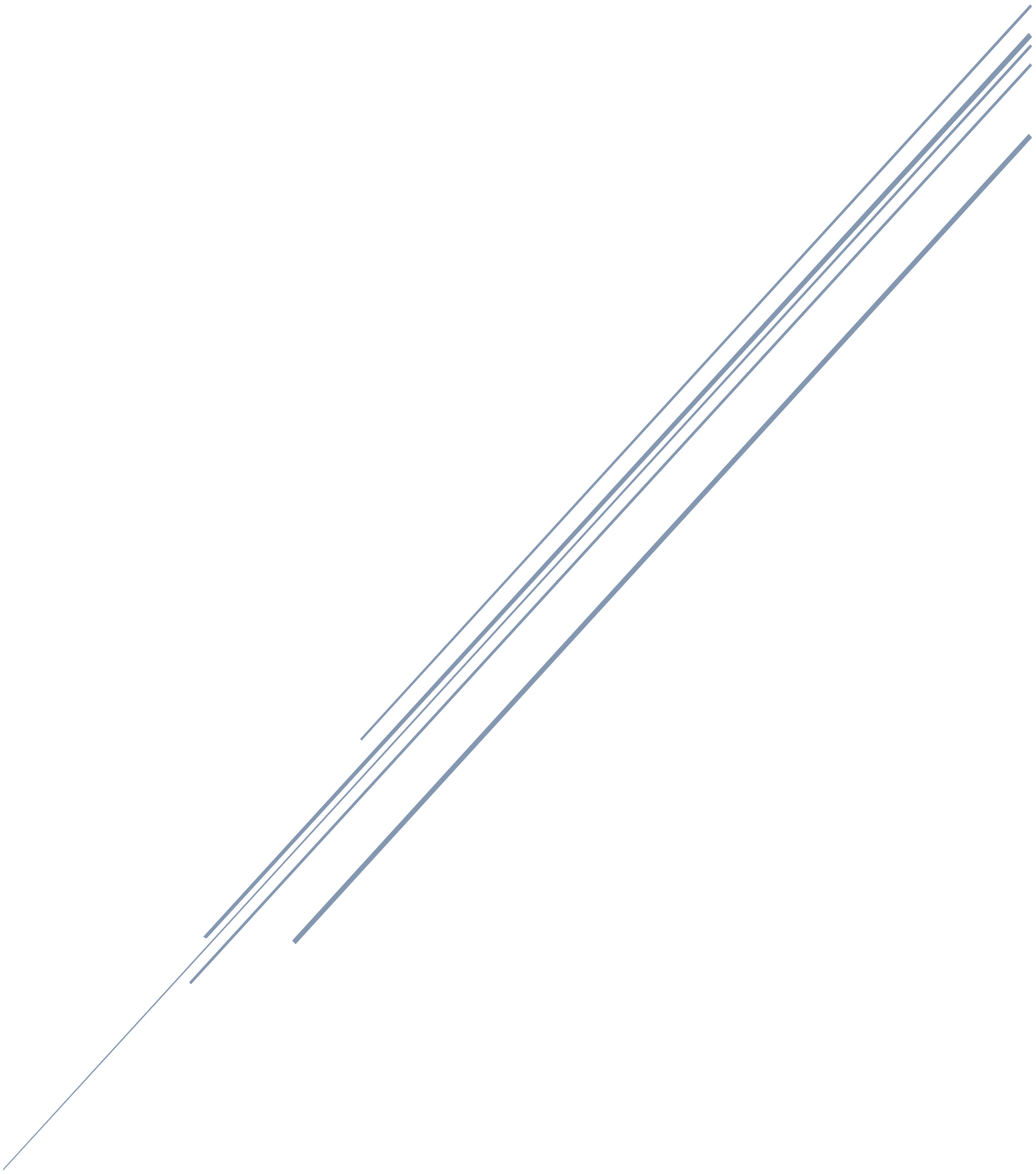
LINEAR REGRESSION

Module 2



v-cardona

Deep Learning with Python and PyTorch

Table of contents

Linear Regression in 1D – Prediction

- Define w and b for y = wx + b with requires_grad = True

- You can make prediction for multiple inputs, like one row per prediction and result will be in each row

- The linear class can be used to make a prediction. We can also use the linear class to build more complex models. Set the random seed because the parameters are randomly initialized:
  torch.manual_seed(1)

- Create Linear Regression Model:
  lr = Linear(in_features=1, out_features=1, bias=True)

- Customize Linear Regression Class, the constructor with input and output size and de prediction funciotion:
  class LR(nn.Module):

```
# Constructor
def __init__(self, input_size, output_size):

    # Inherit from parent
    super(LR, self).__init__()
    self.linear = nn.Linear(input_size, output_size)

# Prediction function
def forward(self, x):
    out = self.linear(x)
    return out
```

- Get parameters of the linear regression:
  lr = LR(1, 1)
  print("The parameters: ", list(lr.parameters()))
  print("Linear model: ", lr.linear)

Linear Regression Training

We use these data points, we train the model, and for the case of linear regression, we are going to get a set of parameters, the bias and slope term. We will make a prediction and in PyTorch this is called the forward step. Therefore, we obtain the equation of the line and we can make a prediction for any of the x-values.

