Problem 1

Setting up the enrivonment and loading the data. No need to change this part.

In [20]:
```
import torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt

seed = 42
torch.manual_seed(seed)
np.random.seed(seed)

# Load tensors from the file
loaded_tensors = torch.load('exercise_2_problem_1_data.pth')
X_tensor = loaded_tensors['X_tensor']
Y_tensor = loaded_tensors['Y_tensor']
dataset = TensorDataset(X_tensor, Y_tensor)
```

a) Model definition

Finalize the model definition as instructed in the exercise sheet.

In [21]:
```
class RegressionModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, 400),
            nn.ReLU(),
            nn.Linear(400, 200),
            nn.ReLU(),
            nn.Linear(200, 100),
            nn.ReLU(),
            nn.Linear(100, output_dim)
        )

    def forward(self, x):
        return self.layers(x)
```

b) Write optimization loop

You can use any optimizer you want, but remember to set all the hyperparameters it requires, including the batch size and number of iterations etc that are defined outside the optimizer function.

In [22]:
```
# Helper for retrieving minibatches of desired size
device = "cuda" if torch.cuda.is_available() else "cpu"

batch_size = 64
```

```
print(f"Batch size is {batch_size}")

dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)


model = RegressionModel(100, 2).to(device)
criterion = nn.MSELoss()
lr = 0.0001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
print(f"Optimizer is {optimizer}\n")

num_epochs = 200
printing_interval = num_epochs // 10 # Avoid printing hundreds or thousands of l
losses_ = []

model.train()
for epoch in range(num_epochs):
    running_loss = 0.0

    for x_batch, y_batch in dataloader:
        # send only the current batch to GPU (if available)
        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)

        optimizer.zero_grad()
        y_pred = model(x_batch)
        loss = criterion(y_batch, y_pred)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    avg_loss = running_loss / len(dataloader)
    losses_.append(avg_loss)

    if (epoch % printing_interval == 1):
        print(f"Epoch [{epoch+1}/{num_epochs}] - MSE Loss: {avg_loss:.6f}")

    if avg_loss < 1e-6:
        print("Reached training error of < 1e-6, stopping")
        break
print(f"Final MSE Loss {losses_[-1]} after {len(losses_)} epochs")
```

```
Batch size is 64
Optimizer is Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.0001
    maximize: False
    weight_decay: 0
)

Epoch [2/200] - MSE Loss: 0.880157
Epoch [22/200] - MSE Loss: 0.148335
Epoch [42/200] - MSE Loss: 0.042844
Epoch [62/200] - MSE Loss: 0.011118
Epoch [82/200] - MSE Loss: 0.002442
Epoch [102/200] - MSE Loss: 0.000474
Epoch [122/200] - MSE Loss: 0.000075
Epoch [142/200] - MSE Loss: 0.000010
Epoch [162/200] - MSE Loss: 0.000001
Reached training error of < 1e-6, stopping
Final MSE Loss 9.7856562675247e-07 after 166 epochs
```

d) Computing the gradient norms

Practice extracting information about a trained model, by computing the gradient norms that were used to analyse SGD behavior during the lectures. No need to do anything with the gradient norms, just show that you can compute them.

In [23]:
```python
# Switch to evaluation mode -- we are no longer training!
model.eval()

grad_norms = []  # to store gradient norms for each batch

for x_batch, y_batch in dataloader:
    x_batch = x_batch.to(device)
    y_batch = y_batch.to(device)

    y_pred = model(x_batch)

    loss = criterion(y_batch, y_pred)
    loss.backward()

    batch_norm = 0
    for p in model.parameters():
        p_norm = torch.linalg.norm(p.grad)
        batch_norm += p_norm.item()**2 # squared norm

    grad_norms.append(batch_norm)

# Convert to numpy array for statistics
grad_norms = np.array(grad_norms)
mean_gn = grad_norms.mean()
var_gn  = grad_norms.var()
```

```
print("Gradient Norm stats for evaluation pass:")
print(f"  Mean   : {mean_gn:.6f}")
print(f"  Std    : {var_gn:.6f}")
```

```
Gradient Norm stats for evaluation pass:
  Mean   : 0.002544
  Std    : 0.000004
```

e) Reporting

Edit this cell directly to write your answers. You should print the values within the code blocks, but also copy them here for ease of grading. You can also write any other remarks you may have in this cell.

