INTRODUCTION TO ARTIFICIAL INTELLIGENCE [EARIN]

LABAROTORY 5

VARIANT 5

AUTHOR: AKSA FATIMA [317740]

Introduction

Multilayer perceptron is an architecture of Artificial Neural Network, which is widely used in machine learning and deep learning. It is a feedforward neural network consisting of multiple layers of neurons, each fully connected to the next layer. MLPS can be used in classification and regression problems and are widely used in many application domains, including image recognition, natural language processing, recommender systems, etc.

Multilayer Perceptron (MLP) is a common feedforward neural network architecture, which consists of an input layer, a hidden layer, and an output layer. It is used to solve classification and regression problems and is widely used in many fields. MLPS are trained by a backpropagation algorithm and can be combined with other neural network architectures to handle more complex tasks.

Data sets

Kuzushiji-MNIST is a drop-in replacement for the MNIST data set, provided in the original MNIST format as well as NumPy format. Since MNIST restricts us to 10 classes, we choose one character to represent each of the 10 rows of Hiragana when creating Kuzushiji-MNIST.

Algorithm:

MLP:

MLP (Multilayer Perceptron) is a popular algorithm in the field of artificial neural networks which is commonly used for classification problems. It consists of an input layer, one or more hidden layers, and an output layer. The input layer takes in the data and the output layer produces the desired output. The hidden layers consist of neurons which perform calculations based on the input and weight values and pass the result to the next layer. MLP is trained using a backpropagation algorithm, which adjusts the weights in the network based on the difference between the predicted and actual output. MLP is a versatile algorithm which can produce accurate results and can be implemented easily using popular open-source frameworks like PyTorch.

Backpropagation:

Backpropagation algorithm is a widely used algorithm for training neural networks, including MLP. It is an iterative technique that adjusts the weights in the network to minimize the difference between the predicted and actual. outputs. The algorithm works by first making a forward pass through the network to obtain the predicted output, then computing the loss function, which measures the difference between the predicted and actual outputs. The gradient of the loss function with respect to the weights is then computed using the chain rule of calculus, which is propagated backwards through the network. Finally, the weights are updated using a learning rate and the calculated gradients. This process is repeated for each input in the training set until convergence is reached. Backpropagation is a powerful algorithm that enables neural

networks to learn complex patterns and relationships in data, which makes it widely used in the field of artificial intelligence.

Details:

1. In our code we imports the necessary libraries, including torch, torchvision, and matplotlib.pyplot, for building and training the neural network model and for visualizing the results.

2. loads the KMNIST dataset and splits it into a training set and a validation set. The path of the data set is'. / data, through torchvision. Transforms. ToTensor () converts the image to a tensor format.

3. creates training and validation data loaders, which are used to bulk load the data.

4. main part:
   The MLP class is the implementation of a Multi-layer Perceptron neural network using PyTorch. The class takes in the following parameters in its constructor:
   a. input_size: The size of the input layer of the MLP, which is 784 for KMNIST dataset as each image is 28x28 pixels.
   b. output_size: The size of the output layer of the MLP, which is 10 for KMNIST dataset as there are 10 different classes of Hiragana characters to predict.
   c. num_hidden_layers: The number of hidden layers of the MLP, which is set to HIDDEN_LAYERS in this code (i.e., 1,2).
   d. width: The number of neurons in each hidden layer of the MLP, which is set to WIDTH in this code (i.e., 128,64).
   e. Optimizer type: In this program, I'm using SGD and Adam's method for optimization.

5. The CustomMLP class constructs a multi-layer perceptron (MLP) model in PyTorch, enabling flexible configuration of hidden layers and output size. It facilitates forward propagation through linear transformations, serving as the core architecture for classification tasks

6. The forward method performs the feedforward operations of the MLP, including flattening the input tensor, passing it through the hidden layers with SGD and Adam optimizer and passing the output through the final linear layer.

7. creates an instance of the MLP model, defines the loss function (cross entropy loss) and optimizer (stochastic gradient Descent SGD), which are used to train the model.

8. defines the training and evaluation functions. The training function trains the model using the training dataset and returns the training loss and accuracy. The evaluation function evaluates the model performance using the validation dataset and returns the validation loss and accuracy.

