Introduction to Neural Networks and Pytorch (M4)

Created	@December 16, 2025 9:18 PM
Module Code	IBMM04 : Introduction to Neural Networks and Pytorch

Module 2: Linear Regression

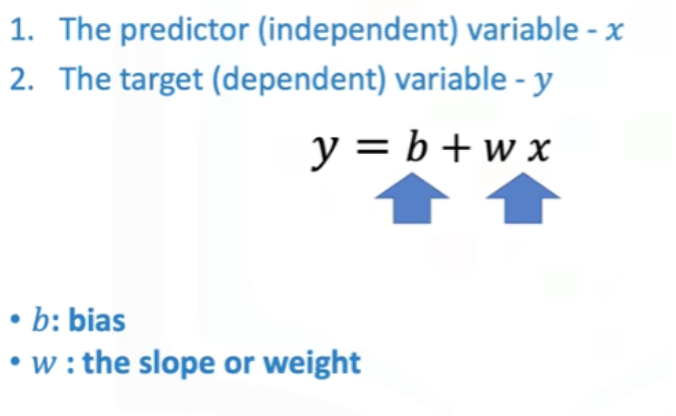
Linear Regression Prediction

1. What 1D Linear Regression is

- Models relationship between **one input feature** `x` and **output** `y`
- Equation:

y^=wx+b

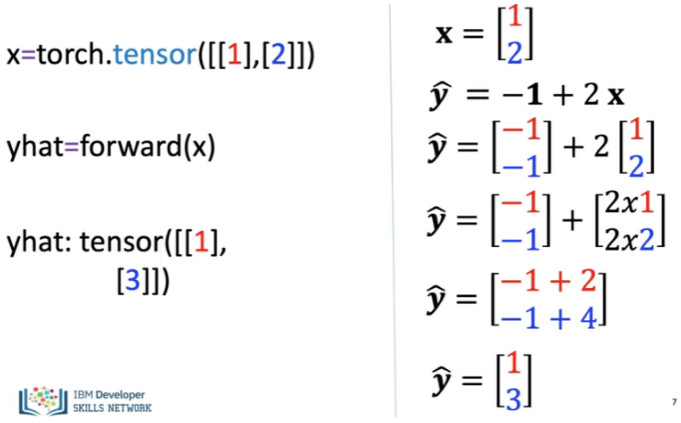
- `w` → weight (slope)
- `b` → bias (intercept)
- `ŷ` → predicted value

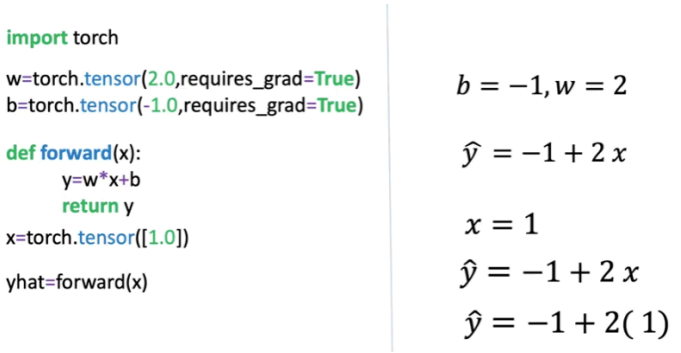


2. Prediction = Forward Pass

- Given `x`, compute `ŷ`
- In PyTorch, this computation is called the **forward step**
- Same formula applies to:
  - Single value (`1×1` tensor)
  - Multiple samples (`N×1` tensor)

Each row is treated as an **independent sample**.





