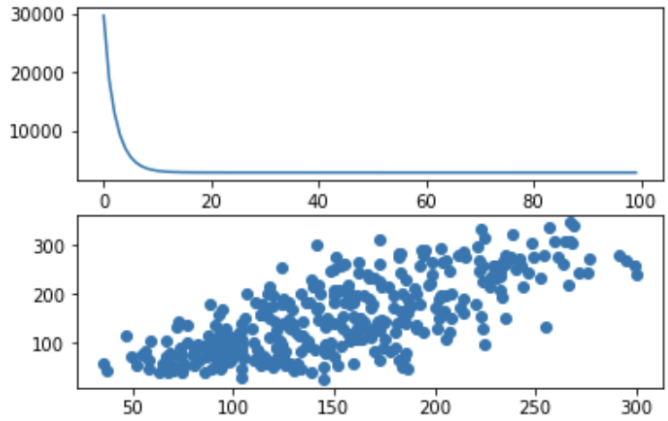



```
Parameter containing:
tensor([[  1.9400, -11.4488,  26.3047,  16.6306,  -9.8810,  -2.3179,  -7.6995,
           8.2121,  21.9769,   2.6065, 153.7365]])
Minimum cost: 2890.406494140625
```

Q5. What conclusion, if any, can be drawn from the weight values?

- Larger magnitudes suggest that the feature has a stronger influence on the prediction.

- The sign of each weight indicates the direction of the relationship between the feature and the target variable. Positive weights suggest that as the feature value increases, the predicted value also increases. Negative weights suggest the opposite.

How do gender and BMI affect blood sugar levels?

Gender: The weight for "SEX" is -11.4488. Given the encoding, this suggests that being female (encoded as 2) is associated with lower blood sugar levels compared to being male (encoded as 1). The negative weight indicates that as the gender value increases, the blood sugar level decreases.

BMI: The positive weight for "BMI" (26.3047) remains consistent with the previous interpretation: an increase in BMI is associated with increased blood sugar levels.

What are the estimated blood sugar levels for the below examples? [2 marks]

AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6
25	F	18	79	130	64.8	61	2	4.1897	68
50	M	28	103	229	162.2	60	4.5	6.107	124

```python
# Given that "M" is 1 and "F" is 2
table = torch.tensor([
    [25, 2, 18, 79, 130, 64.8, 61, 2, 4.1897, 68],  # Removed bias term for normalization
    [50, 1, 28, 103, 229, 162.2, 60, 4.5, 6.107, 124]  # Removed bias term for normalization
])

# Compute mean and standard deviation for each feature
mean = torch.mean(table, dim=0)
std = torch.std(table, dim=0)

# Normalize the table
normalized_table = (table - mean) / std

# Add the bias term back to the normalized table
normalized_table_with_bias = torch.cat([normalized_table, torch.ones(normalized_table.shape[0], 1)], dim=1)

# Get predictions from the model using the normalized data
predicted_blood_sugar = model(normalized_table_with_bias)

print(predicted_blood_sugar)
```

```
tensor([[ 93.9010],
        [213.5720]])
```

For the female, the estimated blood sugar level (Y) is approximately 93.90.
For the male, the estimated blood sugar level (Y) is approximately 213.57.

Q6. Try the code with a number of learning rates that differ by orders of magnitude and record the error of the training and test sets. What do you observe on the training error? What about the error on the test set? [3 marks]

```python
learning_rates = [0.0001, 0.001, 0.01, 0.1, 1, 10] # Store learning rates in a list
training_errors = list()
test_errors = list()

for alpha in learning_rates: # Iterate through each learning rate
    new_model = LinearRegression(x_train.shape[1]) # Initialise model

    new_cost_lst = list()
    for i in range(100): # Train the model with the current lr
        new_prediction = new_model(x_train)
        new_cost = mean_squared_error(y_train, new_prediction)
        new_cost_lst.append(new_cost)
        gradient_descent_step(new_model, x_train, y_train, new_prediction, alpha)

    training_errors.append(min(new_cost_lst).item()) # Record all training errors

    new_test_prediction = new_model(x_test) # Predict and record all test errors
    test_error = mean_squared_error(y_test, new_test_prediction)
    test_errors.append(test_error.item())

for lr, train, test in zip(learning_rates, training_errors, test_errors):
    print(f"Learning Rate: {lr}, Training Error: {train}, Test Error: {test}")

