Homework 1 Part 1
An Introduction to Neural Networks

11-785: Introduction to Deep Learning (Spring 2021)

OUT: **February 8, 2021**
DUE: **February 28, 2021, 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 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.
  - **Appendix**: This contains information and formulas about some of the functions you have to implement in the homework.
  - **Glossary and detailed technical explanation**: 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
- autograder
  - hw1_autograder
    - runner.py
    - test.py
- toyproblem
  - toy.py
  - Toyproblem.pdf
- expected_graphs
  - train_error.png
  - train_loss.png
  - val_error.png
  - val_loss.png
- 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
```

[1] Make 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.

- **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.

[2] We 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. Each question has only a single correct answer. (*HINT: You can try to code some questions and verify your answers.)

---

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

```
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 ($\mu_{\mathcal{B}}$) and standard deviation ($\sigma^2_{\mathcal{B}}$) refer to?**

(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?**

(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.

(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?**

(A) Yes

(B) No

[3]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.

(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`**.**

(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.

NOTE:
This task is actually as simple as putting the given formulas into code. If you don't understand how the class structure works, watch Rec0A. This is an example of inheritance, where the activation class forms a blueprint, which we use for the sigmoid, tanh, Relu functions. In order to compute the derivative you need the result of the activation, (its state) so don't forget to save that.

REFERENCE: Understanding the derivative of the sigmoid function.

3.1.1 Sigmoid Forward [2 points]

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

3.1.2 Sigmoid Derivative [2 points]

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

3.1.3 Tanh Forward [2 points]

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

3.1.4 Tanh Derivative [2 points]

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

3.1.5 ReLU Forward [2 points]

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

3.1.6 ReLU Derivative [2 points]

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

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 [18 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, the batch norm will be applied with sigmoid non-linearities.
- You will not be able to test your BatchNorm code 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 12 points are for Batch Norm's correctness during training and 6 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.

Forward

$$u_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_i \tag{1}$$

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

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

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

x_i is the input to the batch norm layer. Within the forward method, compute the mean, variance, and norm. Then return the final output, y_i. All other variables are defined in the appendix or code.

Inference

$$E[x] = \alpha * E[x] + (1 - \alpha) * \mu_{\mathcal{B}} \tag{5}$$

$$Var[x] = \alpha * Var[x] + (1 - \alpha) * \sigma_{\mathcal{B}}^2 \tag{6}$$

During training (and only training), your forward method should be maintaining a running average of the mean and variance. These running averages should be used during inference. (Check the **inference** section of appendix under Batch Norm for why we use them.) You will need to manually pass the eval value in 3.5 when using batch norm in your Neural Network, checkout **self.train_mode**.

Backward

$$\frac{\partial L}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \gamma \tag{7}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \tag{8}$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \hat{\mathbf{x}}_i \tag{9}$$

$$\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i} \frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2} \tag{10}$$

The input to the backward method (delta) is the derivative of the loss w.r.t. the output of batch norm layer, $\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}$. Within the backward method calculate (dgamma $\frac{\partial L}{\partial \gamma}$) and (dbeta $\frac{\partial L}{\partial \beta}$) so that you can update those parameters later. To complete the backward pass, return $\frac{\partial L}{\partial \mathbf{x}_i}$, whose equation can be found in the appendix. All other derivative derivations for batch norm can be found in the appendix.

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 [62 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 [6 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 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 [46 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 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`: 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:

$$\nabla W^k = \beta \nabla W^{k-1} - \eta \nabla_W Loss(W^{k-1})$$
$$W^k = W^{k-1} + \nabla W^k$$

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

TIP: Debugging your code:
All the above tasks will also help you to develop basic debugging skills and its always advisable to attempt solving such problems with toy examples of your own. In order to motivate this habit, we want you to be able to debug your neural network either by constructing a toy example of your own or by using the one which we are providing.
Try experimenting with the toy problem mentioned in the toyproblem pdf to test out your MLP on a small toy dataset at this point.

