Published in BiaslyAI

Andrea Eunbee Jang Follow

Feb 26, 2019 · 7 min read · ✦ · Listen

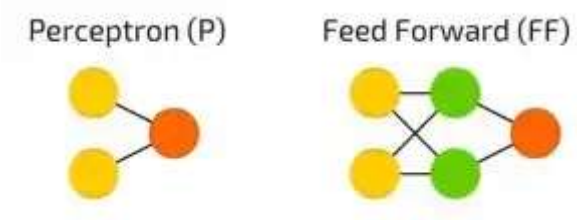
PyTorch: Introduction to Neural Network — Feedforward / MLP

In the last tutorial, we've seen a few examples of building simple regression models using PyTorch. In today's tutorial, we will build our very first neural network model,

namely, the feedforward neural network model.

- Autograd
- Single-layer Perceptron
- So what is the activation function?
- Feedforward Neural Network
- Training Example

Introduction



Yellow: Input layer | **Green**: Hidden layer | **Orange**: Output layer

The feedforward neural network is the simplest network introduced. It is an extended version of perceptron with additional hidden nodes between the input and the output layers. In this network, data moves in the only forward direction without any cycles or loops.

Autograd

Before jumping into building the model, I would like to introduce `autograd`, which is an automatic differentiation package provided by PyTorch. This is a must-have package when performing the gradient descent for the optimization of the neural network models. To use this function, you need to initialize your tensor with `requires_grad=True`. If it is set to `False` then the gradient will not be calculated automatically. Moreover, the gradient can only be calculated if your variable is a function of the variable you want to differentiate. In our case, it is the function of x as seen below.


```
import torch

x = torch.ones(1, requires_grad=True)

print(x.grad)    # returns None
```

As you see above, `print(x.grad)` will return `None` since `x` is a tensor we initialized, a scalar, so there is nothing to be calculated. On the other hand, the gradient for `x` is calculated in the code below because `z` is a function of `y` and `y` is a function of `x`.

```
x = torch.ones(1, requires_grad=True)
y = x + 2
z = y * y * 2

z.backward()     # automatically calculates the gradient
print(x.grad)    # ∂z/∂x = 12
```

Here, `.backward()` performs the backpropagation automatically and `x.grad` holds the calculated gradient for `x`. In mathematical notation, `x.grad` is ∂z/∂x. I've included the more read-friendly version of the same calculation below.

