Homework 1
An Introduction to Neural Networks

11-785: Introduction to Deep Learning (Fall 2019)

OUT: **September 09, 2019**
DUE: **September 28, 2019, 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:**
  - **Part 1**: All of the problems in Part 1 will be graded on Autolab. You can download the starter code (hw1.py) from Autolab as well. This assignment has 100 points total, including 90 that are autograded.
  - **Part 2**: This section of the homework is an open ended competition hosted on Kaggle.com, a popular service for hosting predictive modeling and data analytics competitions. The competition page can be found here.

  The handout also contains:

  - **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

  In addition, you will find the page:

  `http://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2019/www/hwnotes/HW1p1.html`

  extremely helpful. It contains detailed information about the key topics needed to complete the homework.

- **Submission:**
  - **Part 1**: The compressed `handout` folder hosted on Autolab contains a single python file, `hw1.py`. This contains a template for solving all of the problems and includes some useful helper functions. Make sure that all class and function definitions originally specified (as well as class attributes and methods) in `hw1.py` are fully implemented to the specification provided in this writeup and in the docstrings or comments in `hw1.py`.

    Your submission must be titled `handin.tar` (gzip format) and it is minimally required to contain a directory called `hw1`, which contains the `hw1.py` file implementing the classes, functions, and methods specified in the writeup. After completing **Training Statistics** `hw1` will also contain images. Other source files that are imported by `hw1.py` are allowed. **Please do not import**

any other external libraries other than Numpy, as extra packages that do not exist in the autograder image will cause a submission failure.

- **Part 2**: See the the competition page for details.

- **Local autograder (Part 1):** To make your life easier, we provide in the handout for part 1 a local autograder that you can use on your own 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.

  To run the local autograder : move your `hw1` folder in `local_autograder/` and run `runner.py` from the `local_autograder` folder.

1 Part 1: A Simple Neural Network

For part 1 of the homework you will 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. 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.

In the file `hw1.py` we provide you with classes whose methods you have to fill. The autograder tests will compare the outputs of your methods and attributes of your classes with a reference solution. Therefore we enforce for a large part the design of your code ; however, you still have a lot of freedom in your implementation . 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.

Data

The tar folder for Part 1 has the following structure:

- data
- example_training_stats
- hw1
- local_autograder
- test

You will be editing the python file in the hw1 directory. The local autograder can be used for testing locally. example_training_stats contains example plots for the training function.

1.1 Task 1: Activations [12 points]

For each of the activation functions, implement the **forward** and **backward** functions of the class. No helper functions or additional classes should be necessary. Keep your code as concise as possible, and leverage Numpy as much as possible.

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 further used for calculating the derivative during the backward pass.

1.1.1 Sigmoid Forward [2 points]

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

1.1.2 Sigmoid Backward [2 points]

$$\mathrm{S'(z)} = S(z) \cdot (1 - S(z))$$

1.1.3 Tanh Forward [2 points]

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

1.1.4 Tanh Backward [2 points]

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

1.1.5 ReLU Forward [2 points]

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

1.1.6 ReLU Backward [2 points]

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

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

1.2 Task 2: Loss [4 points]

For this homework, 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.

Implement the methods specified in the class definition `SoftmaxCrossEntropy`. This class inherits the base `Criterion` class.

Again no helper methods should be needed and you should stride to keep your code as simple as possible. Use 1e-8 for eps.

1.2.1 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, 1)

1.2.2 Backward [2 points]

Perform the 'backward' pass of softmax cross entropy operation using intermediate values saved in the forward pass.

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

1.3 Task 3: Batch Normalization [15 points]

In this problem you will implement the Batch Normalization technique from Ioffe and Szegedy [2015]. Implement the `BatchNorm` class and make sure that all provided class attributes are set correctly. Apply the BatchNorm transformation of the first `num_bn_layers` specified as an argument to `MLP`. For the sake of the autograder tests, you can assume that batch norm will be applied to networks with sigmoid non-linearities. 10 points are for BatchNorm's correctness during training, and 5 for during inference.

The appendix has the information necessary to complete the forward and backward functions of BatchNorm.

1.4 Task 4: Initializations [0 points]

Good initialization has been shown to make a great difference in the training process. You are expected to implement the initializers for weights and bias. There won't be any autograder tests that check for weight initializer implementations, but you will need to implement a random normal initializer and a zeros initializer to test your implementation for the last task (Training Statistics).

1.4.1 Weight Initialization [0 points]

The `random_normal_weight_init` function accepts two ints specifying the number of inputs and units. It should return an appropriately shaped `ndarray` with each entry initialized with an independent sample from the standard normal distribution.

1.4.2 Bias Initialization [0 points]

The `zeros_bias_init` function accepts the number of bias units and returns an appropriately shaped ndarray with each entry initialized to zero.

1.5 Task 5: Simple MLP [59 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.

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 1.5.3).
- `num_bn_layers`: Number of `BatchNorm` layers start from upstream (Should be 0 until completing 1.3).

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

- `@W`: The list of weight matrices.
- `@dW`: The list of weight matrix gradients.
- `@b`: A list of bias vectors.
- `@db`: A list of bias vector gradients.
- `@bn_layers`: A list of `BatchNorm` objects. (Should be `None` until completing 1.3).

