Introduction to Neural Networks and Pytorch (M4)

Created	@December 17, 2025 6:48 PM
Module Code	IBMM04 : Introduction to Neural Networks and Pytorch

Module 3: Linear Regression Pytorch

Stochastic Gradient Descent

1. Why Stochastic Gradient Descent (SGD)?

In **batch gradient descent**:

- Gradient is computed using **all samples**
- Update is **smooth but expensive**

In **Stochastic Gradient Descent**:

- Gradient is computed using **one sample at a time**
- Update is **fast but noisy**

Mathematically, instead of minimizing:

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

SGD minimizes:

$$\ell_i(w,b) = (wx_i + b - y_i)^2$$

one sample at a time

one sample at a time

---

2. What actually happens in SGD

For each sample (xi,yi)(x_i, y_i)(xi,yi):

1. Compute prediction y^i\hat{y}_iy^i
2. Compute **loss for that sample**
3. Compute gradient from **that single loss**
4. Update parameters immediately

This means:

- Some updates **reduce total error**
- Some updates **increase total error** (outliers)

➡️ Loss **fluctuates**, instead of decreasing smoothly.

---

3. Epoch and iterations (important distinction)

- **Iteration**: one parameter update (using one sample)
- **Epoch**: one full pass over the dataset

If dataset has 3 samples:

- 1 epoch = 3 iterations

---

4. Why SGD is noisy

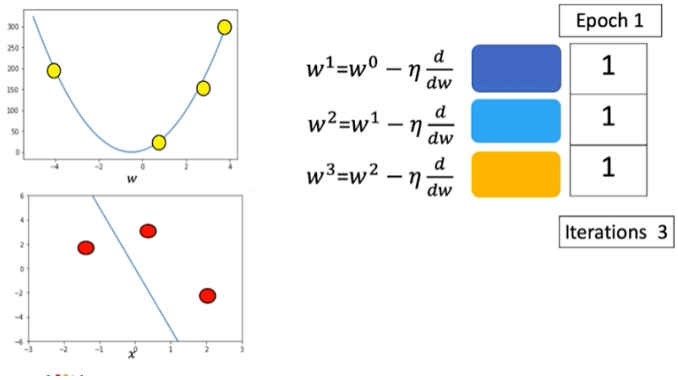
Because:

- Each sample gives a **different gradient direction**
- Outliers can temporarily increase loss

But over many updates:

- Parameters move toward a region of **low average cost**

This makes SGD an **approximation** of true cost minimization.



5. Gradient expression in SGD

For one sample:

For one sample:

$$\frac{\partial \ell_i}{\partial w} \propto (wx_i + b - y_i)\, x_i$$

Update rule:

$$w \leftarrow w - \eta \frac{\partial \ell_i}{\partial w}$$

$$b \leftarrow b - \eta \frac{\partial \ell_i}{\partial b}$$

Same rule as batch GD — **different data size**.

6. SGD in PyTorch (core idea)

Key points:

- `requires_grad=True` → track gradients
- Loss computed per **single sample**
- `loss.backward()` computes gradients
- Parameters updated **inside inner loop**

Structure:

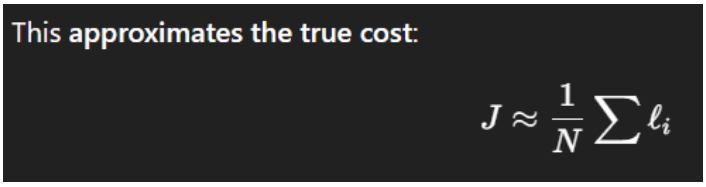
```
for epochin epochs:
for x_i, y_iin data:
        loss = criterion(model(x_i), y_i)
        loss.backward()
        update parameters
        zero gradients
```

7. Tracking loss in SGD

Since per-iteration loss is noisy:

- Accumulate loss across samples
- Average it per epoch

This **approximates the true cost**:



Used for:

- Monitoring training
- Comparing epochs

8. Dataset object (PyTorch)

A custom Dataset must define:

- `__len__()` → number of samples
- `__getitem__(index)` → returns `(x_i, y_i)`

This allows:

```
dataset[i]
dataset[0:3]
```

```
len(dataset)
```

9. DataLoader (why we use it)

`DataLoader`:

- Wraps the Dataset
- Handles batching, shuffling, iteration

With:

```
DataLoader(dataset, batch_size=1)
```

You get:

- **Pure SGD** (one sample at a time)

Iteration becomes:

```
for x, yin trainloader:
    ...
```

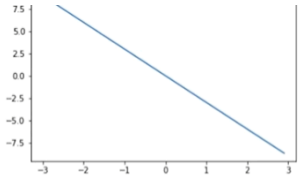
Cleaner, scalable, and standard.