Optimizer settings:

I used Adam with with a stepsize of $\mu = 0.0001$ and the rest kept at default values, i.e. $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. The batch size was $B = 64$.

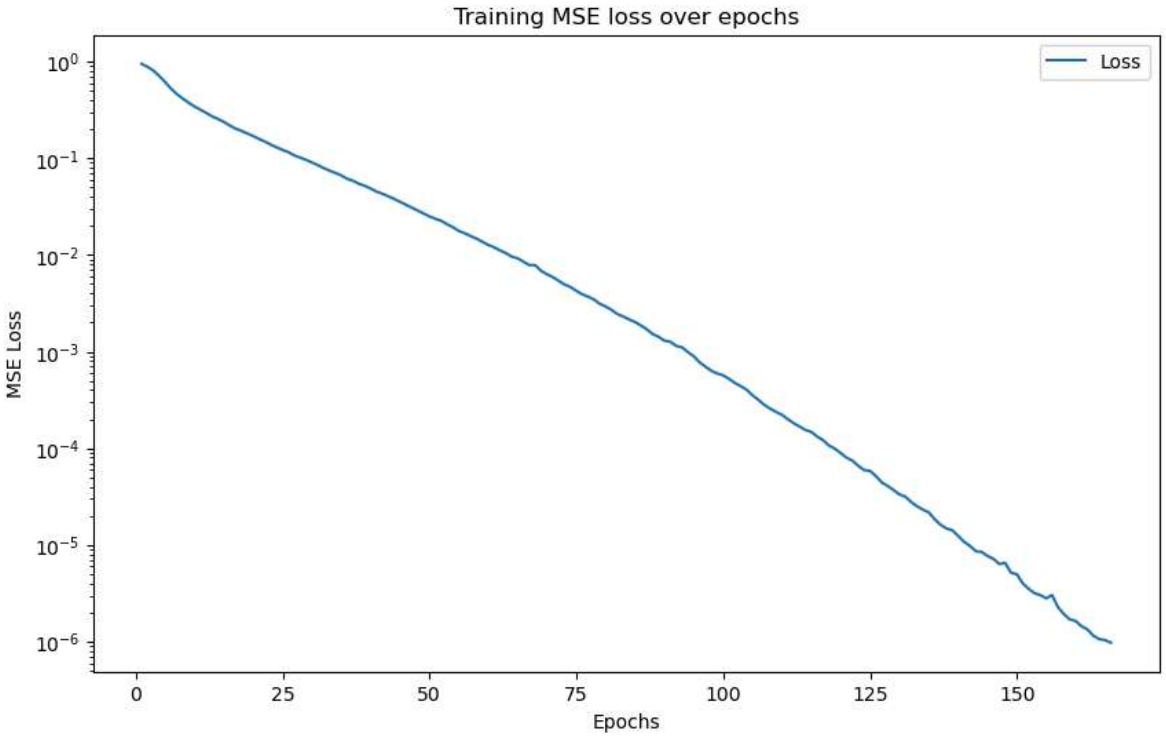
Optimization speed:

I reached the loss threshold of 10^{-6} after 166 epochs.

The gradient norms:

After training the model, the gradient norm is 0.002544 and the variance over the batches is 0.000004.

In [24]:
```
plt.figure(figsize=(10,6))
plt.semilogy(range(1, len(losses_) + 1), losses_, label="Loss")
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("MSE Loss")
plt.title("Training MSE loss over epochs")
plt.show()
```



==========

Problem 2

a) Generate training/testing data

Do not change this part. The code snippet greates synthetic data by feeding random inputs through randomly initialized neural network.

