Homework 1

[1]

CMU 10-417/617: Intermediate Deep Learning (Fall 2022)

https://rsalakhucmu.github.io/10417-22/
OUT: Sep 14, 2022

DUE: Oct 3, 2022
TAs: Sedrick Keh, Olivier Filion, Tarun Chiruvolu, Michael Agaby

START HERE: Instructions

Homework 1 covers topics on regression, classification, backpropagation and neural networks basics. The homework includes short answer questions, derivation questions, and a coding task.

- **Collaboration policy:** Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 2.1"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the Academic Integrity Section on the course site for more information: https://rsalakhucmu.github.io/10417-22/#policies
- **Late Submission Policy:** See the late submission policy here: https://rsalakhucmu.github.io/10417-22/#policies
- **Submitting your work:**
  - **Written:** For both the programming portion and the written problems, we will be using Gradescope (https://gradescope.com/). For the programming portion, please submit your filled out "mlp.py" file to Gradescope under the assignment name "Homework 1 Programming", and please submit your writeup to "Homework 1 Written". Please write your solution in the LaTeX files provided in the assignment and submit in a PDF form. Put your answers in the question boxes (between `\begin{soln}` and `\end{soln}`) below each problem. Please make sure you complete your answers within the given size of the question boxes. **Handwritten solutions are not accepted and will receive zero credit.** Regrade requests can be made, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are

[1] Compiled on Wednesday 14th September, 2022 at 01:15

found then points will be deducted. For more information about how to submit your assignment, see the following tutorial (note that even though the assignment in the tutorial is handwritten, submissions must be typed): https://www.youtube.com/watch?v=KMPoby5g_nE&feature=youtu.be

– **Code:** All code must be submitted to Gradescope. **If you do not submit your code to Gradescope, you will not receive any credit for your assignment.** There is no limit on the number of submissions that can be made to Gradescope.

Problem 1 (7 pts): l_1 loss as MLE under Laplace noise

Consider data generated by a linear model with the addition of Laplace noise. Namely, the labels $y(\mathbf{x}, \mathbf{w})$ are generated as

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \mathbf{x} + \varepsilon \tag{1}$$

where ε is a random variable sampled from a Laplace distribution with parameter b, i.e. $\text{Lap}(0, b)$. Equivalently, we can write $y \sim \text{Lap}(\mathbf{w}^\top \mathbf{x}, b)$, for which the probability density function is

$$p(y; \mathbf{w}^\top \mathbf{x}, b) = \frac{1}{2b} e^{-\frac{|y - \mathbf{w}^\top \mathbf{x}|}{b}} \tag{2}$$

1. (5 pts) Show that, given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, the maximum likelihood estimate for the above model is

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} |y_i - \mathbf{w}^\top \mathbf{x}_i| \tag{3}$$

Thus, the l_1 loss can be recovered as a maximum-likelihood estimation problem under Laplace noise.

Solution

2. (2 pts) Describe, in 1-2 sentences, a situation where the l_1 loss would be preferable over the l_2 loss (mean-squared error loss). Remember that the l_2 loss objective for a linear model is defined as

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} \left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 \tag{4}$$

Solution

Problem 2 (10 pts): Neural network intuition

This question's purpose is to help you build some intuition for neural networks.

Part 2.1 (7 pts)

1. (3 pts) Consider an n-layer neural network written as a function of x as follows:

