Mini-project 5

CMPSCI 670, Fall 2019, UMass Amherst

Instructor: Subhransu Maji
TAs: Aruni RoyChowdhury, Archan Ray

Guidelines

Submission. Submit a *single pdf document* via moodle that includes your solutions, figures and code. The latex source file for the homework is provided which you can modify to produce your report. You are welcome to use other typesetting software as long as the final output is a pdf. For readability you may attach the code printouts at the end of the solutions within the same pdf. Note that we will not run your code. Similarly figures should be included in a manner which makes it easy to compare various approaches. Poorly written or formatted reports will make it harder for the TA to evaluate it and may lead to a partial deduction of credit.

Late policy. You could have 24 hours late submission with a 50% mark down. Late submission beyond 24 hours will not be given *any* credits.

Plagiarism. We might reuse problem set questions from previous years, covered by papers and webpages, we expect the students not to copy, refer to, or look at the solutions in preparing their answers. We expect students to want to learn and not google for answers.

Collaboration. The homework must be done individually, except where otherwise noted in the assignments. 'Individually' means each student must hand in their own answers, and each student must write their own code in the programming part of the assignment. It is acceptable, however, for students to collaborate in figuring out answers and helping each other solve the problems. We will be assuming that you will be taking the responsibility to make sure you personally understand the solution to any work arising from such a collaboration.

Using other programming languages. We made the starter code available in Python and Matlab. You are free to use other languages such as Octave or Julia with the caveat that we won't be able to answer or debug non Matlab/Python questions.

Python requirements. We will be using Python 2.7. The Python starter code requires `scipy`, `numpy` (at least v1.12), and `scikit-image`. If you are not familiar with installing those libraries through some package manager (like `pip`), the easiest way of using them is installing Anaconda.

The MNIST dataset

In this part your goal is to implement several image features and evaluate their performance using logistic-regression classifiers and convolutional networks on three variants of the MNIST digits dataset:

- **normal** – the standard MNIST dataset (File `digits-normal.mat`)
- **scaled** – where the pixel values of each image in the normal dataset are replaced as $I \leftarrow a * I + b$ where a and b are random numbers $\in [0, 1]$ (File `digits-scaled.mat`).
- **jittered** – where the image is translated within each image randomly between $[-10, 10]$ pixels both in the horizontal and vertical directions (File `digits-jittered.mat`).

The "normal" dataset can be loaded into Matlab by typing `load('../data/digits-normal.mat')`. In Python, we added a function in `utils.py` called `loadmat` that does the same thing. This loads a structure called `data` which has the images, labels and the splits in variables `x`, `y`, and `set` respectively. E.g., `data.x` is an array of size $28 \times 28 \times 1 \times 2000$ containing 2000 digits. Class labels are given by the variable `data.y` $\in \{0, 1, \ldots, 9\}$. Thus, pixel `(i,j)` in the the `k'th` training example is `data.x(i,j,1,k)` with label `data.y(k)`. The `data.set` has values $\in \{1, 2, 3\}$ corresponding to train, val, and test splits respectively. In Python, `data` is a dictionary and you can access the fields in a similar way: `data.y` is `data['y']`, `data.set` is `data['set']`. An important difference is that `data['x']` is an array of size $28 \times 28 \times 2000$ instead of $28 \times 28 \times 1 \times 2000$. So, the pixel `(i,j)` in the the `k'th` training example is `data['x'][i,j,k-1]` with label `data['y'][k-1]`.