In [25]:
```
# N samples with D inputs and O outputs
N = 100
D = 10
O = 5

# Random mapping from x to y, as small neural network
class CreationModel(nn.Module):
  def __init__(self, D, O, M):
    super().__init__()
    self.layers = nn.Sequential(
      nn.Linear(D, M),
      nn.Tanh(),
      nn.Linear(M, O)
    )

  def forward(self, x):
    return self.layers(x)

# For getting the same data
torch.manual_seed(78798)

# Training data
x = torch.randn(N, D)
```

```
noiselevel = 0.1
modelGenerate = CreationModel(D,O,5)
modelGenerate.eval()
y = modelGenerate(x).clone().detach() + noiselevel*torch.randn(N, O)

# Test data
N_test = 10000
x_test = torch.randn(N_test, D)
y_test = modelGenerate(x_test).clone().detach()  + noiselevel*torch.randn(N_test
```

Pre-defined model

Do not change this part. The parameter M controls the size/complexity of the network.

In [26]:
```
class RegressionModel(nn.Module):
  def __init__(self, D, O, M):
    super().__init__()
    self.layers = nn.Sequential(
      nn.Linear(D, M),
      nn.ReLU(),
      nn.Linear(M, M),
      nn.ReLU(),
      nn.Linear(M, O)
    )

  def forward(self, x):
    return self.layers(x)
```

b) Train model and validate the double descent principle

In [27]:
```
# Use full data for gradients
B = N
data_loader = DataLoader(TensorDataset(x,y), batch_size=B, shuffle=True)

# Loop over some range of M values. With these parameters logspace generates M=2
Mvalues = np.logspace(np.log10(2),np.log10(60),num=15,dtype='int')[1:]
losses = np.zeros((len(Mvalues),2))
for M_index, M in enumerate(Mvalues):
    model = RegressionModel(D, O, M).to(device)

    criterion = nn.MSELoss()
    lr = 0.001
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    num_epochs = 2000

    model.train()

    for epoch in range(num_epochs):
        running_loss = 0.0

        for x_batch, y_batch in data_loader:
            # send only the current batch to GPU (if available)
            x_batch = x_batch.to(device)
            y_batch = y_batch.to(device)
```