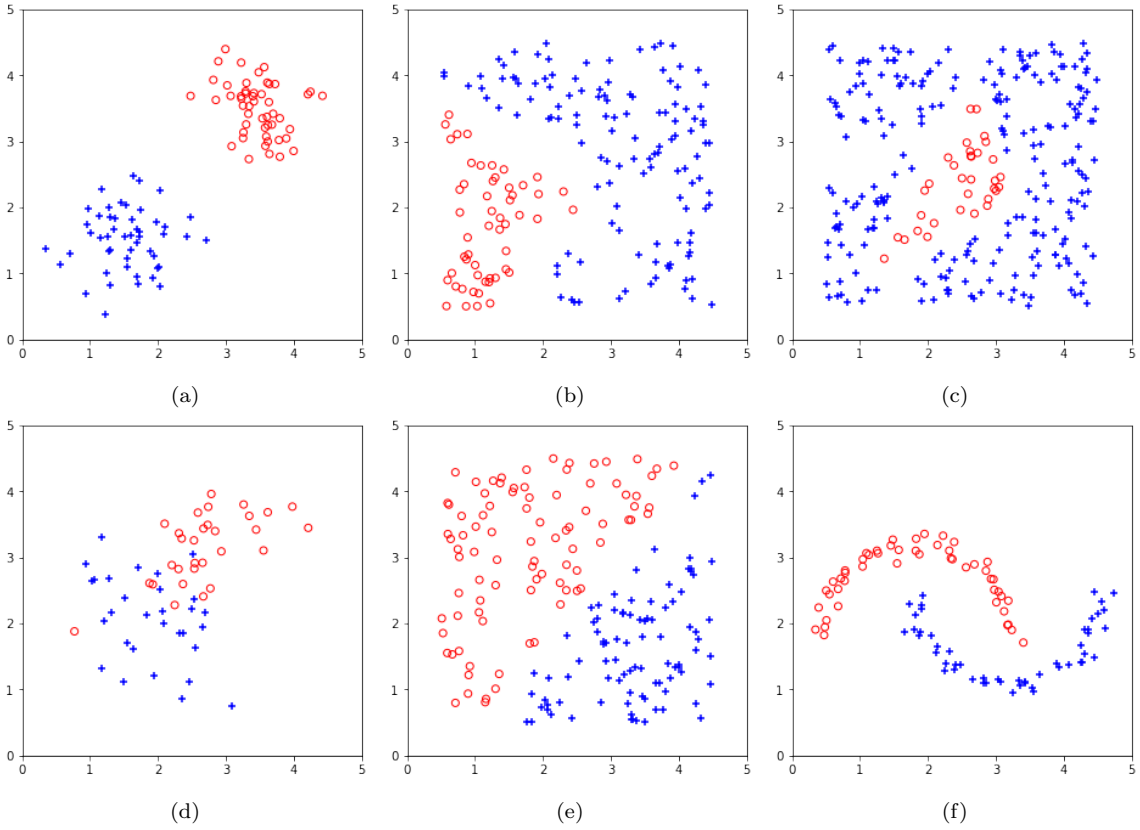
$$f(x) = \phi_n(W_n^T \phi_{n-1}(W_{n-1}^T ... \phi_2(W_2^T \phi_1(W_1^T x + b_1) + b_2) ... + b_{n-1}) + b_n) \tag{5}$$

where W_i are the weight matrices, b_i are the biases, and ϕ_i are the activation functions.

Show that **linear** activation functions (that is, $\phi_i(x) = \alpha_i x + \beta_i$) give rise to a **linear map** f – that is, show that $f(x) = W^T x + b$ for some matrix W and vector b.

Solution

Once you have done this, consider these 6 datasets with data points in $\mathbb{R}^2$ belonging to two classes denoted by red circles ($\circ$) and blue crosses ($+$).



(a) (b) (c)

(d) (e) (f)

For questions 2-5, we look at the power of different neural network architectures with sigmoid output layers. For each architecture, give all the datasets from the above that it can perfectly classify (zero misclassification error on this particular dataset).

2. (1 pt) Feed-forward neural network with 100 hidden layers, each containing 100 nodes, with linear activation functions

Solution

3. (1 pt) Feed-forward neural network with 1 hidden layer containing 2 nodes and a sigmoid activation function

Solution

4. (1 pt) Feed-forward neural network with 1 hidden layer containing 3 nodes and a sigmoid activation function

Solution

5. (1 pt) Feed-forward neural network with 1 hidden layer with sigmoid activation function that can contain any number of nodes

Solution

Part 2.2 (2 pts)

As you know, when initializing a neural network, it is common practice to initialize the weight matrices with values from a random number generator. Here, we will explore why we don't just initialize to 0.

For simplicity, we will consider an arbitrary layer of a fully-connected neural network with ReLU activations, namely:

$$\mathbf{a}^{(n)} = \text{ReLU}(\mathbf{W}^\top \mathbf{a}^{(n-1)} + \mathbf{b}) \tag{6}$$

for $n \in \mathbb{N}$ and an arbitrary loss function, L.

