Homework 5, Machine Learning, Fall 2023

***IMPORTANT* Homework Submission Instructions**

1. All homeworks must be submitted in one PDF file to Gradescope.
2. Please make sure to select the corresponding HW pages on Gradescope for each question
3. For all coding components, complete the solutions with a Jupyter notebook/Google Colab, and export the notebook (including both code and outputs) into a PDF file. Concatenate the theory solutions PDF file with the coding solutions PDF file into one PDF file which you will submit.
4. You must show work or give an explanation for every question. "Yes" is not a sufficient answer. Always show the steps of your calculations. Only partial credit will be given if you do not explain your answer.
5. Failure to adhere to the above submission format may result in penalties.

All homework assignments must be your independent work product, no collaboration is allowed. You may check your assignment with others or the internet after you have done it, but you must actually do it by yourself. **Please copy the following statements at the top of your assignment file**:

Agreement: This assignment represents my own work. I did not work on this assignment with others. All coding was done by myself.

1 Reading Question: ProtoTree (Jon) 10pts

In the last assignment, we became familiar with the basics of decision trees. Decision trees are both interpretable and powerful, but a traditional decision tree may not be sufficient for every domain. In this problem, you are to read the paper at this link and answer the following questions about prototype driven trees for computer vision.

(a) The ProtoTree framework uses the idea of a soft decision tree to enable learning prototypes. What is the difference between a "soft" decision tree and a "hard" decision tree? Name at least one advantage of each.

(b) We will now explore how each component of the ProtoTree is learned, with minor simplifications.[1] In ProtoTree, each prototype is represented as a d dimensional vector $\mathbf{p}_j \in \mathbb{R}^d$. For the latent representation $\mathbf{z}_i \in \mathbb{R}^d$ of an image at a given location, the similarity of $\mathbf{p}_j$ to $\mathbf{z}_i$ is computed as

$$\text{sim}(\mathbf{p}_j, \mathbf{z}_i) = \exp(-\|\mathbf{z}_i - \mathbf{p}_j\|),$$

where $\text{sim}(\mathbf{p}_j, \mathbf{z}_i) = 1$ indicates that $\mathbf{p}_j = \mathbf{z}_i$ and $\text{sim}(\mathbf{p}_j, \mathbf{z}_i) \approx 0$ indicates that the two are not at all similar. The overall model output is then computed as:

$$\text{traverse}(\text{node}, \mathbf{z}_i) = \begin{cases} \text{node.c} & \text{if node is a leaf} \\ \text{sim}(\text{node.p}, \mathbf{z}_i) \cdot \text{traverse}(\text{node.r}, \mathbf{z}_i) & \\ \quad +(1 - \text{sim}(\text{node.p}, \mathbf{z}_i)) \cdot \text{traverse}(\text{node.l}, \mathbf{z}_i) & \text{otherwise,} \end{cases}$$

where, for a given node, node.p denotes the prototype for a non-leaf node, node.l denotes the left child (which may or may not be a leaf) of a non-leaf node, node.r denotes the right child (which may or may not be a leaf) of a non-leaf node, and node.c denotes the classification vector $\mathbf{c} \in \mathbb{R}^{n_{\text{classes}}}$ for a leaf node.

Write a (non-recursive) function $f : \mathbb{R}^d \to \mathbb{R}^{n_{\text{classes}}}$ that describes the forward reasoning of the simple ProtoTree shown in Figure 1. This function takes a vector $\mathbf{z}_i$, and produces a vector of class logits. The function "traverse" should not appear in your answer. Hint: Roll out the recursive definition for the case in Figure 1.

(c) Consider the negative log-likelihood loss, an analog to cross-entropy loss for an arbitrary number n_{classes} of classes:

$$\ell(\mathbf{y_i}, \hat{\mathbf{y}}_\mathbf{i}) = \sum_{j=0}^{n_{\text{classes}}} -y_{i,j} \log(\hat{y}_{i,j})$$

For a single sample $(\mathbf{y}_i, \mathbf{z}_i)$ of class a, find the partial derivative $\frac{\partial \ell}{\partial p_1^{(k)}}$ of the model's loss with respect to the k-th entry of prototype 1. Hint: Use backpropagation. The answer may be messier than you expect; naming and consolidating terms that appear often will help.

(d) Since (spoiler alert) the loss function applied to the overall model is differentiable with respect to each prototype vector, we know it is possible to learn these prototypes via gradient descent. However, these prototypes are really vectors in a high-dimensional space. How are prototypes modified so they can be visualized as training image patches?