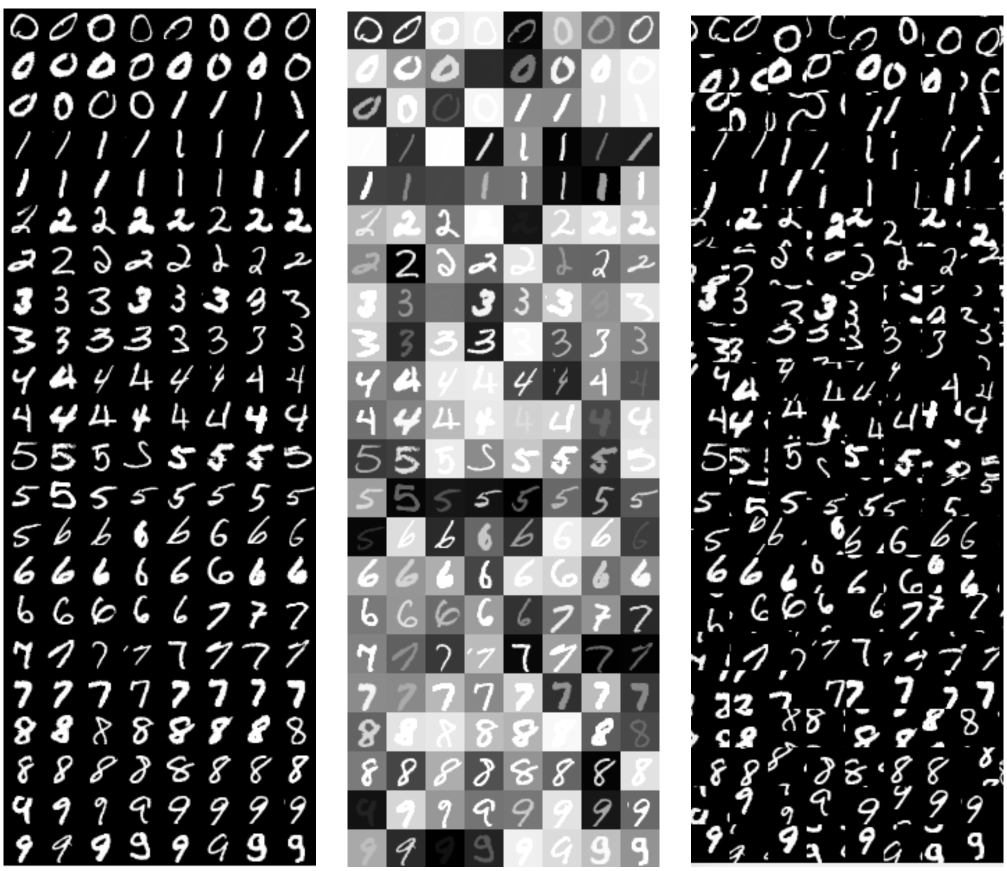
(a) normal (b) scaled (c) jittered

Figure 1: Example images from different variants of the MNIST dataset. You can visualize the images using `montageDigits` function included in the codebase.

1 Handcrafting image representations

Implement three different image representations described below.

1. **Pixel features.** Implement the function `features = extractDigitFeatures(x, 'pixel')` that takes as input the images `x` and returns their pixel values as a feature. You have to simply reshape the 28×28 pixel image into a vector of size 784×1. The variable `features` is a array of size $784 \times N$ where N is the number of images in `x`.

2. **Histogram of oriented gradients.** Implement the function `features = extractDigitFeatures(x, 'hog')` that takes as input the images `x` and returns their histogram of oriented gradients (HoG) as a feature. Recall the steps in the HOG feature computation are:

   (a) Apply non-linear mapping to the values of the image. In the HOG paper of Dalal and Triggs, CVPR 2005, mapping the pixel intensities through a non-linearity such as $x \leftarrow x^{\gamma}$ or $x \leftarrow \log(x)$ was found to be useful.

   (b) Compute horizontal and vertical gradients g_x and g_y at each pixel using derivative operators. Compute the magnitude $m = \sqrt{g_x^2 + g_y^2}$ and orientation $\theta = \arctan(g_y/g_x)$ at each pixel. Note that this version of angle lies between $-\pi/2$ to $\pi/2$. You may also use the version that computes angles between $-\pi$ to π using $\theta = \arctan(g_x, g_y)$. Construct the orientation bins accordingly.

   (c) Given a `binSize` and number of orientations (`numOri`) define a grid of `binSize` $\times$ `binSize` pixels across the image. Within each cell compute the orientation histogram by assigning each pixel to the nearest orientation in `numOri` and with vote proportional to the gradient magnitude. Concatenate these histograms to obtain an image representation. Use a spatial binSize of 4 and 8 orientation bins.

   (d) Local and global feature normalization.

   To begin with you can ignore initial non-linear mapping (Step 1) and local normalization (Step 4). You will investigate the effect of global normalization as described in the next section.

3. **Local binary patterns.** In class we discussed various ways of constructing a bag-of-words representation for images. The steps are (1) extract local features on a dense grid or at interest points, (2) assign them to codewords based on a learned dictionary, and (3) build a histogram of word counts per image. Local binary patterns (LBP) combine feature extraction and codeword assignment and have been shown to be highly effective for image classification.

   Your goal is to implement a simplified version of LBP in this homework. In this version you will represent each image as a collection of all of its 3x3 patches (overlapping). If each pixel is represented by a 8 bit intensity value then the number of such patches are $8^{(3\times3)} = 134,217,728$, which is far larger than the number of pixels in an image. Clustering these patches across images would give you a dictionary.

   The LBP approach instead directly maps each 3x3 patch into a number between 0 and 255 as follows. Consider the pixels in a 3x3 patch. Subtract from each pixel the value of the center pixel and assign them a bit value of 1 if the value of the pixel is greater than zero, and a bit value of 0 otherwise. This gives us a 8-bit representation (excluding the center which is always 0) of every 3x3 patch. Thus each 3x3 patch can be represented by a integer between 0-255. The scheme is shown in the Figure 2.

   To obtain an image representation apply this scheme to each 3x3 patch in the image and obtain a histogram that indicates how many patches get mapped to each integer. Implement the function `features = extractDigitFeatures(x, 'lbp')` that takes as input the images `x` and returns a local binary pattern (LBP) histogram for each image. This feature should be 256 dimensional.

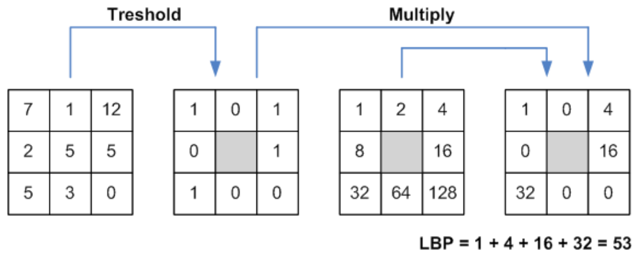

Figure 2: Illustration of the LBP feature computation for a 3x3 patch.

1.a Feature normalization

After you implement these features you may find that the scales of the features vary across images. For example, on the "scaled" version of the dataset the images exhibit varying contrast levels making pixel representations less robust. Feature normalization can alleviate some of these issues and improve classification performance with simple machine learning models.

Your goal is to implement the following normalization steps:

1. **Square-root scaling –** replace the value of each feature by its square-root, i.e., $x \leftarrow \sqrt{x}$.
2. **L2-normalization –** scale each feature vector to unit length in Euclidean sense, i.e., if $\mathbf{x}$ is a vector then the L2-normalized version would be $\mathbf{x} \leftarrow \mathbf{x}/\sqrt{\sum_i \mathbf{x}_i^2}$.