Let $\mathbf{z}^{(n)} = \mathbf{W}^\top \mathbf{a}^{(n-1)} + \mathbf{b}$ and assume that $\frac{\partial L}{\partial \mathbf{z}^{(n)}}$ is a vector. Show that if all elements of $\mathbf{W}$ and $\mathbf{b}$ are 0, then all the elements of $\frac{\partial L}{\partial \mathbf{W}}$ are also 0.

Solution

Part 2.3 (1 pt)

Considering the network described in part 2.2, if we keep all the weights of the network fixed as $\mathbf{W}^{(n)} = \mathbf{0}$ and only train the biases, what would go wrong? (2-3 sentences)

Solution

Problem 3 (9 pts): Activation functions

The purpose of this question is to learn about the similarities and differences between three important activation functions.

In recent years, the Rectified Linear Unit (ReLU) activation function has become the non-linearity of choice for deep learning practitioners:

$$\text{ReLU}(x) = \max\{x, 0\} \tag{7}$$

However, in the early days of deep learning, the sigmoid function (σ) was the most frequently used activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{8}$$

In addition, the hyperbolic tangent function (tanh) is often used as an activation function for neural networks:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{9}$$

We will now explore some properties of the sigmoid and tanh activation functions.

1. (2 pts) One advantage of the sigmoid function is that it has a derivative that makes back-propagation very simple. Show that

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{10}$$

Solution

2. (1 pt) Unfortunately, the sigmoid function suffers from the **vanishing gradient problem**, which is part of the reason it has fallen out of favor. Show that

$$\lim_{x \to \infty} \sigma'(x) = \lim_{x \to -\infty} \sigma'(x) = 0 \tag{11}$$

Solution

3. (1 pt) In 1-2 sentences, why do you think the vanishing gradient problem would be an issue?

Solution

4. (2 pts) The tanh function is closely related to the sigmoid function. Show that

$$\tanh(x) = 2\sigma(2x) - 1 \tag{12}$$

Solution

5. (2 pts) Consider the following two-layer neural network that uses the sigmoid function as a hidden activation function

$$y_k = w_{k0}^{(2)} + \sum_{j=1}^{M} w_{kj}^{(2)} \sigma \left(\sum_{i=1}^{M} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) \tag{13}$$

and the following two-layer neural network that uses, instead, the tanh function

$$y'_k = u_{k0}^{(2)} + \sum_{j=1}^{M} u_{kj}^{(2)} \tanh \left(\sum_{i=1}^{M} u_{ji}^{(1)} x_i + u_{j0}^{(1)} \right) \tag{14}$$

Using the relationship between the activation function derived in part 4, find expressions for $u_{kj}^{(2)}$, $u_{k0}^{(2)}$, $u_{ji}^{(1)}$, and $u_{j0}^{(1)}$ such that two networks are equivalent.

Solution

6. (1 pt) Does the tanh activation function solve the vanishing gradient problem? Explain, in 2-3 sentences, why or why not.

Solution

Problem 4 (14 pts): Back-propagation

Introduction and Notation

In this question, you will derive the necessary back-propagation operations for an efficient implementation of a feed-forward neural network for classification in Problem 6. Remember that the back-propagation algorithm calculates the gradient of each of the network's parameters to determine by how much to change them to achieve a better loss.

Let $f(x_1, x_2, x_3, \ldots, x_n) = f(\mathbf{x})$ be a scalar output function of multiple scalar inputs, or a scalar output function of a single vector input. Recall the operator ∇, defined as

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \cdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{15}$$

In this homework, we will abuse the notation and extend ∇. First let W be a $r \times c$ matrix and $g(W)$ be a scalar output function. Define

$$\nabla_W[g] = \begin{bmatrix} \frac{\partial g}{\partial W_{11}} & \cdots & \frac{\partial g}{\partial W_{1c}} \\ \cdots & & \cdots \\ \frac{\partial g}{\partial W_{r1}} & \cdots & \frac{\partial g}{\partial W_{rc}} \end{bmatrix} \tag{16}$$

(Note, this is not the Hessian, this is just a way to write and refer to each of the partial derivatives.) In addition, suppose $h(\mathbf{x}, \mathbf{y}, W)$ is a scalar function of vectors $\mathbf{x}$,$\mathbf{y}$, and a matrix W. Define

$$\nabla_{\mathbf{x}}[h] = \begin{bmatrix} \frac{\partial h}{\partial x_1} \\ \cdots \\ \frac{\partial h}{\partial x_n} \end{bmatrix} \tag{17}$$

and similarly for $\nabla_{\mathbf{y}}[h]$ and $\nabla_W[h]$.

