

Homework 1 Part 1

An Introduction to Neural Networks

11-785: INTRODUCTION TO DEEP LEARNING (SPRING 2020)

OUT: **January 19, 2020**

DUE: **February 8, 2020, 11:59 PM**

Start Here

- **Collaboration policy:**

- You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
- You are allowed to talk with / work with other students on homework assignments
- You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.

- **Overview:**

- **MyTorch:** An introduction to the library structure you will be creating as well as an explanation of the local autograder and the way you will submit your homework.
- **Multiple Choice:** These are a series of multiple choice (autograded) questions which will speed up your ability to complete the homework, if you thoroughly understand their answers.
- **A Simple Neural Network:** All of the problems in Part 1 will be graded on Autolab. You can download the starter code/mytorch folder structure from Autolab as well. This assignment has 100 points total, including 95 that are autograded.
- **Appendix:** This contains information and formulas about some of the functions you have to implement in the homework.
- **Glossary:** This contains basic definitions to most of the technical vocabulary used in the handout.

- **Directions:**

- You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.
- We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.

1 MyTorch

The culmination of all of the Homework Part 1's will be your own custom deep learning library, which we are calling *MyTorch* [©]. It will act similar to other deep learning libraries like PyTorch or Tensorflow. The files in your homework are structured in such a way that you can easily import and reuse modules of code for your subsequent homeworks. For Homework 1, MyTorch will have the following structure:

- mytorch
 - loss.py
 - activation.py
 - batchnorm.py
 - linear.py
- hw1
 - hw1.py
 - mc.py
- autograder
 - hw1_autograder
 - * runner.py
 - * test.py
- create_tarball.sh

-
- **Install** python (version 3), numpy, pytest. In order to run the local autograder on your machine, you will need the following libraries installed in python (version 3):

```
pip3 install numpy
pip3 install pytest
```

- **Hand-in** ¹ your code by running the following command from the top level directory, then **SUBMIT** the created *handin.tar* file to autolab:

```
sh create_tarball.sh
```

- **Autograde** ² your code by running the following command from the top level directory:

```
python3 autograder/hw1_autograder/runner.py
```

- **DO NOT:**

- Import any other external libraries other than numpy, as extra packages that do not exist in autolab will cause submission failures.
- Add, move, or remove any files or change any file names.

¹Make sure that all class and function definitions originally specified (as well as class attributes and methods) are fully implemented to the specification provided in this writeup and in the docstrings or comments.

²We provide a local autograder that you can use for your code. It roughly has the same behavior as the one on Autolab, except that it will compare your outputs and attributes with prestored results on prestored data (while autolab directly compares you to a solution). In theory, passing one means that you pass the other, but we recommend you also submit partial solutions on autolab from time to time while completing the homework.

2 Multiple Choice

- These questions are intended to give you major hints throughout the homework.
 - Please try to thoroughly understand the questions and answers for each one.
 - Answer the questions by returning the correct letter as a string in the corresponding question function in `hw1/mc.py`.
 - Each question has only a single correct answer.
 - Verify your solutions by running the local **autograder**.
 - To get credit (5 points), you must answer **all** questions correctly.
-

(1) **Question 1: What are the correct shapes of b and c from the code below?** [1 point]

```
a = np.arange(60.).reshape(3,4,5)
b = np.sum(a, axis=0, keepdims=True)
c = np.sum(a, axis=0)
```

- (A) `b.shape=(3, 4, 5)` `c.shape=(4, 5)`
- (B) `b.shape=(1, 4, 5)` `c.shape=(4, 5)`
- (C) `b.shape=(3, 4, 5)` `c.shape=(1, 4, 5)`
- (D) `b.shape=(3, 4, 5)` `c.shape=(3, 4, 5)`

(2) **Question 2: First, read through the appendix on Batchnorm. In the appendix we discuss reducing covariate shift of the data. What does the mean (μ_B) and standard deviation (σ_B^2) refer to?** [1 point]

- (A) Every neuron in a layer has a mean and standard deviation, computed over an entire batch
- (B) Every layer has a mean and a standard deviation computed over all the neurons in that layer
- (C) Every neuron in a layer has a mean and a standard deviation computed over the entire testing set

(3) **Question 3: For Batchnorm, is it necessary to maintain a running mean and running variance of the training data?** [1 point]

- (A) Yes! We cannot calculate the mean and variance during inference, hence we need to maintain an estimate of the mean and variance to use when calculating the norm of x (\hat{x}) at test time.³
- (B) No! Life is a simulation. Nothing is real.

