CSE 4554
Machine Learning Lab

Experiment No: 4
Name of the experiment: Understanding Neural Networks & Pytorch

Dr. Hasan Mahmud
Professor, Department of CSE
Md. Tanvir Hossain Saikat
Junior Lecturer, Department of CSE

November 2, 2024

Contents

1 Objectives

- To understand how neural networks work
- To implement simple Neural Networks using PyTorch
- To set up a deep learning-based image classification pipeline using only Neural Networks
- Understand the difference between Neural Networks and Linear Regression

1.1 Introduction to Neural Networks

Neural networks are a class of machine learning algorithms inspired by the structure of the human brain. They consist of layers of interconnected neurons (or nodes), which process input data and adjust their internal parameters (weights and biases) to learn complex relationships.

1.1.1 Neural Network Components

A basic neural network consists of:

- **Input Layer**: This layer takes the input data (features) and passes it to the hidden layers.
- **Hidden Layers**: These layers contain neurons that process the input data using activation functions. A network can have one or more hidden layers.
- **Output Layer**: This layer gives the final prediction or classification result.

1.2 Mathematical Representation

In a neural network, the input features $x_1, x_2, \ldots, x_n$ are weighted by parameters $w_1, w_2, \ldots, w_n$ and passed through an activation function. The final prediction is obtained by combining the outputs from the neurons.

For a single neuron:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

Where z is the weighted sum, and b is the bias term.

The output is passed through an activation function $f(z)$ such as:

$$y = f(z)$$

This output is then passed to subsequent neurons (in hidden layers) or directly to the output layer in a single-layer neural network.

1.3 Activation Functions

Activation functions introduce non-linearity into the network, allowing it to learn more complex patterns. Some commonly used activation functions are:

- **Sigmoid**:

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

  Used in binary classification problems.

- **ReLU (Rectified Linear Unit)**:
$$f(z) = \max(0, z)$$
This is the most commonly used activation function due to its simplicity and efficiency.
- **Tanh**:
$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
- **Softmax**: Used in multi-class classification problems. It transforms the raw values into probabilities for each class.

1.4 Forward Propagation

In a neural network, forward propagation refers to the process of passing the input data through the layers of the network, calculating the weighted sums at each neuron, and applying the activation functions. The output of the last layer is the network's prediction.

For a simple feedforward neural network:

$$\text{Layer 1 Output: } a^{[1]} = f(W^{[1]}x + b^{[1]})$$

$$\text{Layer 2 Output: } a^{[2]} = f(W^{[2]}a^{[1]} + b^{[2]})$$

This process continues until the output layer is reached.

1.5 Cost Function

The cost function measures the error in the network's predictions. For a classification task, a common cost function is cross-entropy:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))\right]$$

Where $y^{(i)}$ is the true label, and $h_\theta(x^{(i)})$ is the predicted probability for the i-th example.

1.6 Backpropagation Algorithm

Backpropagation is the process used to update the weights of the network based on the gradient of the cost function. The steps include:

1. Compute the error at the output layer.
2. Propagate the error backward through the hidden layers.
3. Update the weights using gradient descent.

The gradients of the cost function with respect to the weights are calculated using the chain rule of calculus.

1.6.1 Weight Update Rule

In gradient descent, the weights are updated as follows:

$$W = W - \alpha\frac{\partial J}{\partial W}$$

Where α is the learning rate, and $\frac{\partial J}{\partial W}$ is the gradient of the cost function with respect to the weights.

1.7 Neural Networks vs Linear and Logistic Models

While neural networks are capable of learning complex patterns, linear and logistic regression models are limited to simpler relationships.

1.7.1 Linear Regression

Linear regression predicts a continuous target variable by fitting a line to the data. The model is:

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

It assumes that the relationship between the input features and the target is linear.

1.7.2 Logistic Regression

Logistic regression is used for binary classification tasks. The prediction is made using a logistic (sigmoid) function:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Logistic regression is essentially a linear model with a non-linear transformation applied to the output.