With these constructs at hand, let us derive back-propagation for a one hidden layer neural network with a softmax output and cross-entropy loss function. Let column vectors $\mathbf{x} \in \mathbb{R}^D$ be a data-point and $\mathbf{y} \in \mathbb{R}^M$ be a one-hot encoding of the the corresponding label. Consider the neural network defined by the following equations.

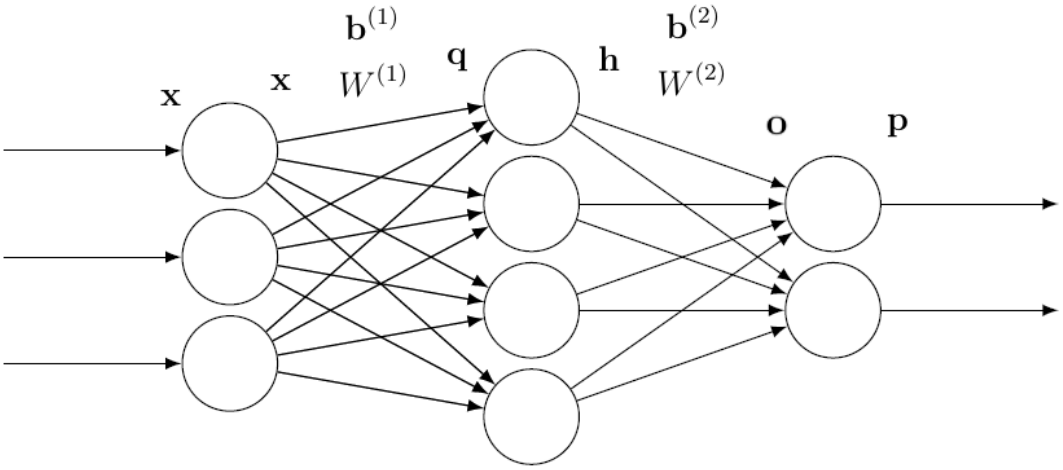


Figure 2: One layer fully connected neural network

$$\mathbf{q} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{18}$$

$$\mathbf{h} = \mathrm{ReLU}(\mathbf{q}) = \max(0, \mathbf{q}) \qquad \text{which is applied element-wise} \tag{19}$$

$$\mathbf{o} = W^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \tag{20}$$

$$\mathbf{p} = \mathrm{softmax}(\mathbf{o}) \qquad \text{which is defined as } p_i = \frac{e^{o_i}}{\sum_{k=1}^{M} e^{o_k}} \tag{21}$$

$$L(\mathbf{p}, \mathbf{y}) = -\sum_{i=1}^{M} y_i \log(p_i) \tag{22}$$

Note that $W^{(1)} \in \mathbb{R}^{H \times D}$, $\mathbf{b}^{(1)} \in \mathbb{R}^{H}$, $W^{(2)} \in \mathbb{R}^{M \times H}$ and $\mathbf{b}^{(2)} \in \mathbb{R}^{M}$.

Our ultimate goal is to calculate the gradients of the loss function with respect to the parameters $W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)}$.