The effectiveness of one or both of these steps depends on the dataset and feature types.

2 Multiclass logistic regression

In multiclass logistic regression with K classes the predicted probability of class k for input $\mathbf{x}$ is given by:

$$\hat{y}^{(k)} = P(Y = k|X = x) = \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_{i=1}^{K} \exp(\mathbf{w}_i^T \mathbf{x})}, \tag{1}$$

where $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_k$ are the model parameters. Training minimizes the cross-entropy loss over the training set $\{(\mathbf{x}_i, y_i)\}$ plus a regularization term to encourage simpler models given by

$$\min_{\mathbf{w}_1,\ldots,\mathbf{w}_K} L = \min_{\mathbf{w}_1,\mathbf{w}_2,\ldots,\mathbf{w}_k} \sum_i \ell(y_i, \hat{y}_i) + \lambda \sum_k ||\mathbf{w}_k||_2^2, \tag{2}$$

where $\hat{y}_i = P(Y|X = x_i)$ and $\ell(y, \hat{y})$ is the cross-entropy loss

$$\ell(y, \hat{y}) = -\sum_{k=1}^{K} y^{(k)} \log \hat{y}^{(k)}. \tag{3}$$

While there is no closed form solution to minimize the objective we can still find the solution with gradient descent. Derive the expression for the i^{th} component in the vector gradient of L with respect to $\mathbf{w}_k$ and write down the update rule for $\mathbf{w}_k$ using η for the step size. Note that in class we discussed the online and batch version of the gradient. This dataset is small enough that batch gradients can be computed.

Implement the logistic-regression classifier inside the function `model = trainModel(x, y)`, which in turn calls the function `model = multiclassLRTrain(x, y, param)`. The `multiclassLRTrain` method runs batch gradients using a fixed learning rate for a given number of iterations to compute the model parameters. You might have to adjust these hyperparamters (eta, lambda, maxiter) to obtain good performance across feature types.

Similarly implement the function `ypred = multiclassLRPredict(model,x)` that computes predictions for new data points.

One important note is that you may find adding a bias term to the end of the features can be helpful. You can accomplish this by simply adding a constant feature to the end of the all the feature vectors. Don't forget to account for this at test time. For this task you will find that running gradient descent with a fixed learning rate for a large number of iterations finds good model parameters. This shouldn't be surprising since the objective is convex.

2.a Training, cross-validation, testing

The entry code for the homework is in the file `testMNIST`. It loops over different variants of the dataset, calls the function `extractDigitFeatures` for three different feature types $\in$ {'pixel', 'hog', 'lbp'}, trains a logistic-regression model on the training set and evaluates the model on the validation set. The function `[acc, conf] = evaluateLabels(y, ypred, false)` returns the accuracy and confusion matrix (`conf`) given the true labels `y`, and predictions `ypred`. When the last parameter is `true` it additionally displays the confusion matrix. You may find this helpful for intermediate debugging.

Currently the `extractDigitFeatures` simply returns zero features for all methods and all methods perform at 10% accuracy on all datasets which is the chance performance. You will implement your approach here. Each training step has several hyperparameters, such as normalization method, size of bin for oriented gradient features, as well as classifier-specific ones such as learning rates, regularization term. These should be tuned on the validation set (`testSet=2`). Once you have found the optimal values, you can change the `testSet=3` to evaluate the method on the test set.

The confusion matrix (c) is a 10x10 array where the $c(i,j)$ entry counts how many times a class i got classified as class j. A perfect classifier should have all non-zero values in the diagonal. You can use this to visualize which pair of classes get confused most by different features.

3 Convolutional neural nets

Investigate a convolutional neural network (CNN) for these tasks using a library of your choice. Your starting point will be architectures that consists of several convolutional and pooling layers, followed by a one or more linear layers similar to the LeNet architecture discussed in class. Your goal is to compare CNNs to handcrafted representations on the provided datasets. Most deep learning libraries provide example code for MNIST classification using CNNs. Perhaps the simplest way is to convert the datasets provided in the codebase to a format readable by the library to get started. Links to popular deep learning frameworks are linked below.

Note that the dataset in the homework is *different* from the standard MNIST dataset that comes with example code in various packages. Make sure you modify the code and load the provided dataset instead. If you are working in Python, you may find scipy.io.loadmat() useful for loading data from mat files. Since our dataset is relatively small and the training models with many parameters can lead to over-fitting. Therefore, you may want to experiment with shallower CNNs with a few convolutional, pooling, non-linearity blocks, followed by a linear layer or two for predicting class labels.

Explore a few different architecture choices and hyperparamters such as learning rates, weight decay (regularization term) on the validation set and report the result on the test set.

3.a Software packages

PyTorch and TensorFlow are two popular deep learning frameworks based on python, while MatConvNet is a popular one for on Matlab. Feel free to use any of these for this part of the homework, or any other package that you are familiar with. Here are links to some of these frameworks:

- **TensorFlow.** Tutorials: `https://www.tensorflow.org/tutorials`
- **PyTorch.** Tutorial: `http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html` MNIST example: `https://github.com/pytorch/examples/blob/master/mnist/main.py`.