3.6 Training Statistics

At the bottom of `hw1/hw1.py` complete the function `get_training_stats`.
Note: This function is not graded and is not required for you to get full points in this homework. It is meant for you to try and train the neural network that you have coded.
You will be given 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 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 the above mentioned 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.

```
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: 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 have written a fully functional neural network. You are **not** required to submit your plots. Its for your own verification.

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 ($\sigma^2_{\mathcal{B}}$ is added with $\epsilon = 1e-8$ such that we do not divide by zero):

$$u^{(k)}_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}^{(k)}_i \tag{11}$$

$$(\sigma^2_{\mathcal{B}})^{(k)} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \left(\mathbf{x}^{(k)}_i - u^{(k)}_{\mathcal{B}}\right)^2 \tag{12}$$

$$\hat{\mathbf{x}}_i \leftarrow \frac{\mathbf{x}_i - u_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{13}$$

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,

$$\mathbf{y}_i \leftarrow \gamma\hat{\mathbf{x}}_i + \beta \tag{14}$$

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

$$\frac{\partial L}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\gamma \tag{15}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \beta} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \tag{16}$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \gamma} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\hat{\mathbf{x}}_i \tag{17}$$

$$\frac{\partial L}{\partial \sigma^2_{\mathcal{B}}} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma^2_{\mathcal{B}}} \tag{18}$$

$$= \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial}{\partial \sigma^2_{\mathcal{B}}}\left[(\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{1}{2}}\right] \tag{19}$$

$$= -\frac{1}{2}\sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}(\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{3}{2}} \tag{20}$$

$$\frac{\partial L}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}} \tag{21}$$

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

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

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

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} + (\mathbf{x}_i - \mu_{\mathcal{B}})\frac{\partial}{\partial \mu_{\mathcal{B}}}\left[\left(\frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - u_{\mathcal{B}})^2 + \epsilon\right)^{-\frac{1}{2}}\right] \tag{24}$$

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} \tag{25}$$
$$-\frac{1}{2}(\mathbf{x}_i - \mu_{\mathcal{B}})\left[\left(\frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - u_{\mathcal{B}})^2 + \epsilon\right)^{-\frac{3}{2}}\frac{\partial}{\partial \mu_{\mathcal{B}}}\left(\frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_{\mathcal{B}})^2\right)\right]$$

$$= -(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}(\mathbf{x}_i - \mu_{\mathcal{B}})\left(\sigma_{\mathcal{B}}^2 + \epsilon\right)^{-\frac{3}{2}}\left(-\frac{2}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right) \tag{26}$$

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

$$\frac{\partial L}{\partial \mu_{\mathcal{B}}} = -\sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}\sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}(\mathbf{x}_i - \mu_{\mathcal{B}})\left(\sigma_{\mathcal{B}}^2 + \epsilon\right)^{-\frac{3}{2}}\left(-\frac{2}{m}\sum_{i=1}^{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right) \tag{27}$$

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 $\frac{\partial L}{\partial \mathbf{x}_i}$