Part 4.1 (5 pts)

In these sections, you may find it helpful to use the Kronecker delta (https://en.wikipedia.org/wiki/Kronecker_delta) as a shorthand. First, derive each of the following using chain rule:

$$\frac{\partial p_i}{\partial o_j}, \frac{\partial L}{\partial o_j}, \frac{\partial o_i}{\partial b_j^{(2)}}, \frac{\partial L}{\partial b_j^{(2)}} \tag{23}$$

Solution

Then, show that (by showing each element of the vectors are equal on both sides)

$$\nabla_{\mathbf{o}}[L] = \mathbf{p} - \mathbf{y} \tag{24}$$

$$\nabla_{\mathbf{b}^{(2)}}[L] = \mathbf{p} - \mathbf{y} \tag{25}$$

Solution

Part 4.2 (3 pts)

Derive the following using chain rule

$$\frac{\partial o_i}{\partial h_j}, \frac{\partial L}{\partial h_j} \tag{26}$$

Solution

Then, show that (by showing each element of the vectors are equal on both sides)

$$\nabla_{\mathbf{h}}[L] = W^{(2),\top} \nabla_{\mathbf{o}}[L] \tag{27}$$

Note $W^{(2),\top}$ is the transpose of $W^{(2)}$

Solution

Part 4.3 (3 pts)

Derive the following using chain rule

$$\frac{\partial o_k}{\partial W_{ij}^{(2)}}, \quad \frac{\partial L}{\partial W_{ij}^{(2)}} \tag{28}$$

Solution

Then, show that (by showing each element of the matrices are equal on both sides)

$$\nabla_{W^{(2)}}[L] = \nabla_{\mathbf{o}}[L]\mathbf{h}^\top \tag{29}$$

Solution

Part 4.4 (3 pts)

Derive the following using chain rule. The second one should be in terms of $\frac{\partial L}{\partial h_i}$

$$\frac{\partial h_i}{\partial q_j}, \frac{\partial L}{\partial q_j} \tag{30}$$

Solution

With these expressions at hand, you should be equipped to implement Problem 6 efficiently. The derivative of $\mathbf{q}$ with respect to $\mathbf{x}$, $W^{(1)}$ and $\mathbf{b}^{(1)}$ follows in the same way as $\mathbf{o}$ with respect to $\mathbf{h}$, $W^{(2)}$ and $\mathbf{b}^{(2)}$.

Problem 5 (10 pts) *** 10-617 STUDENTS ONLY ***: L_2 regularization with dropout

NOTE: This problem is required for students enrolled in the graduate version (10-617) of the course only. Students enrolled in the undergraduate version (10-417) of the course may attempt the question if they wish but will not receive any credit for it (no bonus points). If you are currently in 10-417 and considering switching to 10-617 (more info on how to do that on the course website), you should attempt the problem. If you do not attempt the problem, please do **not** delete it because Gradescope won't accept any submission that is missing pages from the template.

Consider a dataset $\mathcal{D}$ of N training points $\left(\mathbf{x}^{(n)}, y^{(n)}\right)$ where $\mathbf{x}^{(n)} \in \mathbb{R}^D$ and $y^{(n)} \in \mathbb{R}$ for all $n \in \{1, \ldots, N\}$. For this question it will be easier to adopt a matrix notation: let $X \in \mathbb{R}^{N \times D}$ be the usual design matrix containing data points as rows, and $\mathbf{y} \in \mathbb{R}^N$ the target vector. We will look at a linear model of the form

$$f_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{D} w_i x_i = \mathbf{w}^\top \mathbf{x} \tag{31}$$

with the sum-of-squares loss written compactly as

$$L(\mathbf{w}) = \|\mathbf{y} - X\mathbf{w}\|^2 \tag{32}$$

Recall the dropout scheme seen in class: any input dimension is retained with probability p and the input can be expressed as $R \odot X$ where $R \in \{0, 1\}^{N \times D}$ is a random matrix with $R_{ij} \sim \text{Bernoulli}(p)$ and $\odot$ denotes an element-wise product. Marginalizing the noise, our loss function becomes

$$L_{\text{dropout}}(\mathbf{w}) = \mathbb{E}_R \left[\|\mathbf{y} - (R \odot X)\mathbf{w}\|^2\right] \tag{33}$$

Part 5.1 (3 pts)

Let $N = D = 1$, so that X and $\mathbf{y}$ are just scalar values, and thus $\mathbf{w}$ is also scalar. Show that dropout with linear regression is equivalent in expectation to a certain form of ridge regression. More specifically, you should show that

$$L_{\text{dropout}}(\mathbf{w}) = \|\mathbf{y} - pX\mathbf{w}\|^2 + p(1-p)\|X\mathbf{w}\|^2 \tag{34}$$

Solution

Part 5.2 (7 pts)

Show the same statement, but for arbitrary values of N and D. That is, show that

$$L_{\text{dropout}}(\mathbf{w}) = \|\mathbf{y} - pX\mathbf{w}\|^2 + p(1-p)\|\Gamma\mathbf{w}\|^2 \tag{35}$$

where $\Gamma = (\text{diag}(X^T X))^{1/2}$ ($\text{diag}(A)$ for a square matrix A is the square matrix containing the diagonal entries of A on its diagonal and zeros in any non-diagonal entries).