³You need to calculate the running average at training time, because you really want to find an estimate for the overall covariate shifts over the entire data. Running averages give you an estimate of the overall covariate shifts.

At test time you typically have only one test instance, so if you use the test data itself to compute means and variances, you'll wipe the data out (mean will be itself, var will be inf). Thus, you use the global values (obtained as running averages) from the training data at test time.

- (4) **Question 4:** Read <https://www.geeksforgeeks.org/zip-in-python/> (zip is useful for creating the weights and biases for the linear layer in one line of code.) Did you enjoy the read? [1 point]
- (A) Yes
 - (B) No
- (5) **Question 5:** Which one of these is a valid layer as defined from class? For this question (and later in the homework), $w.shape=(input, output)$ with $x.shape=(batch\ size, input)$ and $b.shape=(1, output)$. Note, below that anywhere we use dot, we could have instead used matmul. [1 point]
- (A) `z = activationFunction(np.dot(x, b) + w)`
 - (B) `z = activationFunction(np.dot(x, w)) + b`
 - (C) `z = activationFunction(np.dot(x, w) + b)`
 - (D) `baked_potato = activationFunction(potato)`

3 A Simple Neural Network

Write your own implementation of the backpropagation algorithm for training your own neural network, as well as a few other features such as activation and loss functions.

The autograder tests will compare the outputs of your methods and the attributes of your classes with a reference solution. Therefore, we do enforce a large portion of the design of your code; however, you still have a lot of freedom in your implementation.

Keep your code as concise as possible, and **leverage Numpy as much as possible**. No PyTorch!

3.1 Task 1: Activations [12 points]

- In `mytorch/activation.py`, implement the `forward` and `derivative` class methods for each activation function.
- The identity function has been implemented for you as an example.
- The output of the activation should be stored in the `self.state` variable of the class. The `self.state` variable should be used for calculating the derivative during the backward pass.

3.1.1 Sigmoid Forward [2 points]

$$S(z) = \frac{1}{1 + e^{-z}}$$

3.1.2 Sigmoid Derivative [2 points]

$$S'(z) = S(z) \cdot (1 - S(z))$$

3.1.3 Tanh Forward [2 points]

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

3.1.4 Tanh Derivative [2 points]

$$\tanh'(z) = 1 - \tanh(z)^2$$

3.1.5 ReLU Forward [2 points]

$$R(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$$

3.1.6 ReLU Derivative [2 points]

$$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

Note: ReLU's derivative is undefined at 0, however we will implement the above derivative for this homework.

3.2 Task 2: Loss [4 points]

- In `mytorch/loss.py`, implement the `forward` and `derivative` methods for `SoftmaxCrossEntropy`. This class inherits the base `Criterion` class.
- We will be using the **softmax cross entropy loss** detailed in the appendix of this writeup; use the `LogSumExp` trick (see appendix) to ensure numerical stability.

3.2.1 Forward [2 points]

Implement the softmax cross entropy operation on a batch of output vectors.

Hint: Add a class attribute to keep track of intermediate values necessary for the backward computation.

- Input shapes:
 - `x`: (batch size, 10)
 - `y`: (batch size, 10) (one hot vectors)
- Output Shape:
 - `out`: (batch size,)

3.2.2 Derivative [2 points]

Calculate the ‘derivative’ of softmax cross entropy using intermediate values saved in the forward pass.

