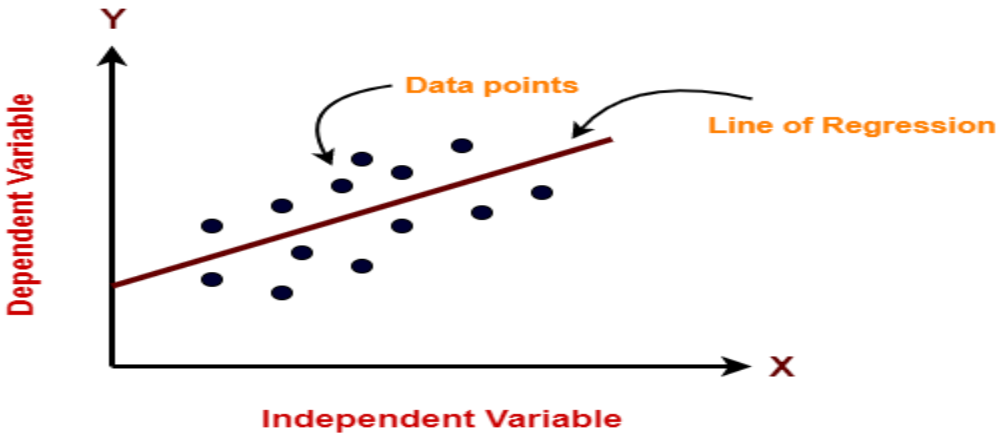
DEEP LEARNING

HANDOUT: REGRESSION

NEURON MODEL, GRADIENTS, OPTIMIZATION, LINEAR MODEL IN ACTION

What is Regression in Machine Learning?

Regression is a type of supervised learning where we try to predict a continuous value based on input features.



Examples:

- Predicting house price based on its size and location
- Predicting medical insurance cost based on age, weight, and smoking status
- Predicting student marks based on study hours

Here, the target or output is a number, not a category.

Task: You work for an e-commerce company that wants to predict the delivery time of a package based on distance, weather, and traffic conditions.

Question:

Why would you choose a regression model over classification for this task?

What kind of prediction output are you expecting?

Answer:

A regression model is used because the output (delivery time) is a **continuous number**. Classification wouldn't work here, as we're not predicting a category like "fast", "medium", or "slow", but the actual time in hours or minutes.

Neuron Model in Regression

The Neuron Model (used in Regression)

In machine learning, a neuron is the smallest unit of a model.

A neuron in regression:

- Takes inputs like age, weight, etc.
- Multiplies each input by a number called a weight
- Adds another number called bias
- Returns a result, which is the prediction

There is no activation function in regression tasks because we want a wide range of outputs (not restricted between 0 and 1 or only positive).

Task: Imagine you're designing a model to predict a student's score based on study hours and number of mock tests taken. You decide to manually build a simple regression neuron.

Question:

If you increase the input value (e.g., more study hours), how should the neuron ideally adjust its weights to increase the predicted score?

Expected Answer:

If more study hours generally lead to higher scores, the neuron should **increase the weight** associated with the study hours so that the prediction increases accordingly.

Loss Function

What is Loss?

Loss means how wrong the prediction is.

The most common loss function used in regression is called Mean Squared Error.

This tells us how far the predicted values are from the actual values. The higher the loss, the worse the model is performing.

Types of Loss Functions in Regression

1) **Mean Squared Error (MSE)**

Average of the squares of the differences between actual and predicted values.

Mean Squared Error (MSE) – "Punish Big Mistakes More"

- **Meaning:** Square the errors before adding. Bigger mistakes feel *much* worse.
- **Use when:**
    - You want to really **minimize large prediction errors**.
    - Your data is **clean**, no extreme outliers.

2) **Mean Absolute Error (MAE)**

Average of the absolute differences between actual and predicted values.

Mean Absolute Error (MAE) – "Treat All Mistakes Fairly"

- **Meaning:** Just measure the size of the error, ignore the direction.
- **What it does:** Adds up how far off each prediction is, then averages it.
- **Use when:**
    - You want your model to be **fair to all errors**
    - **Outliers exist**, but you don't want them to control the model.

3) **RMSE (Root Mean Squared Error)**

"How wrong is your model on average?"

What is RMSE?

- RMSE is the **square root** of the **Mean Squared Error (MSE)**.
- It gives the **average error** in the **same unit** as the output (like rupees, marks, temperature, etc.)

Interpretation:
On average, how much are your predictions **off** from the actual values?

When to Use RMSE:

- When you want to **penalize large errors** more than small ones
- When **scale matters** (e.g., predicting salary, price, etc.)

4) R^2 (R-squared or Coefficient of Determination)

"How well does your model explain the data?"

What is R^2?

- R^2 measures how much **variation in the output** is explained by the model.
- Value is between **0 and 1** (sometimes even negative).

Interpretation:

- **1.0** means perfect predictions (100% variance explained)
- **0.0** means the model does no better than just predicting the average
- **Negative** means the model is worse than the average!

