OCaml Machine Learning Library

By Arjun Bhalla, Akhil Bhandaru, Sam Thomas, and Richard Greenbaum

Welcome to our machine learning library! This file will go over the main features of our program and outline how to use the code we have written. We will start by explaining how to train networks from the example data sets that are provided in the package, and then move on to instructions for loading new data sets and creating new layer types.

The example folder has several additional folders, each of which contains the files needed to train a neural network on a given dataset and evaluate the network created. The “basic” folder contains code for several simpler data sets (‘and’ gates, ‘or’ gates, etc), each of which can be found in the data folder. The iris and yeast folders contain the files and data needed to train networks on more complex data sets that can be found online. Each of these folders should have a main file which handles the set-up, network creation, training, and evaluation. User edits to the main file can be broken down into the following steps:

**Layer type:** At the top of the file, module PLayer is defined according to the type of layer and the type of activation function that should be used in the network. We provide implementations for perceptron layers in perceptron.ml and perceptron layers with biases on each node in perceptron\_bias.ml. We provide modules for the Sigmoidal and Softmax activation functions in differentiable.ml.

**Data import:** Data is imported using the path to the data file (in our cases, they are all csv files) and converted into a data matrix.

**Network creation:** The network is created by ‘consing’ layers onto each other, and the user can specify the number of layers they want in the network as well as the number of nodes they want in each layer. The layer inputs variable must equal the number of nodes in the previous layer (or the dimension of the data if the layer is the first layer) and the outputs variable is equal to the desired number of nodes in that layer.

**Loss function:** The loss function is then chosen with the line “open! Loss.<name of desired loss function>”. The loss functions provided in the Loss.ml file are SquaredDifference, Hinge, and CrossEntropy.

**Training:** The train function has several parameters which allow the user to customize the training of their neural network. These parameters are outlined below:

* Epsilon: Specifies the threshold in the training process where, if the loss function after a given epoch drops below the threshold, training halts and the network is considered to be sufficiently well trained.
* Learning: The learning rate specifies how quickly the weights in the network should be altered by each set of back-propagated deltas. A higher learning rate will cause faster changes in the network weights.
* Max Epoch: Sets an upper bound on the number of epochs that are allowed to be run in a given round of training. If this number is reached, training is halted.
* Gradient Type: (Stochastic, Batch, or MiniBatch <int>) Specifies the number of data vectors that should be sent through the network before deltas are back-propagated through the network and weight updates are made. In Stochastic gradient descent, the deltas are back-propagated through the network after each data vector is sent through. In Batch gradient descent, the entire data set is passed through the network, and then the aggregate delta vector is back-propagated through the network. In MiniBatch <n>, n data vectors are passed through the network, and then the aggregate delta vector from those n data vectors is back-propagated through the network.
* Dropout: Dropout is a form of network normalization that is intended to prevent overfitting of the network to the data and to reduce dependency between neurons. The dropout parameter is set to a float value between 0 and 1, representing the probability that any given node should be dropped (have the activation set to 0) for each epoch. This thins the network by reducing the number of nodes in each epoch, causing the time for each epoch to decrease but the number of epochs to increase. The general concept is that, by randomly removing nodes in each epoch, nodes can be less reliant on each other and become more likely to develop their own unique contribution to classification. The optimal dropout value is generally considered to be at or around .5.

\*Interesting side note: the process by which dropout increases the effectiveness of neural network training has strong ties to the effectiveness of sexual reproduction compared to asexual reproduction. The constant mixing and matching of different genes across generations causes each gene to develop functionality in its own right, or be discarded over time. The most complex organisms all engage in sexual reproduction.

* Output file: the name of the file in which the result of training should be stored. The output file maps each epoch number to the magnitude of the loss vector of that epoch.

**Training and evaluation:** Now that the main file is all set up, its time to train a network! From the directory of the Makefile, run the command ‘make run ex=<name of folder in examples>’. For example, ‘make run ex=iris’ is a valid command. The path to the main file should be displayed, followed by the line “Starting training…” and then the loss for the current epoch and the current epoch number. Training should terminate on its own, either by reaching the epsilon loss threshold or the max epoch threshold. At the termination of training, the network is evaluated on the test data set and the accuracy of the network is displayed. To manually halt training, press the control key and the ‘c’ key together. The current network at that point will be evaluated, and its accuracy will be displayed.

**Saving and loading networks:** The user is also able to save a network that they have trained and load a network that they have saved. To save a network after training it, add or uncomment the line “PNet.save net <desired directory name>” immediately after the network is trained in the main file. A directory will then be added to the main project folder which contains file encodings of each layer. To load a network that has already been saved, add or uncomment the line “let net = PNet.load <directory name>.ann in” in the main file right before the network is trained. If a network is loaded and the code that trains the network is commented out, the loaded network will be evaluated and the accuracy will be printed to the terminal. Otherwise the loaded network will continue to be trained, as if training had been halted and is now being resumed. If training is halted using a ‘control c’ command and the PNet.save command is present, the current network at the time of the halt is saved in the same way it would have been if training had ended gracefully.

**Adding new datasets:** To add a new data set to train on and test, create a new directory and put it in the examples folder. The folder will need to have a main.ml file, a dune file (that can be copied from any of the other folders in examples), and a copy of the data. If the data is in csv format, each row should have the classification value in the column directly after the data. It can be helpful to write a seperate script to download and save the data in this format, but this is not necessary. The main file will be very similar to the main files in the provided data sets. If the data is in a csv file, the Parse.matrix\_of\_file function can be used to import the data into a matrix. Otherwise the user will have to write their own function to import the data from the file and separate the data from the classifications. The rest of the parameters in the main file can be altered as described above, and the file can be run by the same command: make run ex=<directory name>.

**Creating new layer types:** A layer is simply a functor that takes in a module of type Differentiable.sig (a differentiable function that will be used as the activation function) and implements all of the functions specified in the layer.ml file. The user is free to write their own layer implementations in OCaml and use them in the example main files in the same way that perceptron.ml and perceptron\_biases.ml are used.