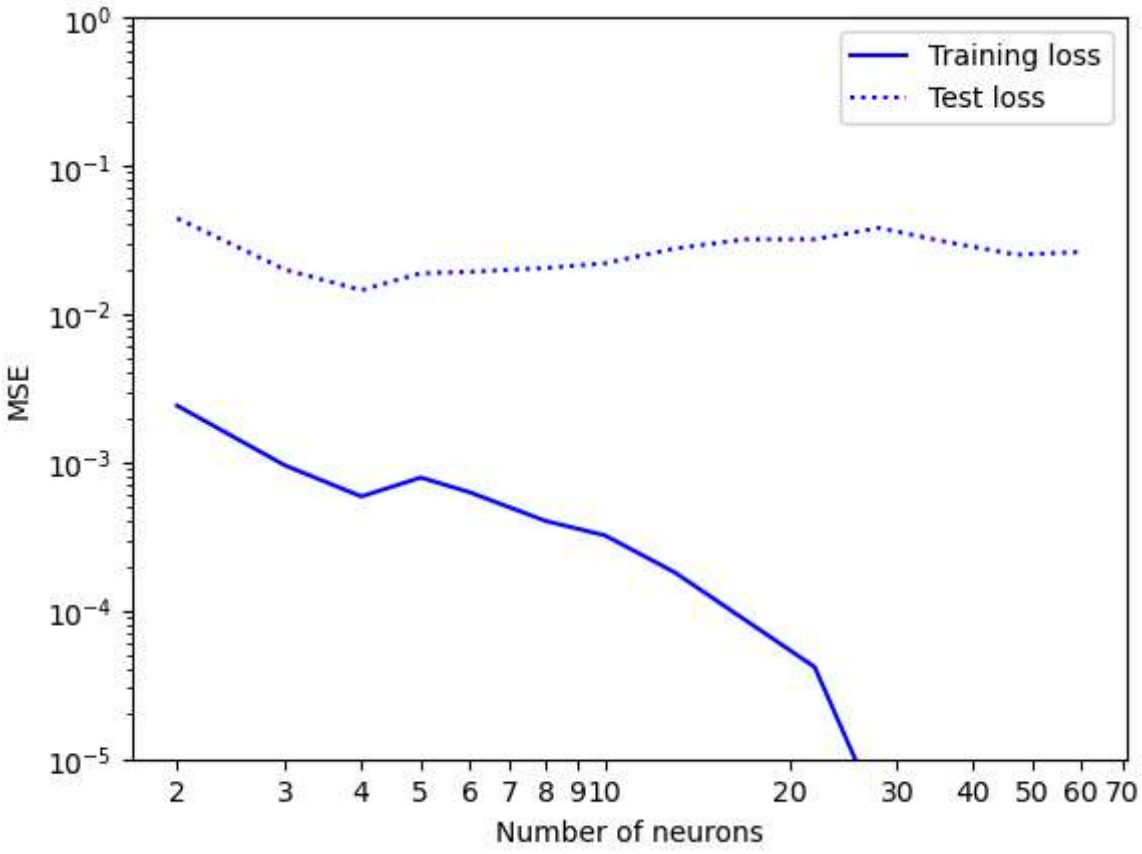
```
            optimizer.zero_grad()
            y_pred = model(x_batch)
            loss = criterion(y_batch, y_pred)

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        trainloss = running_loss / len(dataloader)

    model.eval()

    y_pred = model(x_test.to(device))
    testloss= criterion(y_test.to(device), y_pred)

    losses[M_index,0] = trainloss
    losses[M_index,1] = testloss.item()
    print(f"Done training M={M}: Train loss = {trainloss:.6f} Test loss = {testl
```

```
Done training M=2: Train loss = 0.002415 Test loss = 0.044234
Done training M=3: Train loss = 0.000954 Test loss = 0.019938
Done training M=4: Train loss = 0.000589 Test loss = 0.014479
Done training M=5: Train loss = 0.000790 Test loss = 0.018837
Done training M=6: Train loss = 0.000631 Test loss = 0.019241
Done training M=8: Train loss = 0.000403 Test loss = 0.020428
Done training M=10: Train loss = 0.000322 Test loss = 0.021974
Done training M=13: Train loss = 0.000181 Test loss = 0.027629
Done training M=17: Train loss = 0.000086 Test loss = 0.031921
Done training M=22: Train loss = 0.000042 Test loss = 0.031943
Done training M=28: Train loss = 0.000004 Test loss = 0.038065
Done training M=36: Train loss = 0.000000 Test loss = 0.030734
Done training M=47: Train loss = 0.000000 Test loss = 0.025125
Done training M=60: Train loss = 0.000000 Test loss = 0.026199
```

Plot the final training loss and the final test loss

The code below is provided as an example; you may need to modify it to be compatible with your results. Make some effort to make the plot easy to read.

In [28]:
```
plt.loglog(Mvalues, losses[:,0], 'b-')
plt.loglog(Mvalues, losses[:,1], 'b:')
plt.legend(['Training loss', 'Test loss'])
ax = plt.gca(); ax.set_ylim([10**(-5),10**(0)])
ax.set_ylabel("MSE")
ax.set_xlabel("Number of neurons")
# Customize x-axis to show more detailed labels
from matplotlib.ticker import LogLocator, ScalarFormatter
ax.xaxis.set_major_locator(LogLocator(base=10.0, subs=np.arange(1, 10)))
ax.xaxis.set_major_formatter(ScalarFormatter())
plt.show()
```



c) Having more data can hurt

Show that increasing the amount of data helps when around the interpolation threshold, but that for overparameterized model it may hurt.