Method	Samples per update	Behavior
Batch GD	All samples	Stable, slow
SGD	1 sample	Fast, noisy
Mini-batch GD	Small batch	Best trade-off

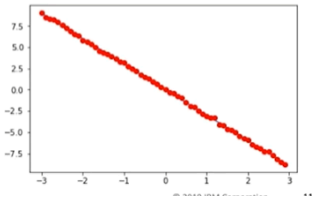
```
w=torch.tensor(-15.0, requires_grad=True)
b=torch.tensor(-10.0, requires_grad=True)

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())
plt.plot(X.numpy(),Y.numpy(),'ro')
plt.show()
```



IBM Developer

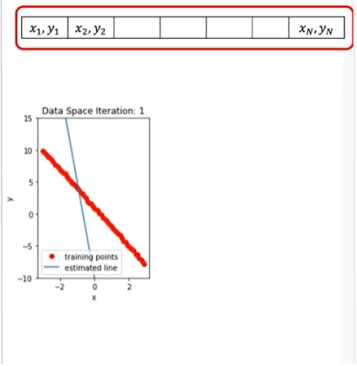
```
def forward(x):
    y=w*x+b
    return y
```

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

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

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

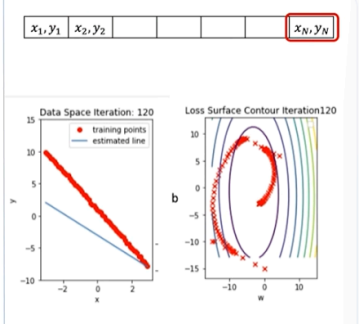
```
lr=0.1
for epoch in range(4):
    for x,y in zip(X,Y):
        yhat=forward(x)
        loss=criterion(yhat,y)
        loss.backward()
        w.data=w.data-lr*w.grad.data
        b.data=b.data-lr*b.grad.data
        w.grad.data.zero_()
        b.grad.data.zero_()
```

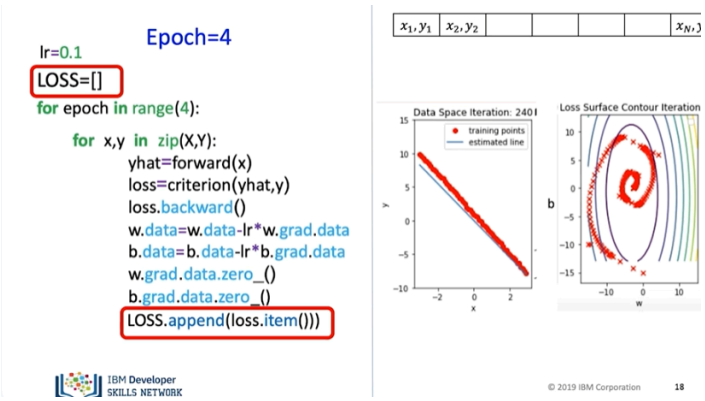


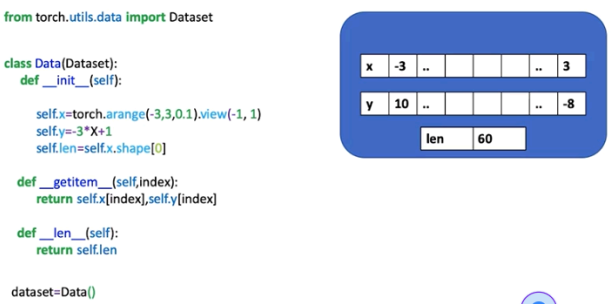
Epoch=2

```
lr=0.1
for epoch in range(4):
    for x,y in zip(X,Y):
        yhat=forward(x)
        loss=criterion(yhat,y)
        loss.backward()
        w.data=w.data-lr*w.grad.data
        b.data=b.data-lr*b.grad.data
        w.grad.data.zero_()
        b.grad.data.zero_()
```

N







Mini-Batch Gradient Descent

1. What is Mini-Batch Gradient Descent?

Mini-batch gradient descent is a **compromise** between:

- **Batch Gradient Descent** (all samples)
- **Stochastic Gradient Descent** (one sample)

👉 Instead of using **1 sample** or **all samples**, we use a **small subset (batch)** of the dataset per update.

