Assignment 1, CS698U: Multi layer perceptron

Nishit Asnani, 14433

January, 2017

1 Objective

To implement and understand the backpropagation algorithm by building a hand written digit classifier for the MNIST dataset with standard split.

2 Setup

The code has been broken up into three files:

- *input.py*: This file is used for loading the MNIST dataset from Yann Lecun's website and storing it in a manner usable by our MLP. For this purpose, input code from **TensorFlow** tutorial's *convolutional.py* [?] has been used as it is, so that obtaining the dataset in a convenient format becomes easy.
- *mlp.py*: This file defines the class `Multi_layer_perceptron`, which forms the crux of the assignment. An object of this class is an MLP model, with a given number of hidden layers, number of neurons in each hidden layer, and activation function as arguments to the constructor. The various class members have been described later on in the report.
- *calling.py*: This forms a python script that loads the dataset by calling the relevant functions in *input.py*, builds models using *mlp.py* and trains them, recording the accuracy and loss statistics after each epoch.

3 Class `Multi_layer_perceptron`

- The constructor takes number of hidden layers, number of nodes (neurons) in each hidden layer (as a list), and the activation function to be used (0 for *tanh* and 1 for *ReLU* as input. It initializes the relevant class members, and also initializes dictionaries corresponding to the parameters (weights), node activations, gradients, square gradients (for ADAM), and records of the previous gradient update (for GD with momentum). It also initializes geometrically decreasing estimates for β_1 and β_2, as required by ADAM.
- `forward_pass` computes a forward pass through the network on a given input image and stores the node activations at each node.
- `nonlinear` is a routine to compute non-linearity on a list of inputs according to the activation function specified.
- `softmax` is a routine that computes the softmax probabilities of each of the output states, given the node activations in the final layer for an input image.

- `compute_loss` computes the softmax loss for a given input and adds that to the class member *loss*.
- `compute_gradients` takes the true label of the processed image, and computes the back propagated error at each node, as well as the gradient of the loss with respect to each parameter in the network. The sum of all gradients over a mini batch is constrained to be at most 5 times the batch size, to cure the problem of exploding gradients.
- `apply_gd_optimizer` uses simple gradient descent to update the weights.
- `apply_momentum_optimizer` applies gradient descent with a standard momentum term to update the parameters. It uses the dictionary storing previous weight updates for this purpose.
- `apply_adam_optimizer` applies the ADAM optimizer with standard settings of β_1 and β_2 by default, but these can be passed as an argument by the *train* function. It uses the accumulated average gradient and accumulated squared average gradient for this purpose.
- `train` takes train and validation data, and a choice of optimizer (sgd, momentum or adam) and performs training for one epoch on the training data, followed by a test on the validation dataset. It returns the loss and accuracy on the validation dataset. Except for standard GD, all the optimizers work for a stochastic minibatch setting, since it converges faster.
- `numerical_gradients` takes a set of data points and computes backpropagated error of the weights on those points. Then, one parameter is perturbed slightly, and its effect on the new loss is seen after a forward pass. This is used to compute the numerical gradient of the loss with respect to that parameter, and this is done for all the parameters one after another.

4 Experiments and Findings