Each row of the tensor represents a different data point.

One important thing to notice is the higher the variance or standard deviation of the noise the more the points deviate from the line, noise power.

Loss

In order to find the parameter we need to find how good our model is. We need a quantity that is near zero when our model provides a good estimate and large when our model estimate is bad. We call this quantity the loss. We do this by subtracting our model estimate, in this case Y hat, with their actual value, Y. We are essentially finding the distance from a model estimate to the actual value we are trying to predict.

Gradient Descent

In general, gradient descent is recursively calculated. We start with an initial guess and for the first iteration we simply add a value proportional to the derivative. We update the parameter value and we repeat for the second iteration. And we continue to the k-th iteration. The parameter theta is known as a learning rate, usually selected empirically. But if it is too large it will miss the minimum and if it is too small it will take a lot to converge.

$$w^0$$

$$w^1 = w^0 - \eta \frac{dl(w^0)}{dw}$$

$$w^2 = w^1 - \eta \frac{dl(w^1)}{dw} \qquad \eta\text{:learning rate}$$

$$\vdots$$

$$w^{k+1} = w^k - \eta \frac{dl(w^k)}{dw}$$

Cost

The sum or average of the loss is called the cost. The first variable is the slope, and as we adjust the slope the value for the slope parameter changes. Similarly, it is also a function of the bias; the bias controls the offset of the line.

Training parameters in PyTorch

- Create forward function for prediction:

```
def forward(x):
    return w * x
```

- Define the cost or criterion function using MSE (Mean Square Error):

```
def criterion(yhat, y):
    return torch.mean((yhat - y) ** 2)
```

- Create Learning Rate and an empty list to record the loss for each iteration:

```
lr = 0.1
LOSS = []
```

- Create a model parameter by setting the argument `requires_grad` to `True` because the system must learn it:

```
w = torch.tensor(-10.0, requires_grad = True)
```

- Define a function for train the model:

```
def train_model(iter):
  for epoch in range (iter):

    # make the prediction as we learned in the last lab
    Yhat = forward(X)

    # calculate the iteration
    loss = criterion(Yhat,Y)

    # plot the diagram for us to have a better idea
    gradient_plot(Yhat, w, loss.item(), epoch)

    # store the loss into list
    LOSS.append(loss)

    # backward pass: compute gradient of the loss with respect to all the learnable parameters
    loss.backward()

    # updata parameters
    w.data = w.data - lr * w.grad.data

    # zero the gradients before running the backward pass
    w.grad.data.zero_()
```

- The parameter value is sensitive to initialization

Training with slope and bias

The slope controls a relationship between x and y and there is a function of the bias parameter. The bias parameter controls the vertical offset of the line. A true linear regression, the loss function is a function of two variables, the slope and bias.

Stochastic, Batch and Mini-Batch Gradient Descent

Stochastic Gradient Descent

In stochastic gradient descent, we are minimizing or cost or total loss by minimizing it one sample at a time. The difference with normal gradient descent is that we use all the data in each iteration and here only one data.

In one iteration, we apply the gradient choosing one sample each time and for all data samples.

Model train:

```
def train_model_SGD(iter):
  # Loop
  for epoch in range(iter):
    # SGD is an approximation of out true total loss/cost, in this line of code we calculate our true loss/cost and store it
    Yhat = forward(X)
    # store the loss
    LOSS_SGD.append(criterion(Yhat, Y).tolist())
    for x, y in zip(X, Y):
      # make a pridiction
      yhat = forward(x)
      # calculate the loss
      loss = criterion(yhat, y)
      # Section for plotting
      get_surface.set_para_loss(w.data.tolist(), b.data.tolist(), loss.tolist())
      # backward pass: compute gradient of the loss with respect to all the learnable parameters
      loss.backward()
      # update parameters slope and bias
      w.data = w.data - lr * w.grad.data
      b.data = b.data - lr * b.grad.data
      # zero the gradients before running the backward pass
      w.grad.data.zero_()
      b.grad.data.zero_()
    #plot surface and data space after each epoch
    get_surface.plot_ps()
```

Mini-Batch Gradient Descent

We choose a number of samples dates to calculate the iteration gradient. It is like stochastic but instead of choose only one, we choose a number of samples to do in a iteration, and then we continue with the rest of samples until we have done with all of them. Iterations = training size / batch size

In PyTorch in the constructor of DataLoader we can choose the size of batch (batch_size=4). And proceed like working with stochastic gradient.