Hint: Try proving the case for $N = 1$ and arbitrary values of D, and extend that to datasets of N points.

Solution

Problem 6 (60 pts): Programming

For this question you will write your own implementation of the foward and backward path for some core layers in neural networks, and use them to build a trainable network. After implementing the components, you will use them to solve a concrete task. Please do not use any toolboxes except those already imported in the template code.

Warning: It takes multiple hours to train all of the networks. Please start early and leave ample time for training/debugging.

Part 1: Implementation (24 pts)

In this part, you will first implement several classes of important neural network modules, and then use these modules to build up the networks. For this part, we provide a template code. Please FOLLOW the templates, and implement the classes methods. Do NOT change the interface. The classes will be used for auto-grading. Code that does not pass the autograder WILL NOT be graded.

Hint: Be sure to vectorize all of your computations to ensure they can run fast enough (so make computations in the form of vector and matrix multiplications as much as possible instead of for loops).

1.1 Introduction to the Codebase

In the code we provide, you can find three files: **mlp.py** and **test.py** and **test.pk**. (Please use **Python 3.6** or above, and the newest version of numpy.)

mlp.py is the file you need to implement. In this file, there is a base class called *Transform*. A *Transform* class represents a (one input, one output) function done to some input x. In symbolic terms, if f represents the transformation, *out* is the output, x is the input, $out = f(x)$. The forward operation is called via the forward function. The derivative of the operation is computed by calling the backward function. The layers in neural networks can be represented by inheriting this *Transform* class. And in each class, you will need to implement *forward* and the *backward* (and possibly the *step* and *zerograd*) function as instructed.

test.py and **tests.pk** contain some unit tests we provide to you. These files do not need any modification, and you can choose to use them or not. You can use these to test your implementations with these commands:

To run one test: `python -m unittest tests.TestLinearMap`
To run all tests: `python -m unittest tests`

Note that passing the tests does not necessarily mean that you will get full mark in auto-grading, while failing the tests almost surely indicate that you will fail the auto-grading.

1.2 Basic Layers

There are several basic layers to implement:

- **LinearMap**: This is the linear transformation layer, i.e. fully-connected layers. Please implement the *forward*, *backward* and *step* of it. You can use tests.TestLinearMap to test your implementation.
- **ReLU**: This is the layer of ReLU function, a popular kind of activation function. Please implement the *forward* and *backward* function. You can use tests.TestReLU to test your implementation.
- **SoftmaxCrossEntropyLoss**: This is the layer of softmax and cross-entropy loss. The inputs are the pre-softmax logits, and the forward output is the **mean** cross-entropy loss across samples in a batch. You can use tests.TestLoss to test your implementation.

1.3 Momentum

For more stable and faster training, in deep learning we can use momentum on the gradient. Instead of directly update with gradient G, we update it with momentum G_M. G_M is initialized as 0, and then updated with:

$$G_M^{new} = \alpha * G_M^{old} + G$$

where α is the coefficient controlling stability of descent direction. In **LinearMap**, please implement momentum. After, you can use tests.TestMomentum to test your implementation.

1.4 Dropout

Dropout is another common trick we use in deep learning to avoid overfitting. Dropout can be viewed as following: During training, for each neuron, there is a probability p of masking out it to 0. During inference, there is no masking out, but instead, we multiply the neuron values by $1 - p$ to be consistent with the expectation of the training phase.