- **MatConvNet.** Example: `https://github.com/vlfeat/matconvnet/tree/master/examples/mnist`. Tutorial: `http://www.vlfeat.org/matconvnet/training/`

4 Answer the following questions [70 points]

1. *[10 points]* Implement the multi-class logistic regression training and evaluation. Include the gradient update equation for the parameters in your writeup and attach the code for training and evaluation at the end.
2. *[5 points]* Describe your experiments using pixel features and logistic regression. In particular you should:
   - Report the accuracy on the validation and test set.
   - Describe what normalization steps were effective on what datasets.
   - Include the values of hyperparameters that worked best on the validation set.
3. *[15 points]* Answer all the above questions for HoG features. Don't forget to include the hyperparameters specific to HoG features such as numOri and binSize.
4. *[15 points]* Answer all the above questions for LBP features.
5. *[20 points]* Answer all the above questions for CNN. In particular note what architectures you tried.
6. *[5 points]* Which feature worked the best on each version of the dataset? Briefly explain your answer.

In your submission include your implementation of `extractDigitFeatures`. As a reference my implementation of various features with suitable normalization obtains the following accuracy on the validation set. With careful cross-validation you may be able to outperform these numbers.

```
+++ Accuracy Table [trainSet=1, testSet=2]
-------------------------------------
dataset           pixel hog   lbp
-------------------------------------
digits-normal.mat 87.00 82.00 74.00
digits-scaled.mat 77.40 79.00 74.00
digits-jitter.mat 20.40 34.00 71.60
```

With some modifications of the LeNet architecture the CNN can reach about 95%, 90%, 70% accuracy on normal, scaled and jittered set respectively. Training CNNs takes longer so plan accordingly. As a rough estimate, a network with two convolutional layers and one fully-connected layer took about 30 minutes to train on a laptop CPU. Most libraries can take advantage of GPU hardware if your machine has any. That could speed up your training times by a factor of 10 or more.

While we don't expect you to match these results exactly, if your numbers that are significantly off might suggest a bug in the code. If that happens, we expect you to investigate why that's the case.

5 Image denoising with deep image priors [30 points]

Given a noisy image x_0 we can find its denoised version x^* through the following minimization:

$$x^* = \min_x E(x, x_0) + P(x) \tag{4}$$

where $E(x, x_0)$ is a *data term* that measures how close is x to x_0 and $P(x)$ is the *prior term*. This formulation can be interpreted as a *maximum a posteriori* (MAP) inference in a Bayesian setting. However, defining a good prior $P(x)$ for natural images is challenging. In class we discussed a Gaussian mixture model over patches as an example, but the resulting minimization and learning is challenging. We also discussed procedural priors such as Gaussian smoothing, median filtering, and non-local means, which you implemented in Mini-project 2. In this part we will investigate convolutional networks as image priors.

Recent research shows that convolutional networks for image generation can be effectively used to induce a prior over natural images (aka "deep image prior"). The main idea is that instead of searching over images x in the optimization, we look for images that can be generated through a convolutional neural network of a fixed architecture on a fixed input. It turns out that random convolutional networks produce "smooth" images and encode natural image priors. We will exploit this strategy for denoising images, by changing the network parameters different to produce the target (noisy) images starting from a random initialization.

Let the network be $f_\theta(\eta)$ where θ are parameters of the network and η is an input. The input is sampled from a standard Gaussian distribution and kept fixed. Now, we can denoise x_0 as follows:

$$\theta^* = \min_\theta E(f_\theta(\eta), x_0); \qquad x^* = f_{\theta^*}(\eta). \tag{5}$$

The error for denoising is the squared distance between the pixels, i.e., $E(f_\theta(\eta), x_0) = \|f_\theta(\eta) - x_0\|_2^2$. Notice that there is no prior term and the objective can simply be minimized by gradient descent.

A question remains about which architecture we should use for f_θ. A good starting point is the following, but feel free to come up with other variations. We will employ a fully-convolutional encoder-decoder architecture with 6 convolutional layers. The first 3 are correspond to the encoder part of the network. It consists of 3 `ConvBlocks`, which are built using a 2D-convolution (with stride= 2) with several 3×3 filters, followed by batch normalization and ReLU activation. The last 3 layers are `DeconvBlocks`, built by using a bilinear upsampling followed by a 2D-convolution (with stride=1), batch normalization and ReLU activation. With this architecture the output size is the same as the input. In this case the input is simply a random image (each pixel is drawn from a zero mean Gaussian or Uniform distribution) of the same size as the target image. Batch normalization speeds up training significantly. Once again feel free to use any software package you like to implement these layers. For example, PyTorch, TensorFlow, and MatConvNet provide implementations of these layers.

Fitting a neural network to a single image poses a huge risk of over fitting. You will find running gradient descent for a long time will cause the training error to go to zero resulting in no denoising!To avoid this can you can stop the optimization early.

For this question we will use the images provided in the `data/denoising` folder which contains:

- `saturn.png, saturn-noisy.png, lena.png, lena-noisy.png`

The first image pair will be familiar to you from Mini-project 2, but this version is slightly resized. Gaussian noise was added to create the noisy versions for both cases. As a reference Gaussian filtering and Median filtering produce the following results on the two images (see the enclosed `evalDenoisingBaseline.m`.)

