HyperionDev

Build a Neural Network

Task

Visit our website

Introduction

After exploring the theoretical underpinnings of Multilayer Perceptron (MLP) neural networks, we'll work through a step-by-step practical implementation using PyTorch, a popular deep learning framework. PyTorch provides a user-friendly interface and efficient GPU acceleration, making it an ideal choice for building and training MLPs.

In this lesson, we'll cover the essential steps involved in creating an MLP using PyTorch. This includes defining the network architecture, initialising weights and biases, defining the loss function and optimiser, training the model on a dataset, and evaluating its performance. By the end of this section, you'll have a solid understanding of how to implement and train MLPs in PyTorch.

Practical implementation

Set-up phase

Before writing the code, it's good practice to set up a virtual environment. A virtual environment is created on top of an existing Python installation, which helps manage dependencies and avoid conflicts between different projects. Each environment has its own independent set of Python packages installed in its directory. Use the additional reading, **Setting Up a Virtual Environment**, in your folder to set up a virtual environment. You may use `venv` or `virtualenv`. Once you have created your virtual environment, ensure that it is activated before installing the required packages for this practical example.

To successfully implement, train and test our MLP model, a deep learning library called PyTorch should be installed. However, if you are using the Google Colab environment, these terminal commands are not necessary since PyTorch is already installed by default. To install PyTorch on your virtual environment or laptop, use the following terminal command:

```
pip install torch torchvision
```

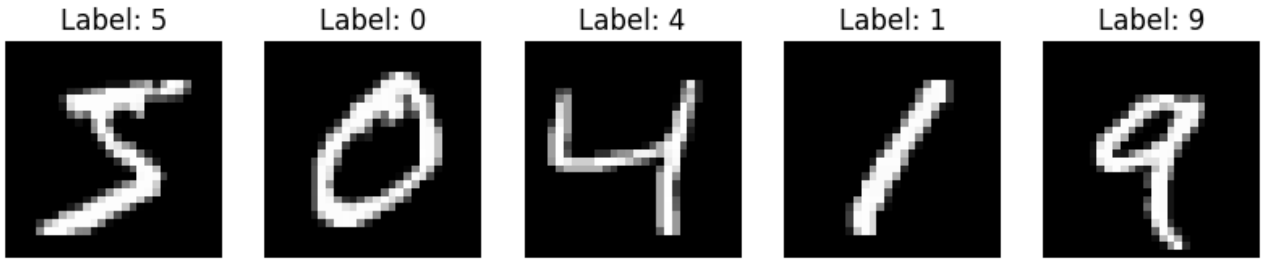
Next, the following lines should be used to import PyTorch and its modules. The first module, `torch`, provides numerous functionalities useful for deep learning and handling computations efficiently. It provides multi-dimensional tensors similar to NumPy arrays but with GPU acceleration. The second module, `torch.nn`, provides classes and functions to define neural network layers, activation functions, loss functions and utilities for initialising forward propagation, and backward propagation of signals through the network layers. The third module, `torch.optim`, implements optimisation algorithms commonly used for training neural networks and updates the

weights of the neural network during training to minimise the defined loss function. By configuring these modules, we establish the groundwork for our model's operations, enabling computations, defining network architectures, and manipulating data.

```
import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import transforms
```

Implementing an MLP

In this project, we'll be working with the **MNIST dataset**, a collection of grayscale images (28x28 pixels) representing handwritten digits from 0 to 9. The specific task at hand is to develop an MLP using the PyTorch framework to classify these handwritten digits.