3. Manual model using tensors

- `w` and `b` are tensors with `requires_grad=True`
- Forward function:

$\hat{y} = wx + b$

- PyTorch applies this **row-wise automatically** for batch inputs

Purpose: understand mechanics, **not used in practice**

---

4. Built-in `nn.Linear`

- PyTorch provides `nn.Linear(in_features, out_features)`
- For 1D regression:

```
in_features = 1
out_features = 1
```

- Internally implements:

$\hat{y} = xW^T + b$

Key points:

- Weights & bias are **learnable parameters**
- Randomly initialized
- Accessible via:

```
model.parameters()
```

---

5. Calling the model

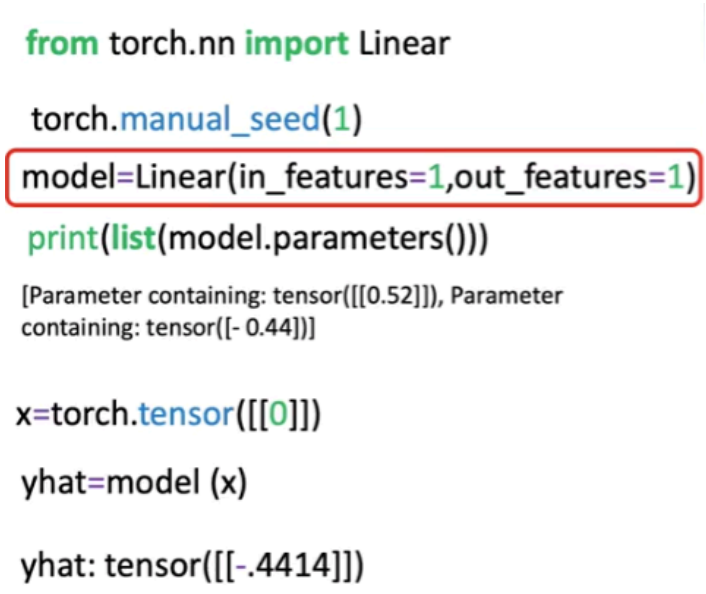
- You **do NOT** call `forward()` explicitly
- This:

```
y_hat = model(x)
```

automatically calls:

```
model.forward(x)
```

This is standard PyTorch design.



---

6. Batch prediction behavior

Input shape: (N,1)

Output shape: N,1)(N,1)

- Same line applied to **every row**
- Bias is broadcasted across rows

---

7. Custom Linear Regression Module

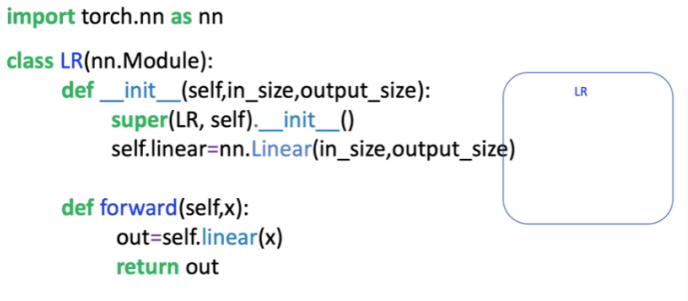
Why create one?

- To combine layers
- To build scalable models

How:

- Subclass `nn.Module`
- Define layers in `__init__`
- Define computation in `forward`

Your custom class becomes a **callable model**.



---

8. Model parameters & `state_dict`

- `parameters()` → learnable tensors
- `state_dict()` → dictionary mapping:

```
layer.weight →values
layer.bias   →values
```

Used for:

- Saving models
- Loading models
- Manually modifying parameters

```
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

    lr = LR(1, 1)

    X= torch.tensor([[1.0], [2.0], [3.0]])

    yhat=lr(X)
```

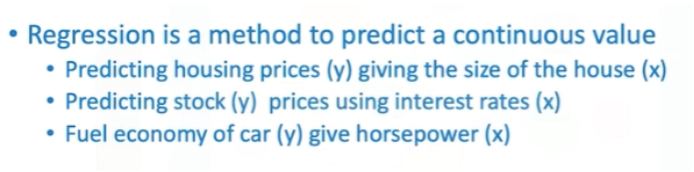
Training Linear Regression

1. What “training” means

- Training = **learning the best values of weight** `w` **and bias** `b`
- Done using a **dataset of input–output pairs**:

{(x1,y1),(x2,y2),...,(xn,yn)}

This is called **simple linear regression** because `x` is 1D.



