ECE421: Introduction to Machine Learning — Fall 2024

Assignment 3: Feed-forward Neural Network

Due Date: Friday, November 15, 11:59 PM

Objectives

In this assignment, we continue implementing our own (small) version of `PyTorch` library. In this assignment you will:

- implement a feed-forward neural network (NN) that will perform classification on the IRIS dataset using functions in the `NumPy` library only.
- start exploring the `PyTorch` library.
- train a simple model on FashionMNIST using `PyTorch`.

You will also be asked to answer several questions related to your implementations.

To avoid any potential installation issue, you are encouraged to develop your solution using Google Colab notebooks.

Requirements

In your implementations, please use the function prototype provided (*i.e.* name of the function, inputs and outputs) in the detailed instructions presented in the starter-code and the remainder of this document. We will be testing your code using a test function that which evokes the provided function prototype. If our testing file is unable to recognize the function prototype you have implemented, you can lose significant portion of your marks. Similar to the previous assignments, you also need to submit a separate `PA3_qa.pdf` file that answer questions related to your implementations.

0 Implementing Feed-Forward Fully-Connected Neural Networks with `NumPy`

This section will provide a background on neural networks that is designed to help you complete the assignment. Neural network libraries such as Tensorflow and PyTorch have made training complicated neural network architectures very easy. However, we want to emphasize that neural networks begin with fundamentally simple models that are just a few steps removed from basic logistic regression.

0.1 Background

In this assignment, you will implement a feed-forward fully-connected network, the fundamental type of neural network models, in plain numpy. The notation that we use in this assignment is different from what you had been introduced to in class. This is intentional, as we want to make sure that you fully understood the basics of neural networks and you are able to switch between different notations and modelings of neural networks.
[**NOTE:** As mentioned above, do not confuse the implementation in this assignment with the Directed Acyclic Graph (DAG) model that was introduced in class. In DAG modeling of neural networks, we clamp the weight that connects to unit to zero, if the two units in the neural network are not connected to each other. While the DAG model simplifies deriving the math for feed-forward and backpropagation, implementing the DAG would result in spending a lot of time multiplying by zero which is wasteful. Instead, in the actual implementation, we keep a set of indices for the connected units and we only some over those indices.]

We will start with the essential elements and then build up in complexity. A neural network model is defined by the following.

- An architecture defining the flow of information between computational layers. This defines the composition of functions that the network performs from input to output.
- A cost function (e.g. cross-entropy or mean squared error).
- An optimization algorithm (e.g. stochastic gradient descent with backpropagation).
- A set of hyperparameters. (Here we use this as a catch-all term to also include algorithm parameters that technically are not "hyperparameters" in the traditional sense because they don't help you change the bias-variance tradeoff, such as the learning rate and the mini-batch size for stochastic gradient descent with mini-batches.)

Each *layer* is defined by the following components.

- A parameterized function that defines the layer's map from input to output (*e.g.* $f(\mathbf{x}) = \sigma(W\mathbf{x} + \mathbf{b})$).
- An activation function σ (*e.g.* ReLU, sigmoid, etc.).
- A set of parameters (*e.g.* weights and bias terms).

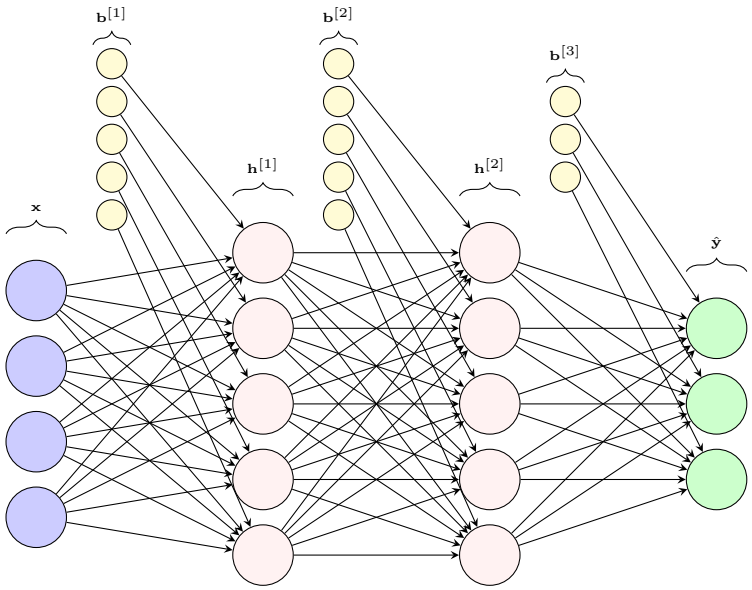
Neural networks are commonly used for supervised learning problems, where we have a set of inputs and a set of labels, and we want to learn the function that maps inputs to labels. To learn this function, we need to update the parameters of the network (*i.e.* the weights, including the bias terms). We do this using **minibatch gradient descent**. This is because iterating on all the data at once (*i.e.* batch gradient descent) is inefficient for large data sets, whereas iterating on just one training point at a time (*i.e.* stochastic gradient descent) introduces excessive stochasticity (randomness) and makes poor use of your computer's caches and potential for parallelism.