- Output shapes:
 - `out`: (batch size, 10)

3.3 Task 3: Batch Normalization [15 points]

- In `mytorch/batchnorm.py`, implement the `forward` and `backward` methods for `BatchNorm`.
- The Batch Normalization technique comes from Ioffe and Szegedy [2015]. The [appendix](#) has the information necessary to complete the forward and backward functions of `BatchNorm`.
- For the autograder tests, you can assume batch norm will be applied to networks with sigmoid nonlinearities.
- **You will not be able to test Batch Norm until you complete section 3.5.** In section 3.5, you will create your own Multi Layer Perceptron and add Batch Norm layers to your MLP. That is where you will gain the 10 points are for Batch Norm’s correctness during training and 5 points during inference.
- Extra Hints:
 - The input to the backward method (`delta`) is the derivative of the loss w.r.t. Batch Norm output in a given MLP.
 - During training, your forward method should be maintaining a running average of the mean and variance. (Check the ”inference” section of appendix under Batch Norm.) These running averages should be used during inference.

3.4 Task 4: Linear Layer [4 points]

- In `mytorch/linear.py`, implement the `forward` and `backward` methods for the `Linear` class.

3.4.1 Forward [2 points]

Hint: Add a class attribute to keep track of intermediate values necessary for the backward computation.

- Input shapes:
 - `x`: (batch size, `in_feature`)
- Output Shape:
 - `out`: (batch size, `out_feature`)

3.4.2 Backward [2 points]

Write the code for the `backward` method of `Linear`.

The input `delta` is the derivative of the loss with respect to the output of the linear layer. It has the same shape as the linear layer output.

- Input shapes:
 - `delta`: (batch size, `out_feature`)

`dW` and `db`: Calculate `self.dW` and `self.db` for the `backward` method. `self.dW` and `self.db` represent the gradients of the loss (**averaged across the batch**) w.r.t `self.W` and `self.b`. Their shapes are the same as the weight `self.W` and the bias `self.b`.

`dx`: Calculate the return value for the `backward` method. `dx` is the derivative of the loss with respect to the input of the linear layer and has the same shape as the input.

- Output Shape:
 - `dx`: (batch size, `in_feature`)

3.5 Task 5: Simple MLP [55 Points]

In this section of the homework, you will be implementing a Multi-Layer Perceptron with an API similar to popular Automatic Differentiation Libraries like PyTorch, which you will be allowed and encouraged to use in the second part of the homework.

In `hw1/hw1.py` go through the functions of the given MLP class thoroughly and make sure you understand what each function in the class does so that you can create a generic implementation that supports an arbitrary number of layers, types of activations and network sizes.

The **parameters** for the MLP class are:

- **input_size**: The size of each individual data example.
- **output_size**: The number of outputs.
- **hiddens**: A list with the number of units in each hidden layer.
- **activations**: A list of `Activation` objects for each layer.
- **weight_init_fn**: A function applied to each weight matrix before training.
- **bias_init_fn**: A function applied to each bias vector before training.
- **criterion**: A `Criterion` object to compute the loss and its derivative.
- **lr**: The learning rate.
- **momentum**: Momentum scale (Should be 0.0 until completing 3.5.3).
- **num_bn_layers**: Number of `BatchNorm` layers start from upstream (Should be 0 until completing 3.3).

The **attributes** of the MLP class are:

- **@linear_layers**: A list of `Linear` objects.
- **@bn_layers**: A list of `BatchNorm` objects. (Should be `None` until completing 3.3).

The **methods** of the MLP class are:

- **forward**: Forward pass. Accepts a mini-batch of data and return a batch of output activations.
- **backward**: Backward pass. Accepts ground truth labels and computes gradients for all parameters.
Hint: Use **state** stored in **activations** during forward pass to simplify your code.
- **zero_grads**: Set all gradient terms to 0.
- **step**: Apply gradients computed in **backward** to the parameters.
- **train** (Already implemented): Set the mode of the network to train.
- **eval** (Already implemented): Set the mode of the network to evaluation.

Note: Pay attention to the data structures being passed into the constructor *and* the class attributes specified initially.

Sample constructor call:

```
MLP(784, 10, [64, 64, 32], [Sigmoid(), Sigmoid(), Sigmoid(), Identity()],  
    weight_init_fn, bias_init_fn, SoftmaxCrossEntropy(), 0.008, momentum=0.9, num_bn_layers=0)
```


3.5.1 Linear MLP [5 points]

- In `hw1/hw1.py` implement a linear classifier (MLP with no hidden layers) using the `MLP` class. We suggest you read through the entire assignment before you start implementing this part.
- For this problem, some useful parameters may be: `input_size`, `output_size`, `hiddens`, `activations`, `weight_init_fn`, `bias_init_fn`, `criterion`.
- The activation function will be a single `Identity` activation, and the criterion is a `SoftmaxCrossEntropy` object.