- trainloader = DataLoader(dataset = dataset, batch_size = 20)

PyTorch Way

- *Optimizer.step() ->* *w.data = w.data - lr * w.grad.data*

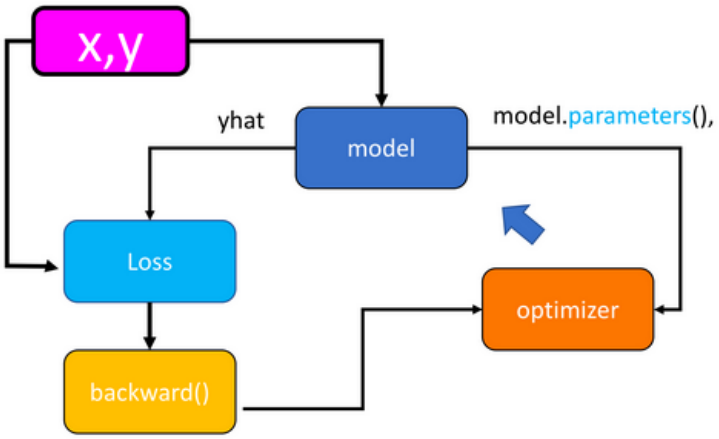
*b.data = b.data - lr * b.grad.data*

- Calculate the total loss or cost:

criterion = nn.MSELoss()

- optimizer object:
  model = linear_regression(1,1)
  optimizer = optim.SGD(model.parameters(), lr = 0.01)

PyTorch randomly initialises your model parameters. If we use those parameters, the result will not be very insightful as convergence will be extremely fast.



- Example with optimizer (only change the optimizer.zero_grad() before, for reset the grad and calculate the new data with optimizer.step()):

```
optimizer = optim.SGD(model.parameters(), lr = 0.1)
trainloader = DataLoader(dataset = dataset, batch_size = 1)
def my_train_model(iter):
    for epoch in range(iter):
        for x,y in trainloader:
            yhat = model(x)
            loss = criterion(yhat, y)
            get_surface.set_para_loss(model, loss.tolist())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        get_surface.plot_ps()

train_model_BGD(10)
```

Model Validation

We usually split our data into three parts: training data, validation data and test data.

Training and validation data

Training data to obtain model parameters, such as slope and bias (w ,b)

Validation data to obtain hyper-parameters such as mini-batch size and learning rate (batch_size, lr). We compare two models; we do this by calculating the cost or total loss of each model.

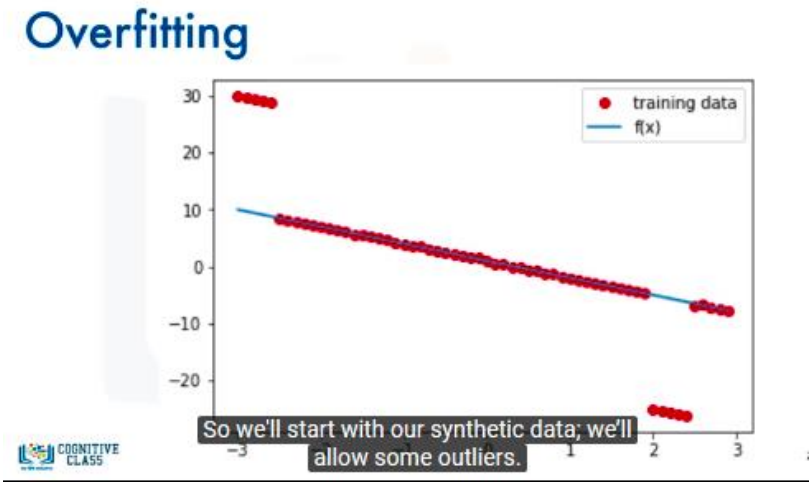
If the best model for validation data and the best for training data contradicts each other, it can be producing overfitting.

The loss on the validation data is usually higher than the loss in the training data.

Learning rate:

- Line minimizes our loss under validation data. This also corresponds to the line that maximizes the total loss or cost on the training data.
- Line is obtained by the maximum number of iterations. You can see it minimizes our total loss or cost on our testing data

When our model performed extremely well on our training data and the extremely poorly on our validation data this implies overfitting, where a model is fitting the noise or the outliers.



Early stopping

In early stopping, for every iteration, we'll take our model, and then we'll calculate the loss using our validation data. We will save the best model and on each iteration we will compare to the best model saved, and if the new one is better we will change it. We will compare each iteration model with the best model saved.

Higher dimensional linear regression

Multiple linear regression prediction

In multiple linear regression, we have many variables, or features. And for every feature we have a parameter, we have the bias parameter. And every variable has its own parameter analogous to the slope: for x1 we have B1; for X2 we had B2; and so on.

When defining linear regression we have to match de in_features and out_features witch the dimension of the samples.

To keep in mind that in the prediction function it has to use matrix multiplication instead of scaler multiplication:

```
def forward(x):
    yhat = torch.mm(x, w) + b
    return yhat
```

- Make a linear regression model using build-in function:

```
model = nn.Linear(2, 1)
```

- Create a linear regression class:

```
class linear_regression(nn.Module):

    # Constructor
    def __init__(self, input_size, output_size):
        super(linear_regression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    # Prediction function
    def forward(self, x):
        yhat = self.linear(x)
        return yhat
```

- Obtain the parameters with .parameters() or .state_dict():

```
print("The parameters: ", list(model.parameters()))
print("The parameters: ", model.state_dict())
```

Multiple linear regression training

Multiple output linear regression

Multiple output linear regression training