COMS 4771 HW3 (Fall 2022)

Due: Wed Nov 16, 2022 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, help taken from/given to specific individuals, etc.) you used to complete your work.

1 Non-parametric Regression via Bayesian Modelling

Here we will study a generative modelling technique via Gaussians for non-parametric regression.

Before getting into regression we need to derive some facts about multivariate Gaussian distributions. Let $x \in \mathbb{R}^d$ be distributed normally as

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right) = \mathcal{N}(\mu, \Sigma)$$

(i) Derive the marginal distribution of x_1?

(ii) (*warning! tedious calculations*) Let $\Sigma^{-1} = \begin{bmatrix} \Sigma^{11} & \Sigma^{12} \\ \Sigma^{21} & \Sigma^{22} \end{bmatrix}$. Using the facts that[1]

- $\Sigma^{11} = (\Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{12}^{\mathsf{T}})^{-1} = \Sigma_{11}^{-1} + \Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^{\mathsf{T}}\Sigma_{11}^{-1}\Sigma_{12})^{-1}\Sigma_{12}^{\mathsf{T}}\Sigma_{11}^{-1}$
- $\Sigma^{22} = (\Sigma_{22} - \Sigma_{12}^{\mathsf{T}}\Sigma_{11}^{-1}\Sigma_{12})^{-1} = \Sigma_{22}^{-1} + \Sigma_{22}^{-1}\Sigma_{12}^{\mathsf{T}}(\Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{12}^{\mathsf{T}})^{-1}\Sigma_{12}\Sigma_{22}^{-1}$
- $\Sigma^{12} = -\Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{22} - \Sigma_{12}^{\mathsf{T}}\Sigma_{11}^{-1}\Sigma_{12})^{-1} = (\Sigma^{21})^{\mathsf{T}}$

Show that the joint distribution on x can be written as

$$\frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\Big\{ -\frac{1}{2}\Big((x_1 - \mu_1)^{\mathsf{T}}\Sigma_{11}^{-1}(x_1 - \mu_1) + (x_2 - b)^{\mathsf{T}}A^{-1}(x_2 - b) \Big) \Big\},$$

where $b = \mu_2 + \Sigma_{12}^{\mathsf{T}}\Sigma_{11}^{-1}(x_1 - \mu_1)$, and $A = \Sigma_{22} - \Sigma_{12}^{\mathsf{T}}\Sigma_{11}^{-1}\Sigma_{12}$.

(iii) Now using the fact that $|\Sigma| = |\Sigma_{11}||\Sigma_{22} - \Sigma_{12}^{\mathsf{T}}\Sigma_{11}^{-1}\Sigma_{12}|$, show that the joint on x can be decomposed as product

$$\mathcal{N}(x_1; \mu_1, \Sigma_{11})\mathcal{N}(x_2; b, A),$$

where b and A are as defined in previous part.

[1] Feel free to prove these facts for yourself, if you are bored :).

(iv) What is the conditional distribution of x_2 given x_1?

Now we are ready to do some regression via generative modelling. Like in any generative model, we first need a prior over our objects of interest (in this case, the regression functions), then given some data (also known as evidence), we shall compute the posterior (in this case, those regression functions that agree with the given data).

A prior over the regression functions. A function $f : \mathbb{R} \to \mathbb{R}$ can be thought of as an infinite length vector. Suppose we want to know the value of this function at positions $x_1, \ldots, x_n \in \mathbb{R}$, we can write it down the result as a vector $\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}$. A simple way to *generate* random functions then is to simply model it as the Gaussian distribution, specifically $\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix} \sim \mathcal{N}(\mu_n, \Sigma_{n\times n})$.

Throughout our discussion below, we will use 500 equal spaced points between the range of -10 and 10 as our locations $x_1, \ldots, x_n$, that is $x_1 = -10, x_2 = -9.959, \ldots, x_{500} = 10$.

(v) For $\mu_n = \vec{0}$ and $\Sigma_{n\times n} = I$, draw 4 random functions and show their plots[2]. What can you say about the smoothness of these functions? What happens if Σ is set to all ones matrix? Play with various values of μ and Σ, what effect does it have on the distribution of the random functions? Explain why these effects are occurring.

(vi) Usually $\mu_n = \vec{0}$ and $\Sigma_{n\times n} = K$, where $K_{ij} = k(x_i, x_j)$ for some kernel function k. A popular choice is $k : (x_i, x_j) \mapsto \exp\{-(x_i - x_j)^2/h\}$, for some fixed parameter h. Draw 4 random functions from this setting of μ and Σ ($h = 5$). and show their plots. What can you say about the smoothness of these functions?

(vii) If one is interested in random periodic functions, qualitatively explain what setting of μ and Σ would be appropriate? Pick a μ and Σ which can generate periodic functions of periodicity 3 units. Draw 4 random functions from that setting and plot them to verify.

The posterior over the regression functions. Of course, in the problem of regression, one is not interested in drawing random functions, but instead, understanding/predicting the trend in data given some observations. Suppose we are given a training data $(\bar{x}_1, \bar{y}_1), \ldots, (\bar{x}_m, \bar{y}_m) = (\overline{\mathbf{X}}, \overline{\mathbf{Y}})$. Then using the suggested model we can model the joint distribution as

[2] use x-axis range -10 to 10, y-axis range -3 to 3 for all your plots.

