Homework #4

CSE 546: Machine Learning
Prof. Kevin Jamieson
Due: 12/5 11:59 PM

1 Regression with Side Information

1. *[10 points]* In linear regression we have seen how penalizing the ℓ_2-norm of the weights (Ridge) and ℓ_1-norm of the weights (Lasso) affect the resulting solutions. Using different loss functions and regularizers to obtain different desired behaviors is very popular in machine learning. In this problem we will explore some of these ideas by using a general convex optimization solver CVXPY: `http://www.cvxpy.org/` to solve the optimization problems we define. Using these kinds of general solvers can be slower than a highly tuned custom solver you write yourself (e.g., accelerated gradient descent with a tuned stepsize) but they make it easy to swap out loss functions or regularizers. One of the benefits of convex optimization is that no matter which solver or method is used (coordinate descent, SGD, gradient descent, Newton's, etc.) they all converge to the same function value (unlike non-convex optimization in neural networks where the optimization method itself affects the resulting solution).

First let's generate some data. Let $n = 30$ and $f(x) = 10\sum_{k=1}^{4} \mathbf{1}\{x \geq \frac{k}{5}\}$, noting that $f(x)$ is non-decreasing in x (i.e., $f(x) \geq f(z)$ whenever $x \geq z$). For $i = 1, \ldots, n$ let each $x_i = \frac{i-1}{n-1}$ and $y_i = \mathbf{1}\{i \neq 15\}(f(x_i) + \epsilon_i)$ where $\epsilon_i \sim \mathcal{N}(0, 1)$. The case where $i = 15$ represents an outlier in the data.

In the last homework we solved a problem of the form $\arg\min_\alpha \sum_{i=1}^{n} \ell(y_i - \sum_{j=1}^{n} k(x_i, x_j)\alpha_j)$ where $\ell(z) = \ell_{ls}(z) := z^2$ was the least squares loss. Least squares is the MLE for Gaussian noise, but is very sensitive to outliers. A more robust loss is the Huber loss:

$$\ell_{huber}(z) = \begin{cases} z^2 & \text{if } |z| \leq 1 \\ 2|z| - 1 & \text{otherwise} \end{cases}$$

which acts like least squares close to 0 but like the absolute value far from 0. Moreover, define a matrix $D \in \{-1, 0, 1\}^{(n-1)\times n}$

$$D_{i,j} = \begin{cases} -1 & \text{if } i = j \\ 1 & \text{if } i = j - 1 \\ 0 & \text{otherwise} \end{cases}$$

so that for any vector $z \in \mathbb{R}^n$ we have $Dz = (z_2 - z_1, z_3 - z_2, \ldots z_n - z_{n-1})$ In what follows let $k(x, z) = \exp(-\gamma\|x - z\|^2)$ where $\gamma > 0$ is a hyperparameter.

a. As a baseline, let

$$\widehat{\alpha} = \arg\min_\alpha \sum_{i=1}^{n} \ell_{ls}(y_i - \sum_{j=1}^{n} K_{i,j}\alpha_j) + \lambda\alpha^T K\alpha \ , \qquad \widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i k(x_i, x)$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is the regularization constant. Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\widehat{f}(x)$ found through leave-one-out CV. (Hint: start with the problem on the homepage of `http://www.cvxpy.org/` and modify it as needed.)

b. Now let

$$\widehat{\alpha} = \arg\min_{\alpha} \sum_{i=1}^{n} \ell_{huber}(y_i - \sum_{j=1}^{n} K_{i,j}\alpha_j) + \lambda \alpha^T K \alpha \ , \qquad \widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i k(x_i, x)$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is the regularization constant. Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\widehat{f}(x)$ found through leave-one-out CV. (Hint: `huber` is a function in `cvxpy`.)

c. The total variation (TV) of a real-valued function g over $\{1, \ldots, n\}$ is defined as $\int_{i=1}^{n-1} |g_i - g_{i-1}| = ||Dg||_1$ and is a common regularizer for de-noising a function (its two-dimensional counterpart is very popular for image de-noising or filling-in missing/damaged parts of photos). Let

$$\widehat{\alpha} = \arg\min_{\alpha} ||K\alpha - y||^2 + \lambda_1 ||DK\alpha||_1 + \lambda_2 \alpha^T K \alpha \ , \qquad \widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i k(x_i, x)$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ_1, λ_2 are regularization constants. For intuition, the penalizer $||DK\alpha||_1$ prefers functions $\widehat{f}$ with sparse jumps in function value while $\alpha^T K \alpha$ perfers functions that are smoothly varying. On your own (not necessary to report plots), plot $\widehat{f}$ for a variety values of $\gamma, \lambda_1, \lambda_2$ to see how they affect the solution. Use leave-one-out cross validation to find a good setting of $\gamma, \lambda_1, \lambda_2$. Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\widehat{f}(x)$ found through leave-one-out CV.