1.7.3 Differences between Neural Networks and Linear/Logistic Models

- **Non-linearity**: Neural networks can learn non-linear relationships due to their activation functions, whereas linear models cannot.
- **Depth**: Neural networks can have multiple hidden layers (deep networks), allowing them to learn hierarchical patterns. Linear and logistic regression are "shallow" models.
- **Feature Engineering**: Linear models often require extensive feature engineering, while neural networks can automatically learn features from data.
- **Complexity**: Neural networks can approximate any continuous function (universal approximation theorem), whereas linear models are limited to linear separability.
- **Performance on Large Data**: Neural networks tend to outperform linear models when provided with large amounts of data, as they can learn complex relationships.

1.8 Practical Considerations

1.8.1 Overfitting and Regularization

Neural networks with many parameters can easily overfit the training data. Regularization techniques such as L2 regularization or dropout can be applied to mitigate overfitting.

1.8.2 Learning Rate and Convergence

The learning rate controls the size of the weight updates during training. If the learning rate is too small, training will be slow. If it is too large, the network may never converge to a good solution.

1.9 Conclusion

Neural networks are a powerful class of models capable of learning complex, non-linear relationships in data. While linear and logistic regression models are limited in their capacity to model complex patterns, neural networks can approximate any function, making them suitable for a wide range of applications. However, they require more data and computational resources and are more prone to overfitting, requiring careful tuning of hyperparameters.

2 What is PyTorch?

PyTorch is a Python-based scientific computing package serving two broad purposes:

1. A replacement for NumPy to use the power of GPUs and other accelerators.
2. An automatic differentiation library that is useful to implement neural networks.

Some key features of PyTorch:

1. PyTorch provides tensors and dynamic neural networks with strong GPU acceleration. Tensors are similar to NumPy arrays and can run on GPUs.
2. It has an autograd system that automatically calculates gradients required for backpropagation in neural networks. This makes training models easier and more intuitive.
3. PyTorch has modular, readable and flexible code making it easy to use and integrate into existing Python programs.
4. It supports rapid prototyping through Python libraries and tools like IPython/Jupyter notebooks.
5. PyTorch can convert models into production-ready models in C++ and Python using TorchScript.
6. It has distributed training capabilities making it easy to train models across multiple GPUs/nodes.

PyTorch is optimized for fast, flexible deep learning research and development. It is gaining popularity due to its ease of use and focus on Python programming. Key advantages include GPU support, automated gradients, modular design and Python focus.

3 Simple Implementation of a Neural Network using Pytorch

A PyTorch implementation of a neural network looks exactly like a NumPy implementation. The goal of this section is to showcase the equivalent nature of PyTorch and NumPy. For this purpose, let's create a simple three-layered network having 5 nodes in the input layer, 3 in the hidden layer, and 1 in the output layer. We will use only one training example with one row which has five features and one target.

```
1 import torch
2 n_input, n_hidden, n_output = 5, 3, 1
```

A simple neural network can be defined and trained in five key steps:

1. parameter initialization
2. Forward Propagation

3. Loss computation
4. Backpropagation
5. Updating the parameters

Let's see each of these steps in a bit more detail.

3.1 Parameter Initialization:

The first step is to do parameter initialization. Here, the weights and bias parameters for each layer are initialized as the tensor variables. Tensors are the base data structures of PyTorch which are used for building different types of neural networks. They can be considered as the generalization of arrays and matrices; in other words, tensors are N-dimensional matrices.

```
## Initialize tensor for inputs and outputs
x = torch.randn((1, n_input))
y = torch.randn((1, n_output))
## Initialize tensor variables for weights
w1 = torch.randn(n_input, n_hidden) # weight for hidden layer
w2 = torch.randn(n_hidden, n_output) # weight for output layer
## initialize tensor variables for bias terms
b1 = torch.randn((1, n_hidden)) # bias for hidden layer
b2 = torch.randn((1, n_output)) # bias for output layer
```

3.2 Forward Propagation:

In this step, activations are calculated at every layer using the two steps shown below. These activations flow in the forward direction from the input layer to the output layer in order to generate the final output.

```
z = weight * input + bias
a = activation_function (z)
## sigmoid activation function using pytorch
def sigmoid_activation(z):
return 1 / (1 + torch.exp(-z))
## activation of hidden layer
z1 = torch.mm(x, w1) + b1
a1 = sigmoid_activation(z1)
## activation (output) of final layer
z2 = torch.mm(a1, w2) + b2
output = sigmoid_activation(z2)
```

The code blocks show how we can write these steps in PyTorch. Notice that most of the functions, such as exponential and matrix multiplication, are similar to the ones in NumPy.

