

16-720B Computer Vision: Homework 5

3D Reconstruction

Instructors: Simon Lucey & Yaser Sheikh

TAs: Chen-Hsuan Lin, Brian Okorn, Yifan Xing, Sree Harsha Kalli,
Siddarth Malreddy, Prakhar Pradeep Naval, Khushi Gupta, Shangxuan Wu,
Bala Siva Jujjavarapu, Jingyan Wang, Yashasvi Agrawal

Due: November 27, 2017, 11:59 PM

- Please pack your system and write-up into a single file `<andrewid>.zip`, in accordance with the complete submission checklist at the end of this document.
- All tasks marked with a **Q** require a submission.
- Please stick to the provided function signatures, variable names, and file names.
- **Start early!** This homework cannot be completed within two hours!
- **Verify your implementation as you proceed** otherwise you will risk having a huge mess of malfunctioning code that can go wrong anywhere.
- If you have any questions, please post them to Piazza, or come to office hours.
- Read through the whole assignment once before starting.



Figure 1: Samples from NIST Special 19 dataset [1]

1 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.

1.1 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.

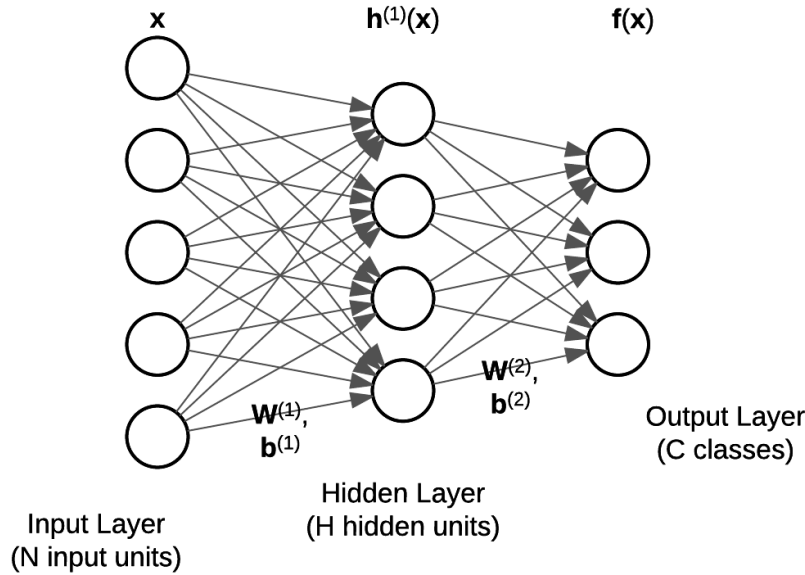


Figure 2: Example of a single hidden layer network

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

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

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

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

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.

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 network for classification, a common output activation function to use is the softmax function. This will allow us to map the real valued activations $\mathbf{a}^{(T)}(\mathbf{x})$ into a probability distribution (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:

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

Q1.1.1 “Theory” [5 points] In training deep networks, the ReLU activation function is generally preferred to the sigmoid activation function. Why might this be the case?

Q1.1.2 Theory [5 points] All types of deep networks use non-linear activation functions for their hidden layers. Suppose we have a neural network with input dimension N and output dimension C and T hidden layers. Prove that if we have a linear activation function \mathbf{g} , then the number of hidden layers has no effect on the representation capability of the network (i.e., that the set of functions that can be represented by a T layer network is exactly the same as the set that can be represented by a $T' \neq T$ layer network).

2 Implement a Fully Connected Network

In this section, you will implement all of the functions needed to initialize, train, evaluate, and use the network. Throughout this assignment, the network will be represented by its parameters \mathbf{W} and \mathbf{b} , the weights and biases of the network.

2.1 Network Initialization

In order to use a deep network, we must first initialize the weights and biases in the network. This is typically done with a random initialization, or initializing the weights from some other training procedure (which we will see in Section 3). If you are interested in issues relating to initialization, [2] is a good paper to look at.

Q2.1.1 Theory [5 points] Why is it not a good idea to initialize a network with all zeros? How about all ones, or some other constant value? (Hint: Consider what the gradients from backpropagation will look like.)