---

2. Why Mini-Batch GD is important

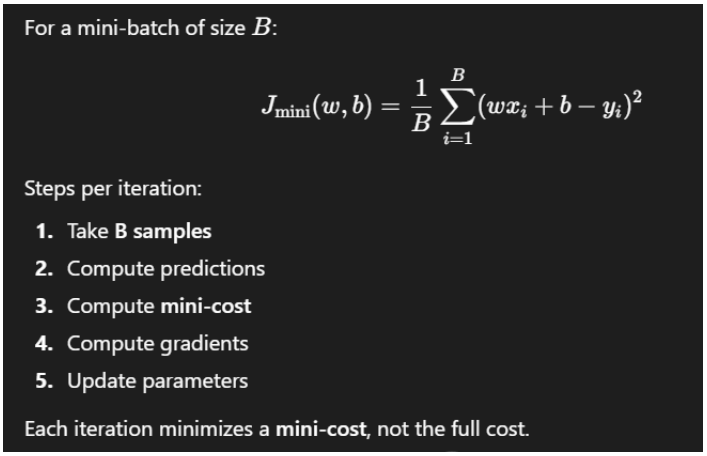
Key advantages:

- Can train on **large datasets** that do not fit in memory

- More **stable** than SGD (less noise)
- Faster than batch GD
- Efficient on **GPUs** due to vectorization

---

3. What happens in one iteration?



Each iteration minimizes a **mini-cost**, not the full cost.

---

4. Epochs, batch size, and iterations

Let:

- N= total samples
- B= batch size

$$\text{Iterations per epoch} = \left\lceil \frac{N}{B} \right\rceil$$

Example (N = 6):

Batch Size	Iterations per Epoch
1 (SGD)	6
2	3
3	2
6 (Batch GD)	1

👉 **Epoch = full pass through dataset**

---

5. Visual intuition

- Batch size = 1 → very noisy updates
- Batch size = large → smooth but slow
- Mini-batch → controlled noise + faster convergence

Loss decreases **more smoothly than SGD**, but **faster than batch GD**.

---

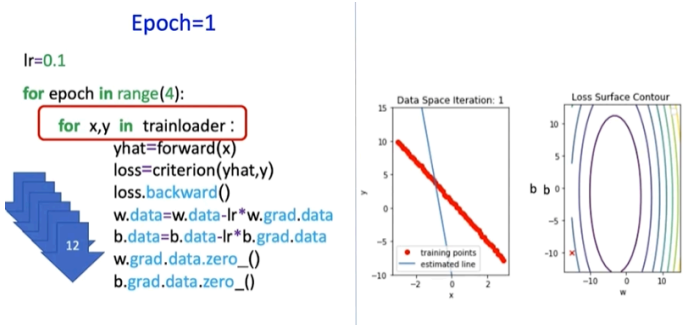
6. Mini-Batch GD in PyTorch (conceptually)

Same as SGD, **only difference is batch size**.

```
trainloader = DataLoader(dataset, batch_size=5, shuffle=True)

for epochin epochs:
for x_batch, y_batchin trainloader:
        loss = criterion(model(x_batch), y_batch)
        loss.backward()
        update parameters
        zero gradients
```

✔ Each update uses **5 samples**



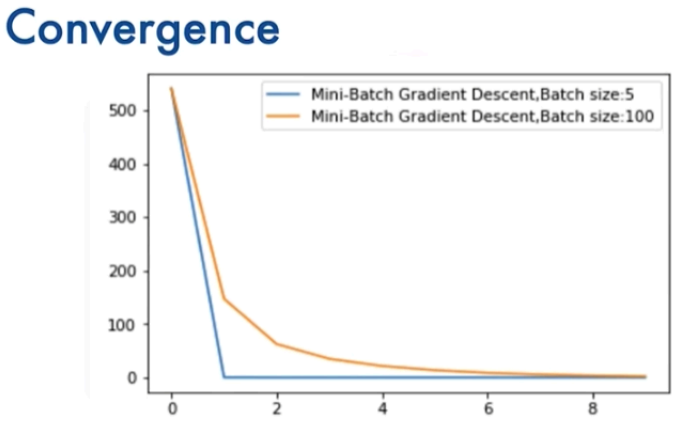
7. Tracking loss

- Loss per batch is **approximate**
- Average batch losses per epoch ≈ true cost
- Used to monitor **convergence**

8. Effect of batch size on convergence

- Small batch → noisy but fast progress
- Large batch → stable but slower
- Medium batch → **best convergence rate**

This is why mini-batch GD is the **default choice in deep learning**.



