CS480/680: Introduction to Machine Learning

Homework 4

Due: 11:59 pm, November 29, 2018, submit on LEARN.

Include your name, student number and session!

Submit your writeup in pdf and all source code in a zip file (with proper documentation). Write a script for each programming exercise so that the TAs can easily run and verify your results. Make sure your code runs! [Text in square brackets are hints that can be ignored.]

Exercise 1: Convolutional Neural Networks (CNN) (50 pts)

Note: Please mention your Python version (and maybe the version of all other packages) in the code.

In this exercise you are going to run some experiments involving CNNs. You need to know Python and install the following libraries: Keras, Tensorflow, Numpy and all their dependencies. To perform any required image transformation, we suggest using pillow. You can find detailed instructions and tutorials for each of these libraries on the respective websites. [To install, try `pip install keras` and `pip install pillow`. For Tensorflow, follow the installation steps on its webpage.]

For all experiments, running on CPU is sufficient. You do not need to run the code on GPUs. Before start, we suggest you review what we learned about each layer in CNN, and read at least this tutorial.

1. Train a VGG11 net on the MNIST dataset. VGG11 was an earlier version of VGG16 and can be found as model A in Table 1 of this paper, whose Section 2.1 also gives you all the details about each layer. The goal is to get as close to 0 loss as possible. Note that our input dimension is different from the VGG paper. You need to resize each image in MNIST from its original size 28×28 to 32×32 [why?], and it might be necessary to change at least one other layer of VGG11. [This experiment will take up to 1 hour on a CPU, so please be cautious of your time. If this running time is not bearable, you may cut the training set by 1/10, so only have ∼600 images per class instead of the regular ∼6000.]

2. Once you've done the above, the next goal is to inspect the training process. Create the following plots:

   (a) (5 pts) test accuracy vs the number of epochs (say $3 \sim 5$)

   (b) (5 pts) training accuracy vs the number of epochs

   (c) (5 pts) test loss vs the number of epochs

   (d) (5 pts) training loss vs the number of epochs

   [If running more than 1 epoch is computationally infeasible, simply run 1 epoch and try to record the accuracy / loss every few minibatches.]

3. Then, it is time to inspect the generalization properties of your final model. Rotate and blur the test set images using any python library of your choice. We recommend pillow, to complete the following two plots:

   (e) (5 pts) test accuracy vs the degree of rotation. Try the following rotations: -45, -40, ... 40, 45, and plot the test accuracy at each rotation. What is the effect?

   (f) (5 pts) test accuracy vs Gaussian blurring. Try blurs with radius 0, 1, 2, 3, 4, 5, 6 and plot the test accuracy vs the blur for each radius. What is the effect?

4. Lastly, let us verify the effect of regularization.

   (g) (10 pts) Re-do parts 1.1 – 1.3 with ℓ_2 regularization on the weights. Create the same plots and indicate the regularization constant that you use.

   (h) (10 pts) Re-do parts 1.1 – 1.3 with data augmentation. Create the same plots and explain what kind of data augmentation that you use.

Exercise 2: Gaussian Processes (GP) (20 pts)

Consider the squared-exponential covariance function on $\mathbb{R} \times \mathbb{R}$:

$$k_\sigma(s,t) = \exp(-\tfrac{1}{2\sigma}(s-t)^2) \tag{1}$$

(a) (10 pts) For each $\sigma \in \{0.1, 1, 10\}$, draw $m = 10$ sample paths of the Gaussian process $\mathcal{GP}(0, k_\sigma)$, where each path consists of $n = 1500$ sample points equally spaced between -1 to 1. What can you observe about the sample paths when we change σ?

(b) (10 pts) Compute the mixed partial derivative $k'_\sigma(s,t) = \frac{\partial^2 k_\sigma}{\partial s \partial t}(s,t)$, and repeat part (a) with covariance function k'_σ (and same $\sigma \in \{0.1, 1, 10\}$). Use the same random seeds for both part (a) and (b). What can you observe about the sample paths in part (a) and part (b)?

Exercise 3: Regularization (30 pts)

Notation: For the vector $\mathbf{x}_i$, we use x_{ji} to denote its j-th element.

Overfitting to the training set is a big concern in machine learning. One simple remedy to avoid overfitting to any particular training data is through injecting noise: we randomly perturb each training data before feeding it into our machine learning algorithm. In this exercise you are going to prove that injecting noise to training data is essentially the same as adding some particular form of regularization. We use least-squares regression as an example, but the same idea extends to other models in machine learning almost effortlessly.

Recall that least-squares regression aims at solving:

$$\min_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^n (y_i - \mathbf{w}^\top \mathbf{x}_i)^2, \tag{2}$$

where $\mathbf{x}_i \in \mathbb{R}^d$ and $\mathbf{y}_i \in \mathbb{R}$ are the training data. (For simplicity, we omit the bias term here.) Now, instead of using the given feature vector $\mathbf{x}_i$, we perturb it first by some independent noise $\boldsymbol{\epsilon}_i$ to get $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i)$, with different choices of the perturbation function f. Then, we solve the following **expected** least-squares regression problem:

$$\min_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^n \mathbf{E}[(y_i - \mathbf{w}^\top \tilde{\mathbf{x}}_i)^2], \tag{3}$$

where the expectation removes the randomness in $\tilde{\mathbf{x}}_i$ (due to the noise $\boldsymbol{\epsilon}_i$), and we treat $\mathbf{x}_i, y_i$ as fixed here. [To understand the expectation, think of n as so large that we have each data appearing repeatedly many times in our training set.]

1. (15 pts) Let $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i) = \mathbf{x}_i + \boldsymbol{\epsilon}_i$ where $\boldsymbol{\epsilon}_i \sim \mathcal{N}(\mathbf{0}, \lambda \mathbb{I})$ follows the standard Gaussian distribution. Simplify (3) as the usual least-squares regression (2), plus a familiar regularization function on $\mathbf{w}$.

2. (15 pts) Let $\tilde{\mathbf{x}}_i = f(\mathbf{x}_i, \boldsymbol{\epsilon}_i) = \mathbf{x}_i \odot \boldsymbol{\epsilon}_i$, where $\odot$ denotes the element-wise product and $p\epsilon_{ji} \sim$ Bernoulli(p) independently for each j. That is, with probability $1-p$ we reset x_{ji} to 0 and with probability p we scale x_{ji} as x_{ji}/p. Note that for different training data $\mathbf{x}_i$, $\boldsymbol{\epsilon}_i$'s are independent. Simplify (3) as the usual least-squares regression (2), plus a different regularization function on $\mathbf{w}$. [This way of injecting noise, when applied to the weight vector $\mathbf{w}$ in a neural network, is known as Dropout (DropConnect).]