---

2. Dataset structure

- Data is usually stored as **tensors**
  - `x` : shape `(n, 1)`
  - `y` : shape `(n, 1)`
- Each row = one sample
- Examples:

- House size → price
- Interest rate → stock price
- Horsepower → fuel efficiency

---

3. Noise assumption

Even if the true relationship is linear:

$y=wx+b$

real data is:

$y=wx+b+\epsilon$

- ε = **noise**
- Assumed to be **Gaussian (normal)**
- Mean ≈ 0
- Standard deviation controls how scattered points are

More noise → points farther from the line.

---

4. Goal of training

- Find a line that **best fits the scattered data**
- "Best" means **minimizing prediction error**, not passing through all points

---

5. Loss (Cost) function

To measure error, we use **Mean Squared Error (MSE)**:

$$\text{MSE}(w,b) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

where:

$$\hat{y}_i = wx_i + b$$

- Squaring penalizes large errors

- Averaging makes it independent of dataset size

---

6. Why minimizing MSE works

- Different values of `w` and `b` → different lines
- Each line gives a different MSE value
- **Best-fit line = line with minimum MSE**

Visually:

- Bad line → high loss
- Better line → lower loss
- Best line → **minimum loss**

---

7. Big picture

1. Assume a linear model
2. Add noise to reflect real-world data
3. Define a loss function (MSE)
4. Find `w` and `b` that minimize the loss

Loss

1. Why loss exists

We want to **learn model parameters** (here: slope `w`, bias `b`).

To learn them, we need a **numerical measure of "how bad" a prediction is**.

That measure is called **loss**.

---

2. Single-sample setup

Given **one data point**:

- Input: $x = -2$

- True output: y=4y = 4y=4

Model (no bias for simplicity):

y^=wx

Here:

- `x` and `y` are fixed
- `w` is unknown → **parameter to learn**

---

3. Loss definition

Loss measures the **distance between prediction and truth**.

For one sample:

$$\text{Loss}(w) = (\hat{y} - y)^2 = (wx - y)^2$$

Why square?

- Makes loss **always ≥ 0**
- Penalizes large errors more
- Smooth and differentiable (important for optimization)

---

4. Loss as a function of the parameter

Key idea:

> Loss is not a function of data — it is a function of the parameter

- Different `w` → different line
- Different line → different prediction
- Different prediction → different loss

So:

Loss=f(w)

---

5. Parameter space vs data space

- **Data space**: where points and lines live (x–y plane)
- **Parameter space**: where loss values live (w–loss plot)

In parameter space:

- Loss curve is a **convex (bowl-shaped) parabola**
- Lowest point = **best parameter**

---

6. Interpreting slopes visually

Slope `w`	Effect
Large positive	Line far from point → high loss
Near optimal	Line close to point → low loss
Large negative	Line far again → high loss

---

7. Role of derivatives

The derivative of loss w.r.t. parameter:

dwdLoss(w)

Properties:

- **Negative** → move right
- **Positive** → move left
- **Zero** → minimum loss

At the minimum:

dwdLoss=0

This gives the **optimal slope**.

---

8. Why calculus matters (deep learning link)

- For simple models → solve analytically
- For neural networks → impossible analytically

- But:
  - Derivatives still tell **direction to move**
  - Used by **gradient descent**

---

9. Final mental model

- **Loss** → error for ONE sample
- **Cost** → average loss over ALL samples
- **Training** → minimize cost using derivatives

Gradient Descent

1. What gradient descent does

Gradient descent is an **iterative optimization algorithm** used to find the **minimum of a loss function** by updating model parameters step by step.

Here, the parameter is **one-dimensional** (e.g., slope `w`).

---

2. Core idea (direction of movement)

- The **derivative of the loss** tells us the **direction of steepest increase**
- To **minimize** the loss, we move in the **opposite direction**

So we update the parameter using:

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

Where:

- $w^{(k)}$ = current parameter estimate
- η (eta) = learning rate
- $\frac{dL}{dw}$ = gradient (derivative of loss)

---

3. Why subtract the derivative

- If gradient is **positive** → loss increases to the right → move left
- If gradient is **negative** → loss increases to the left → move right
- Subtracting the gradient **always moves toward the minimum**

---

4. Iterative nature

Gradient descent works by:

1. Start with an **initial guess**
2. Compute gradient at that point
3. Update parameter
4. Repeat until convergence

Each iteration should **reduce the loss** (if learning rate is reasonable).

