COMS 4721/4771 HW2 (Spring 2024)

Due: Sun March 03, 2024 at 11:59pm

This homework is to be done **alone**. No late homeworks are allowed. To receive credit, a typesetted copy of the homework pdf must be uploaded to Gradescope by the due date. You must show your work to receive full credit. Discussing possible approaches for solutions for homework questions is encouraged on the course discussion board and with your peers, but you must write your own individual solutions and **not** share your written work/code. You **must cite** all resources (including online material, books, articles, ai generative bots, help taken from/given to specific individuals, etc.) you used to complete your work.

It is your responsibility to protect your work, and it is a violation of academic integrity policy to post any part of these questions or your answers to public websites such as: github, bitbucket, chegg, coursehero, etc. **Violators will be reported to the dean for disciplinary action**.

1 On Forecasting Product Demand

One way retail industry uses machine learning is to predict how much quantity Q of some product they should buy to maximize their profit. The optimal quantity depends on how much demand D there is for the product, as well as its cost for the retailer to buy C, and its selling price P to the customer. Assuming that the demand D is distributed as $P(D)$, we can evaluate the expected profit considering two cases:

- if $D \geq Q$, then the retailer sells all Q items and makes a profit $\pi = (P - C)Q$.
- but if $D < Q$, then the retailer can only sell D items at profit $(P - C)D$, but has lost $C(Q - D)$ on unsold items.

(i) What is the expected profit if the retailer buys Q items? Simplify the expression as much as possible.

(ii) By taking the derivative (with respect to Q) of the above expression for expected profit, show that the optimal quantity Q^* to buy satisfies $Q^* = F^{-1}(1 - (C/P))$, where F is the CDF of D. That is, the optimal Q^* is when the cumulative density (of D) equals $1 - (C/P)$.

2 Universality of Decision Trees

(i) Show that any binary classifier $g : \{0,1\}^D \to \{0,1\}$ can be implemented as a decision tree classifier. That is, for any classifier g there exists a decision tree classifier T with k nodes $n_1, \ldots, n_k$ (each n_i with a corresponding threshold t_i), such that $g(x) = T(x)$ for all $x \in \{0,1\}^D$.

(ii) What is the best possible bound one can give on the maximum height of such a decision tree T (from part (i))? For what function g is the bound tight?

3 Learning DNFs with kernel perceptron

Suppose that we have $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ with $x^{(i)} \in \{0,1\}^d$ and $y^{(i)} \in \{-1, 1\}$. Let $\varphi : \{0,1\}^d \to \{0,1\}$ be a "target function" which "labels" the points. Additionally assume that φ is a DNF formula (i.e. φ is a disjunction of conjunctions, or a boolean "or" of a bunch of boolean "and"s). The fact that it "labels" the points simply means that $\mathbf{1}[y^{(i)} = 1] = \varphi(x^{(i)})$.

For example, let $\varphi(x) = (x_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2 \wedge x_3)$ (where x_i denotes the ith entry of x), $x^{(i)} = \begin{pmatrix}1 & 0 & 1\end{pmatrix}^\mathsf{T}$, and $x^{(j)} = \begin{pmatrix}1 & 0 & 0\end{pmatrix}^\mathsf{T}$. Then, we would have $\varphi(x^{(i)}) = 1$ and $\varphi(x^{(j)}) = 0$, and thus $y^{(i)} = 1$ and $y^{(j)} = -1$.

(i) Give an example target function φ (make sure its a DNF formula) and set S such that the data is not linearly separable.

Part (i) clearly shows that running the perceptron algorithm on S cannot work in general since the data does not need to be linearly separable. However, we can try to use a feature transformation and the kernel trick to linearize the data and thus run the kernelized version of the perceptron algorithm on these datasets.