To compute the gradients for gradient descent, we use a dynamic programming algorithm called **backpropagation**. In the backpropagation algorithm, we perform the following sequence of actions.

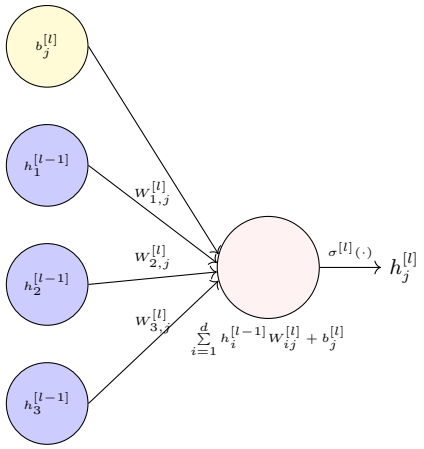
1. **Forward propagation of inputs:** First, we compute what is called a **forward pass** of the network. In the forward pass, we send a mini-batch of input data (*e.g.* 50 training points) through the network, to compute the output.
2. **Computing the loss:** The output is a set of predicted labels, which we use as input to our loss function (along with the true labels from the training data).
3. **Backpropagation and parameter updates:** We then take the gradients of the loss with respect to the parameters of each layer, starting with the output of the network and using the chain rule to propagate backwards through the layers. This is called the **backward pass**. During the backward pass we compute the gradients of the loss function with respect to each of the model parameters, starting from the last layer and "propagating" the information from the loss function backwards through the network. This lets us calculate gradients with respect to all the parameters of our network while letting us avoid computing the same gradients multiple times.

Here are a couple of important notes on implementing batching in your code.

- Every step of your neural network should be defined to operate on mini-batches of data. During a single operation of mini-batch gradient descent, you take a matrix of shape (B, d) where B is the mini-batch size and d is the number of features, and do a forward pass on B training points at once – ideally using vector operations to obtain some parallelism in your computations (as every training point is processed the same way).
- As you are writing the gradient descent algorithm to work on mini-batches, all of your derivations must work for mini-batches. For that reason, many of the derivations you do for this homework will differ from those you have seen in class.
- Thinking in terms of mini-batches often changes the shapes and operations you do. Your derivations **must be for mini-batches** and **cannot** use loops to iterate over individual data points. Be prepared to spend some time working out the tricky details of how to do this.



(a) A 3-layer fully-connected neural network.



(b) The j_{th} neuron in the l_{th} layer.

0.2 Feed-Forward, Fully-Connected Neural Networks

A feed-forward, fully-connected neural network consists of layers of units alternating with layers of edges. Each layer of edges performs an affine transformation of an input, followed by a nonlinear activation function. "Fully-connected" means that a layer of edges connects every unit in one layer of units to every unit in the next layer of units.

We use the following notation when defining fully-connected layers, with superscripts in brackets indexing layers (both layers of units and layers of edges) and subscripts indexing the vector/matrix elements. In this notation, we will use **row vectors** (not column vectors) to represent unit layers so that we can apply successive matrices (edge layers) to them from left to right.