$$\frac{\partial z}{\partial x} = \frac{\partial 2y^2}{\partial x}$$

$$= 4y \frac{\partial y}{\partial x}$$

$$= 4(x+2) \frac{\partial (x+2)}{\partial x}$$

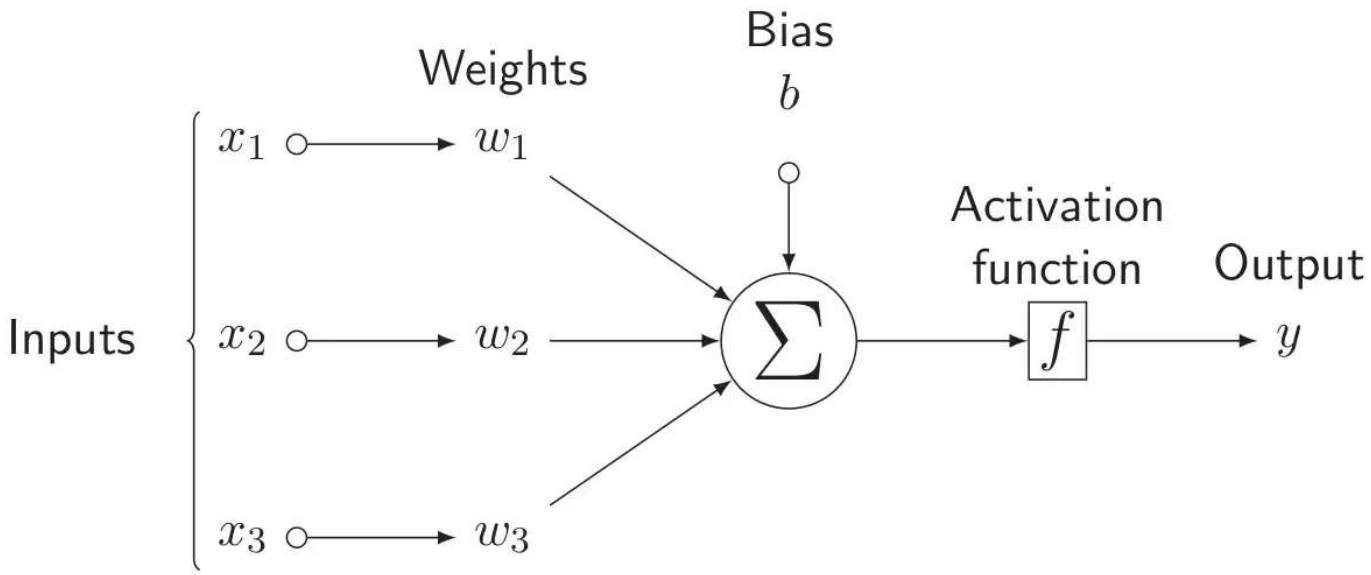
$$= 4(x+2)$$

$$\Rightarrow x = 1$$

$$\Rightarrow \text{Answer is } 12$$

Single-layer Perceptron

Let's first observe how the single-layer perceptron model is implemented and compare it with the feedforward model. Single-layer perceptron takes data as input and its weights are summed up then an activation function is applied before sent to the output layer. For the single-layer perceptron model, the heavy-side step function is the activation that is normally used.



https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network

However, in the example code for the perceptron below I'm using `ReLU()` since heavy-side step function is non-differentiable at x = 0 and it has 0 derivatives elsewhere, meaning the gradient descent won't be able to make a progress in weight updates. Thus, the heavy-side step function is not suitable for the deep neural network.

So what is the activation function?

The activation functions in the neural network introduce the non-linearity to the linear output. It defines the output of a layer, given data, meaning it sets the threshold for making the decision of whether to pass the information or not. You can choose different activation functions depending on the task you would like to achieve. To learn more about activation functions, here is a tutorial I found to be useful.

So in the example we're about to see below, the fully-connected layer `self.fc` outputs linear information and `self.relu` makes it non-linear. One thing to note is that without the activation functions, multiple linear layers are equivalent to a single layer in the neural network.

Now, back to the perceptron model.

```
class Perceptron(torch.nn.Module):
    def __init__(self):
        super(Perceptron, self).__init__()
        self.fc = nn.Linear(1,1)
        self.relu = torch.nn.ReLU() # instead of Heaviside step fn

    def forward(self, x):
        output = self.fc(x)
        output = self.relu(x) # instead of Heaviside step fn
        return output
```

As you can see, the input `x` is passed to a fully-connected layer `self.fc(x)`, then a step function `self.relu(x)` is applied and is returned as an output. But wait, doesn't this seem very similar to the logistic regression model we've built in the previous post? In fact, yes it is. In the logistic regression, we used `Sigmoid` as activation and here we're using `ReLU`. If no activation function is used, then it will be exactly the same as the linear regression model we've seen in the previous tutorial.

Feedforward Neural Network

Now let's look at how to build a simple feedforward network model. Unlike the single-layer perceptron, the feedforward models have hidden layers in between the input and the output layers. After every hidden layer, an activation function is applied to introduce non-linearity. Below is an example feedforward model I built.

```
class Feedforward(torch.nn.Module):
        def __init__(self, input_size, hidden_size):
            super(Feedforward, self).__init__()
            self.input_size = input_size
            self.hidden_size  = hidden_size
            self.fc1 = torch.nn.Linear(self.input_size,
self.hidden_size)
            self.relu = torch.nn.ReLU()
            self.fc2 = torch.nn.Linear(self.hidden_size, 1)
            self.sigmoid = torch.nn.Sigmoid()

        def forward(self, x):
            hidden = self.fc1(x)
            relu = self.relu(hidden)
            output = self.fc2(relu)
            output = self.sigmoid(output)
            return output
```

For this example, I only put one hidden layer but you can add as many hidden layers as you want. When you have more than two hidden layers, the model is also called the **deep/multilayer feedforward model** or **multilayer perceptron model(MLP)**.

After the hidden layer, I use `ReLU` as activation before the information is sent to the output layer. This is to introduce non-linearity to the linear output from the hidden layer as mentioned earlier. What `ReLU` does here is that if the function is applied to a set of numerical values, any negative value will be converted to 0 otherwise the values stay the same. For example, if the input set is [-1,0,4,-5,6] then the function will return [0,0,4,0,6].

As an output activation function, I used `Sigmoid`. This is because the example I want to show you later is a binary classification task, meaning we have binary categories to predict from. `Sigmoid` is the good function to use because it calculates the probability(ranging between 0 and 1) of the target output being label 1. As said in the previous section, the choice of the activation function depends on your task. Now, let's see a binary classifier example using this model.

. . .

Training Example

Create random data points

For this tutorial, I am creating random data points using Scikit Learn's `make_blobs` function and assign binary labels {0,1}. I thought of using a real dataset but my colleague suggested having a separate tutorial for data loading since there are multiple ways to do so in PyTorch.

```
# CREATE RANDOM DATA POINTS

from sklearn.datasets import make_blobs

def blob_label(y, label, loc): # assign labels
    target = numpy.copy(y)
    for l in loc:
        target[y == l] = label
    return target

x_train, y_train = make_blobs(n_samples=40, n_features=2,
cluster_std=1.5, shuffle=True)
x_train = torch.FloatTensor(x_train)
y_train = torch.FloatTensor(blob_label(y_train, 0, [0]))
y_train = torch.FloatTensor(blob_label(y_train, 1, [1,2,3]))

x_test, y_test = make_blobs(n_samples=10, n_features=2,
cluster_std=1.5, shuffle=True)
x_test = torch.FloatTensor(x_test)
y_test = torch.FloatTensor(blob_label(y_test, 0, [0]))
y_test = torch.FloatTensor(blob_label(y_test, 1, [1,2,3]))
```

Model, Criterion, Optimizer

Let's define the model with input dimension 2 and hidden dimension 10. For the loss function (criterion), I'm using `BCELoss()` (Binary Cross Entropy Loss) since our task is to classify binary labels. The optimizer is `SGD` (Stochastic Gradient Descent) with learning rate 0.01.

```
model = Feedforward(2, 10)
criterion = torch.nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
```

Train the model

To see how the model is improving, we can check the test loss before the model training and compare it with the test loss after the training.

```
model.eval()
y_pred = model(x_test)
before_train = criterion(y_pred.squeeze(), y_test)
print('Test loss before training' , before_train.item())
```

`model.eval()` here sets the PyTorch module to evaluation mode. We want to do this because we don't want the model to learn new weights when we just want to check the loss before training. To train, we switch the mode back to training mode.

```
model.train()
epoch = 20

for epoch in range(epoch):

    optimizer.zero_grad()

    # Forward pass
    y_pred = model(x_train)

    # Compute Loss
    loss = criterion(y_pred.squeeze(), y_train)

    print('Epoch {}: train loss: {}'.format(epoch, loss.item()))

    # Backward pass
    loss.backward()
    optimizer.step()
```

Let's start training. First I switch the module mode to `.train()` so that new weights can be learned after every epoch. `optimizer.zero_grad()` sets the gradients to zero before we start backpropagation. This is a necessary step as PyTorch accumulates the gradients from the backward passes from the previous epochs.

After the forward pass and the loss computation, we perform backward pass by calling `loss.backward()`, which computes the gradients. Then `optimizer.step()` updates the weights accordingly.

Evaluation

Okay, the training is now done. Let's see how the test loss changed after the training. Again, we switch the module mode back to the evaluation mode and check the test loss as the example below.

```
model.eval()
y_pred = model(x_test)
after_train = criterion(y_pred.squeeze(), y_test)
print('Test loss after Training' , after_train.item())
```

In order to improve the model, you can try out different parameter values for your hyperparameters(ie. hidden dimension size, epoch size, learning rates). You can also try changing the structure of your model (ie. adding more hidden layers) to see if your mode improves. There is a number of different hyperparameter and model selection techniques popularly used but this is the general idea behind it. In the end, you can select the hyperparameters and the model structure that gives you the best performance.

. . .

PyTorch Tutorial Schedule

1. Learn PyTorch Basics
2. Linear and Logistic Regression Models
3. Introduction to Neural Network (Feedforward)
4. Data loading in PyTorch
5. RNN/LSTM Text Classifier
6. RNN/LSTM Language Model
7. Exploring Text Classification with CNN
8. Can Language Model work with CNN?

Machine Learning Tutorial Pytorch Deep Learning AI