IST 597: Foundations of Deep Learning **(Due: 02/09/22)**

Assignment #00010

Instructor: **Dr. Lee Giles** *TA:* **Ankur Mali** *Name: , PSUID:*

Created by Ankur Mali

Course Policy: Carefully read all the instructions, before you start working on the assignment

- Please typeset your submissions in any LaTeX template, give maximum explanation for each sub-problems. Please include your name and PSUID with submission along with date and time on the first page. {**For this problem you can also attach solution image**}
- Assignments are due before class at 02:29:59 pm {Please check the due date on the **Official course** webportal} Portal.
- Please avoid single line answers, submissions without any explanations would receive **0** points.
- Late assignments will suffer 50 percent points reductions per day, formula is $x/2$, where x=number of days and counter will start at 02:30:00 pm.
- All source materials must be cited. The University Academic Code of Conduct will be strictly enforced.
- We will be creating Canvas submission page for this. Submit all files on Canvas.
- All queries related to Assignment should have a subject line IST597:Assignment00010 Queries

Problem 1. Design Multi-layer Perceptron (MLP) to classify objects and digits (7 points)

1 Multi Layer Perceptron

In this assignment you will implement multilayer perceptron, with 2 hidden layers. A multilayer perceptron is a feed forward neural network architecture composed of multiple layers of stacked perceptron units. See figure 1

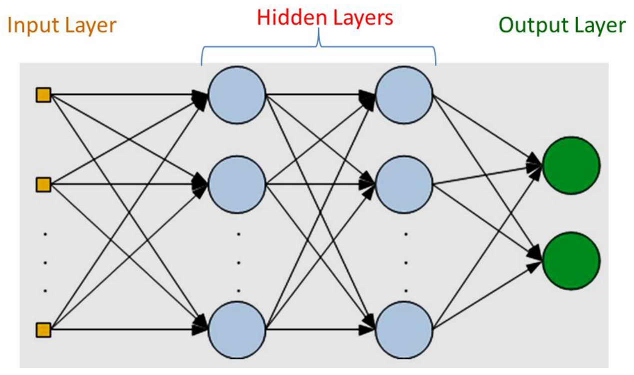

Figure 1: Multi Layer Perceptron

1.1 Forward Pass

In forward pass we compute the activation of all the neurons as:

$$a_j^{(l+1)} = \phi(\sum_i w_{ij}^{(l+1)} a_i^{(l)} + b_j^{(l+1)})$$

Here,

$a^{(l)}$:= activation of layer l.

For $l = 0$, activations ($a^{(0)}$) are the inputs.

$w_{ij}^{(l)}$:= is the weight from i^{th} neuron in layer $l-1$ to j^{th} neuron in layer l

ϕ := non linear activation function

In matrix form, it can be written as :

$$\mathbf{a}^{(l+1)} = \phi(W^T \mathbf{a}^{(l)} + \mathbf{b}^{(l+1)})$$

Note: To better understand each module, try to divide them into pre and post activation. To get post activations, we can use various non-linear activation functions such as Elu, Selu, LeakyRelu. In this excercise you will use Relu as non-linear activation, thus ϕ can can be represented as follows.

$$\phi(x) = \begin{cases} x \; if \; x > 0 \\ 0 \; otherwise \end{cases}$$

Note: Neural network will output set of probabilities or the confidence for each class. One can further explore importance of each class and created a weighted version of softmax. Remember: Everything should be differentiable when model is trained using backprop, when dealing with ,Multi-label sigmoid is the default choice and in scenarios when you have multi-class you will go with softmax.

Therefore for the the last layer, which is also known as output layer, the non linear function depends on the task at hand. For binary classification, logistic function (σ) is applied:

$$\phi(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

For multiclass classification, SoftMax function is used:

$$\phi(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

1.2 Cost Function

Cost function provides us with a measure of difference between the predicted output of the neural network and the actual output. Cost function is derived by minimizing the negative log likelihood of the output given the input. The cost function is parameterized by the weights the of neural network. For classification tasks, the function evaluates to cross entropy:

$$L(W) = -\sum_j y_j \log(h_j)$$

where y_j are the components of vectorized ground truth, and h_j are the corresponding outputs from the neural network.