$$\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{x}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} \tag{28}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma^2_{\mathcal{B}}}\frac{\partial \sigma^2_{\mathcal{B}}}{\partial \mathbf{x}_i} + \frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i}\right] \tag{29}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma^2_{\mathcal{B}}}\frac{\partial \sigma^2_{\mathcal{B}}}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{30}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \sigma^2_{\mathcal{B}}}\frac{\partial \sigma^2_{\mathcal{B}}}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{31}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[\frac{\partial}{\partial \mathbf{x}_i}\left((\mathbf{x}_i - \mu_{\mathcal{B}})(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{1}{2}}\right)\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma^2_{\mathcal{B}}}\frac{\partial \sigma^2_{\mathcal{B}}}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{32}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma^2_{\mathcal{B}}}\frac{\partial \sigma^2_{\mathcal{B}}}{\partial \mathbf{x}_i} + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{33}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma^2_{\mathcal{B}}}\left[\frac{\partial}{\partial \mathbf{x}_i}\left(\frac{1}{m}\sum_{j=1}^{m}(\mathbf{x}_j - \mu_{\mathcal{B}})^2\right)\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{34}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma^2_{\mathcal{B}}}\left[\frac{2}{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\frac{\partial \mu_{\mathcal{B}}}{\partial \mathbf{x}_i} \tag{35}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma^2_{\mathcal{B}}}\left[\frac{2}{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\left[\frac{\partial}{\partial \mathbf{x}_i}\left(\frac{1}{m}\sum_{j=1}^{m}\mathbf{x}_j\right)\right] \tag{36}$$

$$= \frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\left[(\sigma^2_{\mathcal{B}} + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L_{\mathcal{B}}}{\partial \sigma^2_{\mathcal{B}}}\left[\frac{2}{m}(\mathbf{x}_i - \mu_{\mathcal{B}})\right] + \frac{\partial L_{\mathcal{B}}}{\partial \mu_{\mathcal{B}}}\left[\frac{1}{m}\right] \tag{37}$$

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

Forward

$$u_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i \tag{38}$$

$$(\sigma^2_{\mathcal{B}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_i - u_{\mathcal{B}})^2 \tag{39}$$

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - u_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{40}$$

$$\mathbf{y}_i \leftarrow \gamma\hat{\mathbf{x}}_i + \beta \tag{41}$$

Backward

$$\frac{\partial L}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \hat{\mathbf{x}}_i} = \frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\gamma \tag{42}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \beta} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i} \tag{43}$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \gamma} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \mathbf{y}_i}\hat{\mathbf{x}}_i \tag{44}$$

$$\frac{\partial L}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m}\frac{\partial L_{\mathcal{B}}}{\partial \hat{\mathbf{x}}_i}\frac{\partial \hat{\mathbf{x}}_i}{\partial \sigma_{\mathcal{B}}^2} \tag{45}$$

Inference

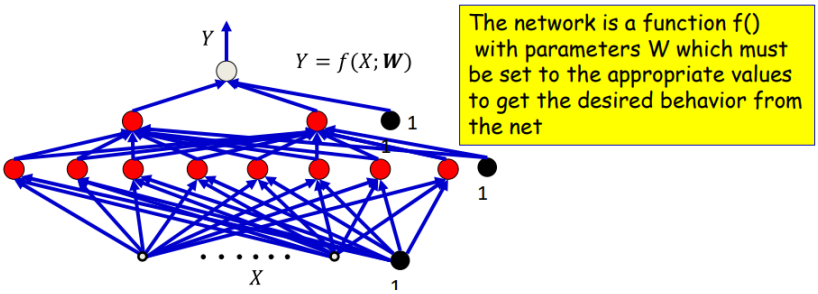
$$E[x] = \alpha * E[x] + (1-\alpha) * \mu_{\mathcal{B}} \tag{46}$$

$$Var[x] = \alpha * Var[x] + (1-\alpha) * \sigma_{\mathcal{B}}^2 \tag{47}$$

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. 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.

D Glossary and detailed technical explanation

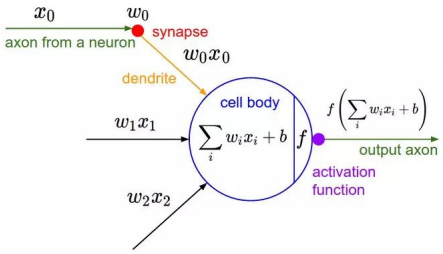
Multi-Layer Perceptron: Explaining the structure: An MLP is a class of feedforward networks, which consists of at least 3 layers: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. The network is a function f() with parameters W which must be set to the appropriate values to get the desired behavior from the net.



The MLP can be constructed to represent any complex decision boundary and this flexibility is one of the most important features of an MLP. To explore this in greater detail, refer to lecture 3, slides 11 onward.