Consider the feature transformation $\phi : \{0,1\}^d \to \{0,1\}^{3^d}$ which maps a vector x to the vector of all the conjunctions of its entries or of their negations. So for example if $d = 2$ then

$$\phi(x) = \begin{pmatrix}1 & x_1 & x_2 & \bar{x}_1 & \bar{x}_2 & x_1 \wedge x_2 & x_1 \wedge \bar{x}_2 & \bar{x}_1 \wedge x_2 & \bar{x}_1 \wedge \bar{x}_2\end{pmatrix}^\mathsf{T}$$

(note that 1 can be viewed as the empty conjunction, i.e. the conjunction of zero literals).

Let $K : \{0,1\}^d \times \{0,1\}^d \to \mathbb{R}$ be the kernel function associated with ϕ (i.e. for $a, b \in \{0,1\}^d : K(a,b) = \phi(a) \cdot \phi(b)$). Note that the naive approach of calculating $K(a,b)$ (simply calculating $\phi(a)$ and $\phi(b)$ and taking the dot product) takes time $\Theta(3^d)$.

Also let $w^* \in \{0,1\}^{3^d}$ be such that $w_1^* = -0.5$ (this is the entry which corresponds to the empty conjunction, i.e. $\forall x \in \{0,1\}^d : \phi(x)_1 = 1$) and $\forall i > 1 : w_i^* = 1$ iff the ith conjunction is one of the conjunctions of φ. So for example in the above case where $d = 2$ and $\phi(x) = \begin{pmatrix}1 & x_1 & x_2 & \overline{x_1} & \overline{x_2} & x_1 \wedge x_2 & x_1 \wedge \overline{x_2} & \overline{x_1} \wedge x_2 & \overline{x_1} \wedge \overline{x_2}\end{pmatrix}^\mathsf{T}$ and letting $\varphi(x) = (x_1 \wedge x_2) \vee (\overline{x_1})$ we would have:

$$w^* = \begin{pmatrix}-0.5 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0\end{pmatrix}^\mathsf{T}$$

(ii) Find a way to compute $K(a,b)$ in $O(d)$ time.

(iii) Show that w^* linearly separates $\phi(S)$ ($\phi(S)$ is just a shorthand for $\{(\phi(x^{(i)}), y^{(i)})\}_{i=1}^n$) and find a lower bound for the margin γ with which it separates the data. Remember that $\gamma = \min_{(\phi(x^{(i)}),y^{(i)})\in\phi(S)} y_i\left(\frac{w^*}{\|w^*\|} \cdot \phi(x^{(i)})\right)$. Your lower bound should depend on s, the number of conjunctions in φ.

(iv) Find an upper bound on the radius R of the dataset $\phi(S)$. Remember that

$$R = \max_{(\phi(x^{(i)}),y^{(i)})\in\phi(S)} \|\phi(x^{(i)})\|.$$

(v) Use parts (ii), (iii), and (iv) to show that we can run kernel perceptron efficiently on this transformed space in which our data is linearly separable (show that each iteration only takes $O(nd)$ time per point) but that unfortunately the mistake bound is very bad (show that it is $O(s2^d)$).

There are ways to get a better mistake bound in this same kernel space, but the running time then becomes very bad (exponential). It is open whether there are ways to get both polynomial mistake bound and running time.

4 Neural Networks as Universal Function Approximators

Neural networks are a flexible class of parametric models which form the basis for a wide range of algorithms in machine learning. Their widespread success is due both to the ease and efficiency of "training" them, or optimizing over their parameters, using the backpropagation algorithm, and their ability to approximate any smooth function. Recall that a feed-forward neural network is simply a combination of multiple 'neurons' such that the output of one neuron is fed as the input to another neuron. More precisely, a neuron ν_i is a computational unit that takes in an input vector $\vec{x}$ and returns a weighted combination of the inputs (plus the bias associated with neuron ν_i) passed through an activation function $\sigma : \mathbb{R} \to \mathbb{R}$, that is: $\nu_i(\vec{x}; \vec{w}_i, b_i) := \sigma(\vec{w}_i \cdot \vec{x} + b_i)$.