Q2.1.2 Code [5 points] Implement `[W, b] = InitializeNetwork(layers)`. This function should take as input the sizes of the layers for your neural network. The `layers` parameters will be a vector of at least 3 integers. The first element denotes the size of the data layer N , the last element denotes the size of the output layer/number of classes C , and the intermediate elements denote the size of the hidden layers. You should return \mathbf{W} and \mathbf{b} which are both cell arrays containing `length(layers)-1` elements. \mathbf{W} should contain at least two weight matrices of appropriate size, and \mathbf{b} should contain at least two bias vectors.

Q2.1.3 Writeup [5 points] Describe the initialization you implemented in Q2.1.2 and any reasoning behind why you chose that strategy.

2.2 Forward Propagation

Section 1 has the math for forward propagation. The loss function generally used for classification is the cross-entropy loss.

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

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

Q2.2.1 Code [10 points] Implement `[out, act_a, act_h] = Forward(W, b, X)` runs forward propagation on an input data sample \mathbf{X} , using the network defined by the parameters \mathbf{W} and \mathbf{b} . Both \mathbf{W} and \mathbf{b} are cell arrays containing the network parameters as initialized by `InitializeNetwork(...)`. \mathbf{X} is a vector of dimensions $N \times 1$. This function should return the final softmax output of the network in the variable `out`, which is of size $C \times 1$. The function must also return the cell arrays `act_a` and `act_h`, which contain the pre- and post-activations of the network on this input sample.

Q2.2.2 Code [5 points] Implement `[outputs] = Classify(W, b, data)`. This function should accept the network parameters `W` and `b`, as well as a $D \times N$ matrix of data samples. `outputs` should contain the softmax output of the network for each of the data samples.

Q2.2.3 Code [10 points] Implement `[accuracy, loss] = ComputeAccuracyAndLoss(W, b, data, labels)`. This function should compute the accuracy on the network with respect to the provided `data` matrix of size $D \times N$ and `labels` matrix of size $D \times C$. You should also compute and return the average cross-entropy loss on the dataset.

2.3 Backwards Propagation

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.

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

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

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

Q2.3.1 Code [30 points] Implement `[grad_W, grad_b] = Backward(W, b, X, Y, act_h, act_a)` that runs the back propagation algorithm on a single input example `X` and the ground truth vector `Y`. The function takes as input the network parameters `W` and `b` and the network pre- and post-activations `act_a` and `act_h` from calling `Forward` on `X`. The function should return the computed gradient updates for the network parameters. `grad_W` and `grad_b` are both cell arrays identical in shape to the kinds produced by `InitializeNetwork(...)`, but containing the gradient updates to make to each of the network parameters.

Q2.3.2 Code [5 points] Implement `[W, b] = UpdateParameters(W, b, grad_W, grad_b, learning_rate)` that computes and returns the updated network parameters `W` and `b`. The function is given the old parameters, the parameters gradients `grad_W` and `grad_b` returned by `Backward(...)`, and the supplied `learning_rate`.

2.4 Training Loop

When training a neural network using Gradient Descent, there are two options one can take for the updates, batch or stochastic updates. In batch gradient descent, we compute the gradient for all the examples in the training set and then average the gradients before updating the weights. In stochastic, we compute the gradient with one sample, update the weights with that gradient and carry on with the next sample. Each update is called a *iteration*, one complete pass through the entire dataset is called an *epoch*. In case of stochastic gradient descent, number of iterations in an epoch equals the number of training samples.

Q2.4.1 Theory [5 points] Give pros and cons for both stochastic and batch gradient descent. In general, which one is faster to train in terms of number of epochs? Which one is faster in terms of number of iterations?

Q2.4.2 Code [10 points] Implement

`[W, b] = Train(W, b, train_data, train_label, learning_rate)` which trains the network for one epoch on `train_data` and ground-truth `train_label`. The function should return the updated network parameters.

2.5 Gradient Checker [25 points]

Often, when implementing new layers in popular packages like Caffe or Torch you will have to implement the forward and backward pass for that layer. While the forward is fairly easy, a lot of things can go wrong in the backward pass. One common way to check if the backward pass is correct is by writing a gradient checker. A good explanation of a gradient checker is in the link: <http://ufldl.stanford.edu/tutorial/supervised/DebuggingGradientChecking/>

Q2.5.1 Code [25 points] Implement a script `checkGradient.m` that checks gradients of the loss with respect to a few random weights in each layer. Submit the code.

