Report - Classifying Near Earth Objects using Artificial Neural Networks

Meduri Harshith Chowdary
21CS10042
EONN

Implementation

Implementation differences between the PyTorch model and the custom neural network model

1. **Framework Used:**
    - **PyTorch Model:** Utilizes the PyTorch framework, which is a popular deep learning library that provides pre-defined modules for building neural networks and automatic differentiation for gradient computation**.**
    - **Custom Neural Network Model:** Implemented from scratch using basic Python and NumPy, without relying on any deep learning frameworks. This approach requires manual implementation of forward and backward passes, including activation functions and weight updates.
2. **Model Architecture**
    - **PyTorch Model:** Defines the neural network architecture using pre-defined PyTorch modules such as **nn.Linear** for fully connected layers. The model structure is defined within a class by subclassing **nn.Module**.
    - **Custom Neural Network Model:** Specifies the model architecture manually, including the number of layers, neurons in each layer, and activation functions. The weights are initialized randomly, and forward and backward passes are implemented explicitly.
3. **Training Loop:**
    - **PyTorch Model:** Utilizes PyTorch's built-in optimization functions (**torch.optim**) and loss functions (**nn.CrossEntropyLoss**). The training loop involves iterating over data batches using DataLoader and performing forward and backward passes, followed by weight updates using optimization algorithms.
    - **Custom Neural Network Model:** Implements a custom training loop without using optimization libraries. It calculates the loss manually, computes gradients, and updates weights using gradient descent. Batch processing is also implemented manually.
4. **Data Handling:**
    - **PyTorch Model:** Utilizes PyTorch's data handling utilities such as **Dataset** and **DataLoader** for loading and processing data batches. Data normalization and splitting into training and test sets are performed using PyTorch functions or libraries such as sklearn.
    - **Custom Neural Network Model:** Handles data preprocessing, normalization, and splitting using basic Python and **Pandas** functionalities. The dataset is manually

divided into training and test sets, and normalization is performed using custom functions.

5. **Inference and Evaluation:**
    - **PyTorch Model:** Inference and evaluation are performed using PyTorch functions. The model predicts on the test set, and evaluation metrics such as accuracy and classification reports are calculated using **sklearn** functions.
    - **Custom Neural Network Model:** Implements custom functions for prediction and evaluation. After training, the model predicts on the test set, and evaluation metrics such as accuracy and classification reports are calculated manually.

Custom Neural Network Implementation Details

1. ***' calculate_accuracy (predictions, targets) '***
    - Purpose: This function calculates the accuracy of the model's predictions compared to the true labels.
    - Parameters:
        - **predictions**: Predicted labels by the model.
        - **targets**: True labels.
    - Explanation:
        - Calculates the total number of correct predictions by comparing each prediction with the corresponding true label.
        - Divides the total number of correct predictions by the total number of predictions to compute accuracy.
    - Return Value: Accuracy of the model as a floating-point number.
2. ***' sigmoid (x) '***
    - Purpose: Sigmoid activation function to introduce non-linearity in the neural network.
    - Parameters:
        - **x**: Input to the sigmoid function
    - Explanation:
        - Applies the sigmoid function element-wise to each element of the input array.
        - Sigmoid function maps input values to the range **[0, 1].**
    - Return Value: Output of the sigmoid function.
3. ***' sigmoid_derivative (x) '***
    - Purpose: Computes the derivative of the sigmoid activation function for backpropagation.
    - Parameters:
        - **x**: Input to the sigmoid function
    - Explanation:
        - Computes the derivative of the sigmoid function element-wise.

- ➢ The derivative of the sigmoid function is calculated as ***sigmoid(x) * (1 - sigmoid(x)).***
- Return Value: Output of the sigmoid derivative function.