- $\mathbf{x}$: A single data vector, of shape $1 \times d$, where d is the number of features. You can think of it as "unit layer zero." We present a training point or a test point here.
- $\hat{\mathbf{y}}$: A single output vector, of shape $1 \times k$, where k is the number of output units. These could be regression values or they could symbolize classifications (and you can mix output units of both types). Each training point $\mathbf{x}$ is accompanied by a label vector $\mathbf{y}$, and the goal of training is to make $\mathbf{x}$'s output $\hat{\mathbf{y}}$ be close to $\mathbf{y}$.
- $n^{[l]}$: The number of units (neurons) in unit layer l.
- $W^{[l]}$: A matrix of weights connecting unit layer $l-1$ with unit layer l, of shape $n^{[l-1]} \times n^{[l]}$. This matrix represents the weights of the connections in edge layer l. At edge layer 1, the shape is $d \times n^{[l]}$.
- $\mathbf{b}^{[l]}$: The bias vector for layer l, of shape $1 \times n^{[l]}$.
- $\mathbf{h}^{[l]}$: The output of edge layer l. This is a vector of shape $1 \times n^{[l]}$.
- $\sigma^{[l]}(\cdot)$: The nonlinear activation function applied at layer l.

A fully-connected layer l is a function

$$\phi^{[l]}(\mathbf{h}^{[l-1]}) = \sigma^{[l]}(\mathbf{h}^{[l-1]} W^{[l]} + \mathbf{b}^{[l]}) = \mathbf{h}^{[l]}.$$

The output $h^{[l]}$ of edge layer l is computed then used as the input to edge layer $l+1$. (At edge layer 1, $h^{[0]}$ is simply the input vector $\mathbf{x}$.) A neural network is thus a composition of functions. We want to find weights such

that the network maps each training point $\mathbf{x}$ to its label $\mathbf{y}$.

In a multiclass classification problem with more than two classes, it is common to set k equal to the number of classes and have each output unit represent a true/false value for one class. This is called one-hot encoding. A one-hot encoded output unit used for classification might employ the labels

$$y_i = \begin{cases} 1, & \text{if } \mathbf{x} \text{ is from class } i, \\ 0, & \text{otherwise.} \end{cases}$$

For example, for a classification problem with 3 classes, the label for a training point in class 3 might be $(0, 0, 1)$ and the label for a training point in class 2 might be $(0, 1, 0)$. However, the precise values you choose ought to depend on the activation function σ. Moreover, you might get better results by using less extreme labels, such as 0.15 and 0.85, in lieu of 0 and 1, if σ is the logistic (sigmoid) function. If you have only two classes, there is usually no advantage to one-hot encoding; one output unit for the class label should suffice.

0.3 The Neural Network Package

In the codebase we have provided, each layer is an object with a few relevant attributes.

- `parameters`: An OrderedDict containing the weights and biases of the layer.
- `gradients`: An OrderedDict containing the gradients of the loss with respect to the weights and biases of the layer, with the same keys as parameters.
- `cache`: An OrderedDict containing intermediate quantities calculated in the forward pass that are useful for the backward pass.
- `activation`: An Activation instance that is the activation function applied by this layer.
- `n_in`: The number of input units.
- `n_out`: The number of output units.

You will pass the layer a parameter that selects an activation function from those defined in `activations`. This will be stored as an attribute of the layer, which can be called as `layer.activation()`. The forward and backward passes of the layer are defined by the following methods.

- `forward`: This method takes as input the output X from the previous layer (or input data). This method computes the function $\phi(\cdot)$ from above, combining the input with the weights `W` and bias `b` that are stored as attributes. It returns an output `out` and saves the intermediate value `Z` to the cache attribute, as it is needed to compute gradients in the backward pass.
- `backward`: This method takes the gradient of the downstream loss (*i.e.*, the gradient of the loss with respect to the output of the current layer) as input. Then it uses the `cache` from the forward pass to compute the gradient of its output with respect to its inputs and weights. Via chain rule, it then returns the gradient of the loss with respect to the input of the layer.

Each activation function has a similar (but simpler) structure.

1 Implementing Feed-Forward Neural Network Using `NumPy`

In this section, you implement a feed-forward neural network (NN) that will perform classification on the IRIS dataset using functions in the `NumPy` library only.