5.a Implementing the deep image prior

We have provided a python-based skeleton for the deep image prior in `evalDenoising` script, which uses the PyTorch library. The `evalDenoising` script loads the clean and the noisy image, visualizes them, and computes the error (squared-distance) between them. It then initializes a convolutional network (you will implement this) and runs several iterations of gradient descent. Your goal is to denoise the images using

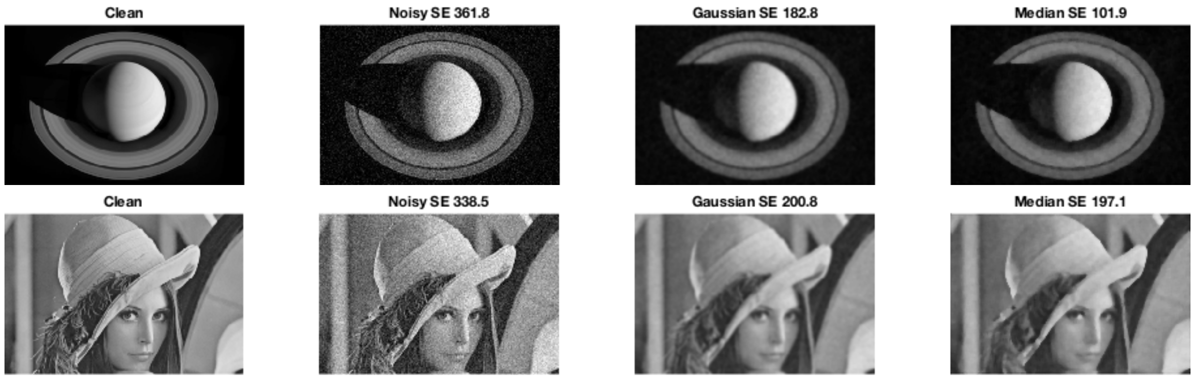

Figure 3: Image denoising with Gaussian and Median filter.

the deep image prior, which should result in a lower error value. With careful architecture design and early stopping you should be able to reduce the noise level significantly.

- *[20 points]* **Network architecture and random image.** The first step is to implement the architecture described before. You will use the following number of filters for each layer: 16, 32, 64. The kernel size of each filter is 3×3. The decoding part of the architecture is simply a mirrored version. Once the architecture is implemented, you will execute `evalDenoising`, which will instantiate your network, sample a random input η and show the image generated by the random architecture. In Python, you will implement your network in the file `dip.py`. Show five images generated by the network in your report by sampling different random inputs and network parameters.

- *[6 points]* **Using the deep image prior.** In this step you will use the network you created in the previous step to minimize Equation 5. You will use squared error as a data-term ($E(x, y) = \|x - y\|_2^2$). Run 500 gradient steps using Adam optimizer with learning rate$= 0.01$. Report the results for both the images as well as the error with respect to the reference. Also plot the training and testing error as a function of the number of iterations.

  Tip: Start with a shallower network, say with just one or two encoder-decoder layers, to debug your code.

- *[4 points]* Run gradient descent for 5000 iterations. Are the images generated after 5000 iterations better than the ones generated after 500 iterations? Why?

As a reference 500 iterations of gradient descent takes 3 minutes on my laptop.

5.b Extensions [Extra 10 points]

Here are some suggestions for extra credit. You can implement one of these:

- **Improving the results.** Take a look at the original paper `https://arxiv.org/abs/1711.10925` and explore different strategies to improve the results. For example deeper architectures, those with skip connections, as well as averaging the results across iterations. Also take a look at this paper that characterizes why the deep image prior works `https://arxiv.org/abs/1904.07457`.

- **Image demosaicing with deep image prior.** Come up with a data-term in the loss function that can be used instead of the squared error in order to perform image demosacing with the deep image prior. Run your algorithm on the images from Mini-project 2 and compare them to the baselines.

6 Report Writing and Conclusion [10 points]

As usual, please follow the guidelines for writing a good report to earn points for writing and presentation. We request that you do not share your solutions publicly or privately. Making these took quite a bit of effort and we would like to reuse them in the future.

Finally, congratulations on finishing all mini projects. Hope it was fun and have a cake!

7 My Answers of Problem 1 to 4:

7.a Gradient update equation for the parameters

$$W_i = W_i - \eta\left(\frac{exp\left(w_k^T x\right)}{\sum_{i=1}^{k} exp\left(w_i^T x\right)} + 2\lambda w_i - 1\right)$$

In the equation, $2\lambda w_i$ is the regularization term. -1 is the bias.

Code for training and evaluation is attached at the end.

7.b Pixel features and Logistic regression

Pixel features:

```
if featureType == 'pixel'
   if NormalizationType is 'square',
      return np.square(features)
   if NormalizationType is 'L2',
      return features/np.sqrt(np.sum(features**2, axis=0))
```

Logistic regression:

```
After Initialize weights randomly
Initialize eta, lambda, and maxiter
for i in range(maxiter):
   h = dot of w and input x
   max_h = the max value among the array h.
   Base = sum of exp^f
   Calculate the loss function which given us in the problem description. By using:
   loss = np.sum(y * (-f[y, np.arange(numData)] + np.log(Base))) + np.sum(w*w)

   Probability = np.exp(f)/np.sum(np.exp(f), axis=0)
   Minus the bias. Assume bias is 1in here.
   dProb_w = the dot of probability and input x.
   w = w- learning rate (eta) * (dProb_w + 2*lm* w),
   2*lm* w is regularization term

w is parameters of model.
return model
```

The accuracy on the validation set.

```
-------------------------------------
Feature Type: Pixel
-------------------------------------
dataset            L2   Square
-------------------------------------
digits-normal.mat 83.20 82.80
digits-scaled.mat 81.80 82.20
digits-jitter.mat 14.00 13.80
```

Comparing the results above,
For dataset " digits-normal.mat", L2 normalization steps were more effective.
For dataset " digits-scaled.mat", Square normalization steps were more effective.
For dataset " digits-jitter.mat", L2 normalization steps were more effective.