With this notation, we can define a layer ℓ of a neural network $\mathcal{N}^\ell$ that takes in an I-dimensional input and returns a O-dimensional output as

$$\begin{aligned} \mathcal{N}^\ell(x) &:= \left(\nu_1^\ell(\vec{x}), \nu_2^\ell(\vec{x}), \cdots, \nu_O^\ell(\vec{x})\right) && x \in \mathbb{R}^I \\ &= \sigma(W_\ell^\mathsf{T}\vec{x} + \vec{b}_\ell) && W_\ell \in \mathbb{R}^{d_I \times d_O} \text{ defined as } [\vec{w}_1^\ell, \ldots, \vec{w}_O^\ell], \\ & && \text{and } \vec{b}_\ell \in \mathbb{R}^{d_O} \text{ defined as } [b_1^\ell, \ldots, b_O^\ell]^\mathsf{T}. \end{aligned}$$

Here $\vec{w}_i^\ell$ and b_i^ℓ refers to the weight and the bias associated with neuron ν_i^ℓ in layer ℓ, and the activation function σ is applied pointwise. An L-layer (feed-forward) neural network $\mathcal{F}_{L\text{-layer}}$ is then defined as a network consisting of network layers $\mathcal{N}^1, \ldots, \mathcal{N}^L$, where the input to layer i is the output of layer $i-1$. By convention, input to the first layer (layer 1) is the actual input data.

(i) Consider a nonlinear activation function $\sigma : \mathbb{R} \to \mathbb{R}$ defined as $x \mapsto 1/(1+e^{-x})$. Show that $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$.

(ii) Consider a *single layer* feed forward neural network that takes a d_I-dimensional input and returns a d_O-dimensional output, defined as $\mathcal{F}_{\text{1-layer}}(x) := \sigma(W^T x + b)$ for some $d_I \times d_O$ weight matrix W and a $d_O \times 1$ vector b. Given a training dataset $(x_1, y_1), \ldots, (x_n, y_n)$, we can define the *average error* (with respect to the network parameters) of predicting y_i from input example x_i as:

$$E(W, b) := \frac{1}{2n} \sum_{i=1}^{n} \|\mathcal{F}_{\text{1-layer}}(x_i) - y_i\|^2.$$

What is $\frac{\partial E}{\partial W}$, and $\frac{\partial E}{\partial b}$?

(note: we can use this gradient in a descent-type procedure to minimize this error and learn a good setting of the weight matrix that can predict y_i from x_i.)

(iii) Single layer neural networks—though reasonably expressive—are not flexible enough to approximate arbitrary smooth functions. Here we will focus on approximating some fun two-dimensional parametric functions $f : [0,1]^2 \to \mathbb{R}$ with a *multi-layer* neural network.

Consider an L-layer neural network $\mathcal{F}_{L\text{-layer}}$ as described above. Given a sample of input-output pairs $(x_1, y_1), \ldots, (x_n, y_n)$ generated from an unknown fixed function f, one can approximate f from $\mathcal{F}_{L\text{-layer}}$ by minimizing the following error function over the parameters W^T and b.

$$\begin{aligned} E(W_1, b_1, \ldots, W_L, b_L) &:= \frac{1}{2n} \sum_{i=1}^{n} \|\mathcal{F}_{L\text{-layer}}(x_i) - y_i\|^2 \\ &= \frac{1}{2n} \sum_{i=1}^{n} \|\mathcal{N}^L \circ \cdots \circ \mathcal{N}^1(x_i) - y_i\|^2. \end{aligned}$$

