Homework #3

CSE 446: Machine Learning
Prof. Byron Boots
Due: **Monday** 3/01/2021 11:59 PM
92 points

Please review all homework guidance posted on the website before submitting to Gradescope. Reminders:

- Please provide succinct answers along with succinct reasoning for all your answers. Points may be deducted if long answers demonstrate a lack of clarity. Similarly, when discussing the experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. In other words, all your explanations, tables, and figures for any particular part of a question must be grouped together.
- When submitting to gradescope, please link each question from the homework in gradescope to the location of its answer in your homework PDF. Failure to do so may result in point deductions. For instructions, see `https://www.gradescope.com/get_started#student-submission`.
- If you collaborate on this homework with others, you must indicate who you worked with on your homework. Failure to do so may result in accusations of plagiarism.

Conceptual Questions

1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. *[2 points]* True or False: The maximum margin decision boundaries that support vector machines construct have the lowest generalization error among all linear classifiers.

b. *[2 points]* Say you trained an SVM classifier with an RBF kernel ($K(u, v) = exp(-\frac{\|u-v\|_2^2}{2\sigma^2})$). It seems to underfit the training set: should you increase or decrease σ?

c. *[2 points]* True or False: Training deep neural networks requires minimizing a non-convex loss function, and therefore gradient descent might not reach the globally-optimal solution.

d. *[2 points]* True or False: It is a good practice to initialize all weights to zero when training a deep neural network.

e. *[2 points]* True or False: We use non-linear activation functions in a neural network's hidden layers so that the network learns non-linear decision boundaries.

f. *[2 points]* True or False: Given a neural network, the time complexity of the backward pass step in the backpropagation algorithm can be prohibitively larger compared to the relatively low time complexity of the forward pass step.

Kernels

2. *[5 points]* Suppose that our inputs x are one-dimensional and that our feature map is infinite-dimensional: $\phi(x)$ is a vector whose ith component is

$$\frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x^i$$

for all nonnegative integers i. (Thus, ϕ is an infinite-dimensional vector.) Show that $K(x, x') = e^{-\frac{(x-x')^2}{2}}$ is a kernel function for this feature map, i.e.,

$$\phi(x) \cdot \phi(x') = e^{-\frac{(x-x')^2}{2}}.$$

Hint: Use the Taylor expansion of e^z. (This is the one dimensional version of the Gaussian (RBF) kernel).

3. This problem will get you familiar with kernel ridge regression using the polynomial and RBF kernels. First, let's generate some data. Let $n = 30$ and $f_*(x) = 4\sin(\pi x)\cos(6\pi x^2)$. For $i = 1, \ldots, n$ let each x_i be drawn uniformly at random on $[0, 1]$ and $y_i = f_*(x_i) + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, 1)$.
For any function f, the true error and the train error are respectively defined as

$$\mathcal{E}_{true}(f) = E_{XY}[(f(X) - Y)^2], \qquad \widehat{\mathcal{E}}_{train}(f) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2.$$

Using kernel ridge regression, construct a predictor

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

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is the regularization constant. Include any code you use for your experiments in your submission.

a. *[10 points]* Using leave-one-out cross validation, find a good λ and hyperparameter settings for the following kernels:

- $k_{poly}(x, z) = (1 + x^T z)^d$ where $d \in \mathbb{N}$ is a hyperparameter,
- $k_{rbf}(x, z) = \exp(-\gamma\|x - z\|^2)$ where $\gamma > 0$ is a hyperparameter[1].

Report the values of d, γ, and the λ values for both kernels.

b. *[10 points]* Let $\widehat{f}_{poly}(x)$ and $\widehat{f}_{rbf}(x)$ be the functions learned using the hyperparameters you found in part a. For a single plot per function $\widehat{f} \in \{\widehat{f}_{poly}(x), \widehat{f}_{rbf}(x)\}$, plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, and $\widehat{f}(x)$ (i.e., define a fine grid on $[0, 1]$ to plot the functions).

Neural Networks for MNIST

4. In Homework 1, we used ridge regression for training a classifier for the MNIST data set. In Homework 2, we used logistic regression to distinguish between the digits 2 and 7. In this problem, we will use PyTorch to build a simple neural network classifier for MNIST to further improve our accuracy.

We will implement two different architectures: a shallow but wide network, and a narrow but deeper network. For both architectures, we use d to refer to the number of input features (in MNIST, $d = 28^2 = 784$), h_i to refer to the dimension of the ith hidden layer and k for the number of target classes (in MNIST, $k = 10$). For the non-linear activation, use ReLU. Recall from lecture that

$$\text{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Weight Initialization

Consider a weight matrix $W \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$. Note that here m refers to the input dimension and n to the output dimension of the transformation $Wx + b$. Define $\alpha = \frac{1}{\sqrt{m}}$. Initialize all your weight matrices and biases according to $\text{Unif}(-\alpha, \alpha)$.