When to Use R^2:

- When you want to evaluate the **overall fit** of the model
- To compare how much better one model performs over another

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}|$$

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}$$

$$R^2 = 1 - \frac{\sum(y_i - \hat{y})^2}{\sum(y_i - \bar{y})^2}$$

Where,

$\hat{y}$ – *predicted value of y*

$\bar{y}$ – *mean value of y*

Task: You are building a model to **predict house prices (in lakhs)** based on size, location, and number of rooms.

You test your model on a sample and get these metrics:

- MAE = 5
- MSE = 42
- RMSE = 6.48
- $R^2 = 0.87$

How to Interpret These:

Metric	**Meaning in This Scenario**
MAE = 5	On average, your predictions are off by **5 lakhs** (up or down)
MSE = 42	Your errors squared average to 42. Big errors hurt more here
RMSE = 6.48	On average, your model is **off by 6.48 lakhs** — similar to MAE but big errors matter more
$R^2 = 0.87$	Your model explains **87% of the variation** in house prices

When to Use Each Metric

Metric	**Best When...**
MAE	You want **simple average error**, less sensitive to outliers
MSE	You want to **penalize big mistakes more** (e.g., for safety or finance)
RMSE	You want an **interpretable error in the same units**, but still care about big errors
R^2	You want to know **how well the model fits the data overall** (not actual error size)

Task: You created a model to predict monthly electricity bills, but you notice that one prediction is way off by 1000 rupees, while others are off by only 50 rupees.

Question:

Why might MSE be more appropriate than just averaging all errors (like in MAE)?

Answer:

MSE penalizes larger errors more, so the big mistake (off by 1000) has a much stronger influence on the loss. This forces the model to correct large mistakes more seriously than small ones.

What Are Gradients?

A gradient tells us how much and in what direction to change each weight and bias to make the model better.

Imagine you're climbing down a mountain to reach the lowest point (minimum loss). A gradient tells you which direction to go and how steep that direction is.

- A positive gradient means the loss increases if you go in that direction.
- A negative gradient means the loss decreases, and you should go that way.

In regression, we calculate gradients for each weight and bias to understand how changing them affects the model's predictions.

Task: While training your regression model, you notice the predictions are too high. You calculate the gradient of the loss with respect to the weight and find it is positive.

Question:

What does this tell you, and what should you do with the weight to improve performance?

Answer:

A positive gradient means increasing the weight made the prediction worse (too high). So, to improve performance, the model should **reduce** the weight.

What is Learning Rate?

The learning rate is a number that tells the model how big or small a step it should take while updating weights and bias.

- A small learning rate takes tiny steps and may take a long time to reach the best values.
- A large learning rate moves faster but may skip over the best solution or become unstable.

Think of it like turning the volume knob you want just the right setting, not too high or too low.

Task: You're training two models. One learns very slowly, and the other fluctuates wildly without improving. On checking, you find:

Model A uses a learning rate of 0.0001

Model B uses a learning rate of 1.0

Question:

Which model has what kind of issue and how should you fix it?

Answer:

Model A is learning too slowly due to a **very small learning rate**

Model B is unstable due to a **very large learning rate**
You should increase the learning rate slightly for Model A and reduce it for Model B to find a balance.

Optimization Using Gradient Descent

The process of improving the model by adjusting the weights and bias to reduce the loss is called optimization.

The most common method is Gradient Descent, which works like this:

1. Start with random weights and bias.
2. Make a prediction.

3. Calculate the loss (how wrong the prediction is).
4. Calculate gradients (how to improve).
5. Update weights and bias in the correct direction using the learning rate.
6. Repeat this process for many steps (called epochs) until the loss is as low as possible.

Task : You trained a model for predicting rental prices. Initially, the loss decreased well, but after 300 steps, it stopped improving even though your data is good.

Question:

What could be the reason the loss stopped improving, and how can you try to fix it?

Answer:

Possible reasons:

The learning rate is too low — model is stuck and learning very slowly

The model reached a local minimum

Fix:

Try increasing the learning rate slightly

Use momentum-based optimization or try different initial weights

Linear Regression in Action

Let's say you want to predict house price based on size in square feet.

The steps are:

- Take input: house size
- Multiply it by a weight
- Add a bias
- Get the predicted price
- Adjust weight and bias until predictions are close to actual prices

This is called a linear regression model, and it creates a straight line to fit the data.

.

Task: You used linear regression to predict car prices based on mileage. But the results are not accurate because luxury cars with high mileage are still expensive.

Question:
Why might linear regression not work well here, and what could be a better alternative?

Answer:
Linear regression assumes a straight-line relationship. But in this case, the relationship is more complex (non-linear). A better model would be **polynomial regression** or **decision trees** which can capture more complex patterns.

PART 2: PRACTICAL EXERCISES

Exercise 1: Build Linear Regression from Scratch Using Python

Dataset

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# House size in square feet
X = np.array([650, 800, 1200, 1500, 1850])


# House price in thousands of dollars
y = np.array([75, 100, 130, 160, 180])
```

```
# Normalize the data (important for better learning)
X = (X - X.mean()) / X.std()
```

- **Purpose:** To bring all input values (house sizes, for example) to a **standard scale**.
- **Why?** Normalization helps the model train **faster and more reliably** by avoiding huge or tiny numbers that could destabilize learning.

