USC CSCI 467 Intro to Machine Learning	**Assignment 2** Due: March 2, 2023, 11:59pm	Spring 2023 Instructor: Robin Jia

This assignment has 4 questions, for a total of 100 points and 7 bonus points. Make sure you also download the `hw2.zip` file from the course website.

When submitting on Gradescope, note that **you must make a submission both for the written portion and programming portion.** For the programming portion, upload the files `regularization.py` and `neural.py` with your completed solution. (There is an "autograder" which will not actually grade your code but it will run it, and should return 0 if it encountered an error and 100 otherwise.) **Please still include the output of your code in the PDF report when requested in the problems.**

Question 1: Logistic Regression and Regularization (24 points)

In this problem, you will implement logistic regression with gradient descent, as well as L2 regularization. We will use the famous MNIST dataset, which has 28×28 images of handwritten digits that must be classified into the digits 0 through 9. For this problem, we will focus on a particular binary classification subtask—distinguishing 5's and 8's. 5's will be our "negative" class and 8's will be our "positive" class. In the last problem, you will do the full 10-way classification problem using neural networks.

(a) (3 points) Before you start, let's get familiar with the data! The folder `mnist_sample` contains a sample of 5's and 8's from the dataset. Browse the images and note three different ways in which examples within the same class differ. (Your observations may be specific to one of the two classes.)

(b) (8 points) Now, it's time to implement logistic regression. More specifically, your code should run gradient descent on the loss function

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} -\log \sigma \left(y^{(i)} \cdot w^\top x^{(i)} \right),$$

where n is the number of training examples and $(x^{(i)}, y^{(i)})$ is the i-th training example.[1] Here, each $x^{(i)}$ is simply the vector of pixels from the image, so it's a $28 \times 28 = 784$-dimensional vector.

Fill in the functions `predict()` and `train()`. As with HW1 Question 3, your code for `predict()` should use no for loops, and your code for `train()` should only use a single for loop. **You should use the `sigmoid()` function that's been imported from scipy**, as this will help avoid some numerical instability issues.

To test your code, run the following command:

```
python3 regularization.py
```

This will run logistic regression with the default settings, which uses no regularization and trains for $10,000$ steps. You should reach training set accuracy of 1.0 and development set accuracy of .9176.

[1] Note that we've added a $1/n$ factor compared to what I had in lecture; this is just to normalize the loss function to be independent of the dataset size. Otherwise, when we do regularization, we would have to choose the λ's to be of the same order of magnitude as n.

This content is protected and may not be shared, uploaded, or distributed.

(Note: This took about 45 seconds on my 3.5-year-old laptop. If you want to check the progress of training, you can print out the training loss every 100 steps, or use `tqdm` to print a progress bar—just use `tqdm(range(num_iters))` in your for loop.)

(c) (5 points) Now, let's add in L2 regularization. The full objective is now

$$L(w) = \frac{1}{n}\sum_{i=1}^{n} -\log\sigma\left(y^{(i)} \cdot w^\top x^{(i)}\right) + \frac{1}{2}\lambda\|w\|^2.$$

(Here we use a common version of L2 regularization that multiplies by 1/2; this will get canceled with the 2 when we take a derivative.)

Add to your implementation in `train()` to use the `l2_reg` parameter. If `l2_reg > 0`, then you should treat it as the λ in the above equation and apply L2 regularization.

You can pass in a value for L2 regularization on the command line with the `--l2 [lambda]` flag. Let's choose a good value of the hyperparmeter λ by trying different values and measuring accuracy on the development set. To help you out, for each value of λ we try, we've chosen a setting of learning rate (`-r` flag) and number of iterations (`-T` flag) that will converge to the minimum.[2] (In practice, you would treat learning rate and number of iterations as hyperparameters and try different values, but we'll just focus on choosing the λ hyperparameter here.)