You will have to implement the `forward` and `backward` method of the `MLP` class so that it can at least work as a linear classifier.

The `step` function also needs to be implemented, as it will be invoked after every backward pass to update the parameters of the network (the gradient descent algorithm).

3.5.2 Hidden Layers [40 points]

Update the `MLP` class that previously just supported a single fully connected layer (a linear model) such that it can now support an arbitrary number of hidden layers, each with an arbitrary number of units.

Specifically, the `hiddens` argument of the `MLP` class will no longer be assumed to be an empty list and can be of arbitrary length (arbitrary number of layers) and contain arbitrary positive integers (arbitrary number of units in each layer).

The implementation should apply the `Activation` objects, passed as `activations`, to their respective layers. For example, `activations[0]` should be applied to the activity of the first hidden layer.

While at risk of being pedantic, here is a clarification of the expected arguments to be passed to the `MLP` constructor once it can support an arbitrary number of layers with an arbitrary assortment of activation functions.

- `input_size`: The size of each individual data example.
- `output_size`: The number of outputs.
- `hiddens`: A list of layer sizes (number of units per layer).
- `activations`: A list of `Activation` objects to be applied after each linear transformation respectively.
- `weight_init_fn`: A function applied to each weight matrix before training.
- `bias_init_fn`: A function applied to each bias vector before training.
- `criterion`: `SoftmaxCrossEntropy` object.

Extra Hints

For slightly more specification, you will be updating the `linear_layers` and `bn_layers` attributes (if you need to). You will also need to update the `forward`, `backward`, and `step` methods to now account for the multitude of layers (and possibly batchnorm).

We will be using the ordering for the forward pass as follows:

Linear Layer \rightarrow Batch Norm (if applicable) \rightarrow Activation \rightarrow Next Layer ...

If applicable, the value `num_bn_layers` indicates that the first `num_bn_layers` linear layers should be followed by a batchnorm in the forward pass.

3.5.3 Momentum [10 points]

Modify the step function present in the MLP class to include momentum in your gradient descent. The momentum value will be passed as a parameter to the MLP.

We will be using the following momentum update equation:

$$\begin{aligned}\nabla W^k &= \beta \nabla W^{k-1} - \eta \nabla_W \text{Loss}(W^{k-1}) \\ W^k &= W^{k-1} + \nabla W^k\end{aligned}$$

as discussed in lecture. The momentum values for the weights and biases are stored within the `Linear` class.

3.6 Task 6: Training Statistics [5 points]

At the bottom of `hw1/hw1.py` complete the function `get_training_stats`.

You will be passed the MNIST data set (provided as a 3-tuple of `{train_set, val_set, test_set}`). (You will not use the test set in this function.)

- **training_losses**: A Numpy `ndarray` containing the average training loss for each epoch.
- **training_errors**: A Numpy `ndarray` containing the classification error on the training set at each epoch (**Hint**: You should not be evaluating performance on the training set after each epoch, but compute an approximation of the training classification error for each training batch).
- **validation_losses**: A Numpy `ndarray` containing the average validation loss for each epoch.
- **validation_errors**: A Numpy `ndarray` containing the classification error on the validation set after each epoch.

Complete the function such that, given a network, a dataset, a number of epochs and a `batch_size`, you train the network and returns those quantities.

Then execute the provided `test` file using the command below. The command will run `get_training_stats` on MNIST for a given network and plot the previous arrays into files. Run the following command, which will automatically add the images to your submission tar.

```
python3 autograder/hw1_autograder/test.py --outpath hw1
```

Your function should perform `nepoch` number of epochs. Remember to invoke the `zero_grad` function after each batch.

Note: Please ensure that you shuffle the training set after each epoch by using `np.random.shuffle` and generate a list of indices and performing a gather operation on the data using these indices.

Note: You will not be autograded on that part. Instead we will grade you manually by looking at your code and plots. We will check that your code and the statistics you obtain make sense. Contrary to the other questions, we do not require an exact match with the solution, only that you run your forward, backward and step functions in the right order, and use the partitions of the dataset correctly (train and val at least), with shuffling of the training set between epochs.

