Exercise 1.1

We want to show that the empirical risk $R_{\text{emp}}(f|\mathcal{D})$ converges to the expected risk $R(f)$ as the number of training samples $N \to \infty$.

Let $Z_i = \mathcal{L}(y_i, f(x_i))$, where (x_i, y_i) are i.i.d. samples and $\mathcal{L}$ - loss function. Then each Z_i is a bounded random variable, and all Z_i are i.i.d.

By the Law of Large Numbers, the sample average converges to the expected value:

$$\frac{1}{N}\sum_{i=1}^{N} Z_i \to \mathbb{E}[Z_i] \quad \text{as } N \to \infty.$$

This implies:

$$R_{\text{emp}}(f|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(y_i, f(x_i)) \to \mathbb{E}_{p(x,y)}[\mathcal{L}(y, f(x))] = R(f).$$

So the empirical risk converges to the expected risk, meaning R_{emp} is a consistent estimator of R.

Exercise 1.2

(a)

Model $f_1 \in \mathcal{H}_1$ has higher bias but lower variance. Since $\mathcal{H}_1$ is a simple linear functions), the model cannot capture complex relationships in the data, which results in high bias. However, due to simplicity, it also means that it behaves more consistently across different training sets, hence the low variance.

Model $f_2 \in \mathcal{H}_2$, a complex hypothesis space with high-degree polynomials has lower bias because it can fit a wider range of functions, including the training data very closely. But this increased flexibility leads to higher variance—small changes in the training data can lead to large changes in the fitted model.

This tradeoff explains how we can observe:

$$R_{\text{emp}}(f_2|\mathcal{D}) < R_{\text{emp}}(f_1|\mathcal{D}) \quad \text{but} \quad R(f_2) > R(f_1).$$

Model f_2 overfits the training data: it achieves a low empirical risk by closely fitting even the noise in the training set, but generalizes poorly, resulting in a higher expected risk.

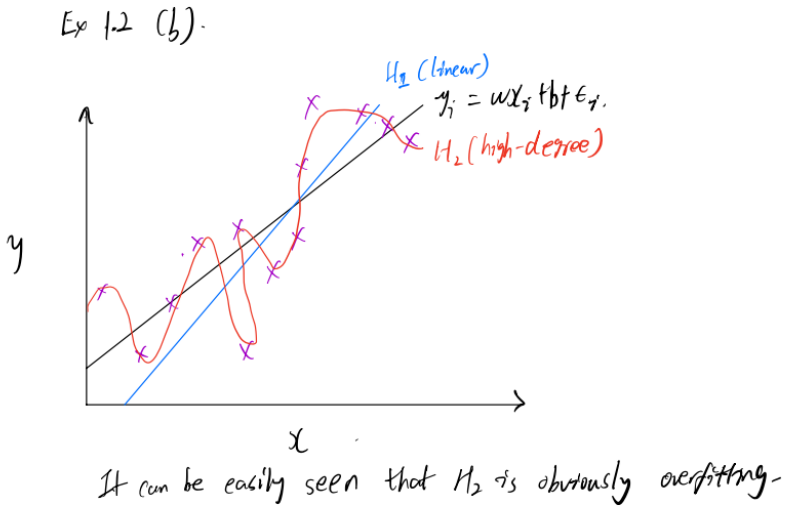


Figure 1: (b) sketch

(c)

k-fold cross validation helps us compare models by checking how well they perform on data they have not seen. By spliting the data into k parts, train on some, and test on the rest, then repeat this k times, it gives a better idea of how the model will work on new data. It's useful because it gives a more accurate estimate of the model's true error $R(f)$, and helps avoid choosing a model that overfits the training data.

Exercise 1.3

(a)

The misclassification loss can be reformulated as:

$$R(f) = \mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{y|\boldsymbol{x}}\left[\mathbb{1}_{\{f(\boldsymbol{x})\neq y\}}\right]\right].$$

Note that the inner expectation is:

$$\mathbb{E}_{y|\boldsymbol{x}}\left[\mathbb{1}_{\{f(\boldsymbol{x})\neq y\}}\right] = 1 - P(y = f(\boldsymbol{x})|\boldsymbol{x}).$$

Minimizing the misclassification loss is equivalent to maximizing $P(y|\boldsymbol{x})$ for given $\boldsymbol{x}$ w.r.t. y (MAP of y):

$$\hat{y}_{\text{MAP}} = \arg\max_{y\in\{0,1\}} = P(y|\boldsymbol{x}).$$

(b)

We have $R(\boldsymbol{f}) = \mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[\|\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x})\|_2^2]\right]$ for the Linear Regression problem. Our task is to find the $\boldsymbol{f}(\boldsymbol{x})$ that minimizes $R(\boldsymbol{f})$.