Try the following and report the training and development accuracy for each:

- $\lambda = 1$: `--l2 1 -r 0.1 -T 1000`
- $\lambda = 0.1$: `--l2 0.1 -r 0.1 -T 2000`
- $\lambda = 10^{-2}$: `--l2 1e-2 -r 0.5 -T 2000`
- $\lambda = 10^{-3}$: `--l2 1e-3 -r 0.5 -T 5000`
- $\lambda = 10^{-4}$: `--l2 1e-4 -T 5000`

To help you debug: For $\lambda = 1$ you should get a train accuracy of 0.9143.

(d) (2 points) Out of the values we tried, which value of λ had the best development set accuracy? Report the test accuracy of the best λ by using the `--test` flag.

(e) (6 points) Finally, let's visualize the learned weights. Since we learn one weight for each pixel of the image, we can visualize the weights as an image!

Run the following commands:

```
python3 regularization.py --plot-weights no_reg
python3 regularization.py --l2 1e-2 -r 0.5 -T 2000 --plot-weights reg_1e-2
```

These will re-run training with no regularization and $\lambda = 10^{-2}$, respectively, and save a plot of the weights to `plot_no_reg.png` and `plot_reg_1e-2.png`.

Comment on the following:

- How do the images differ in appearance?
- For the image with L2 regularization, what does the image tell you about what the model is looking for? Think about what differentiates a 5 from an 8.
- How do the images differ in terms of the scale of the weights (notice the legend to the right). Why does L2 regularization cause this difference?

[2] In case you're wondering: adding L2 regularization to the objective generally makes it easier to optimize, because the problem becomes *strongly convex*. This makes gradient descent converge faster even with a low learning rate. With little or no regularization, we need a larger learning rate and more steps to get close to convergence.

This content is protected and may not be shared, uploaded, or distributed.

Question 2: Connecting Kernels and Features (25 points)

In class, we learned that kernels are motivated by two separate intuitions. First, kernels let us think about model predictions in terms of similarity of the test example to each training example. This is expressed through the kernel function $k(x, z)$, which measures how similar points x and z are. Second, kernels let us do efficient computations with very large, even infinite-dimensional, feature vectors. This is because any valid kernel can be written in the form

$$k(x, z) = \phi(x)^\top \phi(z)$$

for some function ϕ that takes in vectors in our original feature space and returns a (possibly infinite-dimensional) vector in a new feature space. Running a kernelized algorithm, such as kernel logistic regression, is mathematically equivalent to running the original algorithm in the feature space defined by ϕ (although the runtime is very different).

In this problem, you will show how two popular kernel functions—the quadratic kernel and radial basis function kernel—are equivalent to a particular choice of features ϕ. You will also get some hands-on experience working with the quadratic kernel on a toy dataset.

(Note: Parts (b) and (c) of this problem involve a fair bit of arithmetic. You are encouraged to write some simple code to do this arithmetic for you. Your solutions just need to describe what computation you did, and what the result is.)