Note: We provide examples of the plots that you should obtain. If you get similar or better values at the end of the training (loss around 0.3, error around 0.1), then you are very likely to get all the points. If you are a bit above but that the errors and losses still drop significantly during training, it's possible that you get all the points as well.

References

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. CoRR, abs/1502.03167, 2015. URL <http://arxiv.org/abs/1502.03167>.

Appendix

A Softmax Cross Entropy

Let's define softmax, a useful function that offers a smooth (and differentiable) version of the max function with a nice probabilistic interpretation. We denote the softmax function for a logit x_j of K logits (outputs) as $\sigma(x_j)$.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}$$

Notice that the softmax function properly normalizes the k logits, so we can interpret each $\sigma(x_j)$ as a probability and the largest logit x_j will have the greatest mass in the distribution.

$$\sum_{j=1}^K \sigma(x_j) = \frac{1}{\sum_{k=1}^K e^{x_k}} \sum_{j=1}^K e^{x_j} = 1$$

For two distributions P and Q , we define cross-entropy over discrete events X as

$$CE = \sum_{x \in X} P(x) \log \frac{1}{Q(x)} = - \sum_{x \in X} P(x) \log Q(x)$$

Cross-entropy comes from information theory, where it is defined as the expected information quantified as $\log \frac{1}{q}$ of some subjective distribution Q over an objective distribution P . It quantifies how much information we (our model Q) receive when we observe true outcomes from Q – it tells us how far our model is from the true distribution P . We minimize Cross-Entropy when $P = Q$. The value of cross-entropy in this case is known simply as the entropy.

$$H = \sum_{x \in X} P(x) \log \frac{1}{P(x)} = - \sum_{x \in X} P(x) \log P(x)$$

This is the irreducible information we receive when we observe the outcome of a random process. Consider a coin toss. Even if we know the Bernoulli parameter p (the probability of heads) ahead of time, we will never have absolute certainty about the outcome until we actually observe the toss. The greater p is, the more certain we are about the outcome and the less information we expect to receive upon observation.

We can normalize our network output using softmax and then use Cross-Entropy as an objective function. Our softmax outputs represent Q , our subjective distribution. We will denote each softmax output as \hat{y}_j and represent the true distribution P with output labels $y_j = 1$ when the label is for each output. We let $y_j = 1$ when the label is j and $y_j = 0$ otherwise. The result is a degenerate distribution that will aim to estimate P when averaged over the training set. Let's formalize this objective function and take the derivative.

$$\begin{aligned}
L(\hat{y}, y) &= CE(\hat{y}, y) \\
&= - \sum_{i=1}^K y_i \log \hat{y}_i \\
&= - \sum_{i=1}^K y_i \log \sigma(x_i) \\
&= - \sum_{i=1}^K y_i \log \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}} \\
\frac{\partial L(\hat{y}, y)}{\partial x_j} &= \frac{\partial}{\partial x_j} \left(- \sum_{i=1}^K y_i \log \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}} \right) \\
&= \frac{\partial}{\partial x_j} \left(- \sum_{i=1}^K y_i (\log e^{x_i} - \log \sum_{k=1}^K e^{x_k}) \right) \\
&= \frac{\partial}{\partial x_j} \left(\sum_{i=1}^K -y_i \log e^{x_i} + \sum_{i=1}^K y_i \log \sum_{k=1}^K e^{x_k} \right) \\
&= \frac{\partial}{\partial x_j} \left(\sum_{i=1}^K -y_i x_i + \sum_{i=1}^K y_i \log \sum_{k=1}^K e^{x_k} \right)
\end{aligned}$$

The next step is a little bit more subtle, but recall there is only a single true label for each example and therefore only a single y_i is equal to 1; all others are 0. Therefore we can imagine expanding the sum over i in the second term and only one term of this sum will be 1 and all the others will be zero.

$$\frac{\partial L(\hat{y}, y)}{\partial x_j} = \frac{\partial}{\partial x_j} \left(\sum_{i=1}^K -y_i x_i + \log \sum_{k=1}^K e^{x_k} \right)$$