Forward Propagation: The idea of forward propagation is quite straightforward and has 4 main components attached to it. These are as follows:

1. **Inputs:** The inputs are the first layer in the composition of an MLP. While constructing the neural network, the number of neurons is decided by considering the number of values in a single, vectorized input sample. Thus, the input directly dictates the number of neurons in the input layer. For example, within the MNIST dataset, each data sample is a 28x28 grayscale image, this can be vectorized as 784 (product of height and width of the image). Thus we have 784 input neurons, each in accordance with one pixel of the 28x28 image. To visualize the given example, look at this great visualization by Grant Sanderson, creator of the YouTube channel 3Blue1Brown.
2. **Weights:** Weights are the number on the edges that connect the neurons to each other. In essence,they quantify the importance of certain elements of the data in determining the final output. When we say that a neural network is a function that we are trying to learn, it is in fact the weights that we are trying to modify to best fit the function represented by the given data. In other words, weights are parts of the learnable parameters which are updated by gradient descent during backpropagation. The question is, what values of weights do we start with and how much freedom of choice do we have? Weight initialization is an intelligent way of selecting weights to start off the computations of the neural network. The objective of initializing weights in an intelligent fashion is to avoid an explosion or experience a vanishing effect due to the multiplication of results. There are many methods of weight initialization, including randomized / zero / distributed / constrained.
3. **Biases:** 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".
4. **Activation Function:** Neurons don't really know how to put a bound on the value produced as the result of the affine combination. Due to this, we would not be able to decide whether a neuron should fire or not. In order to counter this, the result of the affine combination is passed through a function called activation function. The nature of an activation function can be linear or non-linear. Linear activation functions are not preferred because of certain reasons. To understand this, consider the linear activation function f(x) = x:
    - We observe that the function's derivative is a constant. That means there is constant gradient descent occurring since there is no relation to the value of z.
    - Our model is not really learning as it does not improve upon the error term, which is the whole point of the neural network.
    - Since the activation is linear, nesting in 2 or N number of hidden layers with the same function will have no real effect. The N layers could basically be squashed into one layer.

To understand this in a more granular fashion, let's look at what happens at a single node. There is an interesting way to visualize a neuron and understand how it handles all the above mentioned components, as shown in the image below.

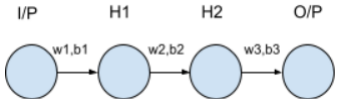


Thus, from a logical perspective, the forward pass would go as follows:

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \ldots D]$
- Set:
  - $D_0 = D$, is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \ldots D;\quad y_0^{(k=1\ldots N)} = x_0 = 1$
- For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$ D_k is the size of the kth layer
    - $z_j^{(k)} = \sum_{i=0}^{D_{k-1}} w_{i,j}^{(k)}\, y_i^{(k-1)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
- Output:
  - $Y = y_i^{(N)}, j = 1..D_N$