1.1 Basic Network Layers

In this question you will implement the layers needed for basic classification neural networks. For each part, you will be asked to 1) derive the gradients, 2) write the matching code, 3) pass the tests.

Keep in mind that your solutions to all layers must operate on mini-batches of data and should not use loops to iterate over the training points in a mini-batch.

1.1.1 ReLU

ReLU is a very common activation function that is typically used in the hidden layers of a neural network and is defined as

$$\sigma_{\text{ReLU}}(\gamma) = \begin{cases} 0, & \text{if } \gamma < 0, \\ \gamma, & \text{otherwise.} \end{cases}$$

Note that the activation function is applied elementwise to a vector or matrix input.
Before you implement the ReLU activation function in `activations.py`, answer the following question.

1.1.1.a Derive the gradient of the downstream loss with respect to the input of the ReLU activation function, Z. You must arrive at a solution in terms of $\partial L/\partial Y$, the gradient of the loss w.r.t. the output of ReLU $Y = \sigma_{\text{ReLU}}(Z)$, and the batched input Z, *i.e.*, where $Z \in \mathbb{R}^{m \times n}$. Include your derivation in your writeup. [**HINT:** you are allowed to use operations like elementwise multiplication and/or division!]

Now, implement the forward and backward passes of the ReLU activation in the script `activations.py`. Do not iterate over training examples, use batched operations.

1.1.2 Fully-Connected Layer

In this section, you will implement the forward and backward passes for the fully-connected layer in the `layers` script. You will write the fully-connected layer for a general input $\mathbf{h}$ that contains a mini-batch of m examples with d features. When implementing a new layer, it is important to manually verify correctness of the forward and backward passes.
Before you implement the forward and backward passes of the fully-connected layer in `layers`, answer the following question.

1.1.2.a Derive the gradients of the loss $L \in \mathbb{R}$ with respect to weight matrix $W \in \mathbb{R}^{n^{[l]} \times n^{[l+1]}}$, i.e., $\partial L/\partial W$, and with respect to the bias row vector $\mathbf{b} \in \mathbb{R}^{1 \times n^{[l+1]}}$, i.e., $\partial L/\partial \mathbf{b}$, in the fully-connected layer. You will also need to take the gradient of the loss with respect to the input of the layer $\partial L/\partial X$, which will be passed to lower layers, where $X \in \mathbb{R}^{m \times n^{[l]}}$ is the batched input. Again, you must arrive at a solution that uses batched X and Z. Please express your solution in terms of $\partial L/\partial Z$, which you have already obtained in question 1.1.1.a, where $Z = XW + \mathbf{1}^\top \mathbf{b}$ and $\mathbf{1} \in \mathbb{R}^{1 \times m}$ is a row of ones. Note that $\mathbf{1}^\top \mathbf{b}$ is a matrix whose each row is the row vector $\mathbf{b}$. So, we are adding the same bias vector to each sample during the forward pass: this is the mathematical equivalent of numpy broadcasting. Include your derivations in your writeup.

Now, implement the forward and backward passes of `FullyConnectedLayer` in `layer.py`. The backward method takes in an argument `dLdY`, the gradient of the loss with respect to the output of the layer, which is computed by higher layers and backpropagated. This should be incorporated into your gradient calculation. Do not loop over training points; use batched operations.

1.1.3 Softmax Activation

Next, we need to define an activation function for the output layer. The ReLU activation function returns continuous values that are (potentially) unbounded to the right. Since we are building a classifier, we want to return probabilities over classes. The softmax function has the desirable property that it outputs a probability distribution. That is, the softmax function squashes continuous values into the range $[0, 1]$ and normalizes the outputs so that they add up to 1. For this reason, many classification neural networks use the softmax

activation. The softmax activation takes in a vector $\mathbf{s}$ of k un-normalized values $s_1, \ldots, s_k$ and outputs a probability distribution over the k possible classes. The forward pass of the softmax activation on input s_i is

$$\sigma_i = \frac{e^{s_i}}{\sum_{j=1}^{k} e^{s_j}},$$

where k ranges over all elements in $\mathbf{s}$. Due to issues of numerical stability, the following modified version of this function is commonly used.

$$\sigma_i = \frac{e^{s_i - m}}{\sum_{j=1}^{k} e^{s_j - m}},$$

where $m = \max_{j \in \{1,\ldots,k\}} s_j$. We strongly recommend implementing the latter formula.