Now we take the partial derivative and remember that the derivative of a sum is the sum of the derivative of its terms and that any term without x_j can be discarded.

$$\begin{aligned}
\frac{\partial L(\hat{y}, y)}{\partial x_j} &= \frac{\partial}{\partial x_j} (-y_j x_j) + \frac{\partial}{\partial x_j} \log \sum_{k=1}^K e^{x_k} \\
&= -y_j + \frac{1}{\sum_{k=1}^K e^{x_k}} \cdot \left(\frac{\partial}{\partial x_j} \sum_{k=1}^K e^{x_k} \right) \\
&= -y_j + \frac{1}{\sum_{k=1}^K e^{x_k}} \cdot (e^{x_j}) \\
&= -y_j + \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}
\end{aligned}$$

That second term looks very familiar, huh?

$$\begin{aligned}
\frac{\partial L(\hat{y}, y)}{\partial x_j} &= -y_j + \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}} \\
&= -y_j + \sigma(x_j) \\
&= \sigma(x_j) - y_j \\
&= \hat{y}_j - y_j
\end{aligned}$$

After all that work we end up with a very simple and elegant expression for the derivative of softmax with cross-entropy divergence with respect to its input. What this is telling us is that when $y_j = 1$, the gradient is negative and thus the opposite direction of the gradient is positive: it is telling us to increase the probability mass of that specific output through the softmax.

B LogSumExp

The LogSumExp trick is used to prevent numerical underflow and overflow which can occur when the exponent is very large or very small. For example, look at the results of trying to exponentiate in python shown in the image below:

```

>>> import math
>>> math.e**1000
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    math.e**1000
OverflowError: (34, 'Result too large')
>>> math.e**(-1000)
0.0

```

As you can see, for exponents that are too large, python throws an overflow error, and for exponents that are too small, it rounds down to zero.

We can avoid these errors by using the LogSumExp trick:

$$\log \sum_{i=1}^n e^{x_i} = a + \log \sum_{i=1}^n e^{x_i - a}$$

You can read more about the derivation of the equivalence [here](#) and [here](#)

C Batch Normalization

Batch Normalization (commonly referred to as “BatchNorm”) is a wildly successful and simple technique for accelerating training and learning better neural network representations. The general motivation of BatchNorm is the non-stationarity of unit activity during training that requires downstream units to adapt to a non-stationary input distribution. This co-adaptation problem, which the paper authors refer to as *internal covariate shift*, significantly slows learning.

Just as it is common to whiten training data (standardize and de-correlate the covariates), we can invoke the abstraction of a neural network as a hierarchical set of feature filters and consider the activity of each layer to be the covariates for the subsequent layer and consider whitening the activity over all training examples after each update. Whitening layer activity across the training data for each parameter update is computationally infeasible, so instead we make some (large) assumptions that end up working well anyway. The main assumption we make is that the activity of a given unit is independent of the activity of all other

units in a given layer. That is, for a layer l with m units, individual unit activities (consider each a random variable) $\mathbf{x} = \{\mathbf{x}^{(k)}, \dots, \mathbf{x}^{(d)}\}$ are independent of each other – $\{\mathbf{x}^{(1)} \perp \dots \mathbf{x}^{(k)} \dots \perp \mathbf{x}^{(d)}\}$.

Under this independence assumption, the covariates are not correlated and therefore we only need to normalize the individual unit activities. Since it is not practical in neural network optimization to perform updates with a full-batch gradient, we typically use an approximation of the “true” full-batch gradient over a subset of the training data. We make the same approximation in our normalization by approximating the mean and variance of the unit activities over this same subset of the training data.