(May, 2024) ***https://www.geeksforgeeks.org/mnist-dataset/***

1. Our practical implementation starts with importing other necessary libraries for our project. Specifically, we utilise the `os` library for operating system dependent functionality, `numpy` for numerical computing, and the **MNIST torchvision dataset class** to work with the specified data.

```
import os

import numpy as np

from torchvision.datasets import MNIST
```

2. Next, we will ensure the reproducibility of our results and define a transformation pipeline. The reproducibility provided below will ensure that every time you run the code; you get the same random numbers, making the results reproducible:

```
torch.manual_seed(1234)

np.random.seed(1234)
```

To define a transformation pipeline and normalise the data, the `transforms.Compose()` function will be used to chain multiple transformations together. In our case, we apply two transformations sequentially: `transforms.ToTensor()` converts the input data into PyTorch tensors, and `transforms.Normalize()` standardises the tensor values using specified mean and standard deviation values. In our example, the mean and standard deviation are set to 0.5 for each channel. This normalisation process should help in stabilising the training process and improving convergence when training networks.

```
transform = transforms.Compose([

   transforms.ToTensor(),

   transforms.Normalize((0.5,), (0.5,))

])
```

3. Next, we will load the MNIST dataset from the specified root directory. We set the `train` parameter to `True` to load the training set, and `download` to `True` to automatically download the dataset if it's not available locally. Additionally, we apply the previously defined normalisation transformation to the dataset during loading. This ensures that the input data is preprocessed before being used for training.

```
mnist = MNIST(root='./data', train=True, download=True, transform=transform)
```

4. After loading the dataset, it will be split into input features (`X`) and corresponding labels (`y`). The dataset consists of image-label pairs, where each image represents a handwritten digit from 0 to 9 and is associated with a specific label indicating the corresponding digit class. To complete the splitting phase, we initialise empty lists `X` and `y` to store the features and labels, respectively. Then, we iterate through each image-label pair in the dataset. For each image, we flatten it to convert it into a one-dimensional array and append it to the list `X`. Similarly, we append the corresponding label to the list `y`. Finally, we convert the generated lists into NumPy arrays for further processing.

```
X = []
y = []
for image, label in mnist:
   X.append(image.flatten().numpy())
   y.append(label)
X = np.array(X)
y = np.array(y)
```

5. The following is to convert `X` and `y` to PyTorch tensors by using the `torch.tensor()` function and specifying the data type for the tensors. This conversion prepares the data for consumption by PyTorch neural network models.

```
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)
```

6. Finally, we are in a position to define the architecture of the MLP model using PyTorch the `nn.Module` class and implementing two essential methods: `__init__()` and `forward()`.

In the `__init__()` method, we define the structure of the neural network by specifying the number of input features, hidden units, and output classes. Two fully connected (dense) layers `self.fc1` and `self.fc2` are defined using the `nn.Linear` module, with `ReLU` activation applied to the output of the first hidden

layer (`self.activation`). Finally, the softmax activation function is applied to the output layer to compute the probabilities for each class.

The `forward()` method defines the forward pass of the network, where the input tensor `x` is passed through each layer sequentially. The ReLU activation function is applied to the output of the first hidden layer, and the softmax function is applied to the output layer to obtain class probabilities. The resulting tensor is returned as the output of the model.

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        # Forward pass through the network
        x = self.activation(self.fc1(x))
        x = self.fc2(x)
        x = self.softmax(x)
        return x
```

7. After defining the MLP architecture, we are in a position to determine the input size, hidden layer size, and output size. The input size (`input_size`) is obtained from the number of features in our dataset, which corresponds to the number of columns in the input feature tensor `X`. The hidden layer size (`hidden_size`) is set to 64, indicating the number of neurons in the hidden layer. The output size (`output_size`) is determined by the number of unique classes in

our dataset, computed using `np.unique(y)` to obtain the unique labels and then calculating the length of the resulting array.

```
input_size = X.shape[1]

hidden_size = 64

output_size = len(np.unique(y))
```

Training phase

8. It is time to train the model using the specified input size, hidden layer size, and output size. We initialise the MLP model (`mlp`) with the defined architecture parameters: `input_size`, `hidden_size`, and `output_size`. Additionally, we define the loss function using the `CrossEntropyLoss()` function, which is suitable for multi-class classification tasks. The `optimizer` is configured using stochastic gradient descent (SGD) with a learning rate of 0.01.

We iterate through a fixed number of epochs, performing the training process in each epoch. During each epoch:

- We perform a forward pass through the MLP model to obtain the predicted outputs (`outputs`) for the input features (`X_tensor`).
- The loss is computed using the `CrossEntropyLoss()` function, comparing the predicted outputs with the ground truth labels (`y_tensor`).
- We initialise the gradients to zero using `optimizer.zero_grad()`, perform the backward pass to compute the gradients (`loss.backward()`), and update the model weights using the optimiser (`optimizer.step()`).
- Finally, we print the current epoch number and the corresponding loss value.

This training loop allows the MLP model to learn from the dataset, optimising its parameters to minimise the classification loss and improve its performance over successive epochs.