The values of hyperparameters that worked best on the validation set.
Learning rate: 0.02, lambda = 0.09

7.c HoG features

```
if featureType == 'hog'x
   get horizontal and vertical gradients gx and gy at each pixel using
   filter [-1, 0, 1],
   bin_size = 8 (try different values later)
   numOri = 11 (try different values later)
   Compute the magnitude m and orientation theta at each pixel.

   Initialize an image which all the pixel value is 0. The image was divided
   into small patches, there are (weight/bin_size*geight/bin_size*numOri)
   patches in the image.

   For each image k,
      Initialize an empty list feature [ ]
      For each bin in the image (i bins in horizontal and j bins in vertical):

          collapsed the theta array (from j*bin_size to (j+1)*bin_size,
          i*bin_size:(i+1)*bin_size, k) into one dimension, get V1.

          collapsed the magnitude m array (from j*bin_size to (j+1)*bin_size,
          i*bin_size:(i+1)*bin_size, k) into one dimension, get V2.

          Get the number of each bins and the bin_edges of each bin by using
          np.histogram.
          Feat = stack arrays in the horizontal level.
          Calculate the feature by using Square-root scaling and L2-normalization.

return feature
```

The accuracy on the validation set.

```
------------------------------------
Feature Type: HoG
------------------------------------
dataset            L2   Square
------------------------------------
digits-normal.mat 82.20 83.80
digits-scaled.mat 81.20 79.80
digits-jitter.mat 14.40 14.00
```

Comparing the results above,
For dataset " digits-normal.mat", Square normalization steps were more effective.
For dataset " digits-scaled.mat", L2 normalization steps were more effective.similar.
For dataset " digits-jitter.mat", L2 normalization steps were more effective.

The values of hyperparameters that worked best on the validation set.
Learning rate= 0.03, lambda = 0.02, numOri =11, binSize =8

7.d LBP features

```
Initialize an empty feature image, the size is (256, N), all the pixel value is 0.
For each image,
   Initialize LBP as an empty image which size is (weight, height)
   Initialize image which padded value 1 around each original image.
   For i in range(1, weight+1):
      for j in range(1, height+1):
         Build a patch1 in the image which size is 3*3. (i, j) is the center
         of the patch.

         Build another patch2 which size is also 3*3, all the pixel value is (i, j)
         results = patch2 { patch1
         All the pixel which equal or bigger than 0 = 1.
         All the pixel which less than 0 = 0.
         Each results multiply with filter1 [[1, 8, 32], [2, 0, 64],[4, 6, 128]],
         patch_LBP = Get sum of last step.
         LBP[j-1, i-1] = patch_LBP
   Take number of each bin as feature by using:
   np.histogram (LBP.flatten(), bins=256, density=True)
   Calculate the feature by using Square-root scaling and L2-normalization.
Return feature
```

The accuracy on the validation set.

```
------------------------------------
Feature Type: LBP
------------------------------------
dataset            L2   Square
------------------------------------
digits-normal.mat 82.60 79.00
digits-scaled.mat 81.00 82.60
digits-jitter.mat 14.20 14.20
```

Comparing the results above,
For dataset ” digits-normal.mat”, L2 normalization steps were more effective.
For dataset ” digits-scaled.mat”, Square normalization steps were more effective.similar.
For dataset ” digits-jitter.mat”, the performance of L2 and Square normalization steps were equal.

The values of hyperparameters that worked best on the validation set.
Learning rate: 0.02, lambda = 0.15

7.e CNN

I used PyTorch (torch.nn) to implement CNN.

```
Take F(torch.nn.functional) as the active function and loss function
Step1: Define the network model
  Define a class named Net, Net has several attributes/ layers:
     Convolution layer 1 (conv1)
     Batch normalization 1(batchnorm1)
     Convolution layer 2 (conv2)
     Batch normalization 2 (batchnorm2)
     Dropout layer (conv2_drop)
     Linear classification layer 1 (fc1)
     Batch normalization 3 (batchnorm3)
     Linear classification layer 2 (fc2)
     Batch normalization 4 (batchnorm4)

  Define a function for Forward Propagation, the input is x
     Add the ReLU active function layer to the network by using F.relu.
     The input x will be convoluted with convolution kernel 2*2
     (self.conv1(x)), 2)
     Then convoluted with Batch normalization 1
     ((self.batchnorm1(self.conv1(x)), 2)))
     Then convoluted with max_pooling layer
     (F.max_pool2d(self.batchnorm1(self.conv1(x)), 2))
     Then convoluted with Batch normalization 2 (self.batchnorm2)
     Then convoluted with Dropout layer (self.conv2_drop)
     Flatten the array of last step to an image,(x.view(-1, 512))
     Then convoluted with Linear classification layer 1 (self.fc1(x))
     Then convoluted with Batch normalization 3 (self.batchnorm3())
     Then Dropout again by using Dropout function
     (F.dropout(x, training=self.training))
   Return the log_softmax function of input x (F.log_softmax(x, dim=1))

Step2: Train the network
  Define the training details of each parameter
  (args, model, device, train_loader, optimizer, epoch)
  Set to training mode
  for batch_idx, (data, target) in enumerate(train_loader):
     data, target = data.to(device), target.to(device)
     convert the data into Variable
     Optimizer gradient is initialized to zero
     Enter data into the network and get output, that is, forward propagation
     Cross entropy loss function
```

```
     Backpropagation gradient
     After finishing a forward pass + back pass, update the parameters
     Print related information
```

```
Step3: Test
   Set to testing mode
   Initialize the loss value of testing as 0 (test_loss = 0)
   Initialize the number of correct prediction data to 0
    for data, target in test_loader:
        data = data.to(device)
        target = target.to(device)
        Before the calculation, the variable must be changed to Variable form.
        Only in this way can get the gradient.
        Using Variable(data), Variable(target)

        output = model(data)
        sum up batch loss
        get the index of the max log-probability
        Accumulate the number of correctly predicted data
        Because all the loss values are accumulated, the average loss must
        be divided by the total data length at the end
        Print the test results (Average loss and accuracy)
```

```
if __name__ == '__main__':
Define Training settings
  GPU enabled
    Load training data (The mean and standard deviation coefficients given
    by the data set, each data set is different)

    Instantiate a network object
    Initialize the optimizer

    Loop in epoch
     Training parameters (args, model, device, train_loader, optimizer, epoch)
     Test (args, model, device, test_loader)
     Save model
```