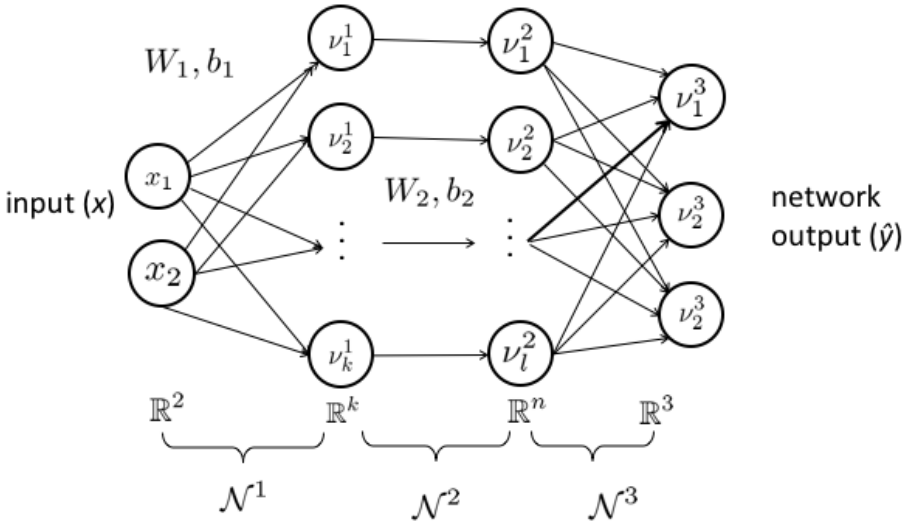
First we will explore how to train such a general network efficiently.

(a) At its core, a neural network is a chain of composed linear transformations connected by non-linearities, i.e. $\mathcal{N}(x) = \sigma_1 \circ f_1 \circ ... \circ \sigma_n \circ f_n$. Using the chain rule, compute the derivative of $\mathcal{N}(x)$. Naively, throwing away the result of each computation, what is the complexity of evaluating the derivative of a chain of n functions, in terms of the length of the chain n?

(b) Noticing that the computation of the derivative $\mathcal{N}'(x)$ involves computing $f_2 \circ ... \circ f_n(x)$, $f_3 \circ ... \circ f_n(x)$, and many other intermediate terms, describe how we can we improve the naive approach from the previous part by reusing computations performed when evaluating the function to compute the derivative at the same point. What is the complexity of this algorithm, again in terms of the length of the chain n? This is the *backpropagation algorithm.*

(iv) Now we will combine all of these calculations to implement a flexible framework for learning neural networks on arbitrary data. Your task is to implement a gradient descent procedure to learn the parameters of a general L-layer feed forward neural network using the backpropogation algorithm.

You will use this framework to learn to approximate complicated patterns in images. The provided data has a 2-dimensional input corresponding to the coordinate of a given pixel, i.e. $X = [(0,0),(0,1),...,(n,m)]$, and the output is a one- or three-dimensional value giving a greyscale or RGB value for the image at that coordinate respectively. Note that since each x_i is 2-dimensional, layer $\mathcal{N}^1$ only contains two neurons. All other layers can contain an arbitrary number, say $k_1, ..., k_L$, neurons. The output layer will have either one or three neurons. Graphically the network you are implementing looks as follows, possibly with more intermediate layers.



Here is the pseudocode of your implementation:

Learn K-layer neural network for 2-d functions
input: data $(x_1, y_1), \ldots, (x_n, y_n)$,
. size of the intermediate layers $k_1, ..., k_L$
- Initialize weight parameters $(W_1, b_1), ..., (W_L, b_L)$ randomly
- Repeat until convergence:
- for a subset of training examples $\{(x_1, y_1), ..., (x_k, y_k)\}$
- compute the network output $\hat{y}_i$ on x_i (this is called the forward pass)
- compute gradients $\frac{\partial E}{\partial W_1}$, $\frac{\partial E}{\partial b_1}$, ..., $\frac{\partial E}{\partial W_L}$, $\frac{\partial E}{\partial b_L}$ using results saved from the forward pass
- update weight parameters:
- For all i, $W_i^{\text{new}} := W_i - \eta \frac{\partial E}{\partial W_i}$, $\quad b_i^{\text{new}} := b_i - \eta \frac{\partial E}{\partial b_i}$

You must submit your code to receive full credit.

For this entire question, you may only use standard Python libraries (math, string, etc.), numpy, pandas, matplotlib, and scipy.io.

Some hints: some example skeleton code `nnskeletoncode.py` has been provided as a possible way to structure your general neural network framework. You are not required to follow this skeleton code or to use Python. It merely provides inspiration for a possible way of structuring the project. This is close to how Tensorflow and PyTorch work.