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)
```

1.5.1 Linear MLP [9 points]

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, you will have to fill in the parts of the MLP class and pass the following parameters:

- `input_size`: The size of each individual data example.
- `output_size`: The number of outputs.
- `hiddens`: An empty list because there are no hidden layers.
- `activations`: A single `Identity` activation.
- `weight_init_fn`: Function name of a weight initializer function you will write for local testing and will be passed to you by the autograder.

- `bias_init_fn`: Function name of a bias initializer function you will write for local testing and will be passed to you by the autograder.
- `criterion`: `SoftmaxCrossEntropy` object.

Consider that a linear transformation with an identity activation is simply a linear model.

You will have to implement the forward and backward function of the `MLP` class so that it can at least work as a linear classifier. All parameters should be maintained as class attributes originally specified in the original handout code (e.g. `self.W`, `self.b`)

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

After all parts are filled, train the model for 100 epochs with a batch size of 100 and try visualizing the training curves using the utilities provided in `test.py`.

1.5.2 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 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 arbitrary numbers 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`: Function name of a weight initializer function you will write for local testing and will be passed to you by the autograder.
- `bias_init_fn`: Function name of a bias initializer function you will write for local testing and will be passed to you by the autograder.
- `criterion`: `SoftmaxCrossEntropy` object.

1.5.3 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. Your function should perform `epoch` number of epochs and return the resulting weights. Instead of calling the step function after completing your backward pass, you would have to call the momentum function to update the parameters. Also, 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.

1.6 Task 6: Training Statistics [10 points]

In this problem you will be passed the MNIST data set (provided as a 3-tuple of {`train_set`, `val_set`, `test_set`}, each in the same format described in 1), and a series of network and training parameters.

- `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 loss for each epoch.
- `validation_errors`: A Numpy `ndarray` containing the classification error on the validation set after each epoch.

Fill the function `get_training_stats` that, given a network, a dataset, a number of epochs and a batch_size, trains the network and returns those quantities. Then execute the provided `test` file, that will run `get_training_stats` on MNIST for a given network and plot the previous arrays into files. Hand-in those image files along with your code on autolab.

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.

2 Part 2: Frame Level Classification of Speech

This part of the homework is a live competition on kaggle.

In this challenge you will take your knowledge of feedforward neural networks and apply it to a more useful task than recognizing handwritten digits: speech recognition. You are provided a dataset of audio recordings (utterances) and their phoneme state (subphoneme) labels. The data comes from articles published in the Wall Street Journal (WSJ) that are read aloud and labelled using the original text. If you have not encountered speech data before or have not heard of phonemes or spectrograms, we will clarify the problem further.

You will be evaluated on the accuracy of the prediction of the phoneme state labels for each frame in the test set.

Please refer to the kaggle page for more details on the task.

Data

- **train.npy**: (24500,)
- **train_labels.npy**: (24500,)
- **dev.npy**: (1100,)
- **dev_labels.npy**: (1100,)

- **test.npy**: (361,)

  The audio data has been transcribed into mel spectrograms. You have 100 40-dimensional mel spectral (row) vectors per second of speech in the recording.

2.1 Task

Your task is to generate predictions for the phonemes of the test set. You will be evaluated based on the accuracy of your predictions. Grade cut-offs are released after the early deadline. For detailed information, please look at the kaggle page

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) $\mathbf{x} = \{\mathbf{x}^{(k)}, \ldots, \mathbf{x}^{(d)}\}$ are independent of each other – $\{\mathbf{x}^{(1)} \perp \ldots \mathbf{x}^{(k)} \ldots \perp \mathbf{x}^{(d)}\}$.

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 $\sigma^2_{\mathcal{B}}$ 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 $\sigma^2_{\mathcal{B}}$ and normalize as follows:

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

$$(\sigma^2_{\mathcal{B}})^{(k)} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \left(\mathbf{x}^{(k)}_i - u^{(k)}_{\mathcal{B}}\right)^2 \tag{2}$$

$$\hat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - u_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{3}$$

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,

$$\mathbf{y}_i \leftarrow \gamma \hat{\mathbf{x}}_i + \beta \tag{4}$$

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 $\mathbf{x}_i \in \mathcal{B}$.

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

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

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

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

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

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

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

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

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

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

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

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \tag{15}$$
$$- \frac{1}{2}(\mathbf{x}_i - \mu_{\mathcal{B}})\left[\left(\frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - u_{\mathcal{B}})^2 + \epsilon\right)^{-\frac{3}{2}}\frac{\partial}{\partial \mu_{\mathcal{B}}}\left(\frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_{\mathcal{B}})^2\right)\right]$$

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}(\mathbf{x}_i - \mu_{\mathcal{B}})\left(\sigma_{\mathcal{B}}^2 + \epsilon\right)^{-\frac{3}{2}}\left(-\frac{2}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right) \tag{16}$$

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

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

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

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

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

$$\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{x}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} \tag{18}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2}\frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i}\right] \tag{19}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2}\frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{20}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2}\frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{21}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[\frac{\partial}{\partial \mathbf{x}_i}\left((\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}}\right)\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2}\frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{22}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2}\frac{\partial \sigma_{\mathcal{B}}^2}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{23}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2}\left[\frac{\partial}{\partial \mathbf{x}_i}\left(\frac{1}{m}\sum_{j=1}^{m}(\mathbf{x}_j - \mu_{\mathcal{B}})^2\right)\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{24}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2}\left[\frac{2}{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{25}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2}\left[\frac{2}{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\left[\frac{\partial}{\partial \mathbf{x}_i}\left(\frac{1}{m}\sum_{j=1}^{m}\mathbf{x}_j\right)\right] \tag{26}$$
$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma_{\mathcal{B}}^2}\left[\frac{2}{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\left[\frac{1}{m}\right] \tag{27}$$
$$\tag{28}$$

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

Forward

$$u_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i \tag{29}$$
$$(\sigma_{\mathcal{B}}^2) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - u_{\mathcal{B}})^2 \tag{30}$$
$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - u_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{31}$$

Backward

$$\frac{\partial L}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\gamma \tag{32}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \beta} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \tag{33}$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \gamma} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\hat{\mathbf{x}}_i \tag{34}$$

$$\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2} \tag{35}$$

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 classfy 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.