[1] In ProtoTree, the probability of traversing the right branch at each node is computed using the piece of the latent representation of the input that maximizes the similarity for the prototype in each node. This piece of the latent representation corresponds to a spatial location in the input image. In this problem, we will assume that $\mathbf{z}_i$ is the minimizer over all prototypes for simplicity, allowing us to ignore spatial indices and more easily compute derivatives.

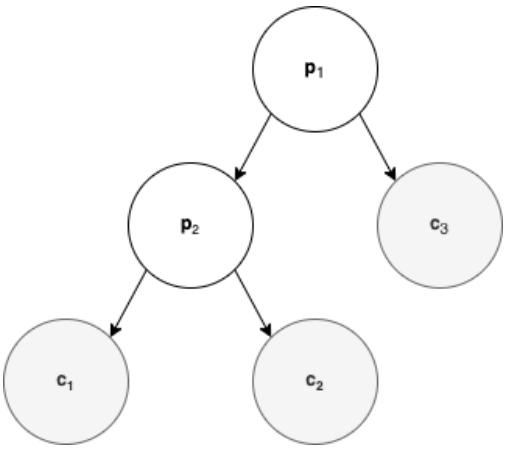


Figure 1: A simple ProtoTree.

2 CNN training, evaluation and Regularizations - Coding (Stark) 15 pts

In this problem, you will implement the training and evaluation code for a simple CNN network for the CIFAR-10 dataset. We supply a Jupyter Notebook template Jupyter Notebook template, and you will try to fill in the missing code. Please do not modify the existing code. This problem ideally requires a GPU, if you do not have access to GPU for training, you could use the free GPU provided on Google Colaboratory.

(a) In the provided problem template, fill in the missing code according to the instructional comments. Here you will implement part of the training and evaluation code.

(b) Load the model trained in (a), and evaluate the model performance on the test set using the accuracy metric. Visualize the training and validation losses recorded in (a). Based on the loss-epoch curves and the test set performance, is this model overfitting?

(c) Here you will write code to train a new CNN network with an additional L1 regularization term in the loss function. (Hint: you can reuse some code from (a), but please remember to initialize a new CNN network before training). Set your REG hyper-parameter value to 1e-4 in the loss function defined below, where W is the model weights, and L_{CE} is the Cross-Entropy loss:

$$L = L_{\text{CE}} + \text{REG} \cdot \sum |W|$$

After you trained this model, visualize the training and validation losses. Apply your trained L1 regularized CNN model to the test set and evaluate test set performance. Is the model doing better than before? Next, visualize the model weights (parameters) using histograms.

(d) Now you will implement a new CNN network with an additional L2 regularization term in the loss function. Set your REG hyper-parameter value to 1e-3 in the loss function defined below:

$$L = L_{\text{CE}} + \text{REG} \cdot \sum W^2$$

Again, visualize the trained model weights using histograms.

(e) Observe the visualizations of trained model weights, and comment on the differences between L1 and L2 regularization and their effects on model weights and the model performance.

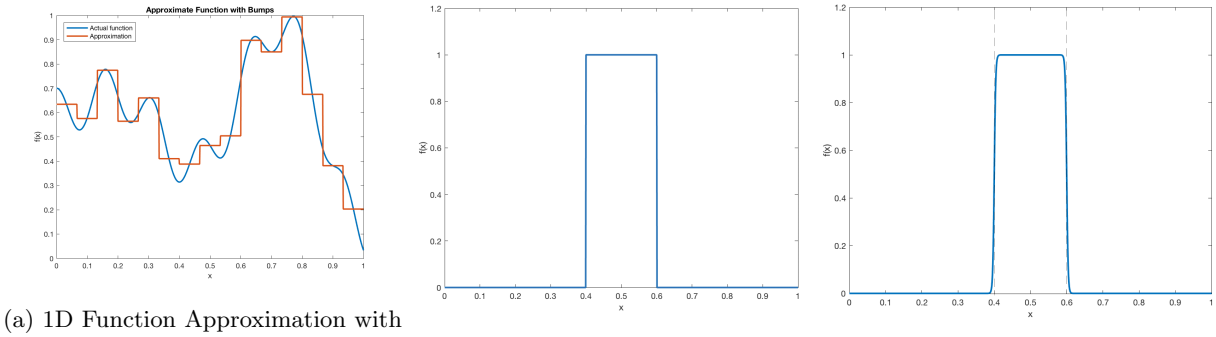
3 Neural Networks and Universal Approximation Theorem (Yiyang) 15pts

3.1

The Universal Approximation Theorem states that a neural network (NN) with one hidden layer can be used to approximate any continuous function within a certain precision. For any function with one input x and one output $f(x)$, one way to approximate it is to construct several "bumps" as shown in Fig. 2a.