Similarly, for regression tasks, the cost function evaluates to mean squared error (MSE)

$$L(W) = \frac{1}{n} \sum_j^n (y_j - h_j)^2$$

1.3 Backward Pass

In backward pass, our aim is to calculate the gradients of the cost function with respect to all the parameters of the neural network.

Consider the forward pass equations:

$$z_j^{(l+1)} = \sum_i w_{ij}^{(l+1)} a_i^{(l)} + b_j^{(l+1)}$$

Taking partial derivative, we get:

$$\frac{\partial z_j^{(l+1)}}{\partial w_{ij}^{(l+1)}} = a_i^{(l)}$$

and,

$$a_j^{(l+1)} = \phi(z_j^{(l+1)})$$

$$\frac{\partial a_j^{(l+1)}}{\partial w_{ij}^{(l+1)}} = \frac{\partial \phi}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial w_{ij}^{(l+1)}} = a_i^{(l)} \frac{\partial \phi}{\partial z_j^{(l+1)}}$$

This allows us to calculate the partial derivatives with respect to parameters using chain rule across the layers of neural network.

2 Training

A training cycle of a neural network involves a forward pass, a backward pass and parameter update. The parameters are updated in each iteration as follows:

$$\hat{w}_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial L(W)}{\partial w_{ij}^{(l)}}$$

This is called Gradient Descent Algorithm as the parameters are updated in the direction maximum negative slope in cost function.

2.1 Stochastic Gradient Descent

In a traditional Gradient Descent Algorithm, entire train set is first passed through forward pass, backward pass and then parameters are updated using gradients aggregated over whole train set. This becomes impossible with very large datasets due to memory constraints. Hence we use Stochastic Gradient Descent. Here we divide the training set into smaller batches of data and parameter update is performed for each batch. This does not necessarily follows the most optimum path for optimization but eventually converges. It sometimes provides better results then Gradient Descent as noise introduced due to mini batches provides an alternate zic-zac path around the global gradient and sometimes avoids problems with diminishing global gradients.

2.2 Early Stopping

Ideally the training continues until the network converges. In practical sense, we define a small value ϵ such that, if the cost function does not reduce at least of ϵ in consecutive iterations, then we stop the training procedure and consider that the training has converged.

$$L(W)_{iter} - L(W)_{iter+1} < \epsilon$$

3 Regularization

Regularization is a way of penalizing the training of neural network model such that we do not end up with extreme parameter values. When the parameters of a neural network take extreme values, the model usually overfits to a certain set of inputs and does not generalize to cover the domain. Two most common types of penalty terms that can be added to cost function are L1 and L2 penalty.

$$L1 := \frac{1}{n}\sum_{i,j,l}^{n} |w_{ij}^{(l)}|$$

$$L2 := \frac{1}{n}\sum_{i,j,l}^{n} (w_{ij}^{(l)})^2$$

This adds the following derivatives to the gradients:

$$\frac{\partial L1}{\partial w_{ij}^{(l)}} = \frac{1}{n}\frac{w_{ij}^{(l)}}{|w_{ij}^{(l)}|}$$

$$\frac{\partial L2}{\partial w_{ij}^{(l)}} = \frac{2w_{ij}^{(l)}}{n}$$

L1 penalty makes the parameters sparse as it forces some parameters to take very small values and only allows few others to survive. It makes the model simpler by removing unwanted parameters. L2 penalty stops parameters from taking very large values. Penalty terms are added to cost function during training.

3.1 Dropout

Dropout is another method of regularizing a model. Instead of adding penalty term to cost function, we directly manipulate the parameter values. At each training step, we choose a fraction of parameters from each layer and set them to zero. This regulates the contribution of each neuron in deciding the final output.

The forward equations can be updated as :

$$m_{ij}^{(l+1)} \sim Bernoulli(p)$$
$$a_j^{(l+1)} = \phi(\sum_i m_{ij}^{(l+1)} w_{ij}^{(l+1)} a_i^{(l)} + b_j^{(l+1)})$$

which updates the derivative calculation as :

$$\frac{\partial z_j^{(l+1)}}{\partial w_{ij}^{(l+1)}} = m_{ij}^{(l+1)} a_i^{(l)}$$

which implies:

$$\frac{\partial a_j^{(l+1)}}{\partial w_{ij}^{(l+1)}} = m_{ij}^{(l+1)} a_i^{(l)} \frac{\partial \phi}{\partial z_j^{(l+1)}}$$

Hence, if a neuron is turned off, all gradients passing through that neuron are also set to 0 while training.

