BT3041: Analysis and Interpretation of Biological Data

ASSIGNMENT 2

Lecturer: Prof. Srinivasa Chakravarty

Teaching Assistant : Sandeep Nair

UTKARSH KUMAR (ME17B123)

DEPARTMENT OF BIOSCIENCES

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

CHENNAI, 600036

The Assignment involves using Neural Networks (Multilayer Perceptron) on the MNIST data given in .pkl format. The following are the results of the assignment:

1)NETWORK ARCHITECTURE:

Template class is used, which comprises of functions relating for auto differentiation. All optimizers we make should inherit template class. Autodifferentiability will follow and we dont have to chain the gradients explicitly.
The neural network architecture comprises of the function definitions of forward, backward propagation, updating weights, accuracy score, fitting and evaluation. Implementation of auto differentiation similar to actual keras/tensorflow/pytorch packages has been tried.
For this assignment, we only use fully connected OR Dense layers where each input neuron is connected to each output neuron, along with a bias unit.

The equation for such a layer is simply:

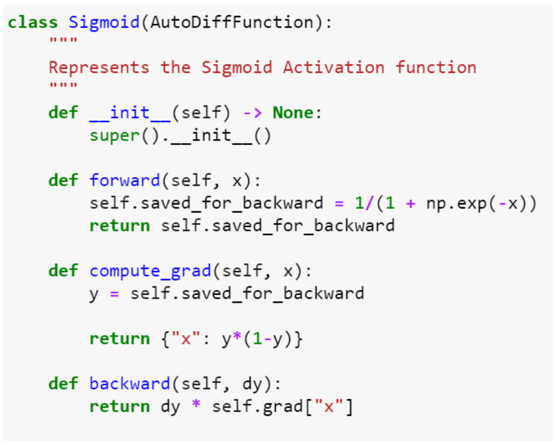
$$y = FullyConnected(x) = wx + b$$

$$\frac{dy}{dw} = x^T \quad \frac{dy}{dx} = w^T \quad \frac{dy}{db} = 1$$

Random and Xavier methods are used in the initialisation of the weights

2) ACTIVATION FUNCTIONS:

The activation functions that are applied, are ReLU and Sigmoid activation function.



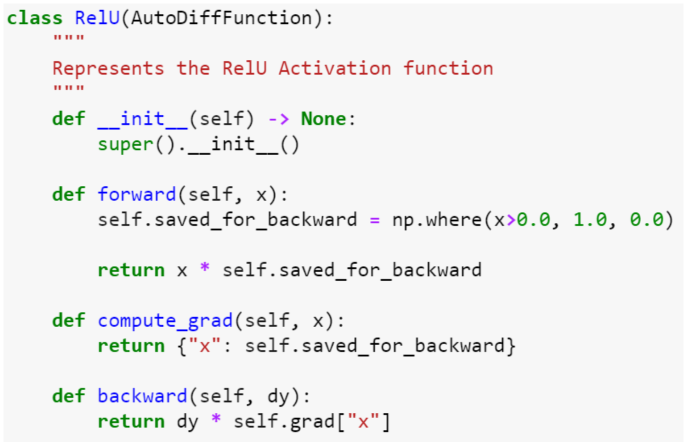
```
class Sigmoid(AutoDiffFunction):
    """
    Represents the Sigmoid Activation function
    """
    def __init__(self) -> None:
        super().__init__()

    def forward(self, x):
        self.saved_for_backward = 1/(1 + np.exp(-x))
        return self.saved_for_backward

    def compute_grad(self, x):
        y = self.saved_for_backward

        return {"x": y*(1-y)}

    def backward(self, dy):
        return dy * self.grad["x"]
```



```
class RelU(AutoDiffFunction):
    """
    Represents the RelU Activation function
    """
    def __init__(self) -> None:
        super().__init__()

    def forward(self, x):
        self.saved_for_backward = np.where(x>0.0, 1.0, 0.0)

        return x * self.saved_for_backward

    def compute_grad(self, x):
        return {"x": self.saved_for_backward}

    def backward(self, dy):
        return dy * self.grad["x"]
```