(a) (5 points) For $x, z \in \mathbb{R}^d$, the quadratic kernel is defined as

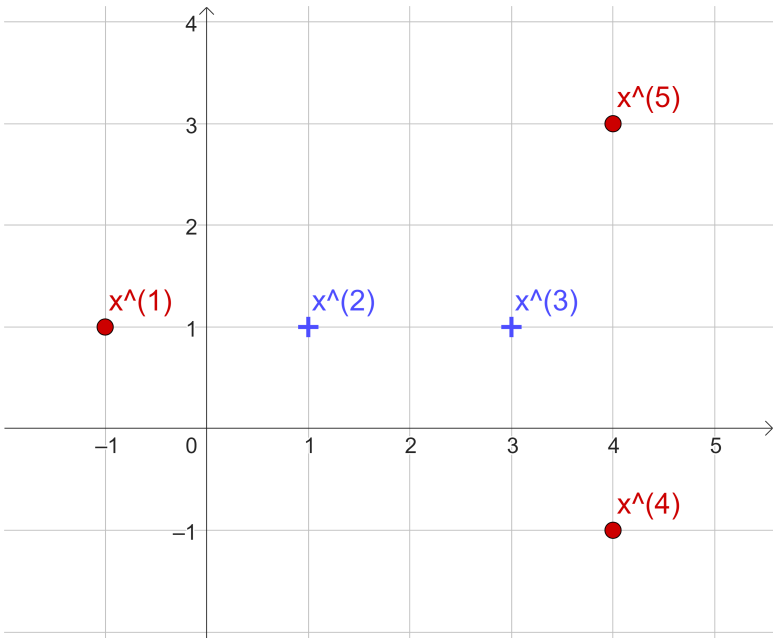
$$k(x, z) = (x^\top z + 1)^2.$$

Prove that when $d = 2$, $k(x, z) = \phi(x)^\top \phi(z)$ where we define

$$\phi(x) = \begin{pmatrix} 1 \\ \sqrt{2} \cdot x_1 \\ \sqrt{2} \cdot x_2 \\ x_1^2 \\ x_2^2 \\ \sqrt{2} \cdot x_1 x_2 \end{pmatrix}$$

(b) (5 points) Consider the following classification dataset of points in $\mathbb{R}^2$:

This content is protected and may not be shared, uploaded, or distributed.



The pluses are positive points; the circles are negative points.

Suppose we try to fit a linear classifier (e.g., standard logistic regression) that uses the feature function ϕ defined in part (a). We are thus looking for a weight vector $w \in \mathbb{R}^6$. Provide a possible value of w that would perfectly classify this data. (It does not have to be the actual value of w learned by logistic regression, although it can be.) Then, open this GeoGebra session https://www.geogebra.org/classic/ubfgwmbx and plot your decision boundary (you can just start writing an equation in the panel on the left and it should graph it). Include the image of your plot.

Hint: One way to perfectly separate the positive and negative points is with a decision boundary shaped like a circle. Recall that the equation for the region inside of a circle in the Cartesian plane of radius r centered at (x_0, y_0) is:

$$(x - x_0)^2 + (y - y_0)^2 \leq r^2.$$

Also note that for the purposes of plotting, x_1 is on the x-axis and x_2 is on the y-axis.

(c) Now suppose we run *kernel* logistic regression using the kernel described in part (a), on the dataset from part (b). This means we will learn a vector $a \in \mathbb{R}^5$, since there are 5 training examples. Suppose further that we learn the vector

$$a = \begin{pmatrix} -14 \\ 14 \\ 5 \\ -1 \\ -3 \end{pmatrix}$$

i. (2 points) The **kernel matrix** K is defined as the $n \times n$ matrix whose ij-th entry is $k(x^{(i)}, x^{(j)})$. Compute the 5×5 kernel matrix for this problem.

ii. (5 points) Using the kernel matrix you computed, the model's predicted score, i.e.

$$\sum_{i=1}^{n} a_i k\left(x, x^{(i)}\right)$$

This content is protected and may not be shared, uploaded, or distributed.

for each of the 5 training examples. Show that a perfectly classifies all 5 training examples. **For this part, do not convert a into the corresponding w vector.**

iii. (3 points) Now compute the w that corresponds to this value of a. Report w to 1 decimal place in each component. Using the same GeoGebra link from part (b), plot the decision boundary defined by this w, and include the picture here. What shape does it have?

(d) (5 points) In class, I said that the Radial Basis Function (RBF) kernel corresponds to an infinite-dimensional feature function. In this question, we will explicitly construct this infinite-dimensional feature function, for the specific case where our original inputs x are 1-dimensional.

For 1-dimensional inputs, the RBF kernel is defined as

$$k(x, z) = \exp\left(-\frac{(x-z)^2}{2\sigma^2}\right)$$

where σ is a hyperparameter (also called the "bandwidth"). A large σ means that two points are considered "similar" even if they're somewhat far away, whereas a small σ means that two are only considered "similar" if they're very close to each other.

Prove that $k(x, z) = \phi(x)^\top \phi(z)$ where we define

$$\phi(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right) \cdot \left[1, \frac{x}{\sigma\sqrt{1!}}, \frac{x^2}{\sigma^2\sqrt{2!}}, \frac{x^3}{\sigma^3\sqrt{3!}}, \dots\right].$$

Note that the left part is simply a scalar, and the right part is an infinite-dimensional(!) feature vector.