- Each class in the skeleton code has at least a `forward()` and `backward()` function, which implement the forward and backward pass respectively. Any neural network framework should have some equivalent variation of these functions.
- In the forward pass, you should apply each layer of the model sequentially and store the intermediate results. The backwards pass then calculates and applies the gradients for each layer according to the gradient descent algorithm.
- You should be able to use different numbers of intermediate layers and different size inputs and outputs to try and improve the convergence and performance of your algorithm.
- We were able to reconstruct both images with a 3-layer network and between 128 and 512 dimensional hidden layers. Training can take as little as 5 minutes with Numpy or Matlab. Minibatch size can be quite small (at least initially) to accelerate learning.
- Try using a simple function like a quadratic function to test your framework before attempting something more complicated like the image data provided.
- Try looking at how PyTorch or Tensorflow structure their computational graphs. Since this is a purely linear network, you can represent it as a linked-list of layers, each of which can pass the gradient to its child node.

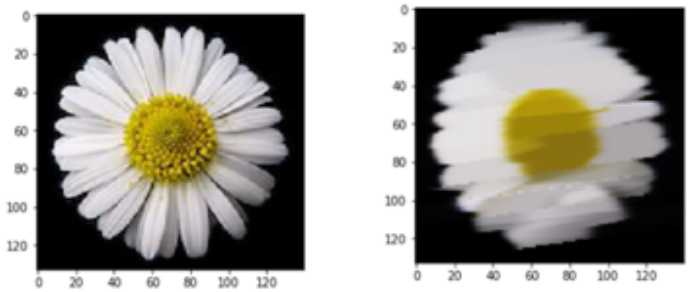
Different gradient descent methods: Once we have computed the gradients, there are many ways to perform gradient descent to optimize the function. **Stochastic Gradient Descent** (SGD) takes a subset of the data, averages the gradients over that subset, and steps iteratively in that direction. In this assignment, we are going to use **Adam** (short for Adaptive Momentum), a more sophisticated method that adapts the step size dynamically for each component of the gradient. The algorithm is:

Optimize a loss function with Adam
input: data $(x_1, y_1), \ldots, (x_n, y_n)$,
- Initialize $\beta_1 = 0.9$, $\beta_2 = 0.999$
- For all i, initialize $m_i = \vec{0}$, $v_i = \vec{0}$, zero vectors with same shape as weights W_i.
- Repeat until convergence:
- for a subset of training examples $\{(x_1, y_1), ..., (x_k, y_k)\}$
- compute the network output $\hat{y}_i$ on x_i
- compute gradients $\frac{\partial E}{\partial W_1}$, $\frac{\partial E}{\partial b_1}$, ..., $\frac{\partial E}{\partial W_L}$, $\frac{\partial E}{\partial b_L}$
- update weight parameters (do the same for bias terms):
- For all i, $m_i^{\text{new}} := (\beta_1) m_i + (1 - \beta_1) \frac{\partial E}{\partial W_i}$, $v_i^{\text{new}} := (\beta_2) v_i + (1 - \beta_2) \left(\frac{\partial E}{\partial W_i} \right)^2$
- For all i, $W_i^{\text{new}} := W_i - \eta \frac{m_i}{\sqrt{v_i} + \epsilon}$

Do the same for the biases as well. For more information, refer to this link. This gradient descent algorithm will allow your network to learn more flexibly. Start by implementing traditional gradient descent, and then add Adam after the network works on simple inputs.

(v) Download `nn_data.mat`. It contains data for two distinct images, each with vector valued variables X and Y (they will be labeled $X1$, $X2$, and $Y1$ and $Y2$ for the two images). Each row in Y gives the pixel value at the corresponding coordinate in X.

You can visualize the data by reformatting the image data Y into a 2D array and displaying it as an image (`imshow` in Python or Matlab may be useful). For example, the second image (and our reconstruction using a very simple network) looks like this:



(a) original image (b) reconstructed

Use your implementation to learn the unknown mapping function which can yield Y from X for both image datasets. Once the network parameters are learned, plot the generated pixel values as an image and include the results with your writeup. Compare the results for at least two settings for the size and number of hidden layers in your network.