3 Training Models

We now have all the code required to train and use a deep network. In this section, you will train and evaluate different models which will then be used in Section 4 for parsing text in images.

3.1 From Scratch

Often times when you encounter new problems in deep learning, you will need to train a new network from scratch. This means that you will randomly initialize the weights in some way, and run multiple iterations of back propagation. The `Train.m` function you implemented in runs one epoch of training, passing over all data samples once. To fully train a deep network you typically must do many epochs.

Once a deep network is trained, one way to gain some insight into what it is doing is to visualize the learned weights. With convolutional neural networks as seen in class, each of the learned filters can be visualized as an image. This can also be done with a fully connected network, as you have implemented here. 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 `nist26_train.mat` contains 299 samples for each of the 26 upper-case letters of the alphabet. This is the set you should use for training your network. The cross-validation set in `nist26_valid.mat` contains 100 samples from each class, and should be used in the training

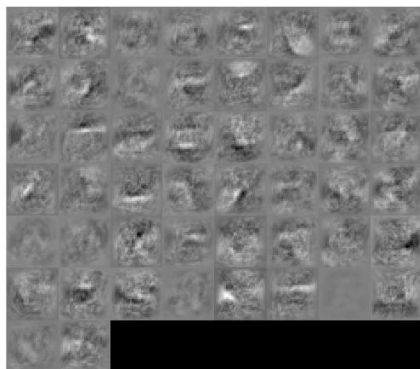


Figure 3: Samples weights from a network with 50 hidden units trained for 30 epochs. Your weights will likely look very different, as you are training with more hidden units.

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 `nist26_test.mat` contains another 100 samples per class, 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 [5 points] Use the provided `train26.m` to train a network from scratch. Use a single hidden layer with 400 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. 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%.

Q3.1.2 Writeup [5 points] Use your modified training script to train two networks, one with learning rate 0.01, and another with learning rate 0.001. 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] Using the best network from the previous question, report the accuracy and cross-entropy loss on the test set, and visualize the first layer weights that your network learned (using `reshape` and `montage`). 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 as a 26×26 image (upscale the image so we can actually see it). Comment on the top two pairs of classes that are most commonly confused.

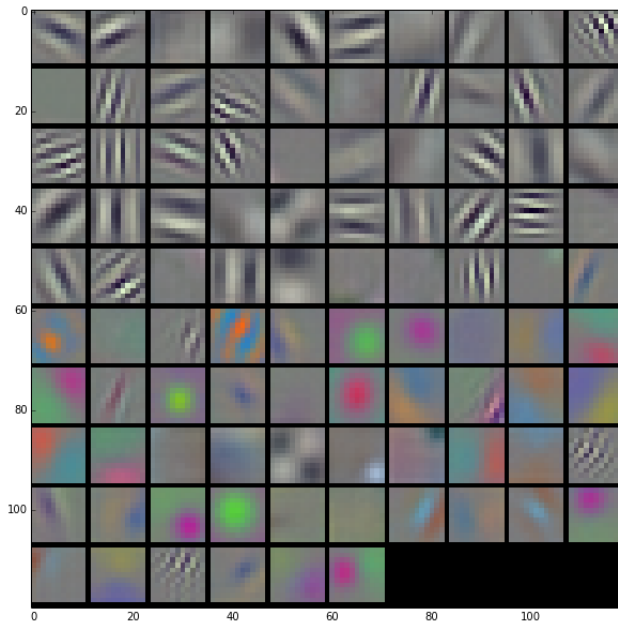


Figure 4: Learned filters from AlexNet trained on the ImageNet dataset. AlexNet is a convolutional neural network whose architecture and weights are often used as the basis for new networks.

3.2 Fine Tuning

When training from scratch, a lot of epochs are often needed to learn anything meaningful. One way to avoid this is to instead initialize the weights more intelligently. Various strategies have been used for initializing neural networks, such as unsupervised pretraining with Auto-Encoders or Restricted Boltzmann Machines.

However thanks to the explosion in popularity of deep learning for computer vision, it is often possible to also 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.

We have trained a network for recognizing capital letters using 800 hidden units, and trained it for 60 epochs. The network parameters are stored in `nist26_model_60iters.mat`. Using these weights, you will initialize and fine-tune a network for recognizing both capital letters as well as numeric digits.