Hint: You will need to use the fact that

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Question 3: Thinking like a Neural Network (22 points)

In class we discussed the notion that a neural network with two layers is a *universal approximator*. That is, given any function, you can create a two-layer neural network to approximate it as well as you would like, as long as you can make the hidden layer as big as you'd like. In this problem, you will try your hand at approximating a few functions in this way.

All parts of this problem work the same way. You are given a function $f : \mathbb{R}^d \to \{0, 1\}$, as well as a domain $\mathcal{X} \subseteq \mathbb{R}^d$. You want to match f on the domain $\mathcal{X}$ with a two-layer neural network classifier $g_\theta(x)$ with parameters θ. More precisely, for every $x \in \mathcal{X}$, you want $g_\theta(x) > 0$ if $f(x) = 1$, and $g_\theta(x) < 0$ if $f(x) = 0$. In other words, g_θ should be a perfect binary classifier between points in $\mathcal{X}$ that have $f(x) = 1$ and points in $\mathcal{X}$ that have $f(x) = 0$. (Note that since you only have to be a good approximation for points in $\mathcal{X}$, this is easier than being a good approximation for every $x \in \mathbb{R}^d$.)

The parameters of your neural network are $\theta = (W, b, v, c)$ where $W \in \mathbb{R}^{h \times d}$, $b \in \mathbb{R}^h$, $v \in \mathbb{R}^h$, and $c \in \mathbb{R}$. h is the dimension of the hidden layer, which you can make as large as you would like. W and b are the weight matrix and bias vector for the first layer; v and c are the weight vector and bias for the second (final) layer. We will use the sigmoid activation function. Recall

This content is protected and may not be shared, uploaded, or distributed.

that $\sigma(z) \approx 1$ when z is large and ≈ 0 when z is very negative. Putting this all together, the neural network computes

$$g_\theta(x) = v^\top \sigma(Wx + b) + c.$$

In each part, you should:

1. State your choice of h, W, b, v, and c. You can use the notation w_i to denote the i-th row of W (and thus w_{ij} is the j-th element of the i-th row). For parts (c) and (d), you may write any of these in terms of d, the dimension of the input.
2. Prove that $g_\theta(x) > 0$ whenever $f(x) = 1$ and $g_\theta(x) < 0$ whenever $f(x) = 0$ for all $x \in \mathcal{X}$.

To make these proofs a bit easier, you may use the approximation that $\sigma(z) = 1$ if $z \geq 10$ and $\sigma(z) = 0$ if $z \leq -10$.[3] We recommend using the notation a_i to denote the value (or "activation") of the i-th hidden unit, i.e. $a_i = \sigma(w_i^\top x + b_i)$.

If you are not sure where to get started, it may help to study the lecture slides where I showed how to implement XOR with a neural network. The same ideas will be used in this problem. It may also be helpful to draw out a diagram of the network to visualize what it is doing.

(a) (7 points) $d = 3$, and $\mathcal{X}$ is the set of 3-dimensional vectors where each entry is an integer. $f(x)$ returns 1 if at least one entry is ≥ 4, and 0 otherwise.

(b) (7 points) $\mathcal{X}$ is the set of 3-dimensional vectors where each entry is a positive integer. $f(x)$ returns 1 if the numbers $\{x_1, x_2, x_3\}$ are valid side lengths of a (non-degenerate) triangle, and 0 otherwise.

Note: For 3 positive numbers to be possible side lengths of a non-degenerate triangle, it is necessary and sufficient for them to satisfy the triangle inequality, $a + b > c$, for all three combinations of the three numbers.[4] For example, $\{4, 7, 10\}$ works because

- $4 + 7 > 10$,
- $4 + 10 > 7$, and
- $7 + 10 > 4$,

whereas $\{4, 7, 12\}$ does not.

(c) (8 points) $\mathcal{X}$ is the set of d-dimensional vectors where each entry is an integer. $f(x)$ returns 1 if at every entry is between -3 and 3 (inclusive), and 0 otherwise. (In other words, f checks if x lies within or on the hypercube with side length 6 centered at the origin.)