The accuracy on the test set.

```
---------------------------------
CNN
---------------------------------
dataset           L2  Square
---------------------------------
digits-normal.mat 87.20 91.30
digits-scaled.mat 82.70 89.60
digits-jitter.mat 78.00 75.20
```

Comparing the results above,
For dataset ” digits-normal.mat”, Square normalization steps were more effective.

For dataset ” digits-scaled.mat”, Square normalization steps were more effective.similar.
For dataset ” digits-jitter.mat”, L2 normalization steps were more effective.

The values of hyperparameters that worked best on the validation set.
Learning rate = 0.01, lambda = 0.03,

7.f Which feature worked the best on each version of the dataset?

```
Feature Normalization: L2
-------------------------------------
dataset           pixel hog   lbp
-------------------------------------
digits-normal.mat 83.20 82.20 82.60
digits-scaled.mat 81.80 81.20 81.00
digits-jitter.mat 14.00 14.40 14.20

Feature Normalization: Square
-------------------------------------
dataset           pixel hog   lbp
-------------------------------------
digits-normal.mat 83.80 82.80 79.00
digits-scaled.mat 82.20 79.80 82.60
digits-jitter.mat 13.80 14.00 14.20
```

Comparing the results above,
For dataset ” digits-normal.mat”, pixel features worked the best. Becasue for normal dataset, the best or simplest way to show their features is returning their pixel values as a feature.

For dataset ” digits-scaled.mat”, LBP features worked the best. We can see the contrast of images in the dataset ”digits-scaled.mat” is different. LBP is robust to gray changes caused by changes in lighting, etc.

For dataset ” digits-jitter.mat”, HOG features worked the best. Because HOG operates on local grid cells of the image, it can maintain good invariance to the geometric and optical deformation of the image. Additionally, when other conditions remain the same, as long as the figures are generally in the same shape, subtle differences in figures can be tolerated. These subtle actions can be ignored without affecting the detection effect.

8 My Answers of Problem 5: Implementing the deep image prior

8.a Network architecture and random image

8.b Using the deep image prior

8.c Run gradient descent for 5000 iterations

9 Code

1. **multiclassLRTrain**

```
import numpy as np

def multiclassLRTrain(x, y, param):

  classLabels = np.unique(y)
  numClass = classLabels.shape[0]
  numFeats = x.shape[0]
  numData = x.shape[1]
  # print('features are:{}'.format(x))

  # Initialize weights randomly (Implement gradient descent)
  eta = param['eta']
  lam = param['lambda']
  maxiter = param['maxiter']
  model = {}
  model['w'] = np.random.randn(numClass, numFeats)*0.01
  model['classLabels'] = classLabels
  w = np.copy(model['w'])
  for i in range(maxiter):
    h = w.dot(x)
    max_h = np.reshape(np.max(h, axis=0), (1, numData))
    h -= max_h
    b = np.sum(np.exp(h), axis=0).reshape(1, numData)

    #loss function
    loss = np.sum(y * (-h[y, np.arange(numData)] + np.log(b))) + np.sum(w*w)
    p = np.exp(h)/np.sum(np.exp(h), axis=0)
    p[y, np.arange(numData)] = p[y, np.arange(numData)] - 1 #bias
    dp_w = p.dot(x.T)
    dl_w = dp_w + 2*lam* w
    w -= eta* dl_w
  model['w'] = w

  return model
```