Q3.2.1 Code/Writeup [10 points] Make a copy of `train26.m` and name it `finetune36.m`. Modify this script to load the data from `nist36_*.mat`, and train a network to classify both written letters and numbers. Finetune (train) this network for 5 epochs with learning rate

0.01, and include plots of the accuracy and cross-entropy loss in your writeup.

Q3.2.2 Writeup [5 points] Once again, visualize the network's first layer weights before and after training. Comment on the differences you see. Also report the network's accuracy and loss on the test set.

Q3.2.3 Writeup [5 points] Visualize the confusion matrix for your best model as a 36×36 image (upscale the image so we can actually see it). Comment on the top two pairs of classes that are most commonly confused. How has introducing more classes affected the network?

4 Extract Text from Images

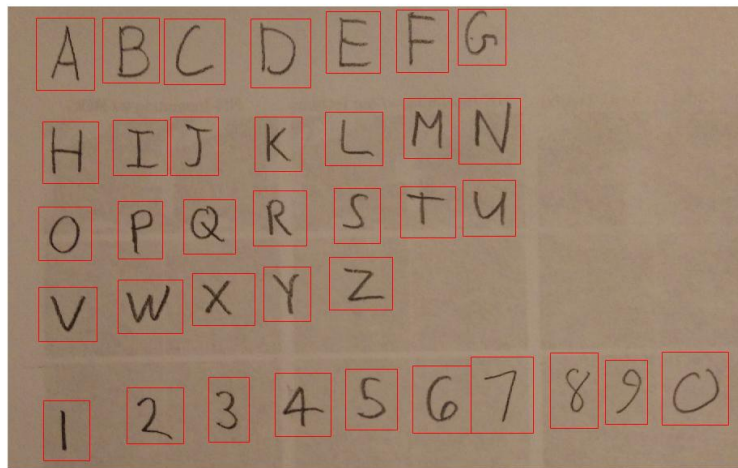


Figure 5: Sample image with handwritten characters annotated with boxes around each character.

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:

1. Process the image (blur, threshold, etc...) to classify all pixels as being part of a character or background.
2. Find connected groups of character pixels (see `bwconncomp`). 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 these 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 [5 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 [25 points] Implement `[lines, bw] = findLetters(im)`. Given an RGB image, this function should return a cell array `lines` containing all of the located handwritten characters in the image, as well as a binary black-and-white version of the image `im`. The `letters` cell array should contain one entry for each line of text that appears in the image. Each entry should be a matrix of size $L_i \times 4$, where L_i represents the number of characters in the i^{th} line of text. Each row of the matrix should contain `[x1, y1, x2, y2]` the positions of the top-left and bottom-right corners of the box. The processed image `bw` should be only black and white (containing pixel values 0.0 and 1.0), with the characters in black and the rest of the image in white. Also include a test script `testFindLetters.m` that runs your function on all of the provided images (and any others you decided to include), and displays a figure showing the bounding boxes around each letter.

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 [15 points] Implement `[text] = extractImageText(fname)`. This function should accept an image path `fname`, and return the text contained in the image. This function will load the image, find the character locations, classify each one with the network you trained in **Q3.2.1**, and return the text contained in the image. Also include a test script `testExtractImageText.m` that runs your function on all of the provided images (and any others you decided to include), and outputs the retrieved text for each image.

Q4.5 Writeup [5 points] Run your `extractImageText(...)` on all of the provided sample images in `images/`. Include the extracted text in your writeup.

5 Image Compression with Autoencoders

Note: You do not need to submit code for this section. However, you need to install the newest MATLAB 2017b with the (new) Neural Network Toolbox. You will not be able to do this section with MATLAB 2017a.

Deep learning is not only useful for supervised learning such as classification; they also perform well in *unsupervised learning* scenarios, where only the data is available to you. In this section, you will explore an important type of network architecture in deep learning research called *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 very useful way of learning compressed representations.

In this section, we will continue using the NIST36 dataset you have from the previous questions.