Optimization in PyTorch

1. Why PyTorch Optimizers exist

In earlier examples, we **manually updated parameters**:

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

This becomes:

- Error-prone
- Hard to extend to many parameters
- Impossible to manage advanced optimizers

👉 **PyTorch optimizers automate parameter updates**.

---

2. What an Optimizer does (precisely)

An optimizer:

1. **Stores references** to model parameters
2. **Reads their gradients** after `backward()`
3. **Updates parameters** according to a rule (SGD, Adam, etc.)
4. Maintains an internal **state** (e.g., momentum)

---

3. Core training components (roles)

Component	Purpose
Dataset	Stores features & targets
DataLoader	Provides batches
Model (`nn.Module`)	Defines forward computation
Criterion (Loss)	Measures prediction error
Optimizer	Updates model parameters

4. Creating an optimizer (SGD example)

```
from torch import nn, optium
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=0.01
)
```

Important details:

- `model.parameters()` → iterator over **all learnable tensors**
- Optimizer **does not own** parameters, it **references** them
- Learning rate controls update magnitude

---

5. Optimizer state

```
optimizer.state_dict()
```

Contains:

- Parameter IDs
- Optimizer-specific buffers (e.g., momentum)
- Hyperparameters

👉 Needed for **checkpointing** and **resuming training**

---

6. Training loop with optimizer (canonical form)

```
for epochinrange(num_epochs):
for x_batch, y_batchin trainloader:
```

```
y_hat = model(x_batch)
loss = criterion(y_hat, y_batch)

optimizer.zero_grad()
loss.backward()
optimizer.step()
```

---

7. Why `optimizer.zero_grad()` is required

PyTorch **accumulates gradients** by default:

$$\text{grad}_{\text{new}} = \text{grad}_{\text{old}} + \text{grad}_{\text{current}}$$

If not cleared:

- Gradients become incorrect
- Updates explode

---

8. What `optimizer.step()` actually does

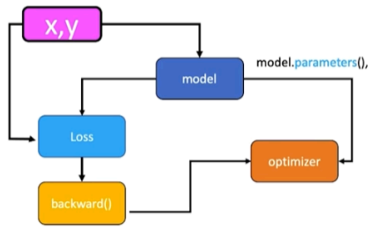
Conceptually equivalent to:

```
for paramin model.parameters():
    param.data -= lr * param.grad
```

But:

- Handles **multiple parameters**
- Supports **momentum, weight decay, adaptive LR**
- Works across **devices (CPU/GPU)**

```
for epoch in range(100):
    for x,y in trainloader :
        yhat=model(x)
        loss=criterion (yhat,y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```



---

9. How optimizer connects to loss (important concept)

Even though:

- Optimizer never sees the loss explicitly

The chain is:

> Loss →backward() → gradients → optimizer.step() → parameters

The connection exists **through the computation graph**.

---

10. Why this matters for deep models

Optimizers:

- Scale to **millions of parameters**
- Enable **advanced methods** (Adam, RMSProp)
- Separate **optimization logic** from model definition

This is why **almost all PyTorch training follows this pattern**.

Training, Validation, and Test Split

Overfitting:

Overfitting happens when:

- A model performs **very well on training data**
- But performs **poorly on unseen (new) data**

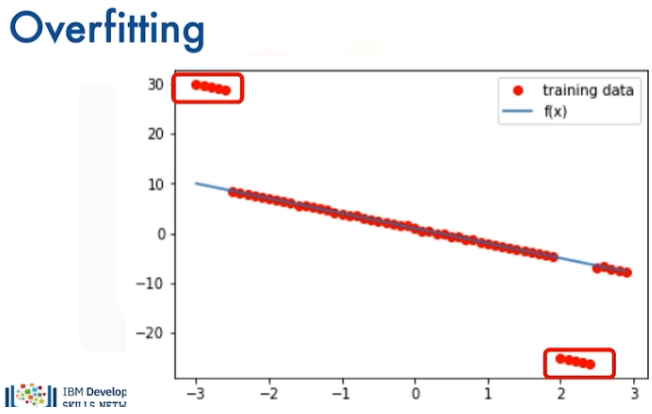
Why does it happen?

- The model is **too complex**
- It learns **noise, outliers, and random fluctuations**
- Instead of learning the true underlying pattern

Example from the video

- Blue line = **true relationship**
- Some points are **far away** from the line
  - These are **noise or errors**
- A complex model may try to pass through *every point*
- That looks good on training data but fails elsewhere

👉 Overfitting = **memorization**, not learning.



Dataset Split (Why we split data)

We split the dataset into **three parts**:

(a) Training Data

- Used to **train the model**

- Model learns parameters like:
  - Weight
  - Bias
  - Slope
- Training uses algorithms like **Gradient Descent**

📌 Example:

- Linear regression learns slope & intercept from training data

---

(b) Validation Data

- Used to **tune hyperparameters**
- NOT used to train the model
- Helps decide **which model is better**

📌 Used during:

- Model selection
- Hyperparameter tuning

(c) Test Data

- Used **only at the end**
- Gives an estimate of **real-world performance**
- Must be **completely unseen**

📌 Rule:

> Never tune your model using test data

Parameters vs Hyperparameters

Parameters

- Learned from **training data**

- Examples:
  - Weight (w)
  - Bias (b)
  - Slope

Learned via **Gradient Descent**

Hyperparameters

- Chosen **before training**
- Control *how* learning happens
- Examples:
  - Learning rate (α)
  - Batch size
  - Number of epochs

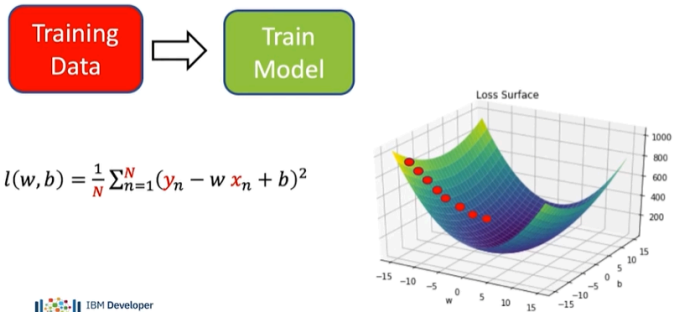
📌 Hyperparameters are **NOT learned automatically**

Gradient Descent

- Loss surface = hills & valleys
- Goal: reach **lowest valley**
- Gradient descent:
  - Takes steps downhill
  - Step size = **learning rate**

📌 Learning rate too small → slow training

📌 Learning rate too large → overshoot minimum



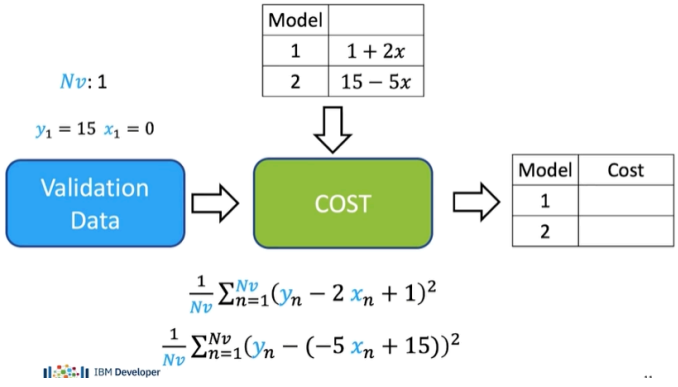
7. Using Validation Data to Select Hyperparameters

What the video does:

- Tries **two different learning rates**
- Trains **two different models**

Steps:

1. Choose learning rate 1
2. Train model → Model 1
3. Choose learning rate 2
4. Train model → Model 2
5. Evaluate both models on **validation data**



Why NOT use training data?

Because training loss is **biased**

- The model has already seen this data

Validation data gives a **fair comparison**

Validation Cost Formula

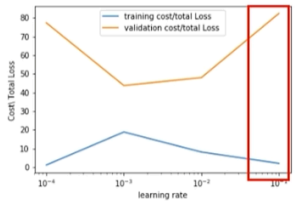
If validation set size = NVN_VNV:

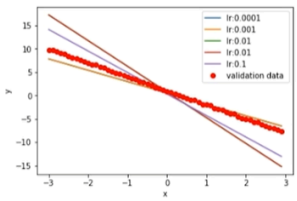
$$J_{val} = \frac{1}{N_V} \sum (y - \hat{y})^2$$

In the video:

- Only **one validation sample**
- So cost simplifies to:

(y−y^)2





Validation Example

Given:

- y=15y = 15y=15
- x=0x = 0x=0

Model 1:

- Prediction ≠ 15
- Loss = **256**

Model 2:

- Prediction = 15
- Loss = **0**

✅ Model 2 is selected because it has **lower validation loss**

Training Loss vs Validation Loss Graph

X-axis:

- Different **learning rates**

Y-axis:

- Cost (loss)

Curves:

- Blue → Training loss
- Orange → Validation loss

📌 Key idea:

> Lowest training loss ≠ best model

We choose the model with **minimum validation loss**

Why NOT use Test Data for Model Selection?

The video shows a critical mistake:

- Learning rate that minimizes **test loss**
- Gives **bad fit** to actual data

Why?

- Test data was indirectly “used”

- Model selection leaked information

📌 This leads to **overfitting to test data**

Correct ML Workflow

1. Split data randomly:
    - Training
    - Validation
    - Test
2. Train models using **training data**
3. Tune hyperparameters using **validation data**
4. Final evaluation using **test data**

Training, Validation, and Test Split PyTorch

---

1. Data Split

- Data is split into **training** and **validation** sets
- Split is **deterministic** (only for demonstration)
- **Training data contains outliers**, validation data does not

  → helps clearly show **overfitting**