---

5. Learning rate (η) problems

Learning rate too large

- Steps are too big
- Can **overshoot the minimum**
- Loss may **increase or diverge**

Learning rate too small

- Steps are tiny
- Converges **very slowly**
- Computationally inefficient

Choosing η is critical.

---

6. Stopping criteria (when to stop)

Common stopping rules:

1. **Fixed number of iterations**
2. **Loss stops decreasing**

3. **Change in loss < threshold**
4. **Gradient ≈ 0** (near minimum)

Practical training often combines multiple criteria.

---

7. Key intuition

- Loss curve is **bowl-shaped (convex)** in simple linear regression
- Gradient descent follows the slope **downhill**
- Each step improves the parameter estimate

Cost Function

1. Why we need a cost function

- **Loss** measures error for **one sample**
- **Cost** measures error over **all training samples**
- We choose parameters (slope `w`, bias `b`) that **minimize total error**, not just one point

---

2. From loss to cost

For one sample:

For one sample:

$$\text{Loss}_i = (y_i - \hat{y}_i)^2$$

For multiple samples:

- **Total cost:**

$$L(w, b) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- **Average cost (MSE):**

$$L(w, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Both are used; average just scales the value.

---

3. Geometric intuition

- Each sample produces a **square error**
- Cost = **sum (or average) of areas of these squares**
- Smaller squares → better fit
- Best line → **minimum total area**

---

4. Cost as a function of parameters

- Cost depends on **slope (`w`) and bias (`b`)**
- Changing parameters changes predictions → changes cost
- Cost surface is **convex (bowl-shaped)** for linear regression

---

5. Gradient descent on cost

We minimize cost using:

w←w−ηdwdL

- Derivative is computed using **all samples**
- Gradients from samples are **summed or averaged**

---

6. Effect of multiple samples on gradient

Case 1: All samples on same side of line

- Gradients have same sign
- Large magnitude gradient
- **Big update step**

Case 2: Samples on opposite sides

- Gradients cancel
- Small derivative
- Parameter is near optimal

---

7. Batch Gradient Descent

- Uses **all training samples** to compute cost and gradient
- One update per full dataset
- Stable but slower for large datasets

Batch=entire training set

Linear Regression Pytorch

1. What we are doing

- We want to **learn the slope** `w` of a line using gradient descent
- We **manually update** `w`, without using optimizers (`torch.optim`)
- This helps understand **how PyTorch autograd works**

---

2. Preparing the parameter

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

- `w` is initialized randomly (here `10`)

- `requires_grad=True` tells PyTorch to **track gradients**
- PyTorch treats `w` symbolically during computation

---

3. Dataset creation

- Create input values `x`
- Generate target `y = -3x + noise`
- `view()` is used to convert 1D tensors into **2D shape (N,1)**, required by models

Noise simulates **real-world data imperfections**.

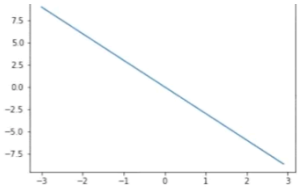
```
import torch
w=torch.tensor(-10.0, requires_grad=True)

X=torch.arange(-3,3,0.1).view(-1, 1)
f=-3*X

import matplotlib.pyplot as plt

plt.plot(X.numpy(),f.numpy())
plt.show()

Y=f+0.1*torch.randn(X.size())
```



---

4. Forward function (model)

```
defforward(x):
return w * x
```

- This is the prediction equation
- No bias here, only slope

---

5. Cost (loss) function

```
loss = torch.mean((y_hat - y)**2)
```

- Uses **Mean Squared Error**

- PyTorch calls this “loss” even though it’s technically cost

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

$$\hat{y} = w\,x$$

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

$$\frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n)^2$$

6. Backpropagation

```
loss.backward()
```

- Computes derivative of loss **w.r.t.** `w`
- Gradient is stored in:

```
w.grad
```