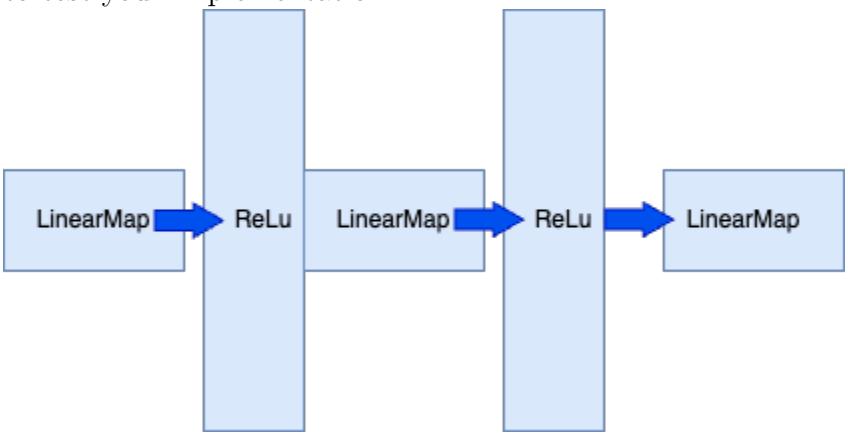
In **ReLU**, there is a dropout chance, which represent the dropout probability p. The default value is 0, meaning no dropout is performed. You need to consider this in your implementation. In the dropout, please use np.random.uniform just once, and only call the random function when train=True, and follow the inline comments instructions. You can use tests.TestDropout to test your implementation.

1.5 Networks to Implement

There are several networks to implement:

- **SingleLayerMLP**: This is a neural network with one hidden layer, and uses ReLU as activation. The output is the logits(pre-softmax) for the classification tasks.
- **TwoLayerMLP**: This is a neural network with two hidden layers, and uses ReLU as activation. The output is the logits(pre-softmax) for the classification tasks.

After implementing the networks, you can use tests.TestSingleLayerMLP, and tests.TestTwoLayerMLP to test your implementation.



This is what the two layer network would look like.

Note that a call to SoftmaxCrossEntropyLoss does not belong in the implementations of the networks, but in the implementation of you training loop in Part 2 below.

On Gradescope, there's a unit test worth 0 points that runs your two layer network under a train loop for 10 epochs. If there is a mismatch in the output with the reference solution, the test will tell you. If the unit test gives no output, then you implemented the code in mlp.py correctly and only have to worry about bugs in your train/test implementation (Part 2 below).

Part 2, Experiments (36 pts)

The objective for your networks is to predict which alphabet system a given character belongs to. The input will be a character from the Omniglot dataset, flattened as a vector of length 105*105. The target will be a integer from 0 to 13, representing 14 different classes of alphabets in the Omniglot dataset. There are 6608 training cases, and 1652 test cases.

How to get the data: On Piazza, under the Resources section, there is a file called `omniglot_14.pkl.tar`. Please download this file and untar it. The resulting file will be `omniglot_14.pkl`, which is a Python pickle file. To unpickle and read the file's contents, use the following snippet.

```
import pickle as pk
with open('omniglot_14.pkl','rb') as f:  data = pk.load(f)
((trainX,trainY),(testX,testY)) = data
```

trainX is a Numpy array of dimension (6608, 105*105) representing the flattened images. trainY is a Numpy array of dimension (6608,), representing the label, which is a number from 0 to 13 representing which alphabet the Omniglot character belongs to.

testX and testY contain 1652 datapoints and use the same format as above ((1652, 105*105) for testX and (1652,) for testY). As a warm up question (not graded), load the data and plot a few examples. Decide if the pixels were scanned out in row-major or column-major order.

Train and Test (12 points each): In this part, there is no template for you. You can implement the training and testing as you like, but please use minibatch SGD (common batch sizes include [16, 32, 64, 128]. Remember to shuffle your training dataset before each epoch for minibatch SGD.

This part will also not be auto-graded: we will be seeing your train and test curves and reading your analysis. For this part, use the classes you implemented to train and test on the dataset we provided, and include the results in the writeup.

For each plot in the following questions, please clearly annotate the axis.

2.1 Single Layer (12 pts)

Please train a network to predict the alphabet for the omniglot character with following settings:

(a) Single hidden layer with 50 nodes, momentum 0, dropout rate 0, learning rate 0.001.