4. ***' relu (x) '***
   - Purpose: Rectified Linear Unit (ReLU) activation function to introduce non-linearity.
   - Parameters:
     - ❖ **x**: Input to the ReLU function.
   - Explanation:
     - ➢ Applies the ReLU function element-wise to each element of the input array.
     - ➢ ReLU function returns the input value if it is positive; otherwise, it returns zero.
   - Return Value: Output of the ReLU function.
5. ***' relu_derivative (x) '***
   - Purpose: Computes the derivative of the ReLU activation function for backpropagation.
   - Parameters:
     - ❖ **x**: Input to the ReLU function.
   - Explanation:
     - ➢ Computes the derivative of the ReLU function element-wise.
     - ➢ The derivative of the ReLU function is 1 for positive input values and 0 for negative input values.
   - Return Value: Output of the ReLU derivative function.
6. ***' compute_loss (predictions, targets) '***
   - Purpose: Computes the loss function for binary classification.
   - Parameters:
     - ❖ **y_true**: True labels.
     - ❖ **y_pred**: Predicted probabilities.
   - Explanation:
     - ➢ Computes the binary cross-entropy loss between the true labels and predicted probabilities.
     - ➢ The loss function penalizes the model based on the difference between true labels and predicted probabilities.
   - Return Value: Loss value.
7. ***' initialize_weights (layers) '***
   - Purpose: Initializes the weights of the neural network randomly.
   - Parameters:
     - ❖ **layers**: List containing the number of neurons in each layer of the neural network.
   - Explanation:

- ➢ Initializes the weights for each layer using random values sampled from a uniform distribution.
- ➢ The shape of the weight matrix for each layer is determined by the number of neurons in the current and next layers.

- Return Value: List of weight matrices for each layer.

8. ***' forward (inputs, weights) '***
   - Purpose: Performs the forward pass through the neural network.
   - Parameters:
     - ❖ **inputs**: Input features.
     - ❖ **weights**: List of weight matrices for each layer.
   - Explanation:
     - ➢ Computes the output of each layer by performing matrix multiplication of inputs and weights followed by activation function.
     - ➢ Applies the sigmoid activation function to the output of each layer except the output layer.
   - Return Value: List of outputs from each layer.
9. ***' backward(inputs, outputs, targets, weights, learning_rate) '***
   - Purpose: Performs the backward pass through the neural network for weight updates.
   - Parameters:
     - ❖ **inputs**: Input features.
     - ❖ **outputs**: List of outputs from each layer.
     - ❖ **targets**: True labels.
     - ❖ **weights**: List of weight matrices for each layer.
     - ❖ **learning_rate**: Learning rate for weight updates.
   - Explanation:
     - ➢ Computes the gradient of the loss function with respect to the weights using backpropagation.
     - ➢ Updates the weights of each layer based on the gradient and learning rate.
   - Return Value: Updated list of weight matrices.
10. ***' train_neural_network(data_train, data_test, layers, epochs, learning_rate, batch_size) '***
    - Purpose: Trains the neural network using the provided training data.
    - Parameters:
      - ❖ **data_train**: Training data.
      - ❖ **data_test**: Test data.
      - ❖ **layers**: List containing the number of neurons in each layer of the neural network.
      - ❖ **epochs**: Number of training epochs.
      - ❖ **learning_rate**: Learning rate for weight updates.
      - ❖ **batch_size**: Batch size for mini-batch gradient descent.

- Explanation:
  - ➢ Initializes the weights of the neural network.
  - ➢ Iterates through the specified number of epochs and mini-batches.
  - ➢ Performs forward and backward passes for each mini-batch to update the weights.
- Return Value: Updated weights of the neural network.

11. ' predict(test_data, weights) '

- Purpose: Predicts the labels for the test data using the trained neural network.
- Parameters:
  - ❖ **test_data**: Test data for prediction.
  - ❖ **weights**: List of weight matrices for each layer.
- Explanation:
  - ➢ Performs forward pass through the neural network using the test data.
  - ➢ Predicts the labels based on the output of the neural network.
- Return Value: Predicted labels for the test data.