(d) (7 points (bonus)) $\mathcal{X}$ is the set of d-dimensional vectors where each entry is either 0 or 1. $f(x)$ returns 1 if the number of 1's in x is odd, and 0 if the number of 1's in x is even. (In other words, f returns the XOR of the d entries in x.)

Question 4: Neural Networks for MNIST (29 points)

In this problem, we will again use the MNIST dataset we first saw in problem 1. Unlike in problem 1, we will do the full 10-way classification problem of identifying handwritten digits between 0 to 9. We will try a few different models: a linear model (softmax regression), a two-layer and three-layer neural network (MLP), and finally a convolutional neural network. This

[3] In fact, $\sigma(10) = 0.999955$ and $\sigma(-10) = 0.000045$

[4] In a "degenerate" triangle, you could have $a + b = c$; this means that the sides of the triangle are on top of each other, e.g. think about $\{1, 1, 2\}$.

This content is protected and may not be shared, uploaded, or distributed.

will all be done using the popular Pytorch library.[5] Pytorch syntax is reminiscent of numpy syntax, except what numpy calls "arrays," Pytorch calls "tensors."

You should be able to install pytorch by running

```
pip3 install -r requirements.txt
```

using the `requirements.txt` file included in the starter code. This should install torch version 1.13.1+cpu. Note that very slight differences are unfortunately still possible between different hardware.

Before you get started, take some time to read through the comments so you understand how the code works. In particular, I have already implemented the training loop itself for you—this is generic code for training models in Pytorch in general. The code runs stochastic gradient descent for a fixed number of epochs (30 by default), and evaluates the training and dev accuracy at the end of each epoch. The code also does a simple version of early stopping—it keeps the model parameters from the epoch that had the highest dev accuracy, and sets those as the final model parameters.

(a) (0 points) The `SoftmaxRegression` class has already been implemented for you as an example. Note that implementing any pytorch model requires implementing two methods:

- `__init__(self)`: This initializes the model. Here, you will define all the components (i.e., "layers") of the model. In this case, we only need a single linear layer, which you create by calling `nn.Linear()`. A linear layer stores its own parameter weight matrix and bias vector, which will get updated during training.
- `forward(self, x)`: This is the code for the model's forward pass—how it goes from input x to the output. In this case, the only thing we need to do is apply the linear layer to x. Note that the actual softmax loss function is computed in the training loop (by `nn.CrossEntropyLoss`), so `forward()` only needs to compute the pre-softmax scores for each class, also called the "logits."

Train this linear model by running:

```
python3 neural.py linear
```

You should get a dev accuracy of .90871, which comes from checkpoint 23. It's a bit better than using the final checkpoint 29. If you don't get these results, you may be on a different Pytorch version.

(b) (5 points) Now implement `TwoLayerMLP`. Your model should do the following:

- Apply a linear layer that maps the input to a vector of size `hidden_dim` (this is h from lecture).
- Apply the ReLU nonlinearity to this vector.
- Apply a second linear layer that maps vectors of size `hidden_dim` to vectors of size `NUM_CLASSES`.

A couple things to note:

- When you create a `nn.Linear()` object, it automatically randomly initializes the parameters for you, so you don't have to worry about doing the initialization yourself.

[5] https://pytorch.org/

This content is protected and may not be shared, uploaded, or distributed.

- You should use the torch function `F.relu()`, which comes from the `torch.nn.functional` module. In general, `torch.nn.functional`[6] implements many useful functions, including other nonlinearities like sigmoid and tanh, as well as operators like max pooling.

When you're done, run your 2-layer MLP with:

```
python3 neural.py mlp2
```

After epoch 0, your dev accuracy should be .88704. **Report the best epoch, and the train and dev accuracy you got at that epoch.**

(c) (3 points) Now let's see how the size of the hidden layer h affects the model. Note that the hidden layer can be as big or as small as we'd like. The code uses $h = 200$ by default, but you can change this by using the `-i [number]` flag. Try $h = 10, 20, 50, 100, 200, 500$. Report the following:

- The dev accuracy from the best checkpoint (i.e., from the best epoch) for each value of h.
- How does dev accuracy change as you increase h?
- How does the runtime of the model change as you increase h?