Before you implement the forward and backward passes of the softmax activation in `activations.py`, answer the following question.

1.1.3.a For a single training point, derive $\partial \sigma_i / \partial s_j$ for an arbitrary (i, j) pair, *i.e.* derive the Jacobian Matrix

$$\begin{bmatrix} \frac{\partial \sigma_1}{\partial s_1}, & \frac{\partial \sigma_1}{\partial s_2}, & \ldots, & \frac{\partial \sigma_1}{\partial s_k} \\ \frac{\partial \sigma_2}{\partial s_1}, & \frac{\partial \sigma_2}{\partial s_2}, & \ldots, & \frac{\partial \sigma_2}{\partial s_k} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \sigma_k}{\partial s_1}, & \frac{\partial \sigma_k}{\partial s_2}, & \ldots, & \frac{\partial \sigma_k}{\partial s_k} \end{bmatrix}.$$

Include your derivation in your writeup. You do not need to use batched inputs for this question; an answer for a single training point is acceptable.

Now, implement the forward and backward passes of the softmax activation in `activations.py`. We recommend vectorizing the backward pass for efficiency. For this question only, you are allowed to use a "for" loop over the training points in the mini-batch.

1.1.4 Cross-Entropy Loss

For this classification network, we will be using the multi-class cross-entropy loss function

$$L = -\mathbf{y} \cdot \ln \hat{\mathbf{y}}$$

where $\mathbf{y}$ is the binary one-hot vector encoding the ground truth labels and $\hat{\mathbf{y}}$ is the network's output, a vector of probabilities over classes. Note that $\ln \hat{\mathbf{y}}$ is the elementwise natural log $\hat{\mathbf{y}}$, and $\cdot$ represents the dot product between $\mathbf{y}$ and $\ln \hat{\mathbf{y}}$. The cross-entropy loss calculated for a mini-batch of m samples is

$$L = -\frac{1}{m} \sum_{i=1}^{m} \mathbf{y}_i \cdot \ln \hat{\mathbf{y}}_i.$$

Let $Y \in \mathbb{R}^{m \times k}$ and $\hat{Y} \in \mathbb{R}^{m \times k}$ be the one-hot labels and network outputs for the m samples, stacked in a matrix. Then, $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$ in the expression above are just the i_{th} rows of Y and $\hat{Y}$.
Before you implement the forward and backward passes of the cross-entropy cost, answer the following question.

1.1.4.a Derive $\partial L / \partial \hat{Y}$ the gradient of the cross-entropy cost with respect to the network's predictions, $\hat{Y}$. You must use batched inputs. Include your derivation in your writeup.
[**HINT:** You are allowed to use operations like elementwise multiplication and/or division!]

Now, implement the forward and backward passes of `CrossEntropyLoss` in `loss.py`. Do not iterate over training examples; use batched operations.

The following is the mark breakdown for Section 1.1:

(i) Questions are answered correctly: 16 marks (4 mark per question)

(ii) Code content is organized well and annotated with comments: 10 marks

(iii) Test file successfully runs the tests: 74 marks (1 mark per test)

1.2 Two-Layer Fully Connected Networks

Now, you will use the methods you've written to train a two-layer network (also referred to as a one-hidden layer network). You will use the Iris Dataset, which contains 4 features for 3 different classes of irises.

1.2.1 Implementing Fully Connected Layer

Fill in the forward, backward, and predict methods for the `NeuralNetworkModel` class in `model.py`. See the provided Google Colab notebook to find out how to define the parameters of your network. We have provided you with several other classes that are critical for the training process.

- The data loader (in `util.py`), which is responsible for loading batches of data that will be fed to your model during training. You may wish to alter the data loader to handle data pre-processing.
- The stochastic gradient descent optimizer (in `optimizer.py`), which performs the gradient updates and optionally incorporates a momentum term.
- A logger (in `uti.pyl`), which saves hyperparameters and learned parameters and plots the loss as your model trains.