5.1 Building the Autoencoder [2 points]

Take a look at the provided code `define_autoencoder.m`. We are going to build a simple autoencoder with the serial architecture

- 2D convolution: 4 filters of size 4×4 , padding=[1,1], stride=[2,2]
- 2D convolution: 8 filters of size 4×4 , padding=[1,1], stride=[2,2]
- 2D convolution: 64 filters of size 8×8 , no padding, stride=[1,1]
- 2D transposed convolution: 8 filters of size 8×8 , no cropping, stride=[1,1]
- 2D transposed convolution: 4 filters of size 4×4 , cropping=[1,1], stride=[2,2]
- 2D transposed convolution: 1 filter of size 4×4 , cropping=[1,1], stride=[2,2]

Note that the output feature of the last 2D convolution will be of size $1 \times 1 \times 64$.

Notice that this roughly follows a symmetric structure. The output dimensions will be exactly the same as the input dimensions. We will use our input images to supervise the output of the network and optimize with pixel-wise L_2 loss; we do not need any labels. *Transposed convolution* here (often misinterpreted as deconvolution) is a similar operation to convolution, but the filters are for increasing spatial resolutions rather than reducing them. They are useful operations for generative models that creates data from hidden representations.

Q5.1 Writeup [2 points] There is something wrong with `define_autoencoder.m`. What is it? Correct the code, and explain how you did it.

5.2 Training the Autoencoder [6 points]

After defining the autoencoder, let's implement `train_autoencoder.m`. You will need to load the training data and reshape/adjust the dimensions appropriately before it can be read into the network. Finally, complete the unfinished call to the built-in function `trainNetwork()`.

Q5.2.1 Writeup [3 points] Using the provided default settings, train the network for 3 epochs. What do you observe in the plotted training loss curve as it progresses?

There are various factors that can impact the training of neural networks. Two of the most important ones are the learning rate and the weight initialization. Adjust these two hyperparameters to make the training loss decrease faster. As a reference, a good training setting in this problem would make the training mini-batch loss decrease to around 4 after the 3rd epoch.

Q5.2.2 Writeup [3 points] What adjustments did you make to improve the training performance? Include a snapshot of the plotted training progress in your report.

5.3 Evaluating the Autoencoder [5 points]

Now let's evaluate how well the autoencoder has been trained. In `eval_autoencoder.m`, load the test data and perform the same data preprocessing as you did with the training data. Complete the unfinished call to the built-in function `predict()`. The output of `predict()` should give you the reconstructed version of the test images. Remember that these reconstructions can be represented with only 64 numbers instead of the original $32 \times 32 = 1024$.

Q5.3.1 Writeup [3 points] From each of the 36 classes, include in your report 2 test images and their reconstruction. You may use `test_labels` to help you find the corresponding classes. The function `subplot` will also be useful. Since dimensionality reduction with autoencoders belong to the class of lossy compression, the exact original data will not be recovered. What differences do you observe that exist in the reconstructed test images, compared to the original ones?

Q5.3.2 Writeup [2 points] Let's evaluate the reconstruction quality using Peak Signal-to-noise Ratio (PSNR). PSNR is defined as

$$\text{PSNR} = 20 \cdot \log_{10}(\text{MAX}_I) - 10 \cdot \log_{10}(\text{MSE}) , \quad (1)$$

where MAX_I is the maximum possible pixel value of the image, and MSE (mean squared error) is computed across all pixels. You may use the built-in function `psnr()` for convenience. Report the average PSNR you get from the autoencoder across all test images.

5.4 Comparing against PCA [14 points]

As a baseline for comparison, we will use one of the most popular methods for data dimensionality reduction – Principle Component Analysis (PCA). PCA allows one to find the best low-rank approximation of the data by keeping only a specified number of principle components. To perform PCA, we will use a factorization method called Singular Value Decomposition (SVD). For further reading, please refer to https://en.wikipedia.org/wiki/Principal_component_analysis and https://en.wikipedia.org/wiki/Singular-value_decomposition.

Run `[U,S,V] = svd(train_data)`. One of the matrices will be an orthonormal matrix that indicates the components of your data, sorted by their importances. Extract the first 64 principle components and form a projection matrix; you will need to figure out how to do these from the `U,S,V` matrices.

Q5.4.1 Writeup [2 points] What is the size of your projection matrix? What is its rank?

This projection matrix was “trained” from our training data. Now let’s “test” it on our test data. Use the projection matrix on `test_data` to obtain the reconstructed test images. Note that these reconstructions can also be represented with only 64 numbers.

