In [29]:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, TensorDataset, random_split
import seaborn as sns
from sklearn.model_selection import train_test_split
data = sns.load_dataset('diamonds')
data.head()
matricule = 2200032 #Insérer le numéro/matricule de la carte d'étudiant
data = sns.load_dataset('diamonds')
dataframe = data.sample(220, random_state = matricule)
dataframe.head()
```

Out[29]:

	carat	cut	color	clarity	depth	table	price	x	y	z
43712	0.51	Ideal	E	SI1	61.7	55.0	1437	5.18	5.20	3.20
51787	0.74	Premium	F	SI1	60.7	59.0	2415	5.85	5.82	3.54
20409	1.34	Ideal	H	VS2	61.9	55.0	8771	7.05	7.08	4.37
21573	1.51	Fair	H	VS2	57.4	61.0	9678	7.49	7.63	4.34
34801	0.31	Ideal	D	VS1	60.5	55.0	877	4.43	4.39	2.67

In [30]:

```python
dataframe.describe()
```

Out[30]:

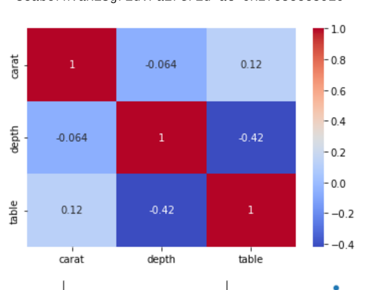
	carat	depth	table	price	x	y	z
count	220.000000	220.000000	220.000000	220.000000	220.000000	220.000000	220.000000
mean	0.775727	61.676364	57.396818	3949.077273	5.657727	5.662545	3.487273
std	0.459105	1.536165	2.312728	4116.433309	1.177658	1.174640	0.719077
min	0.230000	54.300000	53.000000	394.000000	0.000000	0.000000	0.000000
25%	0.380000	61.100000	56.000000	868.750000	4.645000	4.637500	2.877500
50%	0.700000	61.800000	57.000000	2570.500000	5.670000	5.690000	3.505000
75%	1.040000	62.500000	59.000000	5082.000000	6.507500	6.530000	4.022500
max	2.680000	66.900000	70.000000	17554.000000	8.810000	8.770000	5.580000

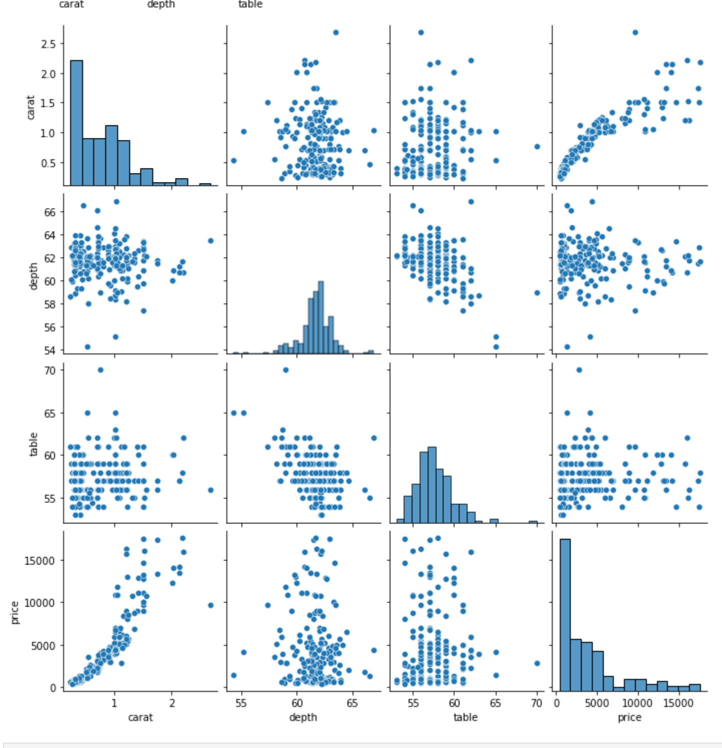
In [33]:

```python
input_cols = ["carat", "depth", "table"]
categorical_cols = ["color", "clarity"]
output_cols = ["price"]
#extraction des données de nos variables quantitatives d'étude et matrice de corrélation:
corr_matrix=dataframe[input_cols].corr()
print(corr_matrix)
#les coefficients de corrélations entre nos variables d'étude:
print(dataframe["carat"].corr(dataframe["table"]))
print(dataframe["carat"].corr(dataframe["depth"]))
print(dataframe["depth"].corr(dataframe["table"]))
#diagramme présentant la matrice de corrélation:
sns.heatmap(corr_matrix,annot=True, cmap='coolwarm')
sns.pairplot(data=dataframe, vars=input_cols + output_cols)
```

```
          carat     depth     table
carat  1.000000 -0.063892  0.121975
depth -0.063892  1.000000 -0.418196
table  0.121975 -0.418196  1.000000
0.12197530270300443
-0.06389177430723464
-0.4181957145559772
```

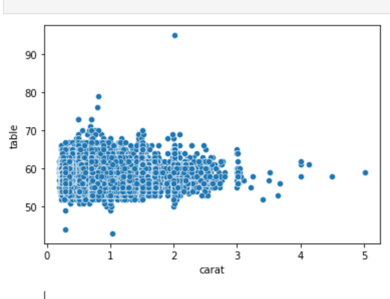
Out[33]: <seaborn.axisgrid.PairGrid at 0x27e5eee5910>

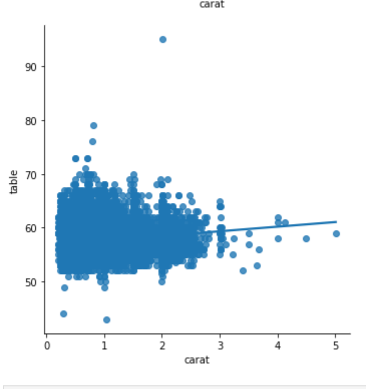




In [25]:

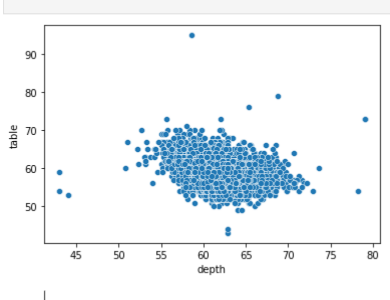
```python
#représentation:
sns.scatterplot(x="carat",y="table",data=sns.load_dataset('diamonds'))
sns.lmplot(x="carat",y="table",data=sns.load_dataset('diamonds'),ci=None)
plt.show()
```

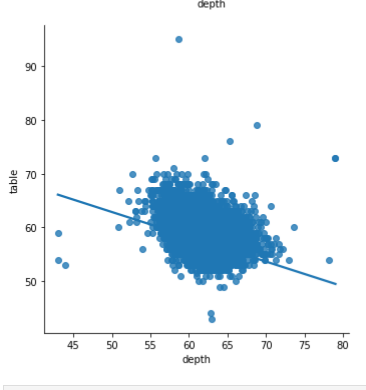




In [26]:

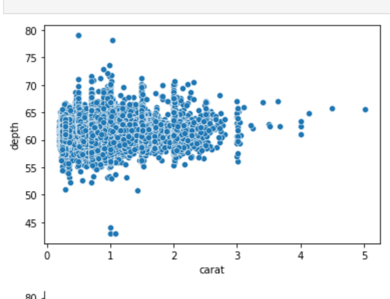
```python
sns.scatterplot(x="depth",y="table",data=sns.load_dataset('diamonds'))
sns.lmplot(x="depth",y="table",data=sns.load_dataset('diamonds'),ci=None)
plt.show()
```

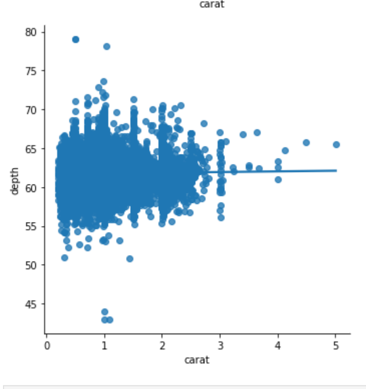




In [27]:

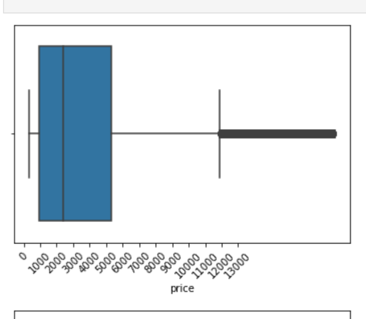
```python
sns.scatterplot(x="carat",y="depth",data=sns.load_dataset('diamonds'))
sns.lmplot(x="carat",y="depth",data=sns.load_dataset('diamonds'),ci=None)
plt.show()
```

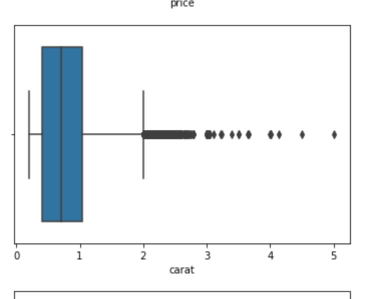


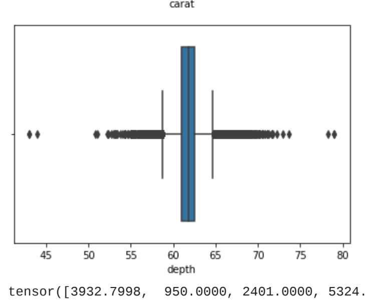


In [7]:

```python
dataframe1 = dataframe.copy(deep=True)
sns.boxplot(x="price",data=dataframe1)
echelle1=range(0,13500,1000)
plt.xticks(echelle1, rotation=45)
plt.show()
sns.boxplot(x="carat",data=dataframe1)
plt.show()
sns.boxplot(x="depth",data=dataframe1)
plt.show()
#Création d'un tableau de statistiques descriptives pour le prix:
data_price=torch.tensor(dataframe1["price"]).float()
mean_price=torch.mean(data_price)
qp1=torch.quantile(data_price, 0.25)
qp2=torch.quantile(data_price, 0.5)
qp3=torch.quantile(data_price, 0.75)
std_p=torch.std(data_price)
stderr_p=std_p / torch.sqrt(torch.tensor(len(data_price)))
#Intervalle de confiance pour les prix :
t_value_p=torch.abs(torch.randn(len(data_price)))
ci_p_lower=mean_price - t_value_p.std() * stderr_p
ci_p_upper = mean_price + t_value_p.std() * stderr_p
tableau_stats_price = torch.tensor([mean_price, qp1, qp2, qp3, std_p, stderr_p, ci_p_lower,ci_p_upper])
print(tableau_stats_price)
#Création d'un tableau de statistiques descriptives pour le carat:
data_carat=torch.tensor(dataframe1["carat"]).float()
mean_carat=torch.mean(data_carat)
qc1=torch.quantile(data_carat, 0.25)
qc2=torch.quantile(data_carat, 0.5)
qc3=torch.quantile(data_carat, 0.75)
std_c=torch.std(data_carat)
stderr_c=std_c / torch.sqrt(torch.tensor(len(data_carat)))
#Intervalle de confiance pour carat :
t_value_c=torch.abs(torch.randn(len(data_carat)))
ci_c_lower=mean_carat - t_value_c.std() * stderr_c
ci_c_upper = mean_carat + t_value_c.std() * stderr_c
tableau_stats_carat = torch.tensor([mean_carat, qc1, qc2, qc3, std_c, stderr_c, ci_c_lower,ci_c_upper])
print(tableau_stats_carat)
#Création d'un tableau de statistiques descriptives pour le depth:
data_depth=torch.tensor(dataframe1["depth"]).float()
mean_depth=torch.mean(data_depth)
qd1=torch.quantile(data_depth, 0.25)
qd2=torch.quantile(data_depth, 0.5)
qd3=torch.quantile(data_depth, 0.75)
std_d=torch.std(data_depth)
stderr_d=std_d / torch.sqrt(torch.tensor(len(data_depth)))
#Intervalle de confiance pour les depth :
t_value_d=torch.abs(torch.randn(len(data_depth)))
ci_d_lower=mean_depth - t_value_d.std() * stderr_d
ci_d_upper = mean_depth + t_value_d.std() * stderr_d
tableau_stats_depth = torch.tensor([mean_depth, qd1, qd2, qd3, std_d, stderr_d, ci_d_lower,ci_d_upper])
print(tableau_stats_depth)
```







```
tensor([3932.7998,  950.0000, 2401.0000, 5324.2500, 3989.4397,   17.1774,
        3922.4189, 3943.1807])
tensor([0.7979, 0.4000, 0.7000, 1.0400, 0.4740, 0.0020, 0.7967, 0.7992])
tensor([6.1749e+01, 6.1000e+01, 6.1800e+01, 6.2500e+01, 1.4326e+00, 6.1684e-03,
        6.1746e+01, 6.1753e+01])
```

In [8]:

```python
#Exemple :
X = torch.tensor(dataframe1[['carat', 'depth']].values, dtype=torch.float32)
Y = torch.tensor(dataframe1['price'].values, dtype=torch.float32)
#Définition d'un modèle linéaire qui prend 2 inputs et rentre 1 output:
model=nn.Linear(2,1)
#Définition de la fonction Loss L1; l'erreur de la valeur absolue de la moyenne et de formule L1 loss = 1/n * ∑|y_pred - y_true| où n représente le nombre
#d'échantillons, y_pred la valeur prédite et y_true la valeur réelle:
loss_fn =nn.L1Loss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
#l'optimizer est responsable de mettre à jour w1 et w2 à chaque étape pour minimiser la perte, le SGD (Stochastic Gradient Descent)fonctionne en utilisant les
#gradients pour ajuster w1 et w2, Le paramètre lr représente le taux d'apprentissage càd pas que l'optimiseur prend pour se rapprocher du minimum de la foncti
for epoch in range(100):
  y_pred = model(X) #calcul de la valeur prédite de y selon notre modèle linéaire
  loss = loss_fn(y_pred.squeeze(), Y) #calcule l'erreur entre la valeur prédite et la valeur réelle
  optimizer.zero_grad() #Réinitialise les gradients de l'optimiseur à zéro.
  loss.backward() #Calcule les gradients de la perte par rapport aux paramètres du modèle.
  optimizer.step() #Actualise les paramètres du modèle en utilisant les gradients calculés.
  if epoch % 10 == 0: #vérifie si le nombre d'itérations est multiple de 10
    print(f"Epoch {epoch}, Loss {loss.item():.4f}") #retourne la valeur de epoch et le loss
```

```
Epoch 0, Loss 3907.3354
Epoch 10, Loss 3526.2908
Epoch 20, Loss 3239.9746
Epoch 30, Loss 3106.7495
Epoch 40, Loss 3035.6121
Epoch 50, Loss 2988.2539
Epoch 60, Loss 2952.6328
Epoch 70, Loss 2924.7341
Epoch 80, Loss 2899.8442
Epoch 90, Loss 2879.7625
```

In [9]:

```python
cut_encoded = pd.get_dummies(dataframe1['cut'], prefix='cut')
color_encoded = pd.get_dummies(dataframe1['color'], prefix='color')
clarity_encoded = pd.get_dummies(dataframe1['clarity'], prefix='clarity')
dataframe1_encoded = pd.concat([dataframe1[['carat', 'depth', 'table']], cut_encoded, color_encoded, clarity_encoded], axis=1)
X = torch.tensor(dataframe1_encoded.values, dtype=torch.float32)
Y = torch.tensor(dataframe1['price'].values, dtype=torch.float32)
print(X.shape, Y.shape)
class Perceptron(nn.Module):
  def __init__(self):
    super(Perceptron, self).__init__()
    self.fc1=nn.Linear(23,1)#création d'une transformation linéaire qui prend input de taille 23(nombre de variables de X) et retourne un output de taille 1
  def forward(self,x):
    x=torch.sigmoid(self.fc1(x))#application de la fonction sigmoid
    return x
model = Perceptron()
criterion=nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
for epoch in range(100):
    optimizer.zero_grad()
    outputs = model(X)
    loss = criterion(outputs, Y.unsqueeze(1))
    loss.backward()
    optimizer.step()
print("Final loss:", loss.item())
```

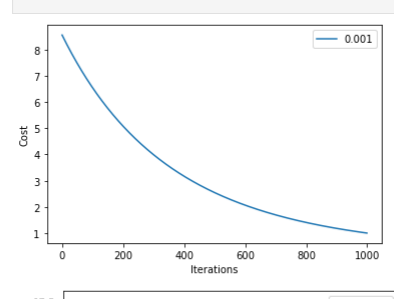
```
torch.Size([53940, 23]) torch.Size([53940])
Final loss: 31374384.0
```

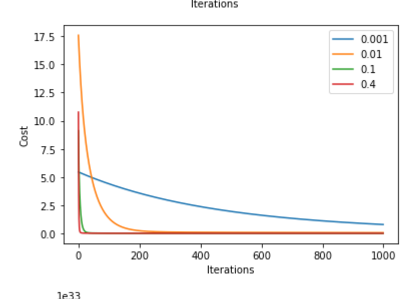
In [17]:

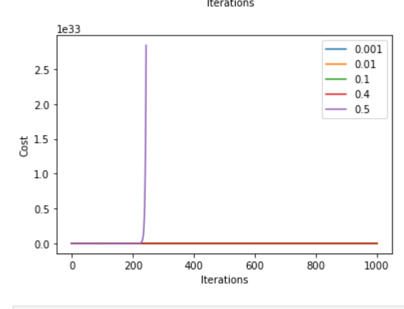
```python
dataframe1 = dataframe.copy(deep=True)
dataframe1 = pd.get_dummies(dataframe1 , columns=['cut', 'color', 'clarity']) #convertit les colonnes catégorielles 'cut', 'color', et 'clarity' en variables
dataframe1= (dataframe1 - dataframe1.mean())/dataframe1.std() #standardise les données De cette manière, chaque variable a une moyenne de 0 et un écart-type d
dataframe1 = dataframe1.sample(frac=1).reset_index(drop=True)#mélange les données de façon aléatoire (frac=1) et réindexe la DataFrame résultante en remettant
# Séparation des données en ensembles d'entraînement et de test
train_data, test_data = train_test_split(dataframe1, test_size=0.2, random_state=42)
# Créer des tenseurs PyTorch à partir des données:
X_train = torch.tensor(train_data.drop(['price'], axis=1).values, dtype=torch.float32)
y_train = torch.tensor(train_data[['price']].values, dtype=torch.float32)
X_test = torch.tensor(test_data.drop(['price'], axis=1).values, dtype=torch.float32)
y_test = torch.tensor(test_data[['price']].values, dtype=torch.float32)
```

In [18]:

```python
# Définir la fonction de coût MSE et ses dérivées partielles :
def mse_cost_function(X, y, w):
  predictions = torch.matmul(X,w)#Multiplication matricielle pour prédire les sorties à partir des entrées X et des poids (weights) w
  errors = predictions - y #l'erreur de la prédiction
  cost = torch.sum(errors ** 2) / (2 * len(X))#calcule la fonction de coût en utilisant la moyenne des carrés des erreurs
  gradients = torch.matmul(torch.transpose(X, 0, 1), errors) / len(X)#calcule le gradient du coût
  return cost, gradients
# Définir la méthode de descente de gradient :
def gradient_descent(X, y, learning_rate, num_iterations):
  # Initialiser les poids:
  w = torch.randn(X.shape[1], 1)
  # Initialiser les coûts
  costs = []
  # Boucle de descente de gradient:
  for i in range(num_iterations):
    cost, gradients = mse_cost_function(X, y, w)
    w -= learning_rate * gradients
    costs.append(cost)
  return w, costs
# Tester différentes valeurs de taux d'apprentissage:
#exemple1 :
learning_rates = [0.001]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
#exemple 2:
learning_rates = [0.001, 0.01, 0.1,0.4]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
#exemple3:
learning_rates = [0.001, 0.01, 0.1,0.4,0.5]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
```

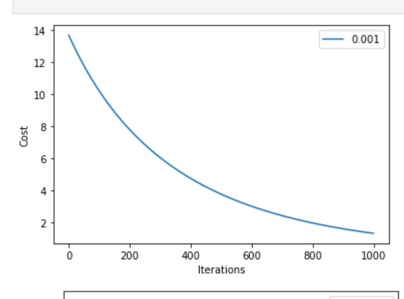


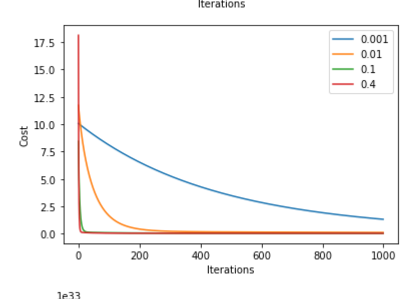


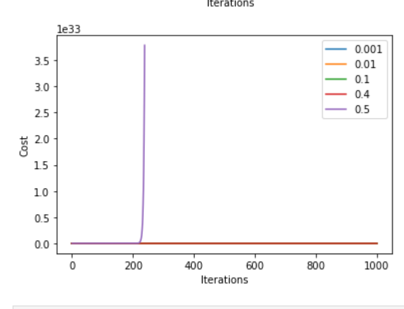


In [21]:

```python
# Définir la fonction de coût MSE et ses dérivées partielles :
def mse_cost_function(X, y, w):
  predictions = torch.matmul(X,w)#Multiplication matricielle pour prédire les sorties à partir des entrées X et des poids (weights) w
  errors = predictions - y #l'erreur de la prédiction
  cost = torch.sum(errors ** 2) / (2 * len(X))#calcule la fonction de coût en utilisant la moyenne des carrés des erreurs
  gradients = torch.matmul(torch.transpose(X, 0, 1), errors) / len(X)#calcule le gradient du coût
  return cost, gradients
#Définir la méthode de descente à pas optimal :
def optimal_step_descent(X, y, num_iterations):
  w = torch.randn(X.shape[1],1)
  costs=[]
  for i in range(num_iterations):
    alpha = torch.matmul(torch.matmul(torch.inverse(torch.matmul(torch.transpose(X, 0, 1), X)), torch.transpose(X, 0, 1)), y)
    cost, gradients = mse_cost_function(X, y, w)
    w -= alpha * gradients
    costs.append(cost)
  return w, costs
# Tester différentes valeurs de taux d'apprentissage:
#exemple1 :
learning_rates = [0.001]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
#exemple2 :
learning_rates = [0.001, 0.01, 0.1,0.4]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
#exemple3:
learning_rates = [0.001, 0.01, 0.1,0.4,0.5]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
```



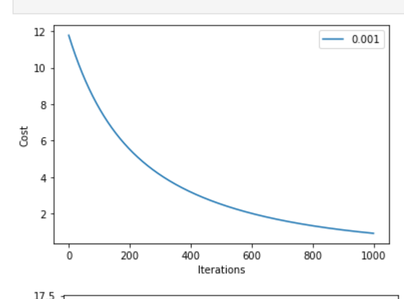


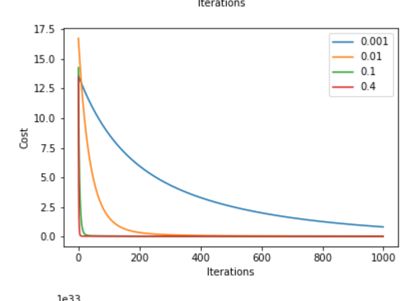


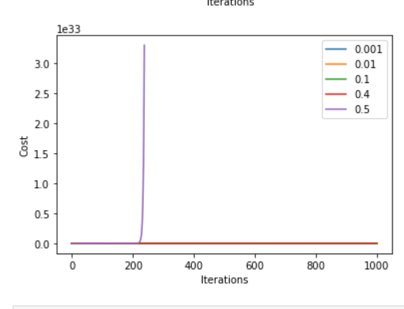
In [20]:

```python
# Définir la fonction de coût MSE et ses dérivées partielles :
def mse_cost_function(X, y, w):
  predictions = torch.matmul(X,w)#Multiplication matricielle pour prédire les sorties à partir des entrées X et des poids (weights) w
  errors = predictions - y #l'erreur de la prédiction
  cost = torch.sum(errors ** 2) / (2 * len(X))#calcule la fonction de coût en utilisant la moyenne des carrés des erreurs
  gradients = torch.matmul(torch.transpose(X, 0, 1), errors) / len(X)#calcule le gradient du coût
  return cost, gradients
#Définir la méthode de descente stochastique :
def stochastic_gradient_descent(X, y, learning_rate, num_iterations):
    w = torch.randn(X.shape[1], 1)
    costs = []
    for i in range(num_iterations):
        cost_total = 0
        for j in range(len(X)):
            rand_index = np.random.randint(len(X))
            X_j = X[rand_index].unsqueeze(0)
            y_j = y[rand_index].unsqueeze(0)
            cost, gradients = mse_cost_function(X_j, y_j, w)
            w -= learning_rate * gradients
            cost_total += cost
        costs.append(cost_total / len(X))
    return w, costs

# Tester différentes valeurs de taux d'apprentissage:
#exemple1 :
learning_rates = [0.001]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
#exemple2 :
learning_rates = [0.001,0.01,0.1,0.4]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
#exemple3 :
learning_rates = [0.001,0.01,0.1,0.4,0.5]
num_iterations = 1000
for j in learning_rates:
  weights, costs = gradient_descent(X_train, y_train, j, num_iterations)
  plt.plot(costs, label=str(j))
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.legend()
plt.show()
```







In [ ]: