CS 229, Fall 2020
Problem Set #2

Due Wednesday, October 21 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be as concise as possible. (2) If you have a question about this homework, we encourage you to post your question on our Piazza forum, at `https://piazza.com/stanford/fall2020/cs229`. (3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work. (4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted. (5) To account for late days, the due date is Wednesday, October 21 at 11:59 pm. If you submit after Wednesday, October 21 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Wednesday, October 21 at 11:59 pm. Please make sure to reserve sufficient time for potentially compiling, scanning, and uploading your homework questions.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via LaTeX, and we will award one bonus point for submissions typeset in LaTeX. Please make sure to tag your solutions properly on Gradescope. The graders reserve the right to penalize incorrectly tagged solutions by 0.5 points per question. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup.

1. **[15 points] Logistic Regression: Training stability**

In this problem, we will be delving deeper into the workings of logistic regression. The goal of this problem is to help you develop your skills debugging machine learning algorithms (which can be very different from debugging software in general).

We have provided an implementation of logistic regression in `src/stability/stability.py`, and two labeled datasets A and B in `src/stability/ds1_a.csv` and `src/stability/ds1_b.csv`.

Please do not modify the code for the logistic regression training algorithm for this problem. First, run the given logistic regression code to train two different models on A and B. You can run the code by simply executing `python stability.py` in the `src/stability` directory.

(a) [2 points] What is the most notable difference in training the logistic regression model on datasets A and B?

(b) [5 points] Investigate why the training procedure behaves unexpectedly on dataset B, but not on A. Provide hard evidence (in the form of math, code, plots, etc.) to corroborate your hypothesis for the misbehavior. Remember, you should address why your explanation does *not* apply to A.

Hint: The issue is not a numerical rounding or over/underflow error.

(c) [5 points] For each of these possible modifications, state whether or not it would lead to the provided training algorithm converging[1] on datasets such as B. Briefly explain your answers. (You are not expected to prove or provide a formal justification for your answers.)

i. Using a different constant learning rate.

ii. Decreasing the learning rate over time by using the learning rate $\alpha_t = c/t^3$ at iteration t, where $c > 0$ is a constant representing the initial learning rate and t is the number of gradient descent iterations thus far.

iii. Linear scaling of the input features.

iv. Adding a regularization term $\|\theta\|_2^2$ to the loss function.

v. Adding zero-mean Gaussian noise to the training data or labels.

(d) [3 points] Are support vector machines vulnerable to datasets like B? Why or why not? Give an informal justification.

[1] Technically, we consider the logistic regression parameters $\theta^{[t]} \in \mathbb{R}^d$ as converging if there exists a vector $\theta^\dagger \in \mathbb{R}^d$ such that $\lim_{t\to\infty} \|\theta^{[t]} - \theta^\dagger\|_2 = 0$. You are not expected to use this formal definition in your answers.

2. **[24 points] Spam classification**

In this problem, we will use the naive Bayes algorithm and an SVM to build a spam classifier.

In recent years, spam on electronic media has been a growing concern. Here, we'll build a classifier to distinguish between real messages, and spam messages. For this class, we will be building a classifier to detect SMS spam messages. We will be using an SMS spam dataset developed by Tiago A. Almedia and José María Gómez Hidalgo which is publicly available on `http://www.dt.fee.unicamp.br/~tiago/smsspamcollection` [2]

We have split this dataset into training and testing sets and have included them in this assignment as `src/spam/spam_train.tsv` and `src/spam/spam_test.tsv`. See `src/spam/spam_readme.txt` for more details about this dataset. Please refrain from redistributing these dataset files. The goal of this assignment is to build a classifier from scratch that can tell the difference the spam and non-spam messages using the text of the SMS message.

(a) [5 points] Implement code for processing the spam messages into numpy arrays that can be fed into machine learning models. Do this by completing the `get_words`, `create_dictionary`, and `transform_text` functions within our provided `src/spam/spam.py`. Do note the corresponding comments for each function for instructions on what specific processing is required.

The provided code will then run your functions and save the resulting dictionary into `spam_dictionary` and a sample of the resulting training matrix into `spam_sample_train_matrix`.

In your writeup, report the vocabular size after the pre-processing step. You do not need to include any other output for this subquestion.

(b) [10 points] In this question you are going to implement a naive Bayes classifier for spam classification with **multinomial event model** and Laplace smoothing. The multinomial event model is presented in section 2.2 of the "Generative Learning algorithms" lecture notes.

Code your implementation by completing the `fit_naive_bayes_model` and `predict_from_naive_bayes_model` functions in `src/spam/spam.py`.

Now `src/spam/spam.py` should be able to train a Naive Bayes model, compute your prediction accuracy and then save your resulting predictions to `spam_naive_bayes_predictions`.

In your writeup, report the accuracy of the trained model on the **test set**.

Remark. If you implement naive Bayes the straightforward way, you will find that the computed $p(x|y) = \prod_i p(x_i|y)$ often equals zero. This is because $p(x|y)$, which is the product of many numbers less than one, is a very small number. The standard computer representation of real numbers cannot handle numbers that are too small, and instead rounds them off to zero. (This is called "underflow.") You'll have to find a way to compute Naive Bayes' predicted class labels without explicitly representing very small numbers such as $p(x|y)$. [**Hint:** Think about using logarithms.]

(c) [5 points] Intuitively, some tokens may be particularly indicative of an SMS being in a particular class. We can try to get an informal sense of how indicative token i is for the SPAM class by looking at:

$$\log \frac{p(x_j = i \mid y = 1)}{p(x_j = i \mid y = 0)} = \log \left(\frac{P(\text{token } i \mid \text{email is SPAM})}{P(\text{token } i \mid \text{email is NOTSPAM})} \right).$$

[2] Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.

Complete the `get_top_five_naive_bayes_words` function within the provided code using the above formula in order to obtain the 5 most indicative tokens.

Report the top five words in your writeup.

(d) [2 points] Support vector machines (SVMs) are an alternative machine learning model that we discussed in class. We have provided you an SVM implementation (using a radial basis function (RBF) kernel) within `src/spam/svm.py` (You should not need to modify that code).

One important part of training an SVM parameterized by an RBF kernel (a.k.a Gaussian kernel) is choosing an appropriate kernel radius parameter.

Complete the `compute_best_svm_radius` by writing code to compute the best SVM radius which maximizes accuracy on the validation dataset. Report the best kernel radius and the corresponding test set accuracy you obtained in the writeup.

(e) [2 points] For linear models such as logistic regression or SVM with linear kernel, the features that we use to represent the data matter. In kernel methods (for example, SVM with RBF kernel in the part above), the features are generated by some fixed feature map $\phi(x)$.

In the deep learning era, high-quality features $\phi(x)$ can be learned from deep learning techniques on potentially other bigger datasets. In this part, we will ask you to use these features and solve the spam classification problem better than the Naive Bayes and SVM approaches. The methods of obtaining these features are beyond the scope of this course. (In a nutshell, the feature map ϕ that you will use, which is called BERT,[3] is learned using large-scale neural networks on language modeling tasks with a huge text corpus.)

You are provided with the features $\{\phi(x^{(1)}), \ldots, \phi(x^{(n)})\}$ of the training set in `src/spam/bert_train_matrix.tsv.bz2`, which contains a matrix $\in \mathbb{R}^{n \times 768}$ where n is the number of training examples (messages). Each row is a tab separated list of floating point values representing the 768 dimensional feature for one example. Similarly, `src/spam/bert_val_matrix.tsv.bz2` and `src/spam/bert_test_matrix.tsv.bz2` contain feature matrices for validation and test examples. The order of the examples in these feature matrices are the same as the order of the examples in the original data and labels. The code to read in the data has already been implemented for you.

A logistic regression model on top of these new features can be used to model the problem, that is,

$$p(y = 1 \mid x; \theta) = h_\theta(x) = \sigma(\theta^\top \phi(x)) \tag{1}$$

where $\sigma(z) = \frac{1}{1+\exp(-z)}$ is the sigmoid function. We have provided you the code to run gradient descent for logistic regression, and you will be asked to tune the learning rate based on the validation set accuracy.

The code to run gradient descent for logistic regression with a specific choice of learning rate is in the function `train_and_predict_logreg` in `src/spam/logreg.py`. (You should not need to modify the code in this file.) This function has the same input and output arguments as `train_and_predict_svm` in `src/spam/svm.py` from the previous sub-question with the exception of the last input argument being a specific learning rate instead of being a specific radius. Please implement the function `compute_best_logreg_learning_rate` in `src/spam/spam.py` to compute and return the best learning rate for logistic regression (among the choices of learning rates provided to you in the code) that maximizes the validation accuracy, similar to the previous sub-question.

[3] A pretrained neural transformer model by Google. Details of the models can be found at `https://arxiv.org/abs/1810.04805`. Code and example usage can be found at `https://github.com/huggingface/transformers`

Report the best learning rate and the corresponding test set accuracy in your writeup.

Note: It is possible that you will see some warnings about "divide by zero" or "invalid value" for certain learning rates. These are expected since some of the learning rates cause gradient descent to diverge.

3. **[15 points] Kernelizing the Perceptron**

Let there be a binary classification problem with $y \in \{0, 1\}$. The perceptron uses hypotheses of the form $h_\theta(x) = g(\theta^T x)$, where $g(z) = \text{sign}(z) = 1$ if $z \geq 0$, 0 otherwise. In this problem we will consider a stochastic gradient descent-like implementation of the perceptron algorithm where each update to the parameters θ is made using only one training example. However, unlike stochastic gradient descent, the perceptron algorithm will only make one pass through the entire training set. The update rule for this version of the perceptron algorithm is given by

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))x^{(i+1)}$$

where $\theta^{(i)}$ is the value of the parameters after the algorithm has seen the first i training examples. Prior to seeing any training examples, $\theta^{(0)}$ is initialized to $\vec{0}$.

(a) [3 points] Let K be a kernel corresponding to some very high-dimensional feature mapping ϕ. Suppose ϕ is so high-dimensional (say, ∞-dimensional) that it's infeasible to ever represent $\phi(x)$ explicitly. Describe how you would apply the "kernel trick" to the perceptron to make it work in the high-dimensional feature space ϕ, but without ever explicitly computing $\phi(x)$. [**Note:** You don't have to worry about the intercept term. If you like, think of ϕ as having the property that $\phi_0(x) = 1$ so that this is taken care of.] Your description should specify:

i. [1 points] How you will (implicitly) represent the high-dimensional parameter vector $\theta^{(i)}$, including how the initial value $\theta^{(0)} = 0$ is represented (note that $\theta^{(i)}$ is now a vector whose dimension is the same as the feature vectors $\phi(x)$);

ii. [1 points] How you will efficiently make a prediction on a new input $x^{(i+1)}$. I.e., how you will compute $h_{\theta^{(i)}}(x^{(i+1)}) = g({\theta^{(i)}}^T \phi(x^{(i+1)}))$, using your representation of $\theta^{(i)}$; and

iii. [1 points] How you will modify the update rule given above to perform an update to θ on a new training example $(x^{(i+1)}, y^{(i+1)})$; *i.e.,* using the update rule corresponding to the feature mapping ϕ:

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))\phi(x^{(i+1)})$$

(b) [10 points] Implement your approach by completing the `initial_state`, `predict`, and `update_state` methods of `src/perceptron/perceptron.py`.

We provide three functions to be used as kernel, a dot-product kernel defined as:

$$K(x, z) = x^\top z, \tag{2}$$

a radial basis function (RBF) kernel, defined as:

$$K(x, z) = \exp\left(-\frac{\|x - z\|_2^2}{2\sigma^2}\right), \tag{3}$$

and finally the following function:

$$K(x, z) = \begin{cases} -1 & x = z \\ 0 & x \neq z \end{cases} \tag{4}$$

Note that the last function is not a kernel function (since its corresponding matrix is not a PSD matrix). However, we are still interested to see what happens when

the kernel is invalid. Run `src/perceptron/perceptron.py` to train kernelized perceptrons on `src/perceptron/train.csv`. The code will then test the perceptron on `src/perceptron/test.csv` and save the resulting predictions in the `src/perceptron/` folder. Plots will also be saved in `src/perceptron/`.

Include the three plots (corresponding to each of the kernels) in your writeup, and indicate which plot belongs to which function.

(c) [2 points] One of the choices in Q4b completely fails, one works a bit, and one works well in classifying the points. Discuss the performance of different choices and why do they fail or perform well?

4. **[25 points] Bayesian Interpretation of Regularization**

Background: In Bayesian statistics, almost every quantity is a random variable, which can either be observed or unobserved. For instance, parameters θ are generally unobserved random variables, and data x and y are observed random variables. The joint distribution of all the random variables is also called the *model* (*e.g.,* $p(x, y, \theta)$). Every unknown quantity can be estimated by conditioning the model on all the observed quantities. Such a conditional distribution over the unobserved random variables, conditioned on the observed random variables, is called the *posterior distribution*. For instance $p(\theta|x, y)$ is the posterior distribution in the machine learning context. A consequence of this approach is that we are required to endow our model parameters, *i.e.,* $p(\theta)$, with a *prior distribution*. The prior probabilities are to be assigned *before* we see the data—they capture our prior beliefs of what the model parameters might be before observing any evidence.

In the purest Bayesian interpretation, we are required to keep the entire posterior distribution over the parameters all the way until prediction, to come up with the *posterior predictive distribution*, and the final prediction will be the expected value of the posterior predictive distribution. However in most situations, this is computationally very expensive, and we settle for a compromise that is *less pure* (in the Bayesian sense).

The compromise is to estimate a point value of the parameters (instead of the full distribution) which is the mode of the posterior distribution. Estimating the mode of the posterior distribution is also called *maximum a posteriori estimation* (MAP). That is,

$$\theta_{\text{MAP}} = \arg\max_{\theta} p(\theta|x, y).$$

Compare this to the *maximum likelihood estimation* (MLE) we have seen previously:

$$\theta_{\text{MLE}} = \arg\max_{\theta} p(y|x, \theta).$$

In this problem, we explore the connection between MAP estimation, and common regularization techniques that are applied with MLE estimation. In particular, you will show how the choice of prior distribution over θ (*e.g.,* Gaussian or Laplace prior) is equivalent to different kinds of regularization (*e.g.,* L_2, or L_1 regularization). You will also explore how regularization strengths affect generalization in part (d).

(a) [3 points] Show that $\theta_{\text{MAP}} = \text{argmax}_\theta\, p(y|x, \theta)p(\theta)$ if we assume that $p(\theta) = p(\theta|x)$. The assumption that $p(\theta) = p(\theta|x)$ will be valid for models such as linear regression where the input x are not explicitly modeled by θ. (Note that this means x and θ are marginally independent, but not conditionally independent when y is given.)

(b) [5 points] Recall that L_2 regularization penalizes the L_2 norm of the parameters while minimizing the loss (*i.e.,* negative log likelihood in case of probabilistic models). Now we will show that MAP estimation with a zero-mean Gaussian prior over θ, specifically $\theta \sim \mathcal{N}(0, \eta^2 I)$, is equivalent to applying L_2 regularization with MLE estimation. Specifically, show that for some scalar λ,

$$\theta_{\text{MAP}} = \arg\min_{\theta} -\log p(y|x, \theta) + \lambda||\theta||_2^2. \tag{5}$$

Also, what is the value of λ? You can continue to assume that $p(\theta) = p(\theta|x)$.

(c) [7 points] Now consider a specific instance, a linear regression model given by $y = \theta^T x + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Assume that the random noise $\epsilon^{(i)}$ is independent for every training

example $x^{(i)}$. Like before, assume a Gaussian prior on this model such that $\theta \sim \mathcal{N}(0, \eta^2 I)$. For notation, let X be the design matrix of all the training example inputs where each row vector is one example input, and $\vec{y}$ be the column vector of all the example outputs.

Come up with a closed form expression for θ_{MAP}.

(d) [5 points] Coding question: double descent phenomenon and the effect of regularization.

The Bayesian perspective provides a justification of using the L_2 regularization term as in equation (5), and it also provides a formula for the optimal choice of λ, assuming that we know the true prior for θ. For real-world applications, we often do not know the true prior, so λ is tuned based on the performance on the validation dataset. In this problem, you will be asked to verify empirically that the choice of λ you derived in part (b) is close to optimal, when the generating process of the data, θ, and the label y are as exactly described in part (c).

Meanwhile, you will empirically observe an interesting phenomenon, often called double descent, which was first discovered in 1970s and recently returned to spotlight. Sample-wise double descent is the phenomenon that the validation loss of some learning algorithm or estimator does not monotonically decrease as we have more training examples, but instead has a curve with two U-shaped parts. Model-wise double descent is a similar phenomenon as we increase the size of the model. For simplicity, here we only focus on the sample-wise double descent.

You will be asked to train on various datasets by minimizing the regularized cost function with various choices of λ:

$$-\log p(\vec{y}|X, \theta) + \lambda ||\theta||_2^2 \tag{6}$$

and plot the validation errors for the choices of datasets and λ. You will observe the double descent phenomenon for relatively small λ but not the optimal choice of λ.

Problem details: We assume that the data are generated as described in part (c) with $d = 500, \sigma = 0.5$. You are asked to have a Gaussian prior $\theta \sim \mathcal{N}(0, \eta^2 I)$ with $\eta = 1/\sqrt{d}$ on the parameter θ. (The teaching staff indeed generated the ground-truth θ from this prior.)

You are given 13 training datasets of sample sizes $n = 200, 250, \ldots, 750$, and 800, and a validation dataset, located at

- `src/bayesianreg/train200.csv`, `train250.csv`, etc.
- `src/bayesianreg/validation.csv`

Let λ_{opt} denote the regularization strength that you derived from part (b). (λ_{opt} is a function of η.)

For each training dataset $(X, \vec{y})$ and each $\lambda \in \{0, 1/32, 1/16, 1/8, 1/4, 1/2, 1, 2, 4\} \times \lambda_{opt}$, compute the optimizer of equation (6), and evaluate the mean-squared-error of the optimizer on the validation dataset. The MSE for your trained estimators $\hat{\theta}$ on a validation dataset $(X_v, \vec{y}_v)$ of size n_v is defined as:

$$MSE(\hat{\theta}) = \frac{1}{n_v} \|X_v \hat{\theta} - \vec{y}_v\|_2^2.$$

Complete the `ridge_regression` method of `src/bayesianreg/doubledescent.py` which takes in a training file and a validation file, computes the θ that minimizes training objective under different regularization strengths, and returns a list of validation errors (one for each choice of λ).

Include in your writeup a plot of the validation errors of these models. The x-axis is the size of the training dataset (from 200 to 800); the y-axis is the MSE on the validation dataset. Draw one line for each choice of λ connecting the validation errors across different training dataset sizes. Therefore, the plot should contain 9×13 points and 9 lines connecting them.

You should observe that for (some of) those choices of λ with $\lambda < \lambda_{opt}$, the validation error may increase and then decrease as we increase the sample size. However, double descent does not occur for λ_{opt} or any regularization larger than λ_{opt}.

Note: Use the Moore-Penrose pseudo-inverse as implemented in `numpy.linalg.pinv` if your matrix is singular.

Remark: If you would like to know more about double descent, please feel free to check out the partial list of references in the introduction and related work [4] but knowing the references or papers are unnecessary for solving this homework problem. Roughly speaking, it is mostly caused by the lack of regularization when $n \approx d$. When $n \approx d$, the data matrix is particularly ill-conditioned and stronger and more explicit regularization is needed.

(e) [5 points] Next, consider the Laplace distribution, whose density is given by

$$f_{\mathcal{L}}(z|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|z-\mu|}{b}\right).$$

As before, consider a linear regression model given by $y = x^T\theta + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Assume a Laplace prior on this model, where each parameter θ_i is marginally independent, and is distributed as $\theta_i \sim \mathcal{L}(0, b)$.

Show that θ_{MAP} in this case is equivalent to the solution of linear regression with L_1 regularization, whose loss is specified as

$$J(\theta) = ||X\theta - \vec{y}||_2^2 + \gamma||\theta||_1$$

Also, what is the value of γ?

Note: A closed form solution for linear regression problem with L_1 regularization does not exist. To optimize this, we use gradient descent with a random initialization and solve it numerically.

Remark: Linear regression with L_2 regularization is also commonly called *Ridge regression*, and when L_1 regularization is employed, is commonly called *Lasso regression*. These regularizations can be applied to any Generalized Linear models just as above (by replacing $\log p(y|x, \theta)$ with the appropriate family likelihood). Regularization techniques of the above type are also called *weight decay*, and *shrinkage*. The Gaussian and Laplace priors encourage the parameter values to be closer to their mean (*i.e.,* zero), which results in the shrinkage effect.

Remark: Lasso regression (*i.e.,* L_1 regularization) is known to result in sparse parameters, where most of the parameter values are zero, with only some of them non-zero.

[4] Nakkiran, P., Venkat P., Kakade, S., Ma, T. Optimal Regularization Can Mitigate Double Descent. arXiv e-prints (Mar. 2020). arXiv:2003.01897

5. **[25 points] A Simple Neural Network**

Let $X = \{x^{(1)}, \cdots, x^{(n)}\}$ be a dataset of n samples with 2 features, i.e $x^{(i)} \in \mathbb{R}^2$. The samples are classified into 2 categories with labels $y^{(i)} \in \{0, 1\}$. A scatter plot of the dataset is shown in Figure 1:

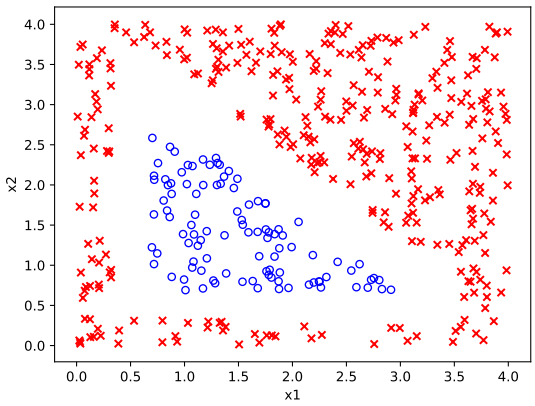

Figure 1: Plot of dataset X.

The examples in class 1 are marked as as "×" and examples in class 0 are marked as "○". We want to perform binary classification using a simple neural network with the architecture shown in Figure 2:

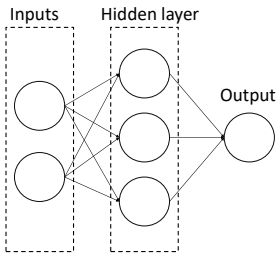

Figure 2: Architecture for our simple neural network.

Denote the two features x_1 and x_2, the three neurons in the hidden layer h_1, h_2, and h_3, and the output neuron as o. Let the weight from x_i to h_j be $w_{i,j}^{[1]}$ for $i \in \{1, 2\}, j \in \{1, 2, 3\}$, and the weight from h_j to o be $w_j^{[2]}$. Finally, denote the intercept weight for h_j as $w_{0,j}^{[1]}$, and the intercept weight for o as $w_0^{[2]}$. For the loss function, we'll use average squared loss instead of the usual negative log-likelihood:

$$l = \frac{1}{n} \sum_{i=1}^{n} \left(o^{(i)} - y^{(i)} \right)^2,$$

where $o^{(i)}$ is the result of the output neuron for example i.

(a) [5 points] Suppose we use the sigmoid function as the activation function for h_1, h_2, h_3 and o. What is the gradient descent update to $w_{1,2}^{[1]}$, assuming we use a learning rate of α? Your answer should be written in terms of $x^{(i)}$, $o^{(i)}$, $y^{(i)}$, and the weights.

(b) [10 points] Now, suppose instead of using the sigmoid function for the activation function for h_1, h_2, h_3 and o, we instead used the step function $f(x)$, defined as

$$f(x) = \begin{cases} 1, x > 0 \\ 0, x \leq 0 \end{cases}$$

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_step_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `step_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

Hint 1: There are three sides to a triangle, and there are three neurons in the hidden layer.

Hint 2: A solution can be found where all weight and bias parameters take values only in $\{-1, -0.5, 0, 1, 3, 4\}$. You are free to come up with other solutions as well.

(c) [10 points] Let the activation functions for h_1, h_2, h_3 be the linear function $f(x) = x$ and the activation function for o be the same step function as before.

Is it possible to have a set of weights that allow the neural network to classify this dataset with 100% accuracy?

If you believe it's possible, please implement your approach by completing the `optimal_linear_weights` method in `src/simple_nn/simple_nn.py` and including the corresponding `linear_weights.pdf` plot showing perfect prediction in your writeup.

If it is not possible, please explain your reasoning in the writeup.

Hint: The hints from the previous sub-question might or might not apply.

6. **[21 points] Batch Normalization**

Machine Learning Algorithms tend to give better results when input features are normalized and uncorrelated. For this reason, data normalization is a common pre-processing step in deep learning. However, the distribution of each layer's inputs may also vary over time and across layers, sometimes making training deep networks more difficult. *Batch Normalization* is a method that draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch.

Given a batch $\mathcal{B} = (x^{(1)}, ..., x^{(m)})$ of vectors in $\mathbb{R}^d$, we first normalize inputs into $(\hat{x}^{(1)}, ..., \hat{x}^{(m)})$ and then linearly transform them into $(y^{(1)}, ..., y^{(m)})$. A Batch Normalization layer also has parameters $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$. The layer works as follows:

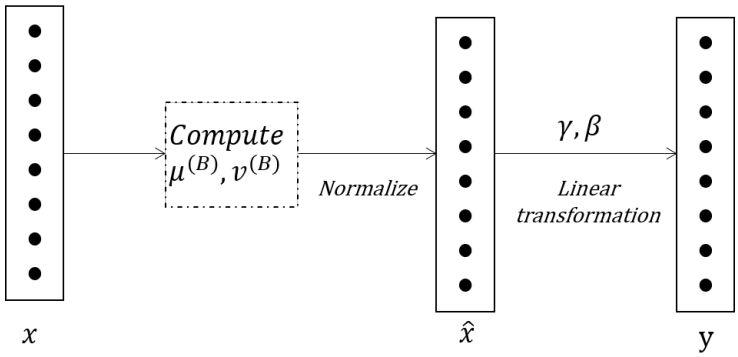

Figure 3: Batch Normalization

First, compute the batch mean vector

$$\mu^{(\mathcal{B})} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \in \mathbb{R}^d \tag{7}$$

Then, compute the batch variance vector $v^{(\mathcal{B})} \in \mathbb{R}^d$, which is defined element-wise for any $j \in \{1, ..., d\}$ by:

$$v_j^{(\mathcal{B})} = \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j^{(\mathcal{B})})^2 \in \mathbb{R} \tag{8}$$

Then, normalize the vectors in $\mathcal{B}$ by computing vectors $(\hat{x}^{(1)}, ..., \hat{x}^{(m)})$ such that $\hat{x}^{(i)} \in \mathbb{R}^d$ is defined element-wise for any $j \in \{1, ..., d\}$ by:

$$\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j^{(\mathcal{B})}}{\sqrt{v_j^{(\mathcal{B})}}} \tag{9}$$

Finally, output vectors $(y^{(1)}, ..., y^{(m)})$ in $\mathbb{R}^d$ defined by

$$y^{(i)} = \gamma \odot \hat{x}^{(i)} + \beta \tag{10}$$

where $\odot$ is the element-wise vector multiplication.

In this question, you will derive the back-propagation rules for batch-normalization. Let $\mathcal{L} = \mathcal{L}(y^{(1)}, \ldots, y^{(m)})$ be some scalar function of the batch-normalization output vectors. We will calculate the gradients of $\mathcal{L}$ with respect to the parameters and some intermediate variables.

(a) [6 points] Calculate the gradient of $\mathcal{L}$ w.r.t. β, γ, and $\hat{x}^{(i)}$ for $i \in \{1, ..., m\}$; i.e. calculate $\frac{\partial \mathcal{L}}{\partial \beta}$, $\frac{\partial \mathcal{L}}{\partial \gamma}$, and $\frac{\partial \mathcal{L}}{\partial \hat{x}^{(i)}}$ for $i \in \{1, ..., m\}$. Your answer can depend on $\frac{\partial \mathcal{L}}{\partial y^{(i)}}$ and the parameters and variables in the forward pass (such as $\hat{x}^{(i)}$'s). Your answer may be vectorized or un-vectorized (e.g. $\frac{\partial \mathcal{L}}{\partial \beta}$ or $\frac{\partial \mathcal{L}}{\partial \beta_j}$ for $j \in \{1, ..., d\}$).