```
mlp = MLP(input_size, hidden_size, output_size)
```

```
criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(mlp.parameters(), lr=0.01)

epochs = 10

for epoch in range(epochs):

   # Forward pass

   outputs = mlp(X_tensor)

   # Compute loss

   loss = criterion(outputs, y_tensor)

   # Zero gradients, backward pass, update weights

   optimizer.zero_grad()

   loss.backward()

   optimizer.step()

   # Print loss

   print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
```

Output:

```
Epoch [1/10], Loss: 2.3010

Epoch [2/10], Loss: 2.3009

Epoch [3/10], Loss: 2.3008

Epoch [4/10], Loss: 2.3007

Epoch [5/10], Loss: 2.3006

Epoch [6/10], Loss: 2.3005

Epoch [7/10], Loss: 2.3004

Epoch [8/10], Loss: 2.3002

Epoch [9/10], Loss: 2.3001

Epoch [10/10], Loss: 2.3000
```

According to the loss values printed below, it could be concluded that the model is not learning effectively during training. The loss remains approximately constant across epochs, suggesting that the model's predictions are not improving. This lack of improvement could be due to various factors, such as insufficient model complexity, inappropriate learning rate, or issues with data preprocessing. Further investigation and adjustments to the model architecture, hyperparameters, or dataset preprocessing may be necessary to improve training performance.

Optional exercise: Make a copy of the above code and try to get better performance by modifying these parameters and re-training the network!

9. We are in a position now to make predictions using the trained MLP model on the input features (`X_tensor`) without computing gradients. This is achieved by enclosing the prediction code within a `torch.no_grad()` context manager, which temporarily disables gradient calculation to reduce memory consumption and improve inference speed. Here, the `torch.argmax()` function is utilised to obtain the index of the maximum value along the specified dimension, effectively identifying the predicted class for each input sample. These predictions are then converted to a NumPy array using the `numpy()` method and printed out.

```
with torch.no_grad():

   predictions = torch.argmax(mlp(X_tensor), dim=1)

print("Predictions:", predictions.numpy())
```

Evaluation phase

10. Evaluation time! Let's download the test data for our dataset.

```
mnist_test = MNIST(root='./data', train=False, download=True,
transform=transform)
```

Now, we split the test dataset into input features (`X_test`) and labels (`y_test`). Then, we convert the test data to PyTorch tensors (`X_test_tensor` and `y_test_tensor`) for compatibility with the model.

Within a `torch.no_grad()` context manager, we perform forward pass inference on the test data using the trained model. The model's outputs (`outputs_test`) are obtained, and predictions are generated by identifying the class with the maximum probability for each sample using the `torch.max()` function. The

number of correctly predicted samples is computed by comparing the predicted labels with the ground truth labels (`y_test_tensor`), and the total number of samples is determined.

Finally, the test accuracy is calculated and the resulting accuracy value is printed to the console.

```
# Split the test dataset into input features (X_test) and labels (y_test)
X_test = []
y_test = []
for image, label in mnist_test:
    X_test.append(image.flatten().numpy())
    y_test.append(label)
X_test = np.array(X_test)
y_test = np.array(y_test)
# Convert test data to PyTorch tensors
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)
# Evaluate the model on the test dataset
with torch.no_grad():
    outputs_test = mlp(X_test_tensor)
    _, predicted = torch.max(outputs_test, 1)
    correct = (predicted == y_test_tensor).sum().item()
    total = y_test_tensor.size(0)
    accuracy = correct / total
print(f'Test Accuracy: {accuracy:.2f}')
```

Output:

```
Test Accuracy: 0.12
```

The test accuracy of 0.12 indicates that the trained MLP model achieves a relatively low level of accuracy when classifying handwritten digits on the test dataset. This suggests that the model's performance is unsatisfactory, as it correctly predicts only 12% of the samples in the test dataset.

When you have finished the exercise, you can deactivate your virtual environment through the command prompt:

```
deactivate
```

When the accuracy of the model is low, further analysis and experimentation are necessary to identify the root causes of the model's poor performance. Possible approaches could include adjusting the model architecture, fine-tuning hyperparameters, or exploring more advanced techniques for data preprocessing and augmentation. Let’s learn more about these approaches in the following section.

Optimising model performance

Optimising the performance of a neural network model is needed in order to achieve higher accuracy and efficiency. Once an initial model is developed, it is often necessary to fine-tune it iteratively to achieve its higher capabilities. Several common fine-tuning techniques will be explained next.

- First, the **learning rate**. Maximising the performance of a neural network commonly depends on adjusting this rate. The learning rate determines the size of the steps the optimisation algorithm takes when updating the model’s parameters (such as weights and biases) during training. By experimenting with various learning rate values in the range between 0 and 1, such as 0.001, 0.01, and 0.1, one can find an appropriate rate that guarantees effective and consistent training. A learning rate that is too high might cause the model to overshoot the optimal solution, consequently leading to unstable training and divergence. On the other hand, a learning rate that is too low may cause the training process to be too slow, and possibly lead to the model becoming caught in local minima and consequently failing to reach the optimal performance. Therefore, a thoughtfully selected learning rate should provide a balance where the model maintains stability but also converges rapidly to a low error value.

- The training process can also depend on **batch size optimisation**. Batch size represents the number of training examples used to compute the gradient in a single iteration. Testing different batch sizes, such as 32, 64, and 128, helps you to strike the ideal mix between computational economy and training stability. Smaller batch sizes often produce more noisy estimates of the gradient, which can assist the model to escape local minima and improve generalisation. But because of the more iterations needed, they can also result in shorter training

times and less steady progress. Although they may need more memory and may result in poorer generalisation, larger batch sizes allow more precise gradient estimations. In general, it is possible to find an optimal batch size that increases performance but also preserves effective use of computational resources by varying it.

- An epoch is a one full pass across the whole training set. Testing varying **epoch counts** (10, 20, 50, etc.) allows you to track how the model's performance changes with time. Should the count of epochs be too low, the model can underfit and fail to identify the trends in the data. On the other hand, if the number of epochs is too high, the model could overfit and capture noise and particular patterns that do not apply generally to new data.

- Determining a model's capacity depends on modifying the **number of neurons in hidden layers**. Hidden layers represent the basic computational units of a neural network, and their size directly affects the network's ability to learn complex patterns. Increasing the ability of the model to capture complicated relationships in the data by adding more neurons can help to perhaps improve performance on challenging tasks. But since a more complicated model might learn noise, this also raises an overfitting risk. On the other hand, lack of neurons could restrict the capacity of the model to learn adequately and result in underfitting.

- Trying different **activation functions**, such as ReLU, sigmoid, or tanh, will provide insight in observing their effects on the learning dynamics of a model. Non-linearity brought into the network via activation functions helps the network to learn more complicated patterns. Popular for its simplicity and efficiency, ReLU helps to lower the possibility of the vanishing gradient problem. If improperly controlled, though, it can cause dead neurons. Although they are more prone to the **vanishing gradient problem**, which can slow down learning, sigmoid and tanh functions might be helpful for issues needing outputs in a specific range.

- Another way to optimise performance is to use rotations, flips, and zooms as **data augmentation** methods to improve model generalisation and resilience. Artificial data augmentation generates modified representations of current data, therefore augmenting the training set and enabling the model to learn to identify patterns under various circumstances. Image classification problems especially benefit from this method since small changes in input data can greatly increase the capacity of the model. Applying these adjustments would help the model to be more resistant to changes in the data, but also lowering overfitting and improving performance on unseen samples.

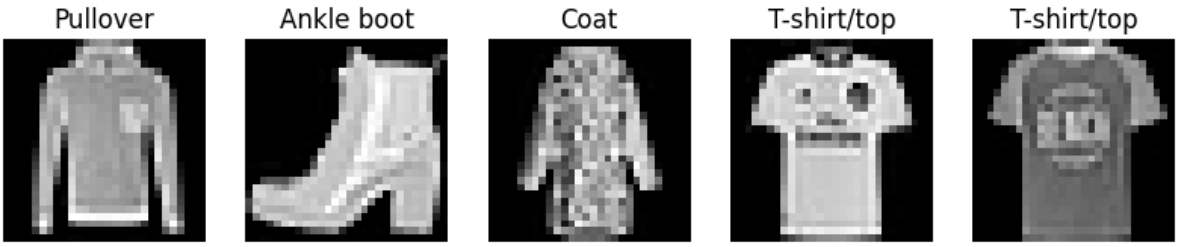
- Be also aware that you can use **different optimisers** to implement different approaches for automatically changing the model parameters. For instance, Stochastic gradient descent (SGD) uses the average gradient over the batch to

update parameters; although slow, this gives stability. Often resulting in speedier convergence, **Adam** computes adaptive learning rates for every parameter, therefore combining the advantages of two prior SGD improvements. To address the vanishing gradient issue, **RMSprop** modulates the learning rate depending on a moving average of recent gradient magnitudes. In the process of testing, these optimisers help to find for a particular model and dataset the approach that offers the best compromise between convergence speed and accuracy.

- As a final remark, ensure you create **training, validation, and test sets** in the beginning of a project. The training set is used to fit the model, providing the data on which the model learns the patterns and relationships. Tuning hyperparameters and choosing model architecture depend on the validation set. It lets you keep an eye on the performance of the model during training and makes adjustments to prevent overfitting. The test set is used for the final evaluation of the model's performance after training is completed. It provides an unbiased assessment of how well the model generalises to new, unseen data.

Instructions

Now that we have gone through the implementation of the MLP model step by step, let's test your knowledge with a similar task. For this practical exercise, the task is to implement an MLP to classify images of fashion items from the MNIST Fashion dataset, achieving an accuracy of **at least 80%** on the test set. This dataset is a collection of grayscale images of various fashion items, such as clothing and accessories. Each image is of size 28x28 pixels and corresponds to one of the following 10 classes: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot.



Practical task

- Create a Jupyter notebook called **fashion_mnist_task.ipynb**.
- Import the dataset, as well as all required libraries, as shown below.