(b) Single hidden layer with 50 nodes, momentum 0.4, dropout rate 0, learning rate 0.001.

(c) Single hidden layer with 50 nodes, momentum 0.4, dropout rate 0.2, learning rate 0.001.

(d) Single hidden layer with 120 nodes, momentum 0.4, dropout rate 0.2, learning rate 0.001

Report the final test accuracy after 150 epochs and create the following two plots:

1. The Train and Test curves for loss (on the same plot) over 150 epochs
2. The Train and Test curves for accuracy (on the same plot) over 150 epochs

Clearly label the train curves and the test curves on both plots and label the axes. Your loss and accuracy plots should each have 8 curves plotted (corresponding to train/test for each of the 4 experiments above).

Compare the results, and write 1-2 sentence on the effect of momentum and dropout.

Solution

2.2 Two Layers (12 pts)

Please train a network to predict the alphabet for the omniglot character with following settings:

(a) Two hidden layers with 50 nodes each, momentum 0, dropout rate 0, learning rate 0.001.

(b) Two hidden layers with 50 nodes each, momentum 0.4, dropout rate 0, learning rate 0.001.

(c) Two hidden layers with 50 nodes each, momentum 0.4, dropout rate 0.2, learning rate 0.001.

(d) Two hidden layers with 120 nodes each, momentum 0.4, dropout rate 0.2, learning rate 0.001.

Report the final test accuracy after 150 epochs and create the following two plots:

1. The Train and Test curves for loss (on the same plot) over 150 epochs
2. The Train and Test curves for accuracy (on the same plot) over 150 epochs

Clearly label the train curves and the test curves on both plots and label the axes. Your loss and accuracy plots should each have 8 curves plotted (corresponding to train/test for each of the 4 experiments above).

Compare the results, and write 1-2 sentence on the effect of momentum and dropout. Did two layers perform better than the one layer network?

Solution

2.3 Learning Rate (8 pts)

Please train

(a) Two hidden layers with 120 nodes each, momentum 0, dropout rate 0, learning rate 0.01.

(b) Two hidden layers with 120 nodes each, momentum 0, dropout rate 0, learning rate 0.001.

(c) Two hidden layers with 120 nodes each, momentum 0, dropout rate 0, learning rate 0.0001.

Create the following two plots:

1. The Train and Test curves for loss for both tasks (on the same plot) over 150 epochs
2. The Train and Test curves for accuracy for both tasks (on the same plot) over 150 epochs

Clearly label the train curves and the test curves on both plots and label the axes. Your loss and accuracy plots should each have 6 curves plotted (corresponding to train/test for each of the 3 experiments above).

Compare the results, and write 1-2 sentence on the effect of the learning rate.

Solution

2.4 Experiments (4 pts)

For this section, we want to give you the chance to experiment with the model by playing around with the hyper-parameters of the neural network. As in the previous sections, produce train and test plots of loss and accuracy for 3 different experiments. Clearly label the hyper-parameters of the experiments, as well as the final test accuracy. Examples of hyper-parameters to play with:

- Number of hidden nodes
- Momentum
- Dropout
- Learning rate
- Weight initialization
- Number of layers

For each experiment, explain how your findings compare to what you expected. (no more than 3 sentences per experiment).

Solution

Write up

Hand in answers to all questions above. For Problem 6, the goal of your write-up is to document the experiments you have done and your main findings, so be sure to explain the results. Be concise and to the point – do not write long paragraphs or only vaguely explain results.

- The answers to all questions should be in pdf form (please use LaTeX). Your answers must fit within the given boxes and you should not change their size or location.
- Submit your PDF write-up to the Gradescope assignment "Homework 1 Written" and your filled-out template "mlp.py" to the Gradescope assignment "Homework 1 Programming".

Collaboration Questions Please answer the following:

After you have completed all other components of this assignment, report your answers to the collaboration policy questions detailed in the Academic Integrity Policies found here.

1. Did you receive any help whatsoever from anyone in solving this assignment? Is so, include full details.
2. Did you give any help whatsoever to anyone in solving this assignment? Is so, include full details.
3. Did you find or come across code that implements any part of this assignment ? If so, include full details even if you have not used that portion of the code.

Solution