9. After each round, the training loss, training accuracy, validation loss, and validation accuracy are printed.

After running the code, I have observed the following output:

Training Accuracy: 72.19%

Validation Accuracy: 66.02%

Batch Size: 64, Learning Rate: 0.001, Optimizer: SGD

Hidden Layers: [128, 64]

Training Loss (first step): 2.3278

Validation Loss (first step): 2.1760

Training Accuracy: 26.48%

Validation Accuracy: 29.02%

Batch Size: 64, Learning Rate: 0.001, Optimizer: Adam

Hidden Layers: [128, 64]

Training Loss (first step): 2.3008

Validation Loss (first step): 0.5443

Training Accuracy: 84.73%

Validation Accuracy: 80.82%

Batch Size: 64, Learning Rate: 0.01, Optimizer: Adam

Hidden Layers: [128]

Training Loss (first step): 2.2933

Validation Loss (first step): 0.7044

Training Accuracy: 83.46%

Validation Accuracy: 74.15%

Batch Size: 64, Learning Rate: 0.1, Optimizer: SGD

Hidden Layers: [128]

Training Loss (first step): 2.2610

Validation Loss (first step): 0.6308

Training Accuracy: 83.70%

Validation Accuracy: 78.27%

Batch Size: 64, Learning Rate: 0.1, Optimizer: Adam

Hidden Layers: [128]

Training Loss (first step): 2.3286

Validation Loss (first step): 2.2573

Training Accuracy: 11.50%

Validation Accuracy: 10.10%

Batch Size: 64, Learning Rate: 0.01, Optimizer: SGD

Hidden Layers: [128, 64]

Training Loss (first step): 2.3130

Validation Loss (first step): 0.9505

Training Accuracy: 63.47%

Validation Accuracy: 64.40%

Batch Size: 64, Learning Rate: 0.01, Optimizer: Adam

Hidden Layers: [128, 64]

Training Loss (first step): 2.3215

Validation Loss (first step): 0.6636

Training Accuracy: 83.42%

Validation Accuracy: 76.30%

Batch Size: 64, Learning Rate: 0.1, Optimizer: SGD

Hidden Layers: [128, 64]

Training Loss (first step): 2.3099

Validation Loss (first step): 0.6206

Training Accuracy: 80.82%

Validation Accuracy: 79.39%

We can conclude the following:

Based on the observed data, let's draw conclusions for each of the factors mentioned.

1. Learning Rate:

- Higher learning rates (0.1) seem to lead to faster initial progress but may require careful tuning to prevent divergence, as seen with SGD optimizer and learning rate 0.1 in the first step.

- Lower learning rates (0.001) can provide more stable training, as seen with both optimizers.

- Optimal learning rate appears to be around 0.001, as it consistently yields high validation accuracy across different optimizers and architectures.

2. Mini-batch Size:

- Smaller batch sizes (64) generally produce better results but slower computation compared to larger batch sizes.

- However, the trend is not consistent across all configurations. Batch size variation might need further exploration.

3. Number of Hidden Layers:

- Adding hidden layers beyond one layer (128) generally improves the model's learning capacity and performance, as seen with the addition of a second hidden layer (64 neurons).

- A linear model (0 hidden layers) performs relatively poorly compared to models with hidden layers, suggesting the importance of non-linear transformations for this task.

4. Width (Number of Neurons in Hidden Layers):

- Increasing the width of the hidden layers (128 neurons to 128, 64 neurons) generally leads to improved performance, as seen with the increase in validation accuracy.

- However, wider layers may require more computational resources and might be prone to overfitting if not regularized properly.

5. Optimizer Type:

- Adam optimizer tends to perform better compared to SGD, particularly with a learning rate of 0.001, as evidenced by consistently higher validation accuracies.

- SGD with momentum shows competitive performance, especially with a learning rate of 0.1, although it might require more careful tuning of hyperparameters.

Overall, based on the observed data, a configuration with Adam optimizer, a learning rate of 0.001, batch size of 64, at least one hidden layer with 128 neurons, and possibly additional layers with widths of 64 neurons each, seems to yield the best performance for this task.