Deep Learning Assignment 1 Solution

1 Solution 1

When performing neural network output classification using softmax activation function for the output node, cross entropy or Average Dross Entropy (ACE) is preferred over Mean Squared Error (MSE) for several reasons:

- MSE gives too much emphasis to the incorrect outputs.
- The ln() function in cross-entropy takes into account the closeness of a prediction and provides a more granular way to compute error.

Moreover, when employing backpropagation for training, ACE is a better choice compared to MSE. In backpropagation, the objective is to drive output node values towards either 1.0 or 0.0 depending on the target values. If MSE is used, the weight adjustment factor (the gradient) contains a term of (output) $\times$ $(1 -$ output). As the computed output approaches either 0.0 or 1.0, the value of (output) $\times$ $(1 -$ output) diminishes, potentially causing training to stall.

On the other hand, using cross-entropy error eliminates the (output) $\times$ $(1-$output) term. Therefore, the weight changes don't decrease as the output approaches the extreme values, reducing the likelihood of training stalling out. This argument assumes neural network classification with softmax output node activation.

2 Solution 2

Given a neural network with linear activation functions, the output $\hat{y}_i$ for a given input x_i can be expressed as a linear function of the input and the network parameters:

$$\hat{y}_i = w \cdot x_i + b$$

where:

- w represents the weights of the network,
- b represents the bias term,
- x_i represents the input,
- $\hat{y}_i$ represents the predicted output.

Now, let's consider the mean squared error (MSE) loss function for a single training example:

$$E = \frac{1}{2}(\hat{y}_i - y_i)^2$$

where:

- y_i represents the true output.

Substituting the expression for $\hat{y}_i$, we get:

$$E = \frac{1}{2}((w \cdot x_i + b) - y_i)^2$$

This expression represents the squared difference between the predicted output and the true output. Since this is a quadratic function of the network parameters w and b, and quadratic functions are convex, the MSE loss function becomes convex.

When using linear activation functions in a neural network, the output $\hat{y}_i$ is simply a linear combination of the input and the network parameters, similar to what we discussed earlier:

$$\hat{y}_i = w \cdot x_i + b$$

where:

- w represents the weights of the network,
- b represents the bias term,
- x_i represents the input,
- $\hat{y}_i$ represents the predicted output.

Now, consider the cross-entropy error function for binary classification:

$$E = -\sum_{i=1}^{N} [y_i \log(w \cdot x_i + b) + (1 - y_i) \log(1 - (w \cdot x_i + b))]$$

where:

- y_i is the true label (0 or 1),
- $\hat{y}_i$ is the predicted probability.

Substituting the expression for $\hat{y}_i$, we get:

$$E = -\sum_{i=1}^{N} [y_i \log(w \cdot x_i + b) + (1 - y_i) \log(1 - (w \cdot x_i + b))]$$

This expression is not a convex function of the network parameters w and b because of the presence of the logarithmic terms inside the summation. The logarithmic terms introduce non-linearities, making the function non-convex.

Therefore, unlike the case with mean squared error (MSE), the cross-entropy error function does not necessarily reduce to a convex optimization problem when using linear activation functions. Instead, it remains a non-convex optimization problem. This property affects the optimization process and may require the use of specialized optimization algorithms to find a good solution, as convex optimization guarantees do not apply in this scenario.

So the answer to the question is Option (B)

3 Solution 3

The following description contains a detailed explanation of the .ipynb file used for solving the question.

3.1 Number of Hidden Layers and Neurons

The number of hidden layers and neurons per layer is specified by the hidden_sizes list. In this case, there are two hidden layers with 128 and 64 neurons, respectively.

3.2 Activation Functions

The activation function used in each hidden layer is ReLU (Rectified Linear Unit), specified by `self.relu = nn.ReLU()` in the NeuralNetwork class.

3.3 Input Image Preprocessing

The input images are preprocessed using the `transforms.Compose` function, which applies a sequence of transformations to the input images. The transformations include converting the images to tensors (`transforms.ToTensor()`) and normalizing them with a mean and standard deviation of 0.5 (`transforms.Normalize((0.5,), (0.5,))`).

3.4 Hyperparameter Tuning Strategies

- **Learning Rate (learning_rate):** The learning rate determines the step size taken during optimization. A common approach is to start with a small learning rate and adjust it based on the model's performance. In this case, a learning rate of 0.001 is used.
- **Number of Epochs (num_epochs):** The number of epochs defines how many times the entire training dataset is passed forward and backward through the neural network. It is typically chosen based on the convergence of the training loss. In this case, the number of epochs is set to 10.
- **Batch Size (batch_size):** The batch size determines the number of samples processed before updating the model parameters. It affects both the speed and stability of training. Common values are powers of 2, and the choice depends on factors such as available memory and computational resources. Here, a batch size of 64 is used.
- **Optimizer (optimizer):** The choice of optimizer affects how the model parameters are updated during training. Common choices include Adam, SGD (Stochastic Gradient Descent), and RMSprop. The Adam optimizer with a learning rate of 0.001 is used in this code.
- **Loss Function (criterion):** The loss function defines the measure of how well the model is performing. For classification tasks, cross-entropy loss is commonly used. It's chosen based on the nature of the problem being solved. Here, `nn.CrossEntropyLoss()` is used.

These hyperparameters are often tuned through experimentation and validation to find the optimal values for the specific task and dataset. Techniques such as grid search, random search, and automated hyperparameter optimization can be employed to search for the best hyperparameters efficiently.

3.5 Complete Explanation

The provided code trains a feedforward neural network (FNN) using the MNIST dataset for handwritten digit classification. Here's a brief explanation of the code:

- **Imports:** The necessary libraries are imported, including PyTorch for building and training the neural network and torchvision for handling datasets and transformations.
- **Hyperparameters:** Hyperparameters such as the input size, hidden layer sizes, output size, batch size, learning rate, and number of epochs are defined.
- **Neural Network Definition:** The NeuralNetwork class is defined, subclassed from nn.Module, representing the architecture of the FNN. It consists of three fully connected (linear) layers with ReLU activation functions in the hidden layers and a softmax activation function in the output layer.
- **Data Preprocessing:** Input images are preprocessed using PyTorch's transforms.Compose function. The images are converted to tensors and normalized to have a mean and standard deviation of 0.5.

- **Dataset Loading:** The MNIST dataset is loaded using torchvision's MNIST class. The training and test datasets are obtained, and DataLoader objects are created to iterate over batches of data during training and testing.
- **Model Initialization:** An instance of the NeuralNetwork class is created,