Given this setup, consider $u_{\mathcal{B}}$ to be the mean and $\sigma_{\mathcal{B}}^2$ the variance of a unit’s activity over a subset of the training data, which we will hence refer to as a batch of data \mathcal{B} . For a training set \mathcal{X} with n examples, we partition it into n/m batches \mathcal{B} of size m . For an arbitrary unit k , we compute the batch statistics $u_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ and normalize as follows ($\sigma_{\mathcal{B}}^2$ is added with $\epsilon = 1e - 8$ such that we do not divide by zero):

$$u_{\mathcal{B}}^{(k)} \leftarrow \frac{1}{m} \sum_{i=1}^m \mathbf{x}_i^{(k)} \quad (1)$$

$$(\sigma_{\mathcal{B}}^2)^{(k)} \leftarrow \frac{1}{m} \sum_{i=1}^m \left(\mathbf{x}_i^{(k)} - u_{\mathcal{B}}^{(k)} \right)^2 \quad (2)$$

$$\hat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - u_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad (3)$$

A significant issue posed by simply normalizing individual unit activity across batches is that it limits the set of possible network representations. A way around this is to introduce a set of learnable parameters for each unit that ensure the BatchNorm transformation can be learned to perform an identity transformation. To do so, these per-unit learnable parameters $\gamma^{(k)}$ and $\beta^{(k)}$, rescale and reshift the normalized unit activity. Thus the output of the BatchNorm transformation for a data example, \mathbf{y}_i is given as follows,

$$\mathbf{y}_i \leftarrow \gamma \hat{\mathbf{x}}_i + \beta \quad (4)$$

We can now derive the analytic partial derivatives of the BatchNorm transformation. Let $L_{\mathcal{B}}$ be the training loss over the batch and $\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}$ the derivative of the loss with respect to the output of the BatchNorm transformation for a single data example $\mathbf{x}_i \in \mathcal{B}$.

$$\frac{\partial L}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \gamma \quad (5)$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \beta} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \quad (6)$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \gamma} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \hat{\mathbf{x}}_i \quad (7)$$

$$\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2} \quad (8)$$

$$= \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial}{\partial \sigma_{\mathcal{B}}^2} \left[(\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] \quad (9)$$

$$= -\frac{1}{2} \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} (\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{3}{2}} \quad (10)$$

$$\frac{\partial L}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}} \quad (11)$$

Solve for $\frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}}$

$$\frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}} = \frac{\partial}{\partial \mu_{\mathcal{B}}} \left[(\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] \quad (12)$$

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} + (\mathbf{x}_i - \mu_{\mathcal{B}}) \frac{\partial}{\partial \mu_{\mathcal{B}}} \left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] \quad (13)$$

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} + (\mathbf{x}_i - \mu_{\mathcal{B}}) \frac{\partial}{\partial \mu_{\mathcal{B}}} \left[\left(\frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_{\mathcal{B}})^2 + \epsilon \right)^{-\frac{1}{2}} \right] \quad (14)$$

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \quad (15)$$

$$\begin{aligned} & - \frac{1}{2} (\mathbf{x}_i - \mu_{\mathcal{B}}) \left[\left(\frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_{\mathcal{B}})^2 + \epsilon \right)^{-\frac{3}{2}} \frac{\partial}{\partial \mu_{\mathcal{B}}} \left(\frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_{\mathcal{B}})^2 \right) \right] \\ & = -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2} (\mathbf{x}_i - \mu_{\mathcal{B}}) (\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{3}{2}} \left(-\frac{2}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_{\mathcal{B}}) \right) \end{aligned} \quad (16)$$

Now sub this expression for $\frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}}$ into $\frac{\partial L}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}}$

$$\frac{\partial L}{\partial \mu_{\mathcal{B}}} = - \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2} \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} (\mathbf{x}_i - \mu_{\mathcal{B}}) (\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{3}{2}} \left(-\frac{2}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_{\mathcal{B}}) \right) \quad (17)$$

Notice that part of the expression in the second term is just $\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2}$

$$\begin{aligned} \frac{\partial L}{\partial \mu_{\mathcal{B}}} &= - \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} + \frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} \left(-\frac{2}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_{\mathcal{B}}) \right) \\ &= - \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} - \frac{2}{m} \frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} \sum_{i=1}^m (\mathbf{x}_i - \mu_{\mathcal{B}}) \end{aligned}$$