[1] Given a dataset $x_1, \ldots, x_n \in \mathbb{R}^d$, a heuristic for choosing a range of γ in the right ballpark is the inverse of the median of all $\binom{n}{2}$ squared distances $||x_i - x_j||_2^2$.

Training

For this assignment, use the Adam optimizer from `torch.optim`. Adam is a more advanced form of gradient descent that combines momentum and learning rate scaling. It often converges faster than regular gradient descent. You can use either Gradient Descent or any form of Stochastic Gradient Descent. Note that you are still using Adam, but might pass either the full data, a single datapoint or a batch of data to it. Use cross entropy for the loss function and ReLU for the non-linearity.

Implementing the Neural Networks

a. *[10 points]* Let $W_0 \in \mathbb{R}^{h \times d}$, $b_0 \in \mathbb{R}^h$, $W_1 \in \mathbb{R}^{k \times h}$, $b_1 \in \mathbb{R}^k$ and $\sigma(z) : \mathbb{R} \to \mathbb{R}$ some non-linear activation function. Given some $x \in \mathbb{R}^d$, the forward pass of the wide, shallow network can be formulated as:

$$\mathcal{F}_1(x) = W_1\sigma(W_0x + b_0) + b_1$$

Use $h = 64$ for the number of hidden units and choose an appropriate learning rate. Train the network until it reaches 99% accuracy on the training data and provide a training plot (loss vs. epoch). Finally evaluate the model on the test data and report both the accuracy and the loss.

b. *[10 points]* Let $W_0 \in \mathbb{R}^{h_0 \times d}$, $b_0 \in \mathbb{R}^{h_0}$, $W_1 \in \mathbb{R}^{h_1 \times h_0}$, $b_1 \in \mathbb{R}^{h_1}$, $W_2 \in \mathbb{R}^{k \times h_1}$, $b_2 \in \mathbb{R}^k$ and $\sigma(z) : \mathbb{R} \to \mathbb{R}$ some non-linear activation function. Given some $x \in \mathbb{R}^d$, the forward pass of the network can be formulated as:

$$\mathcal{F}_2(x) = W_2\sigma(W_1\sigma(W_0x + b_0) + b_1) + b_2$$

Use $h_0 = h_1 = 32$ and perform the same steps as in part a.

c. *[5 points]* Compute the total number of parameters of each network and report them. Then compare the number of parameters as well as the test accuracies the networks achieved. Is one of the approaches (wide, shallow vs. narrow, deeper) better than the other? Give an intuition for why or why not.

Using PyTorch: For your solution, you may not use any functionality from the `torch.nn` module except for `torch.nn.functional.relu` and `torch.nn.functional.cross_entropy`. You must implement the networks $\mathcal{F}$ from scratch. For starter code and a tutorial on PyTorch refer to the sections 6 and 7 material.

Using Pretrained Networks and Transfer Learning

5. So far we have trained very small neural networks from scratch. As mentioned in the previous problem, modern neural networks are much larger and more difficult to train and validate. In practice, it is rare to train such large networks from scratch. This is because it is difficult to obtain both the massive datasets and the computational resources required to train such networks.

Instead of training a network from scratch, in this problem, we will use a network that has already been trained on a very large dataset (ImageNet) and adjust it for the task at hand. This process of adapting weights in a model trained for another task is known as *transfer learning*.

- Begin with the pretrained AlexNet model from `torchvision.models` for both tasks below. AlexNet achieved an early breakthrough performance on `ImageNet` and was instrumental in sparking the deep learning revolution in 2012.
- Do not modify any module within AlexNet that is not the final classifier layer.
- The output of AlexNet comes from the 6th layer of the classifier. Specifically, `model.classifer[6] = nn.Linear(4096, 1000)`. To use AlexNet with CIFAR-10, we will reinitialize (replace) this layer with `nn.Linear(4096, 10)`. This re-initializes the weights, and changes the output shape to reflect the desired number of target classes in CIFAR-10.

We will explore two different ways to formulate transfer learning.

a. *[15 points]* **Use AlexNet as a fixed feature extractor:** Add a new linear layer to replace the existing classification layer, and only adjust the weights of this new layer (keeping the weights of all other layers fixed). Provide plots for training loss and validation loss over the number of epochs. Report the highest validation accuracy achieved. Finally, evaluate the model on the test data and report both the accuracy and the loss.

   When using AlexNet as a fixed feature extractor, make sure to freeze all of the parameters in the network *before* adding your new linear layer:

```
model = torchvision.models.alexnet(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
model.classifier[6] = nn.Linear(4096, 10)
```

b. *[15 points]* **Fine-Tuning:** The second approach to transfer learning is to fine-tune the weights of the pre-trained network, in addition to training the new classification layer. In this approach, all network weights are updated at every training iteration; we simply use the existing AlexNet weights as the "initialization" for our network (except for the weights in the new classification layer, which will be initialized using whichever method is specified in the constructor) prior to training on CIFAR-10. Following the same procedure, report all the same metrics and plots as in the previous question.