In [29]:
```
# Generate the larger training set, using the same process as before
N_large = 500
x_large = torch.randn(N_large, D)
y_large = modelGenerate(x_large).clone().detach() + noiselevel*torch.randn(N_lar

Mvalues_new = [28, 47]
# store the indices of the new M values to find the losses in the original losse
n_100_idx = [np.where(Mvalues == M)[0].item() for M in Mvalues_new]

losses_large = np.zeros((len(Mvalues_new),2))
for M_index, M in enumerate(Mvalues_new):
    model = RegressionModel(D, O, M).to(device)

    criterion = nn.MSELoss()
    lr = 0.001
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    num_epochs = 2000

    model.train()

    for epoch in range(num_epochs):
        running_loss = 0.0

        for x_batch, y_batch in data_loader:
            # send only the current batch to GPU (if available)
```

```
            x_batch = x_batch.to(device)
            y_batch = y_batch.to(device)

            optimizer.zero_grad()
            y_pred = model(x_batch)
            loss = criterion(y_batch, y_pred)

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        trainloss = running_loss / len(dataloader)


    model.eval()

    y_pred = model(x_test.to(device))
    testloss= criterion(y_test.to(device), y_pred)

    losses_large[M_index, 0] = trainloss
    losses_large[M_index, 1] = testloss.item()
    print(f"Done training M={M}: Train loss = {trainloss:.6f} Test loss = {testl
    # count only trainable parameters
    total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f"Parameters = {total_params}")

print()
print("Test losses")
for i in range(len(Mvalues_new)):
    print(f"M = {Mvalues_new[i]}:")
    print(f"\tN = 100 test loss = {losses[n_100_idx[i], 1]:.6f}")
    print(f"\tN = 500 test loss = {losses_large[i, 1]:.6f}" )
    print()
```

```
Done training M=28: Train loss = 0.000003 Test loss = 0.033671
Parameters = 1265
Done training M=47: Train loss = 0.000000 Test loss = 0.027610
Parameters = 3013

Test losses
M = 28:
        N = 100 test loss = 0.038065
        N = 500 test loss = 0.033671

M = 47:
        N = 100 test loss = 0.025125
        N = 500 test loss = 0.027610
```

Reporting

Edit this cell directly to report the requested information.

Interpolation threshold

All models were trained for 2000 epochs to ensure convergence. The optimizer was Adam with default values ($\mu = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$). The

interpolation threshold is around $M = 28$, since this is where the train error goes to (nearly) zero, whereas the test error is high.

For that M, the model has in total 1265 (trainable) parameters.

The pattern predicted in the lecture notes is observed, although quite subtly. The test error decreases at first, then starts rising as we go towards the interpolation threshold. At the interpolation threshold the training error vanishes and the test error is high. Beyond the interpolation threshold the test error starts to decrease again.

More data can hurt

I chose the values of M based on the plot. You can roughly see where the training error takes a dive and the test error is large before starting to decrease again. The values of M were ultimately picked based on the printed train/test errors. I chose the overparametrized regime as $M = 47$, since it has over twice the amount of parameters than $M = 28$, but not too much such that it would fit the $N = 500$ training points as well. When increasing the amount of training data, the interpolation threshold is indeed shifted to the right as expected, such that $M = 28$ has lower test error than previously, while the larger $M = 47$ model does not fit the data as well anymore. Again though, the effect is not that noticeable. Thus more data is not always the best, unless you also increase the size of the model.

The test errors were as follows:

- M = 28:
  - N = 100 test loss = 0.038065
  - N = 500 test loss = 0.033671
- M = 47:
  - N = 100 test loss = 0.025125
  - N = 500 test loss = 0.027610