Generalizability

Question 1: Designing a Regularizer

[3 marks]

Consider training a neural network, represented by the function f,

$$y = f(x; \theta)$$

where x is the input, y is the output, and θ represents the network's weights and biases (hereafter referred to collectively as "weights"). Let your cost function be $E(f(X; \theta), T)$, where X and T represent all the inputs and targets in your dataset. Thus, learning becomes

$$\min_{\theta} E\big(f(X; \theta), T\big) .$$

Gradient descent gives you the weight-update rule

$$\theta \leftarrow \theta - \kappa \nabla_{\theta} E\big(f(X; \theta), T\big) . \tag{1}$$

Suppose, for whatever reason, you want to discourage connection weights that are close to ± 1 as you train the network. That is, once the network is trained, you do not want to see many weights that are close to 1, and you do not want to see many weights that are close to -1.

(a) Design a new cost function, $\tilde{E}$, that discourages weights near ± 1 (as described above). You can include components from the equations above in your cost function. **Display your new cost function prominently**.

(b) For your new cost function, derive the gradient-descent update rule for your weights. Your update rule can include components from the equations above. **Display your update rule prominently**.

Question 2: Combatting Overfitting

[6 marks]

Download the jupyter notebook `a04q2.ipynb`. Note that this notebook requires PyTorch, which you can install on your own machine (if you have one). Alternatively, you can access PyTorch on the jupyter notebook server

https://uwaterloo.syzygy.ca

There were issues importing PyTorch on syzygy, and I have asked to have them addressed. Also, the speed of this server will depend on the load from other students. It might be wise to do preliminary testing with fewer epochs. Once you are confident with your code, you can run the full test. Another alternative is Google's colab,

https://colab.research.google.com

A. The notebook has an implementation of a `Dropout` class. It represents a dropout layer, and is based on the `torch.nn.Module` class. Notice that it has an attribute called `dropprob`, which determines the probability of dropping each node in the layer.

However, its current implementation is simply an identity function; `dropprob` has no effect, so no nodes are dropped.

Complete the implementation so that it performs dropout as described in its documentation. Keep in mind that its implementation should take advantage of the automatic differentiation functionality available in PyTorch.

You should use the `Dropout` class' function `set_dropprob` to set the dropout layer's drop probability.

B. The `RobustNetwork` class is a subclass of the `nn.Module` class and creates a neural network. It is supposed to implement dropout and L_2 weight regularization. The L_2 weight regularization comes from adding a penalty term to the cost function that is the square of the Frobenius norm, yielding the revised cost function

$$\tilde{E}(Y,T) = E(Y,T) + \frac{\lambda}{2}\|\theta\|_F^2 \,. \tag{2}$$

The `learn` function of the `RobustNetwork` class has two extra inputs, `weight_decay` and `dropprob`. The `weight_decay` argument is the same as λ in (2). Your job is to edit the `learn` function so that the neural network uses L_2 regularization on the connection weights and biases, and dropout while training. By default, both parameters are set to zero.

Note: You should use PyTorch's auto-differentiation capabilities. Also, do **not** use PyTorch's optimizers; instead, update the weights manually. You should apply the weight penalty to both the connection weights and the biases.

Hint: The dropout probability should be set to 0 when not learning.

Try your code on the "Test and Train" code in the notebook. Do you observe lower test error when using regularization, or dropout, when compared to using neither. Finally, some marks can be taken off if your code is not readable. Do the grader (and yourself) a favour and put in comments to indicate what your code does.

What to submit

Question 1: You can:

- typeset your solutions using a word-processing application, such as Microsoft Word, LaTeX, Google docs, etc., or
- write your solutions using a tablet computer, or
- write your solutions on paper and take photographs.

In any case, it is **your responsibility** to make sure that your solutions are sufficiently legible.

Submit your solution to Crowdmark only. Crowdwark will accept PDFs or image files (JPEG or PNG). You may submit multiple files for each question, if needed.

Question 2: You must submit your jupyter notebook in two places: **Crowdmark**, and **D2L**.

Crowdmark: Export your jupyter notebook as a PDF, and submit the PDF to Crowdmark.

D2L: Submit your `ipynb` file to Desire2Learn in the designated dropbox. You do not need to zip the file.

Make sure you submit your solutions, and not just a copy of the supplied skeleton code. We suggest you rename the `ipynb` file by replacing "`YOU`" with your WatIAM ID (eg. `a04q2_jorchard.ipynb`).

Important note about submitting code
We re-run your notebook in its entirety. So make sure that all your code cells run, from top to bottom. Any code cell that crashes will stop us from re-running your notebook (and you don't want that). Also, we extract the class and function definitions and autotest them. To facilitate this extraction process, please ensure that:

- all `import` commands are in a single code cell (at the top of the notebook), and
- code cells that contain function definitions and/or class definitions should not include any lines of code that are not part of the definition.