(b) [3 points] Calculate the gradient of $\mathcal{L}$ w.r.t. $v^{(\mathcal{B})}$; i.e. calculate $\frac{\partial \mathcal{L}}{\partial v^{(\mathcal{B})}}$. Your answer can depend on $\frac{\partial \mathcal{L}}{\partial y^{(i)}}$ and the parameters and variables in the forward pass, as well as the gradients that you calculated in part (a). Your answer may be vectorized or un-vectorized ($\frac{\partial \mathcal{L}}{\partial v^{(\mathcal{B})}}$ or $\frac{\partial \mathcal{L}}{\partial v_j^{(\mathcal{B})}}$ for $j \in \{1, ..., d\}$).

(c) [6 points] Show that the gradient of $\mathcal{L}$ w.r.t. $\mu^{(\mathcal{B})}$ is

$$\frac{\partial \mathcal{L}}{\partial \mu^{(\mathcal{B})}} = \frac{-1}{\sqrt{v^{(\mathcal{B})}}} \odot \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial \hat{x}^{(i)}}$$

Please show all of your work. Note that $\frac{1}{\sqrt{v^{(\mathcal{B})}}}$ is the vector where the jth element is $\frac{1}{\sqrt{v_j^{(\mathcal{B})}}}$ (i.e. element-wise square root and reciprocal).

Hint: Consider applying the chain rule using $\hat{x}_j^{(i)}$ for $i \in \{1, ..., m\}$ as the intermediate variables, and note that $\hat{x}_j^{(i)}$ depends on $\mu_j^{(\mathcal{B})}$ directly through $\mu_j^{(\mathcal{B})}$ and indirectly through $v_j^{(\mathcal{B})}$.

(d) [6 points] Show that the gradient of $\mathcal{L}$ w.r.t. $x^{(i)}$ for $i \in \{1, ..., m\}$ is

$$\frac{\partial \mathcal{L}}{\partial x^{(i)}} = \frac{1}{\sqrt{v^{(\mathcal{B})}}} \odot \frac{\partial \mathcal{L}}{\partial \hat{x}^{(i)}} + \frac{2(x^{(i)} - \mu^{(\mathcal{B})})}{m} \odot \frac{\partial \mathcal{L}}{\partial v^{(\mathcal{B})}} + \frac{1}{m} \frac{\partial \mathcal{L}}{\partial \mu^{(\mathcal{B})}}$$

Please show all of your work. Note that $\frac{1}{\sqrt{v^{(\mathcal{B})}}}$ is the vector where the jth element is $\frac{1}{\sqrt{v_j^{(\mathcal{B})}}}$ (i.e. element-wise square root and reciprocal).

Hint: Consider applying the chain rule using $\hat{x}_j^{(k)}$ for $k \in \{1, ..., m\}$ as the intermediate variables, and note that $\hat{x}_j^{(k)}$ for $k \in \{1, ..., m\}$ depends on $x_j^{(i)}$ directly through $x_j^{(i)}$ for $k = i$ and indirectly through $\mu_j^{(\mathcal{B})}$ and $v_j^{(\mathcal{B})}$ for $k \in \{1, ..., m\}$.

Remarks on the broader context: After obtaining $\frac{\partial \mathcal{L}}{\partial x^{(i)}}$ (as a function of $\frac{\partial \mathcal{L}}{\partial y^{(i)}}$ and other quantities known in the forward pass), one can propagate the gradient backwards to other layers that generated the $x^{(i)}$'s (you are not asked to show this). Empirically, it turns out to be important to consider $\mu^{(\mathcal{B})}$ and $v^{(\mathcal{B})}$ as variables (instead of constants), so that in the chain rule, we consider the gradient through $\mu^{(\mathcal{B})}$ and $v^{(\mathcal{B})}$.