Readme | Assignment 1

Members: Divyansh Rastogi (2019464) & Rupanshu Yadav (2019475)

Perceptron

Perceptron training algorithm is implemented in the module `perceptron`

Module files:

- `base.py` Contains the base class for perceptron
- `pta.py` Contains the Perceptron training algorithm (PTA) class

The PTA class inputs the whole dataset, and checks for inconsistent samples on each iteration. When convergence is not possible, it is detected on checking wether one is stuck in a weight update repeat loop using thresholding of the L2 norm of the new weight with the previous weights.

The driver code is in `main.py`.

The current configuration sets random intialization (numpy random seed 7) and bias for the perceptron.

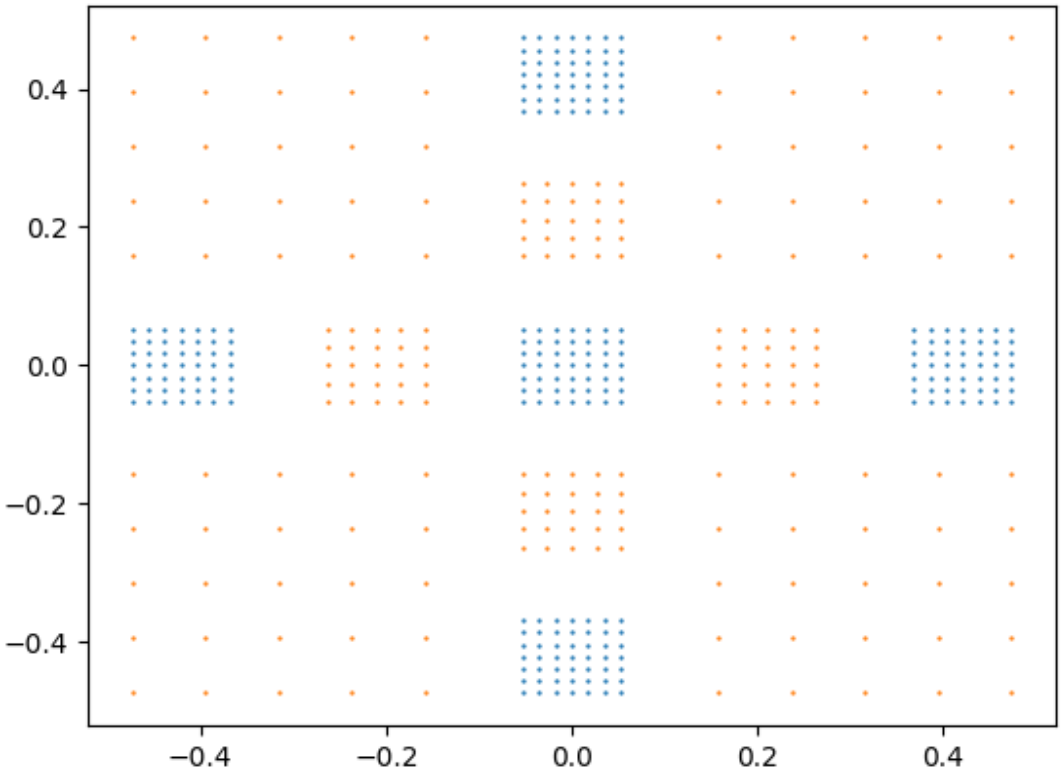
Madaline

Madeline training algorithm is implemented in the module `madeline`

Module files:

- `adaline.py` Contains the base class for the adaline perceptron
- `madaline.py` Contains the base class for the Madaline MLP and the Linear layer

The dataset is generated with class imbalance in mind, to balance the positive and negative classes. Furthermore, the dataset is normalized and mean centered.



The following MLP architecture is used for training:

```
model = MLP(
    Linear(2, 64),
    Linear(64, 32),
    Linear(32, 16),
    Linear(16, 8),
    Linear(8, 1),
)
```

The learning rate is `0.001` with Xavier random initialization. Total neurons used are `121` to achieve a `99.55` accuracy i.e. only `2` misclassifications out of `445` points trained over `10` epochs.

The madaline algorithm is run with an additional heuristic over sorting with the minimum affine value:

> The adaline change for the suitable node (whose change results in model output flip) is committed only and only when it also supplements the overall model accuracy.

The driver code is in `main.py`.

The weights are pickle-stored in `weights/default.pkl`

The above dataset cannot be classified with ≤ 2 hidden neurons, as the architecture is a linear combination (output layer) of the thresholding of two linear combination (hidden layers). This implies an “and” operation between the hidden layers. Thus, given the figure for classification is not convex, it cannot be modelled within a single “and” operation.

With a practical run, we get the following performance of constant and non converging `55.05` accuracy.

MLP

Perceptron training algorithm is implemented in the module `MLP`

Module files:

- `ActivationFn.py` Various activation functions and their derivatives
- `ActivationLayer.py` Base class for activation layer
- `Layer.py` Base class for MLP Linear Dense Layer
- `Loss.py` Loss metrics for MLP such as MSE
- `NN.py` Base class for MLP Model
- `toolkit.py` Main runner class containing model architecture
- `train.py` Main training loop

The data was normalized between 0 and 1.

GD optimizers

- `optim.py` Contains a class for every optimizer, instance of which needs to be passed into NN for optimzation

SGD with Momentum

- SGD with momentum uses the previous descent iteration and accumulates a ‘momentum’ paremter which is added to the

update gradient

- As a result the optimizer converges at a faster rate but fluctuates more around the optimal value

NAG SGD

- NAG is similar to SGD with momentum however the gradient is calculated after the momentum is added rather than before.
- The convergence in this case is similar to the of SGD with slightly less fluctuations. However, in a more general case NAG SGD converges faster and often oscilliates less around the optimial value

Adagrad

- The adaptive gradient allows the learning rate to change depending on the parameters. It performs greater updates for infrequent parameters and smaller updates for frequently parameters.
- Hence, the convergence graph for Adagrad is much more stable. Although the final accuracy after 100 epochs is not as high as NAG or SGD momentum

RMSProp

- RMS prop handle the problem of accumulating squared graidents in Adagrad by applying a decay. Which makes convergence faster and stable in certain cases.
- In this case, the convergence for RMSProp is much faster and stable.

Adam

- Adam introduces the concept of momentum to RMSProp, that is, in addition to storing learning rates for each of the parameters it also stores momentum changes for each of them separately.
- Amongst all the optimizer the most stable convergence was observed with Adam.

Contribution

Both members were actively involved in both of the parts. Although part 1 was primarly handled by Divyansh and part 2 was primarly handled by Rupanshu.