```

```
Learning Rate: 0.0001, Training Error: 28468.08203125, Test Error: 25530.2265625
Learning Rate: 0.001, Training Error: 20040.58203125, Test Error: 18534.30859375
Learning Rate: 0.01, Training Error: 3356.77734375, Test Error: 3431.06884765625
Learning Rate: 0.1, Training Error: 2890.406494140625, Test Error: 2885.619140625
Learning Rate: 1, Training Error: 29711.322265625, Test Error: nan
Learning Rate: 10, Training Error: 29711.322265625, Test Error: nan
```

Observations on both Training and Test Errors:

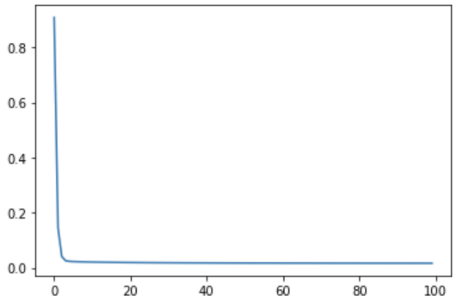
Very Small Learning Rates (0.0001, 0.001): The training error is high. The test error is also high, reflecting the model's suboptimal performance on the training set. This suggests that the model is converging very slowly and hasn't reached a good minimum within the given number of iterations.

Moderate Learning Rates (0.01, 0.1): The training error decreases significantly, indicating that these learning rates allow the model to converge to a better minimum. The test error is much lower, indicating good generalization. The test error for a learning rate of 0.1 is the lowest, suggesting this might be the optimal learning rate for this dataset and model.

High Learning Rates (1, 10): The training error is very high for learning rates of both 1 and 10. This suggests that the model might be diverging and not converging to a good solution. The updates might be too large, causing the model to overshoot the minimum. The test error is "nan", which stands for "not a number". This indicates that the model has diverged during training, and the predictions on the test set are not valid numbers (as they might be infinity).

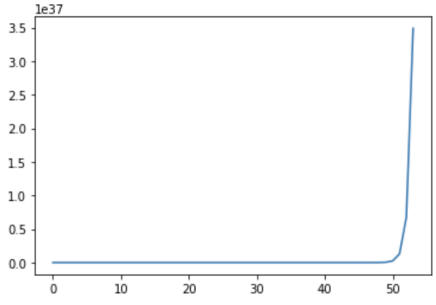
Q8. First of all, find the best value of alpha to use in order to optimize best. Next, experiment with different values of λ and see how this affects the shape of the hypothesis. [3 marks]

```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = 1 # select an appropriate alpha
lam = 0 # select an appropriate lambda
for it in range(100):
    prediction = model(x3)
    cost = mean_squared_error(y, prediction, lam, model.weight)
    cost_lst.append(cost)
    gradient_descent_step(model, x3, y, prediction, alpha, lam)
display.clear_output(wait=True)
plt.plot(list(range(it+1)), cost_lst)
plt.show()
print(model.weight)
print('Minimum cost: {}'.format(min(cost_lst)))
```




```
Parameter containing:
tensor([[-0.2919,  0.0907, -0.7664,  0.1417, -0.4623,  0.3817]])
Minimum cost: 0.01652376540005207
```

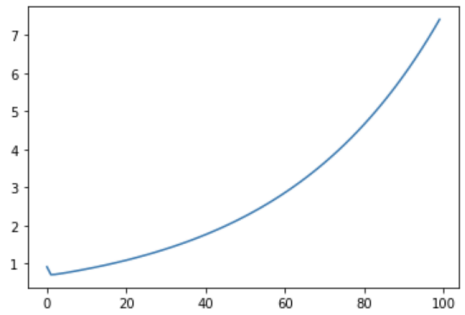
```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = -1 # select an appropriate alpha
lam = 0 # select an appropriate lambda
for it in range(100):
    prediction = model(x3)
    cost = mean_squared_error(y, prediction, lam, model.weight)
    cost_lst.append(cost)
    gradient_descent_step(model, x3, y, prediction, alpha, lam)
display.clear_output(wait=True)
plt.plot(list(range(it+1)), cost_lst)
plt.show()
print(model.weight)
print('Minimum cost: {}'.format(min(cost_lst)))
```




```
Parameter containing:
tensor([[-4.2590e+34, -1.2230e+35,  4.6152e+34, -9.0655e+34,  4.8501e+34,
         -2.6993e+35]])
Minimum cost: 0.9081819653511047
```