The formulae for the above codes are given below:

1. Sigmoid activation

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

$$\frac{dy}{dx} = \frac{-e^{-x}}{(1 + e^{-x})^2} = \sigma(x)(1 - \sigma(x))$$

2. ReLU activation

$$y = ReLU(x) = max(0, x)$$

$$\frac{dy}{dx} = \begin{cases} 1 & x \geq 0 \\ 0 & x \leq 0 \end{cases}$$

2) LOSS FUNCTION:

The loss function dictates how good the output of the neural network is. Since we use MNIST dataset, our job is classification and hence we use the Categorical Cross Entropy loss function.

$$L(p, y) = \Sigma_{i=1}^{N} \Sigma_{k=1}^{K} y_{ik} \log p_{ik}$$

$$y_{ik} = \begin{cases} 1 & x \in class - k \\ 0 & else \end{cases}$$

p_{ik} = probability that i^{th} sample falls in k^{th}class

In our implementation, the given loss function is applied along with the activation function for the last layer i.e. Softmax activation. The formula for Softmax activation is:

$$f : [x_1, x_2, \dots x_k] \rightarrow [p_1, p_2, \dots p_k] \text{ such that } p_i = \frac{e^{x_i}}{\Sigma_{k=1}^{K} e^{x_i}}$$

To find the derivative of loss w.r.t input we have apply the chain rule. Let $p(x)$ represent the softmax activation and L represent the loss. Then the expression turns out to be:

$$\frac{\partial L}{\partial x} = \frac{\partial L(p, y)}{\partial p} \frac{\partial p(x)}{\partial x} = p - y$$

3) LEARNING ALGORITHM:

The following optimizers are used in this notebook, all made from scratch

1) SGD

2) Momentum based gradient descent

3) Adam

- SGD:

  Known as Stochastic Gradient Descent Algorithm and is used to optimize. It is selecting data points at each step to calculate the derivatives. SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

```
for i in range(m):
    $\theta_j = \theta_j - \alpha(\hat{y}^i - y^i)x_j^i$
```

- Momentum based gradient descent:

  Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction

γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

- Adam:

  It computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients vt. Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

4) LEARNING RATE:

We have used a learning rate of **10^-3** in all the three optimizers as this can be taken as a standard learning rate and has been used in the Keras library. All the default parameter values are taken from keras optimizers.

5) LOSS AND ACCURACY FOR THE DATASET:

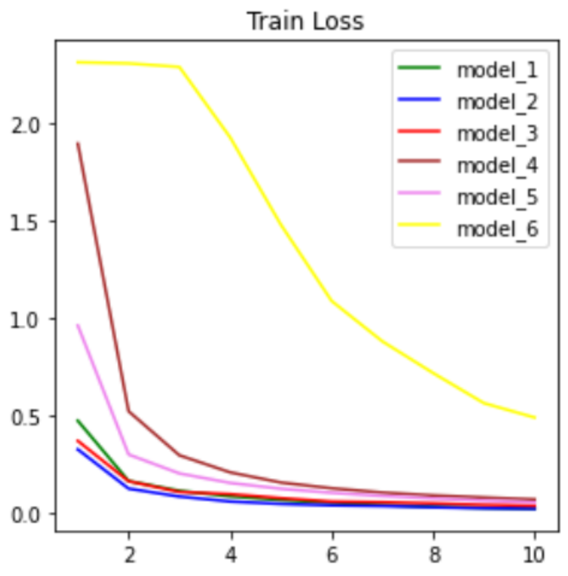
From the implemented code, we finally get the dataset as