```
import torch

import torch.nn as nn

import torch.optim as optim

from torchvision.datasets import FashionMNIST

from torchvision import transforms

from torch.utils.data import DataLoader

from sklearn.metrics import accuracy_score
```

- Set the random seed values to 123, so you are comparing the 'same' set of data with other students.
- Now, your first task is to load a subset of the **MNIST Fashion** dataset using PyTorch tensors
- Next, split the subset of the MNIST Fashion dataset into a training and test set using PyTorch's DataLoader.
- Implement a Multilayer Perceptron (MLP) using PyTorch tensors to create a classification model for the fashion items in the MNIST Fashion dataset.
- Train the MLP model on the training set. Aim to achieve an accuracy of at least 80% on the test set.
- Choose one hyperparameter to tune, and explain why you chose this hyperparameter.
- Select the value for the hyperparameter to test on the test data, and provide reasoning for your selection.
- Here is the code to display the confusion matrix for your MLP model on the test set. Your task is to report which classes the model struggles with the most based on the confusion matrix.

```
def evaluate_model(model, test_loader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.tolist())
            all_labels.extend(labels.tolist())
    return all_preds, all_labels
preds, labels = evaluate_model(model, test_loader)
conf_matrix = confusion_matrix(labels, preds)
print("Confusion Matrix:")
print(conf_matrix)
```

- Calculate and report the accuracy, precision, recall, and F1-score using the ‘macro’ average in the appropriate functions from sklearn.
- If the set performance is not achieved, repeat the training process by further adjusting the model settings.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.