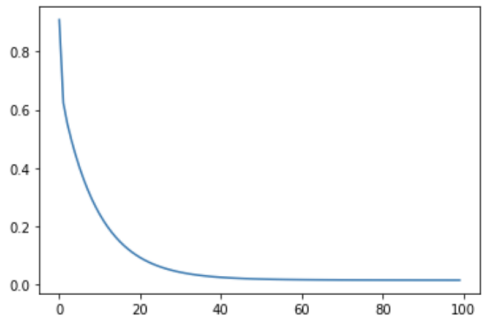
As the value of the learning rate gets closer to 1 or -1, the graph causes a convergence and the minimum cost increases. An alpha value higher than 1.5 causes a divergence in the graph. Therefore, a learning rate between 1-1.5 is more appropriate, since it won't over shoot the minimum and lead to underfitting.

```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = 1.5 # select an appropriate alpha
lam = 0 # select an appropriate lambda
for it in range(100):
    prediction = model(x3)
    cost = mean_squared_error(y, prediction, lam, model.weight)
    cost_lst.append(cost)
    gradient_descent_step(model, x3, y, prediction, alpha, lam)
display.clear_output(wait=True)
plt.plot(list(range(it+1)), cost_lst)
plt.show()
print(model.weight)
print('Minimum cost: {}'.format(min(cost_lst)))
```




```
Parameter containing:
tensor([[ 0.0876, -0.8115, -0.3828, -0.5787,  0.0809, -1.5358]])
Minimum cost: 0.7021127939224243
```

```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = 1.45 # select an appropriate alpha
lam = 0 # select an appropriate lambda
for it in range(100):
    prediction = model(x3)
    cost = mean_squared_error(y, prediction, lam, model.weight)
    cost_lst.append(cost)
    gradient_descent_step(model, x3, y, prediction, alpha, lam)
display.clear_output(wait=True)
plt.plot(list(range(it+1)), cost_lst)
plt.show()
print(model.weight)
print('Minimum cost: {}'.format(min(cost_lst)))
```

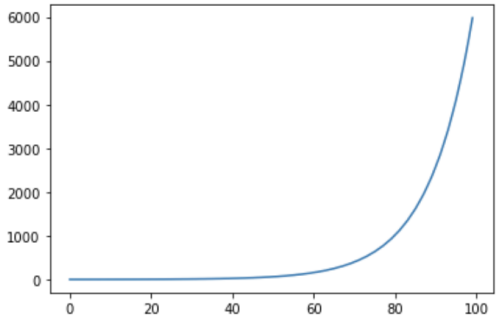



```
Parameter containing:
tensor([[-0.2650,  0.1032, -0.8568,  0.1266, -0.3983,  0.3777]])
Minimum cost: 0.016033830121159554
```

After carefully changing each decimal point, the best value of alpha was 1.45,
as the minimum cost for that was the lowest and the model had the best optimisation.

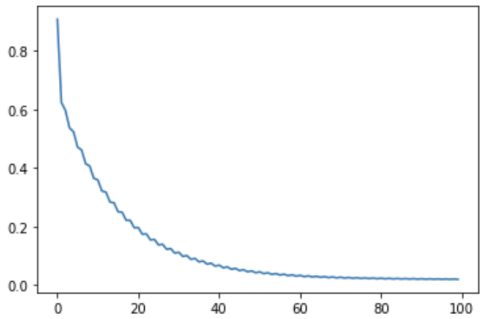
After experimenting with different values of lambda, it was clear that as the value gets closer to -1 or 1 and further away from 0, the shape of the hypothesis goes towards the opposite direction (leads to underfitting) and the minimum cost increases. Therefore, the best value for lambda was 0.

```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = 1.45 # select an appropriate alpha
lam = 0.5 # select an appropriate lambda
for it in range(100):
    prediction = model(x3)
    cost = mean_squared_error(y, prediction, lam, model.weight)
    cost_lst.append(cost)
    gradient_descent_step(model, x3, y, prediction, alpha, lam)
display.clear_output(wait=True)
plt.plot(list(range(it+1)), cost_lst)
plt.show()
print(model.weight)
print('Minimum cost: {}'.format(min(cost_lst)))
```




```
Parameter containing:
tensor([[  9.4302, -26.8454,  13.9253, -20.5532,  13.5903, -56.1262]])
Minimum cost: 0.623335063457489
```

```
cost_lst = list()
model = LinearRegression(x3.shape[1])
alpha = 1.45 # select an appropriate alpha
lam = 0.1 # select an appropriate lambda
for it in range(100):
    prediction = model(x3)
    cost = mean_squared_error(y, prediction, lam, model.weight)
    cost_lst.append(cost)
    gradient_descent_step(model, x3, y, prediction, alpha, lam)
display.clear_output(wait=True)
plt.plot(list(range(it+1)), cost_lst)
plt.show()
print(model.weight)
print('Minimum cost: {}'.format(min(cost_lst)))
```




```
Parameter containing:
tensor([[-0.4251,  0.1118, -0.5621,  0.1749, -0.4199,  0.3456]])
Minimum cost: 0.01922319270670414
```