d. We say a function g over $\{1, \ldots, n\}$ is non-decreasing if $g_i - g_{i-1} \geq 0$ for all i, or $Dg \geq 0$ where the inequality applies elementwise. Perhaps due to domain knowledge, we know that the original unnoisy function we are trying to estimate is non-decreasing. Let

$$\widehat{\alpha} = \arg\min_{\alpha} ||K\alpha - y||^2 + \lambda \alpha^T K \alpha \ , \qquad \widehat{f}(x) = \sum_{i=1}^{n} \widehat{\alpha}_i k(x_i, x)$$
$$\text{subject to} \quad DK\alpha \geq 0$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is a regularization constant. The above is known as a quadratic program because it can be written as $\arg\min_x \frac{1}{2} x^T Q x + p^T x + c$ subject to $Ax \leq b$. On your own (not necessary to report plots), plot $\widehat{f}$ for a variety values of γ, λ to see how they affect the solution. Use leave-one-out cross validation to find a good setting of γ, λ. Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\widehat{f}(x)$ found through leave-one-out CV. Note that the defined constraint only forces $\widehat{f}$ to be monotonic on the training data, not over all $x \in [0, 1]$, but it is instructive to think about how one might achieve this.

2 Deep learning architectures

2. *[15 points]* In this problem we will explore different deep learning architectures for a classification task. Go to `http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html` and complete the following tutorials

- *What is PyTorch?*
- *Autograd: automatic differentiation*
- *Neural Networks*
- *Training a classifier*

The final tutorial will leave you with a network for classifying the CIFAR-10 dataset, which is where this problem starts. Just following these tutorials could take a number of hours but they are excellent, so start early. After completing them, you should be familiar with tensors, two-dimensional convolutions (`nn.Conv2d`) and fully connected layers (`nn.Linear`), ReLu non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`); if there is any doubt of their inputs/outputs or whether the layers include an offset or not, consult the API `http://pytorch.org/docs/master/`.

A few preliminaries:

- Using a GPU may considerably speed up computations but it is not necessary for these small networks (one can get away with using their laptop).
- Conceptually, each network maps an image $x^{in} \in \mathbb{R}^{32\times32\times3}$ (3 channels for RGB) to an output layer $x^{out} \in \mathbb{R}^{10}$ where the image's class label is predicted as $\arg\max_{i=0,1,\ldots,9} x_i^{out}$. An error occurs if the predicted label differs from its true label.
- In this problem, the network is trained via cross-entropy loss, the same loss we used for multi-class logistic regression. Specifically, for an input image and label pair (x^{input}, c) where $c \in \{0, 1, \ldots, 9\}$, if the network's output layer is $x^{out} \in \mathbb{R}^{10}$, the loss is $-\log(\frac{\exp(x_c^{out})}{\sum_{c'=0}^{9} x_{c'}^{out}})$.
- For computational efficiency reasons, this particular network considers *mini-batches* of images per training step meaning the network actually maps $B = 4$ images per feed-forward so that $\widetilde{x}^{in} \in \mathbb{R}^{B\times32\times32\times3}$ and $\widetilde{x}^{out} \in \mathbb{R}^{B\times10}$. This is ignored in the network descriptions below but it is something to be aware of.
- The cross-entropy loss for a neural network is, in general, non-convex. This means that the optimization method may converge to different *local minima* based on different hyperparameters of the optimization procedure (e.g., stepsize). Usually one can find a good setting for these hyperparameters by just observing the relative progress of training over the first epoch or two (how fast is it decreasing) but you are warned that early progress is not necessarily indicative of the final convergence value (you may converge quickly to a poor local minima whereas a different step size could have poor early performance but converge to a better final value).
- The training method used in this example uses a form of stochastic gradient descent (SGD) that uses a technique called *momentum* which incorporates scaled versions of previous gradients into the current descent direction[1]. Practically speaking, momentum is another optimization hyperparameter in addition to the step size.
- We will not be using a validation set for this exercise. Hyperparameters like network architecture and step size should be chosen based on the performance on the test set. This is very bad practice for all the reasons we have discussed over the quarter, but we aim to make this exercise as simple as possible.
- You should modify the training code such that at the end of each epoch (one pass over the training data) compute and print the training and test classification accuracy (you may find the running calculation that the code initially uses useful to calculate the training accuracy).
- While one would usually train a network for hundreds of epochs for it to converge, this can be prohibitively time consuming so feel free to train your networks for just a dozen or so epochs.