(d) (4 points) Now, let's try regularizing using Dropout. The `__init__()` method for `TwoLayerMLP` accepts an option called `dropout_prob` which by default is 0, but can be any probability between 0 and 1. This represents the probability of any neuron getting "dropped out" (i.e., set to 0) during training. Implement Dropout in `TwoLayerMLP`. In particular, you should apply Dropout to the hidden layer, after the ReLU. You should create a `nn.Dropout()` object in `__init__()`, then use it in `forward()`.[7]

Once you're ready, train the model with $h = 200$ and dropout of 0.5 by running the following:

```
python3 neural.py mlp2 -p 0.5
```

Report the best epoch, and the train and dev accuracy you got at that epoch. How does the final train accuracy and final dev accuracy compare with the model trained without dropout?

(e) (6 points) Now let's try adding another hidden layer. Implement `ThreeLayerMLP` with dropout. This will be similar to `TwoLayerMLP` except we have now have a second hidden layer, which takes in the input of the first hidden layer and produces a new set of hidden units by applying another linear mapping followed by ReLU. The second layer is then fed to the final layer to produce the logits. You should add dropout on both hidden layers.

When you're done, run your 3-layer MLP with dropout 0.2:[8]

```
python3 neural.py mlp3 -p 0.2
```

After epoch 0, your dev accuracy should be .87495. **Report the best epoch, and the train and dev accuracy you got at that epoch.** How does this compare with the 2-layer MLP with dropout in terms of both dev accuracy and runtime?

[6] https://pytorch.org/docs/stable/nn.functional.html

[7] https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

[8] I found that lower dropout works a bit better for 3-layer MLP; this is the sort of thing you need to find out by trying different values for the dropout probability.

This content is protected and may not be shared, uploaded, or distributed.

(f) (3 points) Finally, we will implement a convolutional neural network (CNN), which is specially designed for images. Here, we will think of the input as a $1 \times 28 \times 28$ tensor (whereas previously we thought of the input as a 784-dimensional vector). The "1" denotes the number of input channels. Since we have black/white images, there is just one input channel; if we had color inputs, we would have 3 input channels for red, green, and blue. Then 28×28 is the width and height of the image.

You will use the following architecture:

- Apply a convolutional layer with kernel size 3 and producing 5 output channels (called `num_channels` in the code).
- Apply ReLu to this.
- Apply Max pooling with a kernel size of 2.
- Now "flatten" the resulting tensor into a single vector.
- Apply a single hidden layer to this vector, mapping it to a vector of size `hidden_dim`. This consists of a linear layer followed by ReLU
- Apply dropout to the output of this hidden layer
- Finally, apply a final linear layer to map the hidden units to logits.

Before we implement this, we should work out what dimension everything is. Answer the following, assuming the input is a single $1 \times 28 \times 28$ tensor:

- What is the shape of the output of the convolutional layer? Assume that the order of the dimensions is `(output_channels, width, height)`.
- What is the shape of the output of max pooling?
- What is the input dimension of the hidden layer?

(g) (8 points) Now, go ahead and implement the architecture described above in the `ConvNet` class. You will need to use the following:

- `nn.Conv2d`: A 2d convolutional layer.[9] Like `nn.Linear`, it stores its own parameters.
- `nn.MaxPool2d`: A Max-pooling layer.[10]
- The `.reshape()` method, which allows you to take a tensor and reshape it into a different tensor with the same total number of numbers. You can see that the starter code takes the original input x, whose size is $B \times 784$ (where B is batch size), and reshapes it into a tensor whose shape is $B \times 1 \times 28 \times 28$, which is suitable for being the input to a `Conv2d` layer. You will also need to do a `.reshape()` in `forward()` before the hidden layer.

When you're ready, run your CNN model with dropout of 0.2:

```
python3 neural.py cnn -p 0.2
```

After epoch 0, your dev accuracy should be .90621.

Report the best epoch, and the train and dev accuracy you got at that epoch. Finally, since this should perform the best out of all models, report the test accuracy by running the same command with `--test`.

[9] https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html
[10] https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html

This content is protected and may not be shared, uploaded, or distributed.