Regression with Pytorch

This tutorial uses :
Skilearn *LinearRegression*() function, and
Torch **nn. module**. *LinearRegression*()
Torch **nn.MSELoss()** function.

In this lab, you need to use the code with different parameters:

- Loss function
- Bach
- Import data from : `/kaggle/input/random-linear-regression/train.csv`

(use pandas function read_csv)

PyTorch Neural Network Classification

For example, you might want to:

Problem type	What is it?	Example
Binary classification	Target can be one of two options, e.g. yes or no	Predict whether or not someone has heart disease based on their health parameters.
Multi-class classification	Target can be one of more than two options	Decide whether a photo is of food, a person or a dog.
Multi-label classification	Target can be assigned more than one option	Predict what categories should be assigned to a Wikipedia article (e.g. mathematics, science & philosophy).

Classification, along with regression (predicting a number) is one of the most common types of machine learning problems.

Architecture of a classification neural network

Before we get into writing code, let's look at the general architecture of a classification neural network.

Hyperparameter	Binary Classification	Multiclass classification
Input layer shape (`in_features`)	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification

Hyperparameter	Binary Classification	Multiclass classification
Neurons per hidden layer	Problem specific, generally 10 to 512	Same as binary classification
Output layer shape (`out_features`)	1 (one class or the other)	1 per class (e.g. 3 for food, person or dog photo)
Hidden layer activation	Usually ReLU (rectified linear unit) but can be many others	Same as binary classification
Output activation	Sigmoid (`torch.sigmoid` in PyTorch)	Softmax (`torch.softmax` in PyTorch)
Loss function	Binary crossentropy (`torch.nn.BCELoss` in PyTorch)	Cross entropy (`torch.nn.CrossEntropyLoss` in PyTorch)
Optimizer	SGD (stochastic gradient descent), Adam (see `torch.optim` for more options)	Same as binary classification

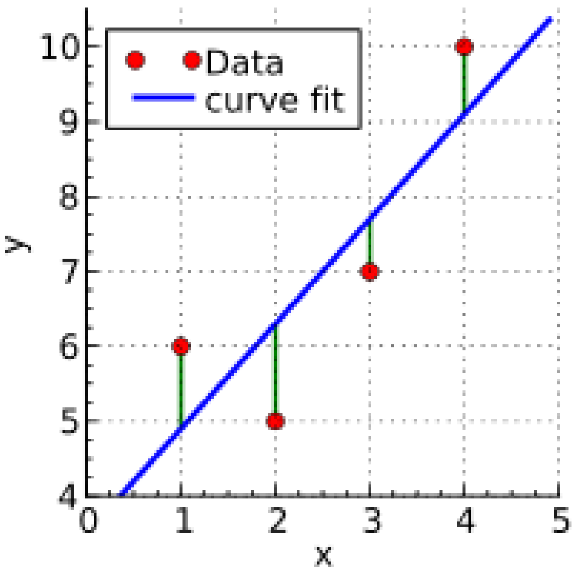
Of course, this ingredient list of classification neural network components will vary depending on the problem you're working on.

Linear regression

The mathematical model:

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^\mathsf{T} \boldsymbol\beta + \varepsilon_i, \qquad i = 1, \ldots, n,$$

In linear regression, the observations (**red**) are assumed to be the result of random deviations (**green**) from an underlying relationship (**blue**) between a dependent variable (*y*) and an independent variable (*x*).



Function: *LinearRegression()*

Library: sklearn

class sklearn.linear_model.LinearRegression(**, fit_intercept=True, copy_X=True, n_jobs=None, positive=False*)[source]

Ordinary least squares Linear Regression.

LinearRegression fits a linear model with coefficients w = (w1, …, wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Library: Pytorch

The class for linear regression, that inherits **torch.nn.Module** which is the basic Neural Network module containing all the required functions.

After that, we initialize the loss (**Mean Squared Error**) and build a ***forward*** function based on a simple linear regression equation that can be used in the training of this model.

Function: MSELoss() # (**Mean Squared Error**)

In pytorch, we can use **torch.nn.MSELoss()** to compute the mean squared error (MSE). In this tutorial, we will use some examples to show you how to use it.

Example: **torch.nn.MSELoss**(size_average=None, reduce=None, reduction='mean')

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE	= mean squared error
n	= number of data points
Y_i	= observed values
$\hat{Y}_i$	= predicted values

Disable gradient calculation

torch. no_grad() : **disables gradient calculation**. So, the reason why it uses less memory is that it's not storing any Tensors that are needed to calculate gradients of your loss. Also, because you don't store anything for the backward pass, the evaluation of your network is quicker (and use less memory).