```
# Initialize weight and bias
weight = 0
bias = 0
```

- **Purpose:** Start with random or zero values.
- The **weight** controls the slope of the prediction line.
- The **bias** shifts the line up or down.
- Both will be **adjusted during training**.

```
# Set learning rate and number of training steps
learning_rate = 0.01
epochs = 1000
```

- **learning_rate**: Controls how big each adjustment (step) is when learning.
  - Small value = slower learning but safer
  - Large value = faster but can overshoot
- **epochs:** Number of times to repeat training on the full data

```
# Training loop
for i in range(epochs):
# Purpose: Repeat the update process many times to gradually improve the model.
    # Make predictions
    predictions = weight * X + bias
```

```
    # Calculate the loss
    error = y - predictions
    loss = (error ** 2).mean()
```

- error: Difference between actual and predicted values
- loss: Mean Squared Error (average of squared errors)
  - Squaring gives more weight to bigger mistakes
  - Lower loss means better model

```
    # Calculate gradients
    gradient_weight = -2 * (X * error).mean()
    gradient_bias = -2 * error.mean()
```

These formulas come from **calculus (derivatives)**.

- Tells us **in which direction and how much** we should adjust the weight and bias to **reduce the loss**.
- -2 comes from the derivative of squared error.

```
    # Update parameters
    weight = weight - learning_rate * gradient_weight
    bias = bias - learning_rate * gradient_bias
```

- Take a **step in the direction that reduces the loss**
- Multiply gradient by learning rate to control the step size

```
# Show final values
print(f"Weight: {weight}")
print(f"Bias: {bias}")
```

- After training is complete, print the final values the model has learned.
- These can be used to make future predictions:
  Predicted y = weight × input + bias

Visualize the Result

```
plt.scatter(X, y, label="Actual")
plt.plot(X, weight * X + bias, color='red', label="Predicted Line")
plt.title("Linear Regression from Scratch")
plt.xlabel("Normalized House Size")
plt.ylabel("House Price ($1000s)")
plt.legend()
plt.grid(True)
plt.show()
```

Exercise 2: Use scikit-learn for Linear Regression

```
from sklearn.linear_model import LinearRegression
```

Q1: Why do we import LinearRegression?

Answer:
Linear Regression is a ready-made class from scikit-learn that builds a linear model (a straight-line predictor). It saves us from coding the entire training process manually (like gradient descent, weight update, etc.).

```
# Reshape input for sklearn
X = X.reshape(-1, 1)
```

Q2: Why do we reshape X?

Answer:

Scikit-learn expects input features to be in a **2D array shape** (rows × columns).

Even if there's only **one feature**, we must reshape it from a 1D array ([x1, x2, x3,...]) to 2D ([[x1], [x2], [x3],...]).

Think of it like saying: "We have multiple samples, and each has 1 feature.

```
# Create and train the model
model = LinearRegression()
model.fit(X, y)
```

Q3: What does this LinearRegression() **do?**

Answer:

It creates an instance of the LinearRegression model.

This object (model) is ready to be trained using .fit().

Q4: What does .fit() do here?

Answer:
This trains the model by:

- Learning the **best weight and bias**
- Minimizing the error between actual and predicted values

```
# Get predictions
y_pred = model.predict(X)
```

Q5: What is happening here?

Answer:

Now that the model has been trained, it can predict outputs for the input X.

So, y_pred contains the predicted prices (or values) for each input feature.

```
# Print results
print("Model weight:", model.coef_)
print("Model bias:", model.intercept_)
```

Q6: What are .coef_ and .intercept_?

Answer:

- .coef_: The weight (slope) of the line — how much the output changes per unit change in input
- .intercept_: The bias (y-intercept) — where the line cuts the y-axis when input is zero

These are the values learned by the model.

Example:

If coef_ = 3 and intercept_ = 2, the model equation becomes:
predicted_price = 3 × input + 2

Visualize

```
plt.scatter(X, y, label="Actual Data")
plt.plot(X, y_pred, color='green', label="Sklearn Prediction")
plt.xlabel("Normalized House Size")
plt.ylabel("Price")
plt.title("Linear Regression with Sklearn")
plt.legend()
plt.grid(True)
plt.show()
```

Q7: Why do we plot scatter and line?

Answer:

- scatter(X, y): Shows the **original actual data points**
- plot(X, y_pred): Draws the **line fitted by the model**

This visualization helps compare **actual vs predicted values** and see how well the model has captured the trend.

Summary:

Step	What You Did	Why It Matters
1	Imported model	To use ready-made linear regression
2	Reshaped data	scikit-learn needs 2D input
3	Created model	To start training
4	Trained model	Found best weight and bias
5	Made predictions	Used the trained model to predict
6	Printed parameters	Saw the learned slope and intercept
7	Visualized data	Compared actual vs predicted results visually