3.2 Normalization

If different input features have values at different scales, e.g. feature 1 has values between 0 and 1 while feature 2 has values between 100 and 10000, then it is good idea to normalize features such that they have 0 mean and unit variance.

$$\mu_i = \frac{1}{m}\sum_{k=0}^{m} x_i^{(k)}$$

$$\sigma_i^2 = \frac{1}{m}\sum_{k=0}^{m} (x_i^{(k)} - \mu_i)^2$$

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

This idea of normalizing data is sometimes extended to normalizing mini-batches rather than the entire train set. That is called Batch Normalization. Batch Normalization can however lead to instability while training.

Other types of normalization include normalizing the features at the output of each layer. That is called Layer Normalization.

Another important normalization technique is normalization of weights (parameters) rather then input or feature maps. Weights are normalized as :

Adding small Gaussian noise to gradients can be helpful in assisting training find new optimization path if it get stuck in a shallow local minima or is rolling slowly on path of low gradients. Small noise provides enough energy to explore the surrounding topology of the cost function.

$$\nabla_w \hat{L} = \nabla_w L + N(0, 0.1)$$

4 Bias-Variance Tradeoff

There is no such thing as perfect training. There is always a tradeoff between how general or how specific we need our neural network to be. If it learns all training examples by heart, but struggles on the test test, the model is said to have high variance. On the other hand if the model shows similar performance on train and test set, but is not good on both, then it is said to have high bias. Figure 2 and Figure 3 shows the behavior of models with high bias or high variance.

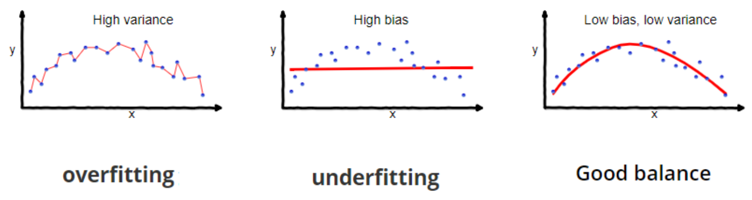

Figure 2: Model fitting

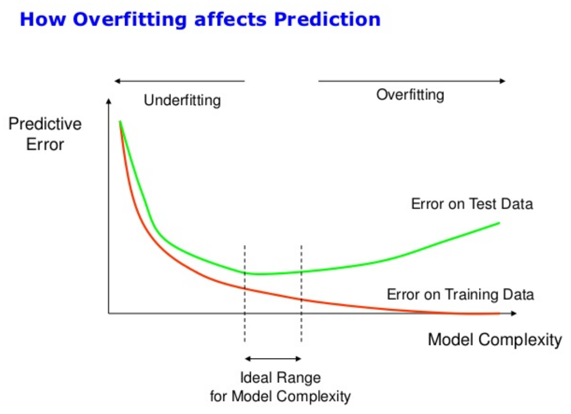

Figure 3: Loss Characteristics of Underfitting and Ovrfitting model

Our goal is to train a model such that it has acceptable and similar performance on train and test sets. It is said to have low bias and low variance.

5 Things to do in this assignment

- Design MLP with 2 hidden layers (Input Layer - 2 hidden Layer - Output layer) to classify objects (fashion MNIST) and digits (MNIST).
- Design regularization approaches, and analyze drop/boost in performance of your model. Report results with atleast 2 regularization variants (droput, L1 penalty, L2, L1+L2, Normalization, Noise). Do you see any tradeoff with respect to bias-variance? Report your findings.
- How did you perform hyper-parameter optimization? Report your approach, also show settings for best model.
- Report your results across multiple trials (minimum 10). Beside accuracy you will also report standard error and plot graph showing variance [1].
- Submit detailed report with graphs, findings and tables. Add GitHub link in your submission.

[1] Note :- Change seeds and run your model (Do inference) K times