Q5.4.2 Writeup [3 points] From each of the 36 classes, include in your report 2 test images and their reconstruction. You may use `test_labels` to help you find the corresponding classes. What differences do you observe that exist in the reconstructed test images, compared to the original ones? How do they compare to the ones reconstructed from your autoencoder?

Q5.4.3 Writeup [3 points] Report the average PSNR you get from PCA. Is it better than your autoencoder? Why?

Q5.4.4 Writeup [3 points] Train another autoencoder with your current settings, but *without* the corrections you made in Q5.1. Include the same 2 test images and their reconstruction in your report, and report the average PSNR you get from the autoencoder and from PCA. What similarity/difference do you observe? Why is there such similarity/difference?

Q5.4.5 Writeup [3 points] Count the number of learned parameters for both your autoencoder and the PCA model. How many parameters do the respective approaches learn? Why is there such a difference in terms of performance?

5.5 Comparing against JPEG-2000 [3 points]

Finally, let's compare against one of the standard image coding algorithms, JPEG-2000. We will not cover the details of JPEG-2000; if you're interested, refer to https://en.wikipedia.org/wiki/JPEG_2000.

Write a for loop to save all your test images in the JPEG-2000 format and read it back into MATLAB. Use `imwrite` and save with the file extension `.jp2` with the option `CompressionRatio` set to 16 (= 1024/64). Use `imread` to read them back.

Q5.5 Writeup [3 points] What is the average PSNR of the test images compressed with JPEG-2000? Is it better than your autoencoder? Explain why we still use hand-crafted algorithms like JPEG-2000 instead of autoencoders for standard image compression.

6 Submission

6.1 Handin Zip Structure

- <andrew id>/
- <andrew id>.pdf
- code/
 - Backwards.m (30 points)
 - checkGradient.m (25 points)
 - Classify.m (5 points)
 - ComputeAccuracyAndLoss.m (10 points)
 - extractImageText.m (10 points)
 - findLetters.m (20 points)

- finetune36.m (5 points)
- Forward.m (10 points)
- InitializeNetwork.m (5 points)
- Train.m (10 points)
- testFindLetters.m (5 points)
- testExtractImageText.m (5 points)
- train26.m (5 points)
- UpdateParameters.m (5 points)
- Any other code or helper functions you used

6.2 Writeup Summary

Q1.1.1 Theory [5 points] ReLU?

Q1.1.2 Theory [5 points] Linear activation function.

Q2.1.1 Theory [5 points] Zero or one initialization.

Q2.1.3 Writeup [5 points] Describe initialization strategy.

Q2.4.1 Theory [5 points] Batch vs. stochastic gradient descent.

Q3.1.2 Writeup [5 points] Training plots and comments on `nist26` data

Q3.1.3 Writeup [5 points] Network test set performance and weights visualization

Q3.1.4 Writeup [5 points] Confusion matrix image visualization and comments

Q3.2.1 Writeup [5 points] Training plot for fine tuning

Q3.2.2 Writeup [5 points] Fine tuned weights visualization

Q3.2.3 Writeup [5 points] Confusion matrix image for fine tuned network

Q4.1 Theory [5 points] Failure cases for sample character detection algorithm

Q4.3 Writeup [5 points] Character detection results on each of the images in `images/`

Q4.5 Writeup [5 points] Extracted text from each of the images in `images/`

Q5.1 Writeup [2 points] Fixing `define_autoencoder.m`

Q5.2.1 Writeup [3 points] Train Autoencoder

Q5.2.2 Writeup [3 points] Improve Training Performance

Q5.3.1 Writeup [3 points] Evaluate Autoencoder

Q5.3.2 Writeup [2 points] Evaluate Autoencoder with PSNR

Q5.4.1 Writeup [2 points] Projection Matrix

Q5.4.2 Writeup [3 points] Evaluate PCA

Q5.4.3 Writeup [3 points]] Evaluate PCA with PSNR

Q5.4.4 Writeup [3 points]] Evaluate Autoencoder without correction

Q5.4.5 Writeup [3 points]] Number of Parameters

Q5.5 Writeup [3 points]] Comparison with JPEG-2000

References

- [1] P. J. Grother. Nist special database 19 handprinted forms and characters database. <https://www.nist.gov/srd/nist-special-database-19>, 1995.
- [2] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feed-forward neural networks.