Outputs will be saved to the folder `experiments/`. You can change the name of the folder a given run saves to by changing the parameter called model name. Be careful about overwriting folders; if you forget to change the name and perform a run with identical hyperparameters, your previous run will be overwritten!

1.2.2 Hyperparameter Tuning (in a very small scale)

Train a 2-layer neural network on the Iris Dataset while varying the following hyperparameters.

- Learning rate
- Hidden layer size (number of units per hidden layer)

answer the following question.

1.2.2.a try at least 3 different combinations of these hyperparameters. Report the results of your exploration, including the values of the parameters you explored and which set of parameters gave the best test error. Provide plots showing the loss versus iterations for your best model and report your final test error.

The following is the mark breakdown for Section 1.2:

(i) Question 1.2.2.a is answered correctly: 10 marks

(ii) Code content is organized well and annotated with comments: 5 marks

(iii) The three method, forward, backward, and predict, are implemented correctly: 30 marks (10 mark per method)

2 Implementing Neural Networks Using `PyTorch`

This part of the assignment is designed to get you familiar with how in the real world we train neural network systems. It isn't designed to be difficult and is based on a tutorial[1] by `PyTorch` itself.

[1] `https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html`

2.1 Understanding PyTorch

PyTorch is based on the "autograd" paradigm. Essentially, you perform operations on multi-dimensional arrays like in NumPy, except PyTorch will automatically handle gradient tracking. To help you understand the full pipeline of creating and training a model in PyTorch, we provided a short induction to PyTorch in the provided Google Colab Notebook. Content in this section closely follows this PyTorch tutorial[2]. Feel free to re-use code from this section in the assigned tasks.

You will also get practice being a real ML engineer by reading documentation and using it to implement models. The notebook covers an outline of what you need to know – we are confident that you can find the rest on your own.

2.2 Implementing a Multi-Layer Perceptron Model Using PyTorch

In this section, you will train a multi-layer perceptron model on an MNIST-like dataset.

Some pytorch components you should definitely use:

- `nn.Linear`
- Some activation function like `nn.ReLU`
- `nn.CrossEntropyLoss`: if you choose to use `nn.CrossEntropyLoss` or `F.cross_entropy`, DO NOT add an explicit softmax layer in your neural network. PyTorch devs found it more numerically stable to combine softmax and cross entropy loss into a single module and if you explicitly attach a softmax layer at the end of your model, you would unintentionally be applying it twice, which can degrade performance.

Your underlying model must be fully connected or dense, and may not have convolutions etc., but you can use anything in `torch.optim` or any layers in `torch.nn` besides `nn.Linear` that do not have weights.
Your deliverables are as follows:

2.2.a Code for training an MLP on MNIST (you should upload your ipynb notebook and also provide your code for this section in `PA3_qa.pdf` (as a screen shot, or in latex typesetting, etc.)).

2.2.b A plot of the training loss and validation loss for each epoch of training after trainnig for at least 8 epochs.

2.2.c A plot of the training and validation accuracy, showing that it is at least 82% for validation by the end of training.

The following is the mark breakdown for Section 2:

(i) Question 2.2.b and 2.2.b are answered correctly: 14 marks (7 marks each)

(ii) Code content is organized well and annotated with comments: 1 marks

(iii) Correct implementation of training an MLP on MNIST: 30 marks

3 Discussion

Please answer the following short questions so we can improve future assignments.

3.a How much time did you spend on each part of this assignment? (5 mark)

3.b Any additional feedback? How would you like to modify this assignment to improve it? (5 mark)

[2] https://pytorch.org/tutorials/beginner/basics/intro.html

4 Turning It In

You need to submit your version of the following files:

- `activations.py`, `layers.py`, `loss.py`, `model.py`, `util.py`
- The modified Google Colab notebook named as `PA3.ipynb`
- `PA3_qa.pdf` that answer questions related to the implementations.
- The cover file with your name and student ID filled. If you use the LaTeX template, you don't need to include the cover file separate;y.

Please pack them into a single folder, compress into a .zip file and name it as PA3.zip.