Results - Table & Plots

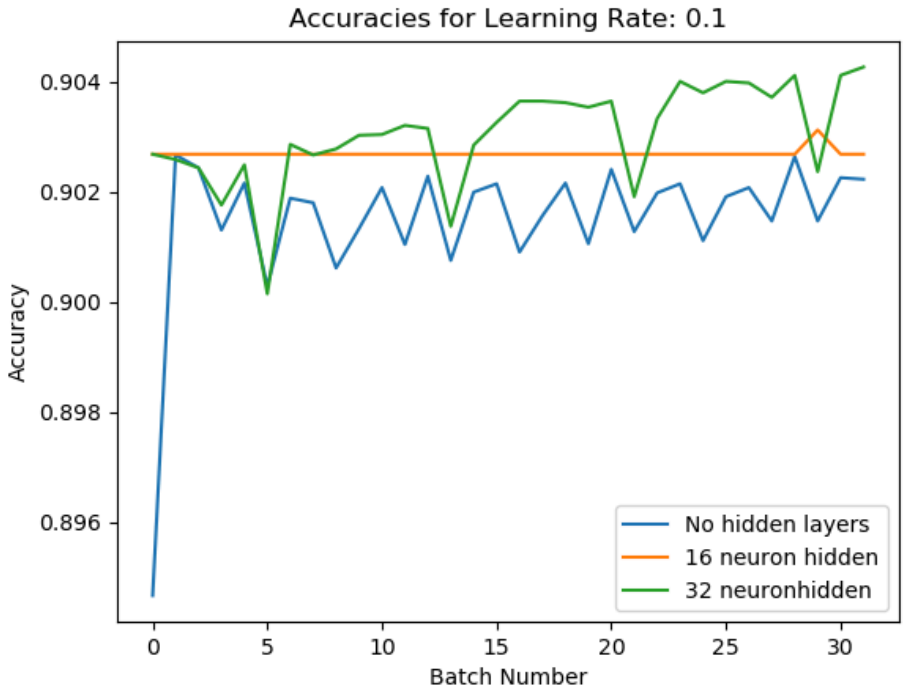
Model	Learning Rate (Hyper-parameter Tuning)				
	0.1	**0.01**	**0.001**	**0.0001**	**0.00001**
pytorch	0.8244165565830031	**0.8266182298546896**	0.8241963892558344	0.8240863055922502	0.8245266402465874
custom_i	**0.9029612505504183**	0.9011999119330691	0.9026860413914575	0.4211800968736239 5	0.6714002642007926
custom_ii	0.9026860413914575	**0.9026860413914575**	**0.9026860413914575**	0.2078379568472038 8	0.7675583443416997
custom_iii	**0.9032364597093792**	0.9023557904007046	0.9026860413914575	**0.8623954205195 95**	**0.9026860413914575**

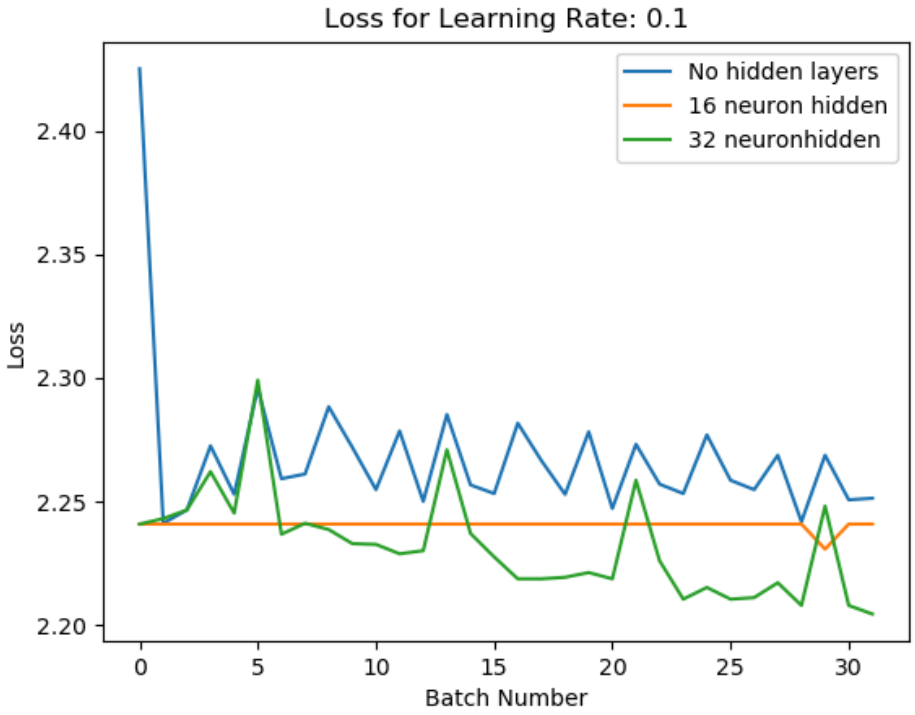
- **custom_i :** No hidden layer
- **custom_ii :** 1 hidden layer with 16 perceptrons
- **custom_iii :** 1 hidden layer with 32 perceptrons