```
from torch.utils.data import Dataset, DataLoader

class Data(Dataset):
    def __init__(self,train =True):
        self.x=torch.arange(-3,3,0.1).view(-1, 1)
        self.f=-3*self.x+1
        self.y=self.f+0.1*torch.randn(self.x.size())
        self.len=self.x.shape[0]
        if train==True:
            self.y[0] = 0
            self.y[50:55] = 20
        else:
            pass

        def __getitem__(self,index):
            return self.x[index],self.y[index]

        def __len__(self):
            return self.len
```

2. Synthetic Data

- Artificial linear data is generated
- True relationship: a straight line
- Outliers are added to training data to distort learning

3. Dataset & Model

- Custom **PyTorch Dataset** class generates data
- Two dataset objects:
    - Training dataset (with outliers)
    - Validation dataset (clean)
- A custom **linear regression model** is defined

```
import torch.nn as nn

class LR(nn.Module):
        def __init__(self,input_size,output_size):
                super(LR,self).__init__()
                self.linear=nn.Linear(input_size,output_size)

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

4. Loss & Hyperparameter

- Loss function: **Mean Squared Error (MSE)**
- Hyperparameter tuned: **learning rate**
- Number of epochs fixed (10)

```
epochs=10

learning_rates=[0.0001,0.001,0.01,0.1,1]

validation_error=torch.zeros(len(learning_rates))

train_error=torch.zeros(len(learning_rates))


MODELS=[]
```

5. Training Process

For each learning rate:

1. Create a **new model**
2. Create an optimizer with that learning rate
3. Train on training data
4. Compute **training loss**
5. Compute **validation loss**

6. Store the model
7.

```
for i,learning_rate in enumerate(learning_rates):

    model=LR(1,1)
    optimizer = optim.SGD(model.parameters(), lr = learning_rate)

    TRAINING

    yhat=model(train_data.x)
    loss=criterion(yhat,train_data.y)
    train_error[i]=loss.item()

    yhat=model(val_data.x)
    loss=criterion(yhat,val_data.y)
    validation_error[i]=loss.item()
```

Learning rates	0.0001	0.001	0.01	0.1	1
train					
Validation					
MODELS					

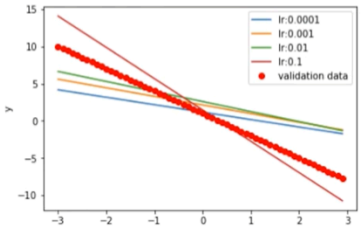
6. Model Selection

- Training loss alone is misleading
- Model with **minimum validation loss** is selected
- Best model fits validation data most closely
-

```
for model,learning_rate in zip(MODELS, learning_rates):

    yhat=model(val_data.x)
    plt.plot(val_data.x.numpy(),yhat.detach().numpy(),label='lr:'+str(learning_rate))

plt.plot(val_data.x.numpy(),val_data.y.numpy(),'or',label='validation data ')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```



7. Visualization & Saving

- Training vs validation loss plotted against learning rate
- Predicted lines plotted to visually verify best fit
- Best model can be saved using:

```
torch.save(model.state_dict(),"model.pth")
```