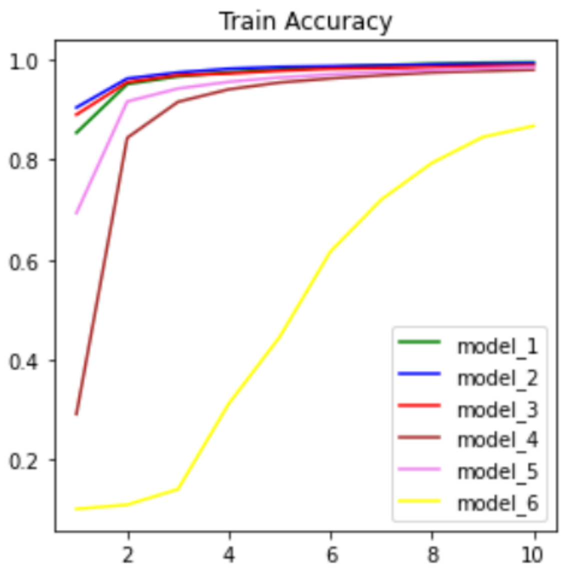
LOSS: 0.08369409363228983

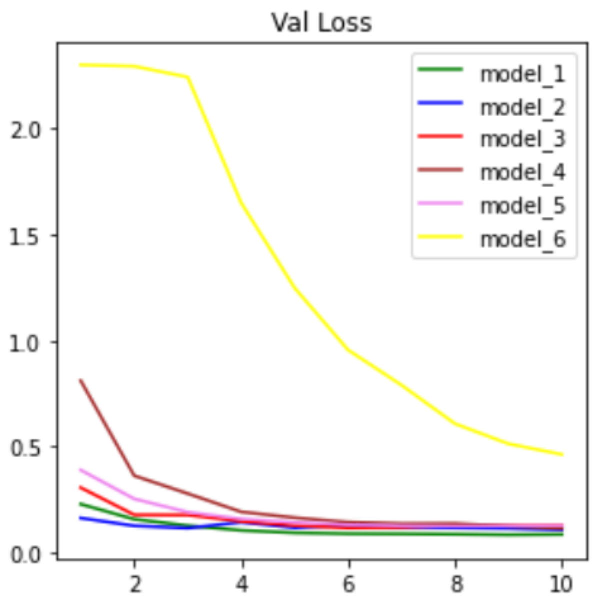
ACCURACY:0.975870253164557

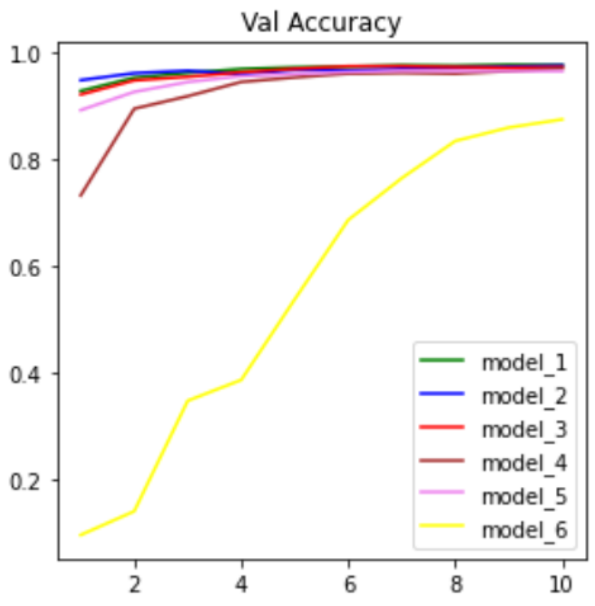
6) RESULT

From the 6 combinations of models that we have run to get the results, we find that the best Activation Model is ReLU and the best Optimizer is Adam. They give the best accuracy values. The plots obtained are given below:


Train Loss
model_1
model_2
model_3
model_4
model_5
model_6


Train Accuracy
model_1
model_2
model_3
model_4
model_5
model_6


Val Loss
model_1
model_2
model_3
model_4
model_5
model_6


Val Accuracy
model_1
model_2
model_3
model_4
model_5
model_6