3.3 Loss Computation:

In this step, the error (also called loss) is calculated in the output layer. A simple loss function can tell the difference between the actual value and the predicted value. Later, we will look at different loss functions available in PyTorch.

```
loss = y - output
```

3.4 Backpropagation:

The aim of this step is to minimize the error in the output layer by making marginal changes in the bias and the weights. These marginal changes are computed using the derivatives of the error term. Based on the Calculus principle of the Chain rule, the delta changes are back passed to hidden layers where corresponding changes in their weights and bias are made. This leads to an adjustment in the weights and bias until the error is minimized.

```
## function to calculate the derivative of activation
def sigmoid_delta(x):
return x * (1 - x)
## compute derivative of error terms
delta_output = sigmoid_delta(output)
delta_hidden = sigmoid_delta(a1)
## backpass the changes to previous layers
d_outp = loss * delta_output
loss_h = torch.mm(d_outp, w2.t())
d_hidn = loss_h * delta_hidden
```

3.5 Updating the Parameters:

Finally, the weights and biases are updated using the delta changes received from the above back-propagation step.

```
learning_rate = 0.1
w2 += torch.mm(a1.t(), d_outp) * learning_rate
w1 += torch.mm(x.t(), d_hidn) * learning_rate
b2 += d_outp.sum() * learning_rate
b1 += d_hidn.sum() * learning_rate
```

Finally, when these steps are executed for a number of epochs with a large number of training examples, the loss is reduced to a minimum value. The final weight and bias values are obtained which can then be used to make predictions on the unseen data.

4 Autograd Feature of PyTorch

PyTorch's Autograd feature is part of what makes PyTorch flexible and fast for building machine learning projects. It allows for the rapid and easy computation of multiple partial derivatives (also referred to as gradients) over a complex computation. This operation is central to backpropagation-based neural network learning.

The power of Autograd comes from the fact that it traces your computation dynamically at run-time, meaning that if your model has decision branches or loops whose lengths are not known until runtime, the computation will still be traced correctly, and you'll get correct gradients to drive learning. This, combined with the fact that your models are built in Python, offers far more flexibility than frameworks that rely on static analysis of a more rigidly structured model for computing gradients.

For more information refer to this YouTube video:
`https://www.youtube.com/watch?v=M0fX15_-xrY&t=9s&ab_channel=PyTorch`.

So, the good news is you do not need to compute derivatives when using Pytorch. Pytorch will handle that for you.

5 Building Models with PyTorch (The PyTorch Way)

The simple implementation that we learned above is not suitable for building neural network models efficiently and with ease. PyTorch has a lot of features that let you create and train various types of neural networks with ease. Learn about the different tools PyTorch makes available to create different types of neural networks from this notebook:
https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/fe726e041160526cf828806536922cf6/modelsyt_tutorial.ipynb

You may also refer to this YouTube video:
https://www.youtube.com/watch?v=OSqIP-mOWOI&list=PL_lsbAsL_o2CTlGHgMxNrKhzP97BaG9ZN&index=4&ab_channel=PyTorch

6 PyTorch Tensorboard Support

TensorBoard is a well-known tool for visualizing the activity of machine learning models and training progress. PyTorch provides support for TensorBoard use, with no dependency on other ML frameworks. This video here
https://www.youtube.com/watch?v=6CEld3hZgqc&list=PL_lsbAsL_o2CTlGHgMxNrKhzP97BaG9ZN&index=5&ab_channel=PyTorch
will show you how to visualize graphs of training progress, input images, model structure, and dataset embeddings with TensorBoard and PyTorch. An example is shown in this notebook
https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/e2e556f6b4693c2cef716dd7f40caaf6/tensorboardyt_tutorial.ipynb

The notebook above shows you how to use Tensorboard in a new browser tab. If you run the notebook in Google Colab you just need to run the following code to start Tensorboard.

```
%load_ext tensorboard
%tensorboard --logdir runs
```