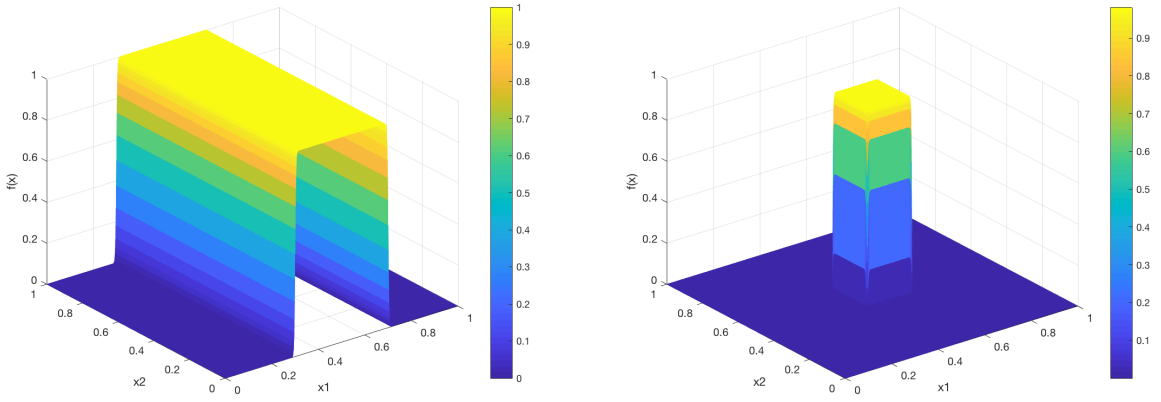
a. Assuming the sigmoid activation function is used, design a NN with one input neuron, one hidden layer, and one output neuron to approximate the bump function shown in Fig. 2b. An example of a reasonable approximation using a single layer NN is shown in Fig. 2c, which goes up at $x = 0.4$ and down at $x = 0.6$. Try to replicate it as closely as possible. (Hint: consider the bump as a combination of a step-up function and a step-down function). Implement your NN and plot your approximation. Describe your NN architecture including all weight and bias values. What is the minimum number of hidden neurons you need to make such an approximation? Note: Please don't use gradient descent algorithms or other parameter estimation algorithms to find the weights and biases of each neuron. Please build your own neural network by hands.

b. Specify what parameters in your NN determine (1) the steepness of the step-up and step-down part of the bump, (2) the step-up and step-down locations (x-coordinates), and (3) the height of the bump.



(a) 1D Function Approximation with Bumps (b) 1D Bump (c) Approximation example

Figure 2: Illustration of Function Approximation with Bumps



(a) An approximation of a bump function (b) An approximation of a tower function

Figure 3: Function Approximation with Two Inputs and One Output

3.2

Now we consider any function with two inputs and one output. We want to show that we can approximate it with a two-layer NN.

a. Design a single layer NN with two inputs (x_1, x_2), and one output $f(\mathbf{x})$ to approximate a bump function in 2D. The bump goes up at $x_1 = 0.3$ and goes down at $x_1 = 0.7$. The height of the bump is 1. An example of a reasonable approximation using a single layer NN is shown in Fig. 3a. Try to replicate it as closely as possible. Implement your NN and plot your approximation. Describe or draw your NN architecture with all weights and bias values. Do not include any edge with zero weight in your graph. What is the minimum number of hidden neurons you need to make such an approximation?

b. Design a two-layer NN with two inputs (x_1, x_2), and one output $f(\mathbf{x})$ to approximate a 2D tower function. The base of the tower is a square centered at $(0.5, 0.5)$ with all side lengths 0.1. The height of the tower is 1. An example of a reasonable approximation using a two-layer NN is shown in Fig. 3b. Try to replicate it as closely as possible. Implement your NN and plot your approximation. (Hint: consider adding an x_1 direction bump and an x_2 direction bump together; you only need one hidden neuron in the second hidden layer–carefully tune the weights and bias of the second layer). Draw your NN architecture with all weights and bias values. Do not include any edge with zero weight. What is the minimum number of hidden neurons you need to make such an approximation?

c. Suppose you have a 2D function $f(x_1, x_2)$ defined on a unit square $(x_1, x_2 \in [0, 1])$. The maximum absolute value of the gradient for both directions is t. You want to approximate this function with several tower functions similar to the one in part **b**. Suppose all tower functions used have the same base square size, but different heights. You want to make sure that the maximum error for each tower function used is ϵ. What is the minimum number of tower functions that can guarantee to make such an approximation for all possible function f that satisfies the conditions? (Hint: think about the worst case for f.) Using the result from part **b**, what can you say about the relationship between the gradient limit, error bound, and the total required hidden neuron number?