 Training Loop: Stochastic Gradient Descent

Q2.4 (5 points): Implement the `get_random_batches()` function that takes the entire dataset (`x` and `y`) as input and splits it into random batches. Write a training loop that iterates over the batches, does forward and backward propagation, and applies a gradient update. The provided code samples batch only once, but it is also common to sample new random batches at each epoch. You may optionally try both strategies and note any difference in performance.

3 Training Models

Follow instructions in the `ipynb` notebook to download the data in `/content/data` folder.

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 overfitting. Finally, the test data in `nist36_test.mat` contains testing data, and should be used for the final evaluation of your best model to see how well it will generalize to new unseen data.

Q3.1 (5 points): Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 50 epochs. The code will generate two plots:

(1) the accuracy on both the training and validation set over the epochs, and

(2) the cross-entropy loss averaged over the data.

Tune the batch size and learning rate for accuracy on the validation set of at least 75%. Hint: Use fixed random seeds to improve reproducibility.

Q3.2 (3 points): The provided code will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after full training. Generate both visualizations. Comment on the learned weights and compare them to the initialized weights. Do you notice any patterns?

Q3.3 (3 points): Use the code in Q3.1 to train and generate accuracy and loss plots for each of these three networks:

(1) one with 10 times your tuned learning rate,

(2) one with one-tenth your tuned learning rate, and

(3) one with your tuned learning rate.

Include total of six plots (two will be the same from Q3.1). Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set. Hint: Use fixed random seeds to improve reproducibility.

Q3.4 (3 points): Compute and visualize the confusion matrix of the test data for your best model. Comment on the top few pairs of classes that are most commonly confused.

4 Image Compression with Autoencoders

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 NIST36 dataset you have from the previous questions.

4.1 Building the Autoencoder

Q4.1 (4 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. We will build an autoencoder with the layers listed below. Initialize the layers with the `initialize_weights()` function you wrote in Q2.1.2.

- 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)

4.2 Training the Autoencoder

Q4.2.1 (5 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 $W = W + M_W$. To implement momentum, populate the parameters dictionary with zero-initialized momentum accumulators M, one for each parameter. Then simply perform both update equations for every batch.

Q4.2.2 (6 points): Using the provided default settings, train the network for 100 epochs. The loss function that you will use is the total squared error for the output image compared to the input image (they should be the same!). Plot the training loss curve. What do you observe?

4.3 Evaluating the Autoencoder

Q4.3.1 (5 points): Now let's evaluate how well the autoencoder has been trained. Select 5 classes from the total 36 classes in the validation set and for each selected class show 2 validation images and their reconstruction. What differences do you observe in the reconstructed validation images compared to the original ones?

Q4.3.2 (5 points): Let's evaluate the reconstruction quality using Peak Signal-to-noise Ratio (PSNR). PSNR is defined as

$$\text{PSNR} = 20 \times \log_{10}(\text{MAX}_I) - 10 \times \log_{10}(\text{MSE}) \tag{1}$$

where MAX_I is the maximum possible pixel value of the image, and MSE (mean squared error) is computed across all pixels. Said another way, maximum refers to the brightest overall sum (maximum positive value of the sum). You may use skimage.metrics.peak_signal_noise_ratio for convenience. Report the average PSNR you get from the autoencoder across all images in the validation set (it should be around 15).

5 (Extra Credit) Extract Text from Images

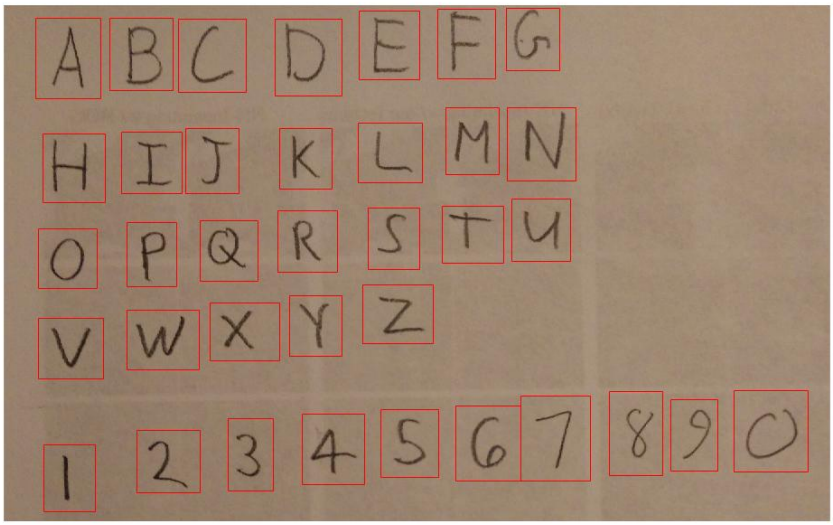

Figure 1: Sample image with handwritten characters annotated with boxes around each character.

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 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 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 character detection, you should be able to place a box on most of the 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.

Q5.1 (Extra Credit) (4 points): The method outlined above is pretty simplistic, and while it works for the given text samples, it makes several assumptions. What are two big assumptions that the sample method makes?

Q5.2 (Extra Credit) (10 points): Implement the `findLetters()` function to 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 between 0.0 to 1.0, with the characters in white and the background in black (consistent with the images in `nist36`). Hint: Since we read text left to right, top to bottom, we can use this to cluster the coordinates.

Q5.3 (Extra Credit) (3 points): Using the provided code, visualize all of the located boxes on top of the binary image to show the accuracy of your `findLetters()` function. Include all the provided sample images with the boxes.

Q5.4 (Extra Credit) (8 points): You will now 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. If you find that your classifier performs poorly, consider dilation under skimage morphology to make the letters thicker.

Your solution is correct if you can correctly detect most of the letters and classify approximately 70% of the letters in each of the sample images.

Run your code on all the provided sample images in `/content/images/`. Show the extracted text. It is fine if your code ignores spaces, but if so, please provide a written answer with manually added spaces.

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 handwritten 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 handwritten text, will output the text contained in the image.

6.1 Mathematical overview

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:

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

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 $\mathbf{y}_i$ denote the i^{th} element of the vector $\mathbf{y}$, 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 optimization 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.

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

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).

6.3 Debugging Checklist for Training Neural Networks

- Input-output pairs should make sense. Inspect them!
- Data loads correctly. Visualize it!
- Data is transformed correctly.
- Model and Data input-target dimensions must match.
- Know what the output should look like. Is the result reasonable?
- Batch size is > 1, else the gradient will be too noisy.
- Learning rate is tuned – not too small, not too large.
- Save the best checkpoint. Check on the validation set for overfitting!

- Run inference in eval mode.
- Log everything!
- Fix the random seed in PyTorch / NumPy / OS when debugging.
- Complex Error Logs are often Simple Bugs. Localize your errors in Tracebacks!
- Establish the invariants. What must be true? (Assert Statement)
- Make small changes between experiments to localize errors.
- Search Google/stack overflow with the error messages.

References

[1] X. Glorot and Y. 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.