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \\ f(\bar{x}_1) = \bar{y}_i \\ \vdots \\ f(\bar{x}_m) = \bar{y}_m \end{bmatrix} = \begin{bmatrix} f(\mathbf{X}) \\ f(\overline{\mathbf{X}}) = \overline{\mathbf{Y}} \end{bmatrix} \sim \mathcal{N}\big(\mu_{n+m}, \Sigma_{(n+m)\times(n+m)}\big) = \mathcal{N}\left(\begin{bmatrix}\mu_n \\ \mu_m\end{bmatrix}, \begin{bmatrix}\Sigma_{nn} & \Sigma_{nm} \\ \Sigma_{mn} & \Sigma_{mm}\end{bmatrix}\right)$$

$$= \mathcal{N}\left(\begin{bmatrix}\mu_n \\ \mu_m\end{bmatrix}, \begin{bmatrix}K(\mathbf{X},\mathbf{X}) & K(\mathbf{X},\overline{\mathbf{X}}) \\ K(\overline{\mathbf{X}},\mathbf{X}) & K(\overline{\mathbf{X}},\overline{\mathbf{X}})\end{bmatrix}\right).$$

Let $y_1, \ldots, y_n = \mathbf{Y}$ be the regression values we are interested in knowing for a set of (test) locations $x_1, \ldots, x_n = \mathbf{X}$, then we can write the above joint model more compactly as

$$\begin{bmatrix}\mathbf{Y} \\ \overline{\mathbf{Y}}\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\mu_n \\ \mu_m\end{bmatrix}, \begin{bmatrix}K(\mathbf{X},\mathbf{X}) & K(\mathbf{X},\overline{\mathbf{X}}) \\ K(\overline{\mathbf{X}},\mathbf{X}) & K(\overline{\mathbf{X}},\overline{\mathbf{X}})\end{bmatrix}\right).$$

Hence, given the training data $(\overline{\mathbf{X}}, \overline{\mathbf{Y}})$ we are interested in knowing the posterior $\mathbf{Y}\big|\overline{\mathbf{Y}}$

(viii) Using the result from part (iv), what is the posterior $\mathbf{Y}\big|\overline{\mathbf{Y}}$?

(ix) For training data $\{(-6, 3), (0, -2), (7, 2)\}$ and K induced by kernel function $k : (x_i, x_j) \mapsto \exp\{-(x_i - x_j)^2/5\}$, draw 4 random functions from the posterior and plot the resulting functions. Make sure to depict the three training datapoints on the same plot. What do you notice?

(x) For the training data in part (ix) and the periodic Σ used in part (vii), draw 4 random functions from the posterior and plot the resulting functions (along with the training data). What do you notice in this case?

Notice that this Bayesian modelling technique provides a (posterior) *distribution* over regression values for the test locations. One can use the mean value as the final prediction over the test locations.

(xi) What is the mean of the posterior $\mathbf{Y}\big|\overline{\mathbf{Y}}$?

(xii) Plot the mean "function" for parts (ix) and (x) as well.

2 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.

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

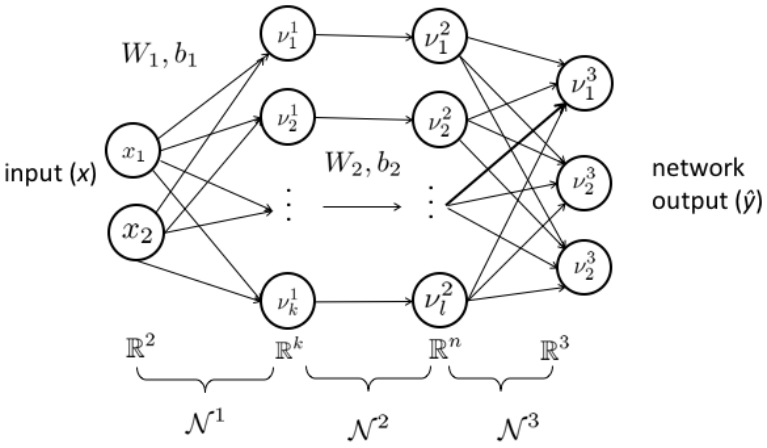
First we will explore how to train such a general network efficiently.

(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
- 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.

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.

- 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**

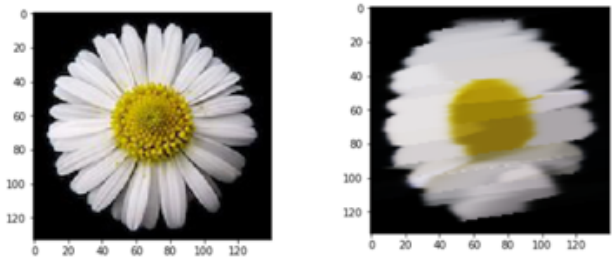
(SGD) takes a subset of the data, averages the gradients over that subset, and steps iteratively in that direction. In this assignment, we are going to use **Adam** (short for Adaptive Momentum), a more sophisticated method that adapts the step size dynamically for each component of the gradient. The 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, look at `http://cs231n.github.io/neural-networks-3/#ada`. 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: Use your implementation



(a) original image (b) reconstructed

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.