2. **extractDigitFeatures**

```
import numpy as np
from scipy.ndimage.filters import convolve1d, convolve
import matplotlib.pyplot as plt
from skimage import feature
np.set_printoptions(threshold=np.nan)


def extractDigitFeatures(x, featureType):

  w = x.shape[1]
  h = x.shape[0]
  N = x.shape[2]
  if featureType == 'pixel':

    features = np.reshape(x,(-1, N))
    #features = FeatrueType(features, 'square') #Comment other one when use one
        of them.
    features = FeatrueType(features, 'L2')

  elif featureType == 'hog':
```

```
    filter_dxy = [-1, 0, 1]
    bin_size = 7
    numOri = 11
    g_x = convolve1d(x, filter_dxy, axis = 1)
    g_y = convolve1d(x, filter_dxy, axis = 0)
    m = np.sqrt(g_x **2 + g_y**2)
    theta = np.arctan(g_y/(g_x+1e-8))

    features = np.zeros((w/bin_size*h/bin_size*numOri, N))
    for k in range(N):
      feat = []
      for i in range(w/bin_size):
        for j in range(h/bin_size):
          v1 = theta[j*bin_size:(j+1)*bin_size, i*bin_size:(i+1)*bin_size,
              k].flatten()
          v2 = m[j*bin_size:(j+1)*bin_size, i*bin_size:(i+1)*bin_size,
              k].flatten()
          hist, bin_edges = np.histogram(v1, bins=numOri,weights=v2,
              density=False)
          feat = np.hstack((feat, hist))
      feat /= (np.sqrt(np.sum(feat**2)) + 1e-8)
      features[:,k] = feat
    features = FeatrueType(features, 'square')
    features = FeatrueType(features, 'L2')

  elif featureType == 'lbp':
    features = np.zeros((256, N))
    patR = 1
    for k in range(N):

      LBP = np.zeros((h,w))
      xx = x[:,:,k]
      image = np.pad(xx, patR, mode='constant').astype('float')
      for i in range(patR,w+patR):
        for j in range(patR, h+patR):
          patch = image[j-patR:j+patR+1, i-patR:i+patR+1]
          subtraction = image[j,i] * np.ones((patR*2+1,patR*2+1))
          binary_patch = patch - subtraction
          binary_patch[binary_patch>=0] = 1
          binary_patch[binary_patch<0] = 0
          filter1 = np.array([[1, 8, 32], [2, 0, 64],[4, 6, 128]])
          patch_LBP = np.sum(binary_patch * filter1)
          LBP[j-1, i-1] = patch_LBP

      (feat1, _) = np.histogram(LBP.flatten(), bins=256, density=True)
      feat1 = feat1.astype("float")
      feat1 = np.sqrt(feat1)

      features[:,k] = feat1
  return features

def FeatrueType(features, normType):
  features = features.astype("float")
  if normType == 'square':
    return np.square(features)
  elif normType == 'L2':
    return features/np.sqrt(np.sum(features**2, axis=0))
```

3. **evaluation**

```
import numpy as np
import matplotlib.pyplot as plt

def evaluateLabels(y, ypred, visualize=True):

    classLabels = np.unique(y)
    conf = np.zeros((len(classLabels), len(classLabels)))
    for tc in range(len(classLabels)):
        for pc in range(len(classLabels)):
            conf[tc, pc] = np.sum(np.logical_and(y==classLabels[tc],
                ypred==classLabels[pc]).astype(float))

    acc = np.sum(np.diag(conf))/y.shape[0]

    if visualize:
        plt.figure()
        plt.imshow(conf, cmap='gray')
        plt.ylabel('true labels')
        plt.xlabel('predicted labels')
        plt.show()

    return (acc, conf)
```

4. **Convolutional neural nets**

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import scipy.io as spio
import os
import utils
import numpy as np
from torchvision import datasets, transforms

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5)
        self.batchnorm1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5)
        self.batchnorm2 = nn.BatchNorm2d(32)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(512, 128)
        self.batchnorm3 = nn.BatchNorm1d(128)
        self.fc2 = nn.Linear(128, 10)
        # self.batchnorm4 = nn.BatchNorm1d(10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.batchnorm1(self.conv1(x)), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.batchnorm2(self.conv2(x))), 2))
        # print(x.size)
        x = x.view(-1, 512)
        x = F.relu(self.batchnorm3(self.fc1(x)))
```

```
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    # lambda_lr = lambda epoch: 0.95** epoch
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item() # sum up
                batch loss
            pred = output.max(1, keepdim=True)[1] # get the index of the max
                log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

if __name__ == '__main__':
    # Training settings
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=10, metavar='N',
                        help='number of epochs to train (default: 10)')
    parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                        help='learning rate (default: 0.01)')
    parser.add_argument('--momentum', type=float, default=0.5, metavar='M',
                        help='SGD momentum (default: 0.5)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training status')
    args = parser.parse_args()
    use_cuda = not args.no_cuda and torch.cuda.is_available()
```

```
torch.manual_seed(args.seed)

device = torch.device("cuda" if use_cuda else "cpu")

kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}

# dataTypes = ['digits-normal.mat', 'digits-scaled.mat', 'digits-jitter.mat']
dataTypes = 'digits-jitter.mat'
path = os.path.join('../data', dataTypes)
# data = todict(spio.loadmat(path, struct_as_record=False,
    squeeze_me=True)['data'])
data = utils.loadmat(path)
print(data['x'].shape)
data_x = np.transpose(data['x'], (2, 0, 1))
print(data_x.shape)
data_x = data_x[:,np.newaxis,:,:]
train_x = data_x[data['set']==1]
train_y = data['y'][data['set']==1]
print(train_x.shape)
tensor_trx = torch.from_numpy(train_x)
tensor_trx = tensor_trx.float()
tensor_try = torch.from_numpy(train_y)
tensor_try = tensor_try.long()
tr_loader = torch.utils.data.TensorDataset(tensor_trx, tensor_try)
train_loader = torch.utils.data.DataLoader(tr_loader,
    batch_size=args.batch_size, shuffle=True)

test_x = data_x[data['set']==3]
test_y = data['y'][data['set']==3]
print(test_x.shape)
tensor_tex = torch.from_numpy(test_x)
tensor_tex = tensor_tex.float()
tensor_tey = torch.from_numpy(test_y)
tensor_tey = tensor_tey.long()
te_loader = torch.utils.data.TensorDataset(tensor_tex, tensor_tey)
test_loader = torch.utils.data.DataLoader(te_loader,
    batch_size=args.test_batch_size, shuffle=True)


model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=args.momentum)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[40, 70, 85,
    90], gamma=0.1)
for epoch in range(1, args.epochs + 1):
   scheduler.step()
   train(args, model, device, train_loader, optimizer, epoch)
   test(args, model, device, test_loader)

torch.save(model, 'model.pth')
```

5. **evalDenoising**

6. **dip**