Backpropagation: The concept of a neural network learning is basically the minimization of cost function. A cost function is a function utilized in supervised machine learning, the cost function returns the error between predicted outcomes compared with the actual outcomes (aka divergence). The aim of supervised machine learning is to minimize the overall cost, thus optimizing the correlation of the model (the neural network) to the system (given data) that it is attempting to represent.

This minimization of the cost function is performed through gradient descent, and the major component of this optimization is the propagation of the derivatives throughout the network. This propagation is a basic chain rule applied across the network. Consider the following example.
For a really simple network with 2 hidden layers, 1 neuron in each layer :



The first and last layers are the input and output layers and H1 and H2 denote the hidden layers. w_i,b_i are the weights and biases going into the respective layers. Let's annotate the activation of the last neuron, i.e the output neuron to be a^L where L indicates the last layer. Thus the activation of H2 would be a^{L-1}. Let the activation be the Sigmoid function $\sigma(x)$.
Thus $a^L = \sigma(z^L)$
where $z^L = w^L * a^{(L-1)} + b^L$

Thus, the cost function for a particular training example is given as:
Where y represents the ground truth for the data sample (the given labels).
Now if we look closely, the cost function varies with variation in $w^L, b^L, a^{(L-1)}$. Our basic aim is to understand how changing one of the terms above affects the cost function. Consider the change observed in cost function by changing the weight.
Thus, we want to find

$$\frac{\partial C_0}{\partial W^L} = \frac{\partial C_0}{\partial a^L} * \frac{\partial a^L}{\partial z^L} * \frac{\partial z^L}{\partial W^L}$$

$$\frac{\partial C_0}{\partial W^L} = 2 * (a^L - y), \frac{\partial a^L}{\partial z^L} = \sigma'(z^L), \frac{\partial z^L}{\partial W^L} = \sigma'(a^{L-1})$$

Therefore,

$$\frac{\partial C_0}{\partial W^L} = 2 * (a^L - y) * \sigma'(z^L) * \sigma'(a^{L-1})$$

The above expression is the derivative with one particular training example. The full cost function involves averaging the cost across all training example, the derivative for the full cost function will be given by:

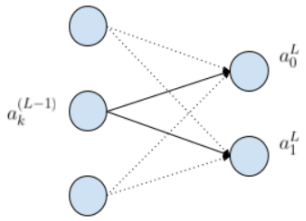
$$\frac{\partial C}{\partial W^L} = \frac{1}{n} * \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial W^L}$$

Similarly, we can find the derivatives for b^L, a^{L-1} as:

$$\frac{\partial C_0}{\partial b^L} = 2 * (a^L - y) * \sigma'(z^L) * 1$$

$$\frac{\partial C_0}{\partial a^{L-1}} = 2 * (a^L - y) * \sigma'(z^L) * W^L$$

To visually understand the above example, look at both one and two by Grant Sanderson, We can iterate the above process throughout the network to see how sensitive the cost function is to previous weights, biases and activations. The example is easily generalized to a network with more number of neurons in each layer. Consider the network given below. We see that the neuron labelled in layer L-1, influences two other neurons in layer L.



Thus the derivative of the cost function with respect to this activation (akL-1) is given by the equation:

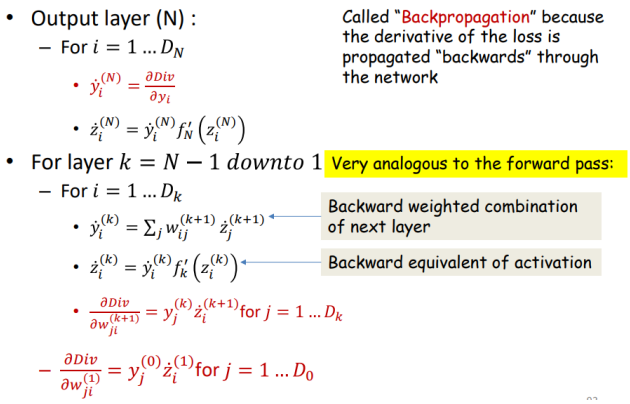
$$\frac{\partial C_0}{\partial a_k^{L-1}} = \sum_{j=0}^{n_L - 1} \frac{\partial C_0}{\partial a_j^L} * \frac{\partial a_j^L}{\partial z_j^L} * \frac{\partial z_j^L}{\partial a_j^{L-1}}$$

This is the sum over number of neurons in layer L.The gradients calculated are used in the gradient descent update rule after being multiplied with the learning rate. Thus the iterative update looks as follows:

do
- $W^{k+1} = W^k - \eta^k \nabla L(W^k)^T$
- $k = k + 1$

while $\left|L\left(W^k\right) - L\left(W^{k-1}\right)\right| > \varepsilon$

This is the learning aspect of the MLP whereby the parameters are modified to achieve the desired objective. Logically, the entire process can be viewed as follows:



Finishing this homework will give you a taste of the math used to construct complex networks for various fields of applied Deep Learning. Kudos on finishing the first step of an exciting series of deep learning concepts! Time to focus on Hw1P2 if you haven't already started!