Best-performing models for each ***learning rate*** and **Best-performing *learning rate*** for each **model** are in **BOLD**

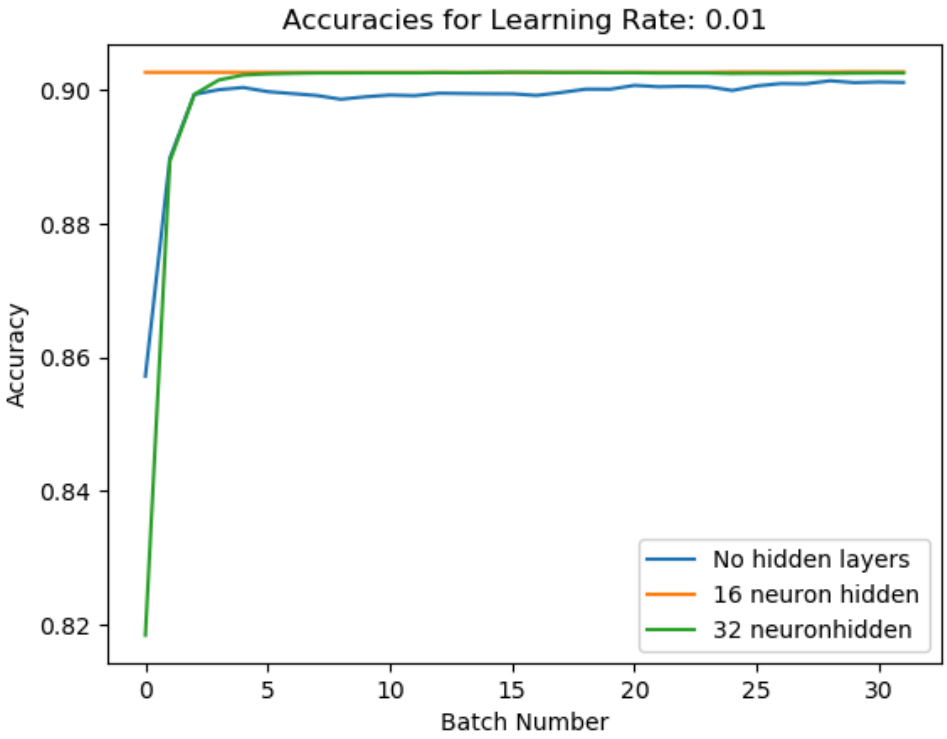
Plots

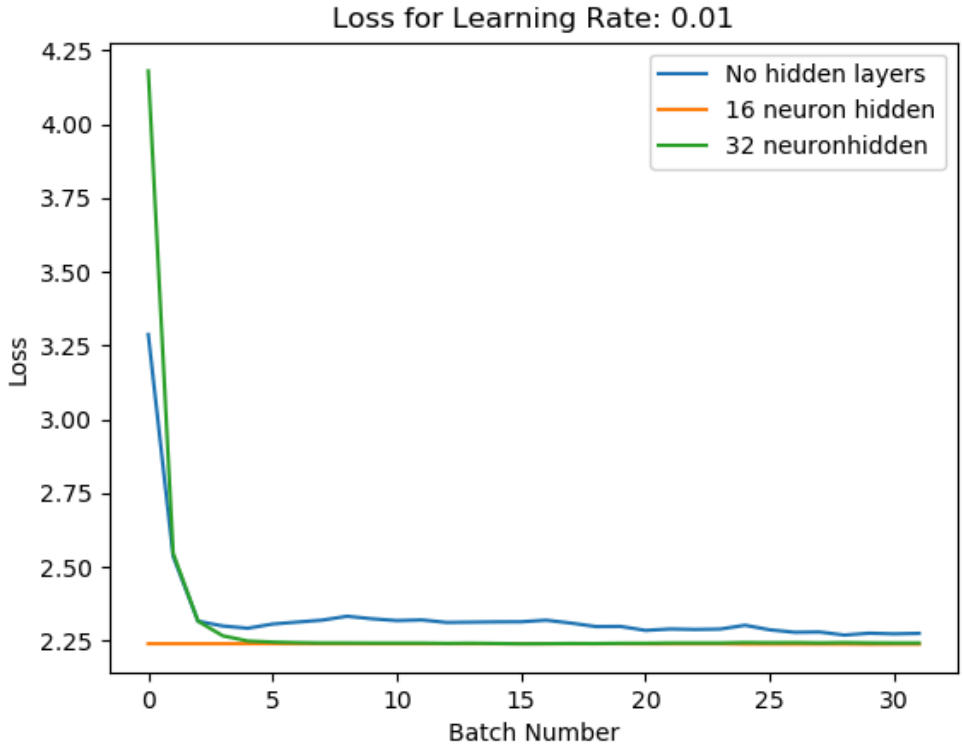
- **Learning_rate = 0.1**



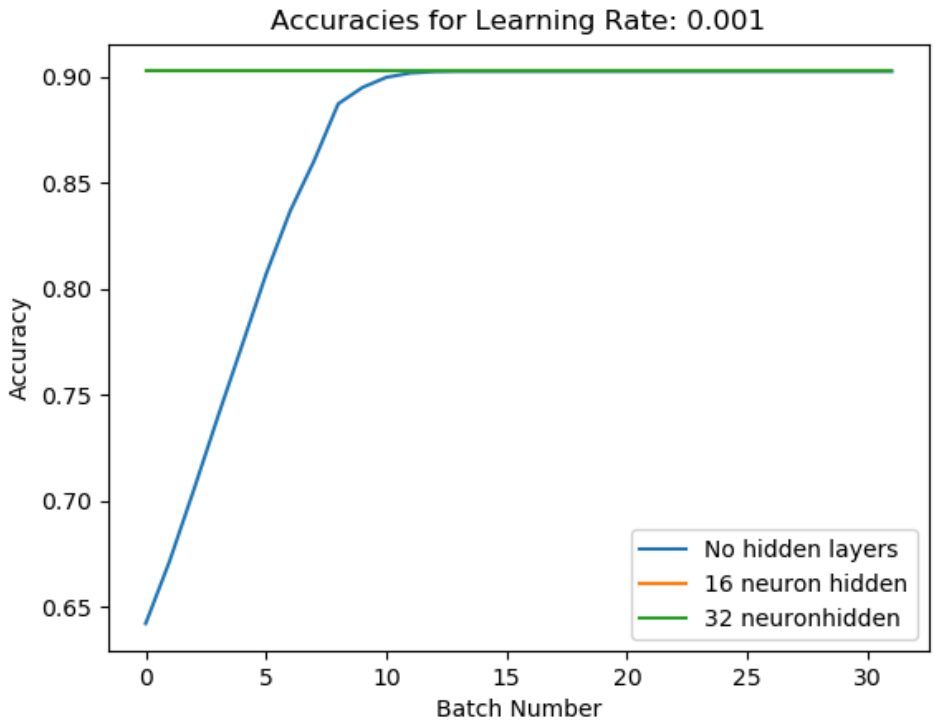


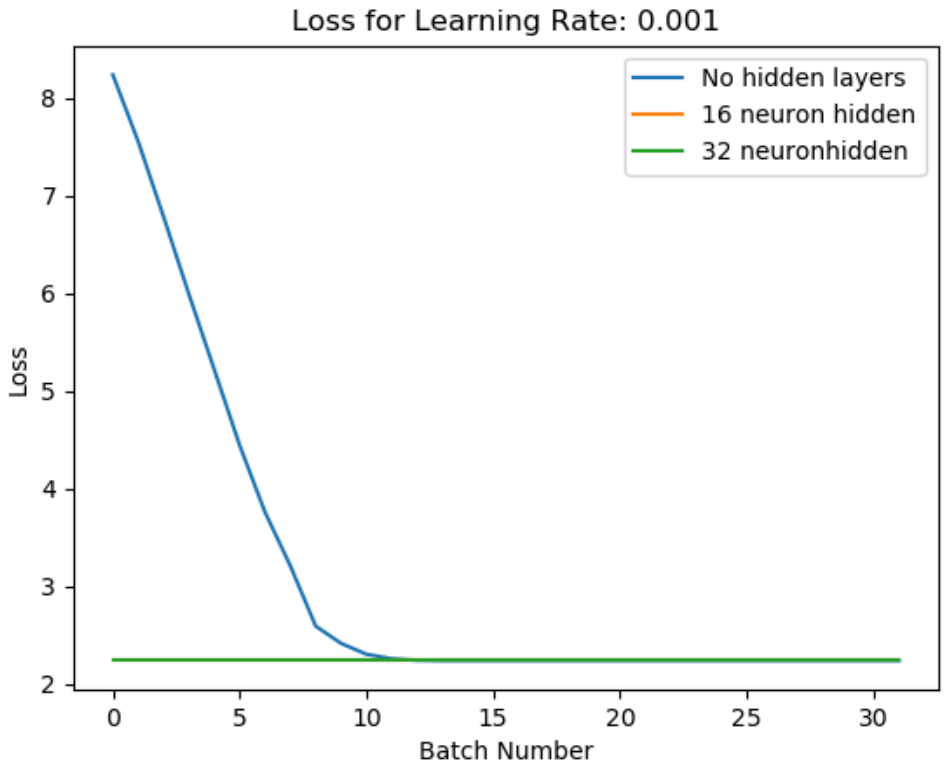
- **Learning_rate = 0.01**



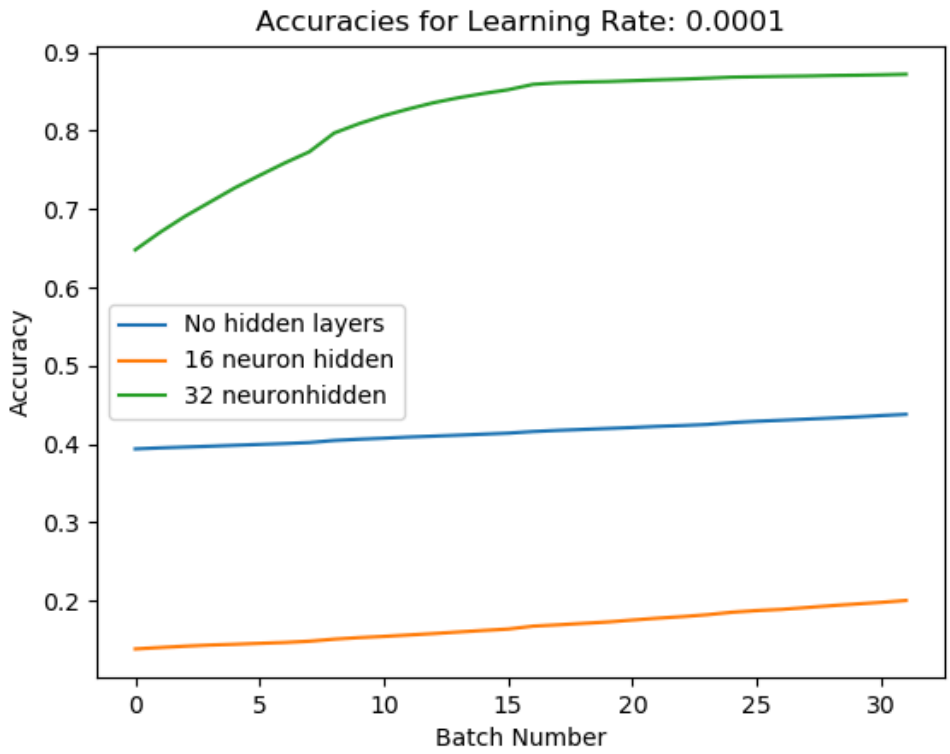


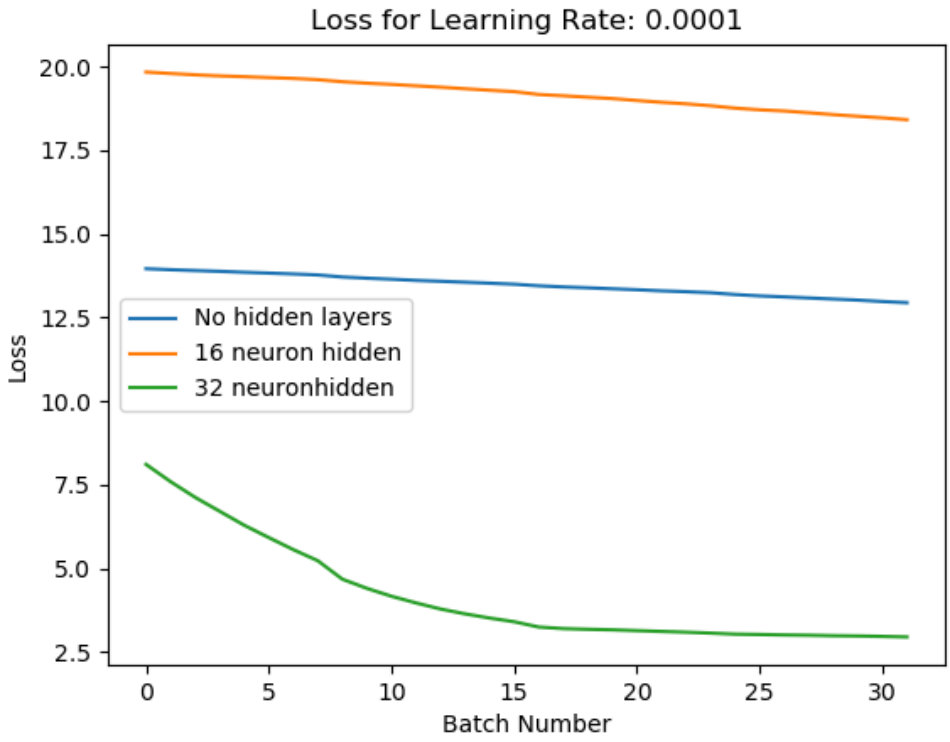
- **Learning_rate = 0.001**