7. Manual parameter update

```
w.data = w.data - lr * w.grad
```

- `.data` accesses the raw value of `w`
- We apply **gradient descent update rule**
- Learning rate (`lr`) controls step size

```
lr=0.1
for epoch in range(4):
    Yhat=forward(X)
    loss=criterion(Yhat,Y)
    loss.backward()

    w.grad
```

$$\hat{y} = -w \begin{bmatrix} -x_1 \\ \vdots \\ -x_N \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

$$l(w) = \frac{1}{N} \sum_{n=1}^{N} (y_n - w\, x_n)^2$$

$$\frac{dl(w)}{dw} \Rightarrow \frac{dl(w=-10)}{dw}$$

8. Resetting gradients

```
w.grad.zero_()
```

- Gradients **accumulate by default**
- Must reset before next iteration

9. Epochs

- **1 epoch = using all data once**
- Repeating epochs:
  - Loss decreases
  - Line moves closer to data
  - Gradient magnitude reduces (slower movement)

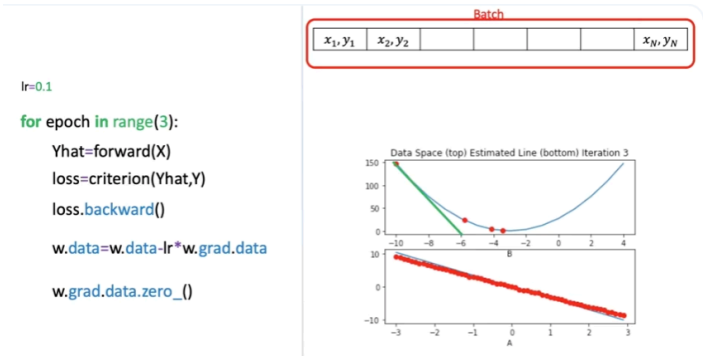
10. Why loss decreases slowly later

- Gradient = slope of tangent
- Near minimum → slope is small
- Smaller slope → smaller updates

This is **expected behavior**.

11. Tracking convergence

- Store `loss.item()` each epoch
- Plot:
  - Loss vs iterations
  - Shows smooth downward curve
- Used when parameter-space plots are not possible



PyTorch Linear Regression Training Slope and Bias

1. Cost as a function of parameters

For linear regression:

$\hat{y} = wx + b$

The **cost function** (average loss) depends on **two parameters**:

- **w (slope / weight)** → controls steepness
- **b (bias)** → controls vertical shift

So mathematically:

$J(w,b) = \frac{1}{N}\sum(wx_i + b - y_i)^2$

Because there are **two parameters**, the cost is a **surface**, not a curve.

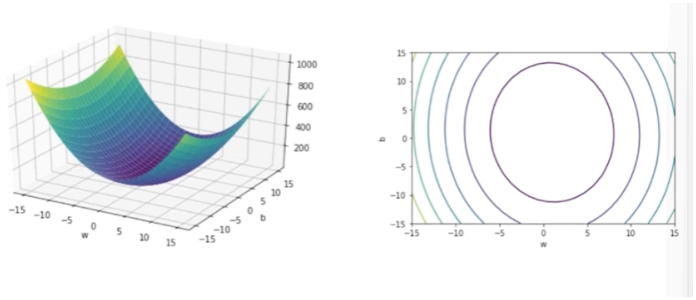
---

2. Cost surface visualization

- **x-axis** → slope (w)
- **y-axis** → bias (b)
- **z-axis (height)** → cost value

Each point on the surface corresponds to **one (w, b) pair** and its cost.

The **minimum point** on this surface = best parameters.



---

3. Contour plots (important intuition)

A **contour plot** is a **top-down view** of the cost surface.