Now for the grand finale, let's solve for $\frac{\partial L}{\partial \mathbf{x}_i}$

$$\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{x}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} \quad (18)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \left[\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2} \frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}} \frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \right] \quad (19)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2} \frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}} \frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \quad (20)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2} \frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}} \frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \quad (21)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \left[\frac{\partial}{\partial \mathbf{x}_i} \left((\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right) \right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2} \frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}} \frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \quad (22)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2} \frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}} \frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \quad (23)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2} \left[\frac{\partial}{\partial \mathbf{x}_i} \left(\frac{1}{m} \sum_{j=1}^m (\mathbf{x}_j - \mu_{\mathcal{B}})^2 \right) \right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}} \frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \quad (24)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2} \left[\frac{2}{m} (\mathbf{x}_i - \mu_{\mathcal{B}}) \right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}} \frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \quad (25)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2} \left[\frac{2}{m} (\mathbf{x}_i - \mu_{\mathcal{B}}) \right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}} \left[\frac{\partial}{\partial \mathbf{x}_i} \left(\frac{1}{m} \sum_{j=1}^m \mathbf{x}_j \right) \right] \quad (26)$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2} \left[\frac{2}{m} (\mathbf{x}_i - \mu_{\mathcal{B}}) \right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}} \left[\frac{1}{m} \right] \quad (27)$$

$$(28)$$

In summary, we have derived the following quantities required in the forward and backward computation for a BatchNorm applied to a single unit:

Forward

$$u_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^m \mathbf{x}_i \quad (29)$$

$$(\sigma_{\mathcal{B}}^2) = \frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - u_{\mathcal{B}})^2 \quad (30)$$

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - u_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad (31)$$

$$\mathbf{y}_i \leftarrow \gamma \hat{\mathbf{x}}_i + \beta \quad (32)$$

Backward

$$\frac{\partial L}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \gamma \quad (33)$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \beta} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \quad (34)$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \gamma} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \hat{\mathbf{x}}_i \quad (35)$$

$$\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^m \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2} \quad (36)$$

$$(37)$$

Inference

$$E[x] = \alpha * E[x] + (1 - \alpha) * \mu_{\mathcal{B}} \quad (38)$$

$$Var[x] = \alpha * Var[x] + (1 - \alpha) * \sigma_{\mathcal{B}}^2 \quad (39)$$

We cannot calculate the mean and variance during inference, hence we need to maintain an estimate of the mean and variance to use when calculating the norm of \mathbf{x} (\hat{x}) at test time. You need to calculate the running average at training time, because you really want to find an estimate for the overall covariate shifts over the entire data. Running averages give you an estimate of the overall covariate shifts. At test time you typically have only one test instance, so if you use the test data itself to compute means and variances, you'll wipe the data out (mean will be itself, var will be inf). Thus, you use the global values (obtained as running averages) from the training data at test time. The running mean is defined as $E[x]$ and the running variance is defined as $Var[x]$ above.

Glossary

Activation function:

An activation function defines a threshold which specifies when a given neuron in our network fires, i.e. what combinations of weights and biases lead the neuron to output a signal. They are modelled based on activation potentials in the brain, which mean neurons only "fire" when the signals are above the threshold.

Auto-differentiation libraries:

Libraries such as PyTorch and Tensorflow which allow internal calculations of partial derivatives.

A comprehensive explanation of auto-differentiation can be found [here](#)

Bias:

Biases act the same way as weights in a neural network, except unlike weights, biases are not dependent on activity, but rather are always "on".

Backpropagation:

Short for "backwards propagation", backpropagation is an algorithm for learning using gradient descent. Backpropagation is a specific type of autodifferentiation.

More details can be found [here](#)

Forward:

The forward pass of a neural network refers to the calculations of outputs, and the comparison of these to the desired outputs (in supervised learning)

Gradient descent:

Gradient descent is an optimization algorithm. Optimizers are introduced on the [supplementary page](#), but you'll have to wait for class to go more in depth.

Loss:

A loss function measures the difference between the output of the neural network (prediction) and the actual desired output (label). The aim is to reduce the value of the loss function.

Multi-Layer Perceptron (MLP):

An MLP is a network of multiple perceptrons, which, as a combination of linear classifiers can be used to classify more complex shapes.

Perceptron:

A perceptron is a linear classifier. Meaning it can be used to distinguish which side of a linear function a given point lies.

Weight:

Weights define the strength of the "connections" between neurons in the network. A weight is a multiplier, and defines how much of a signal is transmitted through the corresponding connection.