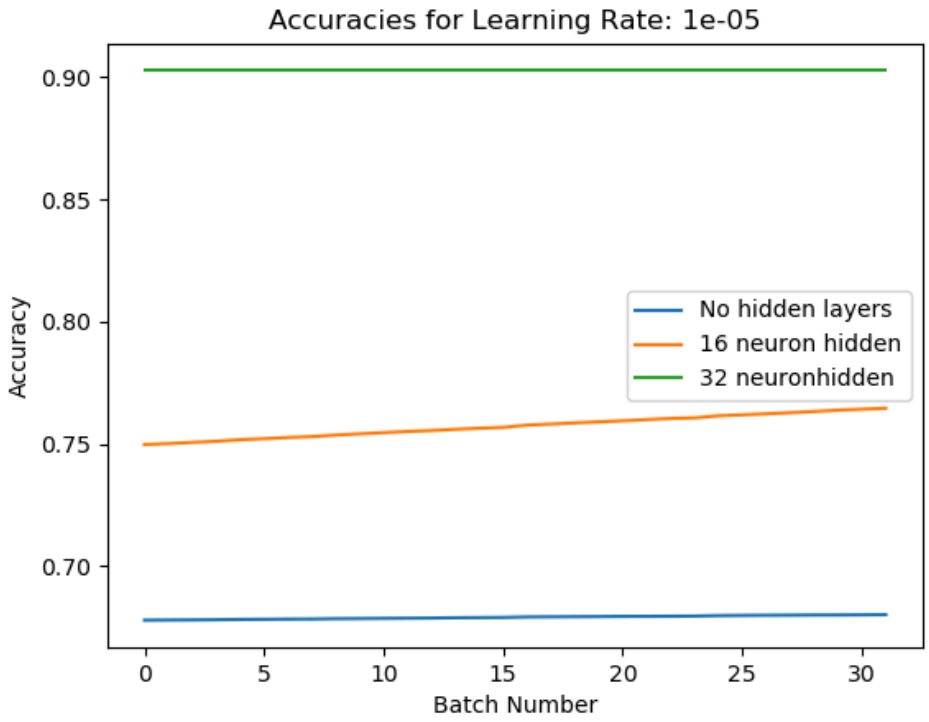


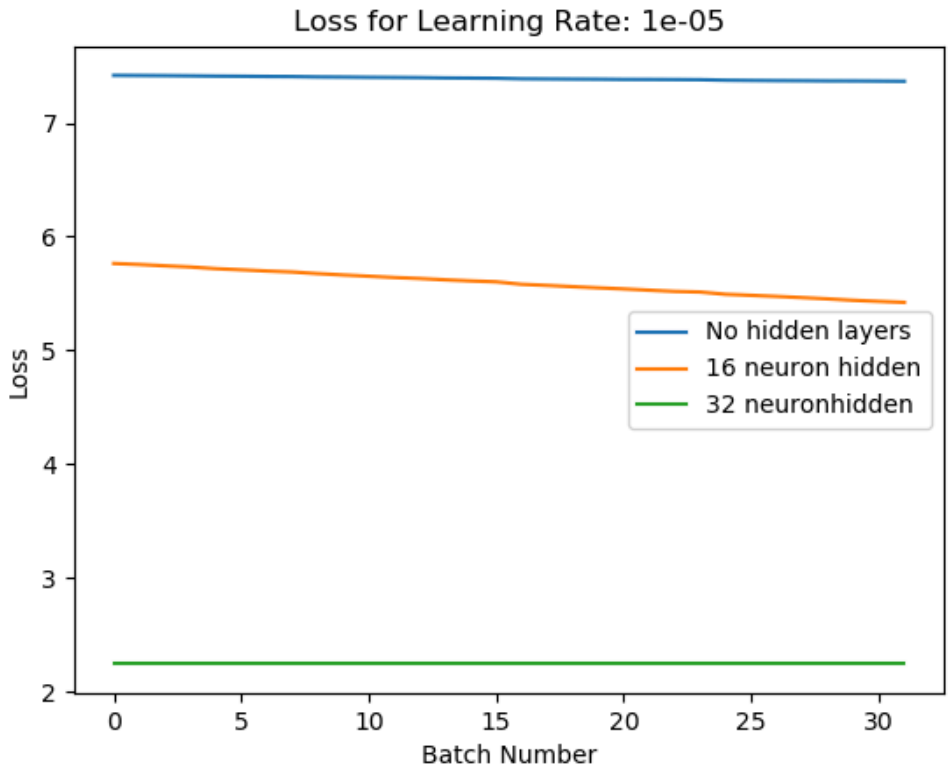
- **Learning_rate = 0.0001**





- **Learning_rate = 1e-5**





Code Usage Instructions

- ★ ***python3 custom.py <type-1/2/3> <learning_rate>***
- ★ ***python3 pytorch.py***
- ★ ***Python3 neo.py <learning_rate>***

Additional

- **CustomNeuralNetwork** as a Class