- Each contour line = same cost value
- Close contours → cost changes fast (steep region)
- Far contours → cost changes slowly (flat region)

Cutting the surface at fixed cost values (200, 400, 600, ...) produces these contour lines.

---

4. Slices of the surface

- **Fix b = constant** → cost becomes a function of w (parabola)
- **Fix w = constant** → cost becomes a function of b (parabola)

Different curvature explains why:

- Some regions require **small parameter changes**
- Others require **large parameter changes** to change cost

---

5. Gradient descent on the surface

Since cost depends on **multiple variables**, we use **partial derivatives**:

$$\frac{\partial J}{\partial w}, \quad \frac{\partial J}{\partial b}$$

Collecting them gives the **gradient vector**:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial w} \\ \frac{\partial J}{\partial b} \end{bmatrix}$$

---

6. Key geometric fact

- The **gradient is perpendicular to contour lines**
- It points in the direction of **steepest increase**
- Gradient descent moves in the **negative gradient direction**

That's why parameter updates move **across contours toward the minimum**.

---

7. PyTorch implementation (first principles)

Steps:

1. Define forward function:

y^=wx+b

```
def forward(x):
    y=w*x+b
    return y
```

$$\hat{y} = w\,x+b$$

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

$$\frac{1}{N}\sum_{n=1}^{N}(y_n - w\,x_n - b)^2$$

1. Define cost (MSE)
2. Initialize `w` and `b` with `requires_grad=True`
3. Compute loss
4. Call `loss.backward()` → computes gradients

5. Update **both** `w` and `b`
6. Zero gradients
7. Repeat for epochs

---

8. What happens over epochs

- Early iterations → large jumps (steep surface)
- Later iterations → smaller jumps (flat near minimum)
- Line gradually fits data better
- `(w, b)` points move inward on contour plot

BEST PRACTISES

Set an Appropriate Learning Rate

The learning rate is critical in determining how quickly or slowly a model learns.

- Moderate Learning Rate: Start with a moderate learning rate (for example, 0.01) to balance convergence speed with stability. A learning rate that's too high may lead to overshooting the optimal solution, whereas a very low rate can result in slow convergence.
- Use Learning Rate Schedulers: Implement learning rate schedulers to adjust the rate dynamically during training, such as decreasing the rate over time for fine-tuning or using cyclic learning rates to overcome local minima.

Data Standardization

Standardizing input features to have zero mean and unit variance speeds up training and improves accuracy by preventing issues due to varying feature scales.

- Scale Features: Use PyTorch's `StandardScaler` or manually standardize features to ensure each feature contributes equally to the model, helping the optimizer converge more effectively.
- Normalize Output (if necessary): If the output is also on a large scale, normalizing it may further improve model training and stability.

Implement Validation Sets

Validation sets are essential for monitoring the model's performance and detecting overfitting.

- Train-Validation Split: Use a portion of the dataset as validation data, evaluating model performance on this set periodically during training.
- Early Stopping: Implement early stopping based on validation loss to halt training when the model stops improving, which helps prevent overfitting.

Gradient Clipping for Stability

In datasets with high variance, gradients can sometimes grow too large, leading to instability.

- Clip Gradients: Apply gradient clipping to limit the magnitude of gradients. PyTorch's `torch.nn.utils.clip_grad_norm_` helps ensure gradients do not exceed a set threshold.
- Avoiding Exploding Gradients: Gradient clipping prevents exploding gradients, which is particularly useful in models with larger datasets or complex feature interactions.

Monitor Loss Function

Monitoring the loss function during training helps diagnose issues and make necessary adjustments.

- Loss Reduction Monitoring: Observe whether the loss steadily decreases during training. If the loss plateaus or increases, consider adjusting the learning rate or re-evaluating data preparation.

- Track with Visualization Tools: Use visualization tools such as TensorBoard to track training and validation loss over epochs for a clear view of model performance.

Conclusion

Applying these best practices can improve the training process and performance of linear regression models in PyTorch. By carefully managing learning rates, standardizing data, using validation, and monitoring the training process, you'll set a strong foundation for building more complex machine-learning models in PyTorch.