You will construct a number of different network architectures and compare their performance. For all, it is highly recommended that you copy and modify the existing (working) network you are left with at the end of the tutorial *Training a classifier*. For all of the following perform a coarse hyperparameter selection (probably by hand) using the test set, report the hyperparameters you found, and plot the training and test classification accuracy as a function of iteration (one plot per network).

a. Fully connected output, 0 hidden layers (logistic regression): we begin with the simplest network possible that has no hidden layers and simply linearly maps the input layer to the output layer. That is, conceptually it could be written as

$$x^{out} = W\text{vec}(x^{in}) + b$$

where $x^{out} \in \mathbb{R}^{10}$, $x^{in} \in \mathbb{R}^{32\times32\times3}$, $W \in \mathbb{R}^{10\times3072}$, $b \in \mathbb{R}^{10}$ where $3072 = 32 \cdot 32 \cdot 3$. For a tensor $x \in \mathbb{R}^{a\times b\times c}$, we let $\text{vec}(x) \in \mathbb{R}^{abc}$ be the reshaped form of the tensor into a vector (in an arbitrary but consistent pattern).

b. Fully connected output, 1 fully connected hidden layer: we will have one hidden layer denoted as $x^{hidden} \in \mathbb{R}^M$ where M will be a hyperparameter you choose (M could be in the hundreds). The nonlinearity applied

[1] See `http://www.cs.toronto.edu/~hinton/absps/momentum.pdf` for the deep learning perspective on this method.

to the hidden layer will be the relu ($\text{relu}(x) = \max\{0, x\}$, elementwise). Conceptually, one could write this network as

$$x^{out} = W_2\text{relu}(W_1\text{vec}(x^{in}) + b_1) + b_2$$

where $W_1 \in \mathbb{R}^{M\times 3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$.

c. Fully connected output, 1 convolutional layer with max-pool: for a convolutional layer W_1 with individual filters of size $p \times p \times 3$ and output size M (reasonable choices are $M = 100$, $p = 5$) we have that $\text{Conv2d}(x^{input}, W_1) \in \mathbb{R}^{(33-p)\times(33-p)\times M}$. Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{input}, W)+b_1$ where b_1 is parameterized in $\mathbb{R}^M$. We will then apply a relu (relu doesn't change the tensor shape) and pool. If we use a max-pool of size N (a reasonable choice is $N = 14$ to pool to 2×2 with $p = 5$) we have that $\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{input}, W_1) + b_1)) \in \mathbb{R}^{\lfloor\frac{33-p}{N}\rfloor\times\lfloor\frac{33-p}{N}\rfloor\times M}$. We will then apply a fully connected layer to the output to get a final network given as

$$x^{output} = W_2\text{vec}(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{input}, W_1) + b_1))) + b_2$$

where $W_2 \in \mathbb{R}^{10\times M(\lfloor\frac{33-p}{N}\rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$. The parameters M, p, N (in addition to the step size and momentum) are all hyperparameters.

d. (Extra credit: *[5 points]*) Returning to the original network you were left with at the end of the tutorial *Training a classifier*, tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully connected layers, stepsize, etc.) and train for many epochs to achieve a *test accuracy* of at least 85%.

The number of hyperparameters to tune in the last exercise combined with the slow training times hopefully gave you a taste of how difficult it is to construct good performing networks. It should be emphasized the networks we constructed are **tiny**; typical networks have dozens of layers, each with hyperparameters to tune. Additional hyperparameters you are welcome to play with if you are so interested: replacing relu $\max\{0, x\}$ with a sigmoid $1/(1 + e^{-x})$, max-pool with average-pool, and experimenting with batch-normalization or dropout.