Let $\frac{\partial}{\partial \boldsymbol{f}}\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[\|\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x})\|_2^2] = 0$, we can solve for that optimal regression function:

$$\boldsymbol{f}^*(\boldsymbol{x}) = \mathbb{E}[\boldsymbol{y}|\boldsymbol{x}].$$

Exercise 1.4

(a)

Substitute X_i by $\mathcal{L}(\boldsymbol{y}_i, \boldsymbol{f}(\boldsymbol{x}_i))$, note that $R_{\text{emp}}(\boldsymbol{f}|\ \mathcal{D}) = \frac{1}{N}\sum_{i=1}^N X_i$ and $R(\boldsymbol{f}) = \mathbb{E}[X_i]$.

According to Hoeffding's inequality:

$$\Pr\left(\left|\frac{1}{N}\sum_{i=1}^N X_i - \mathbb{E}[X_i]\right| \geq \varepsilon\right) \leq 2\exp\left(-\frac{2N\varepsilon^2}{M^2}\right).$$

Thus,

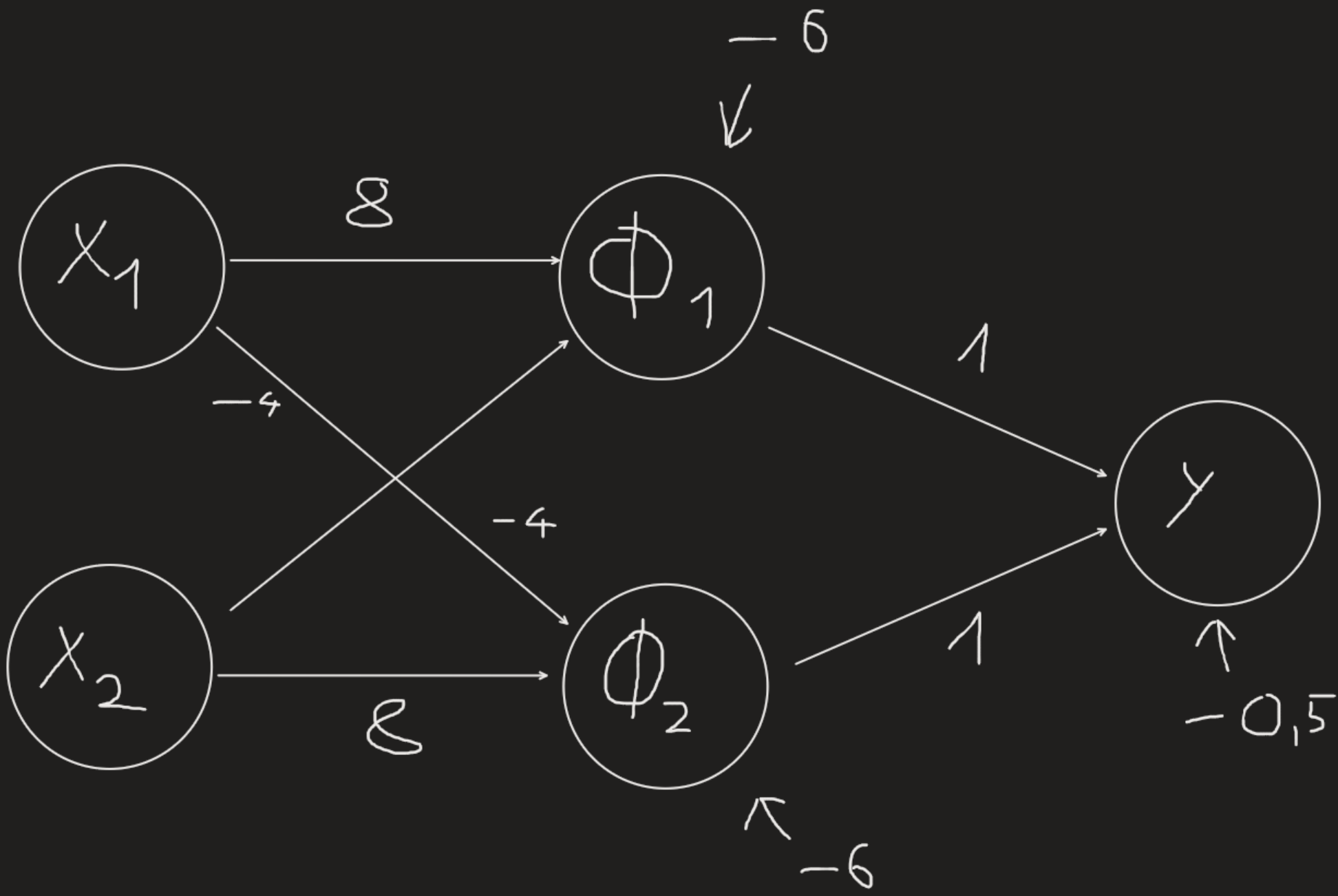
$$\Pr\left(\left|\frac{1}{N}\sum_{i=1}^N X_i - \mathbb{E}[X_i]\right| \leq \varepsilon\right) \geq 1 - 2\exp\left(-\frac{2N\varepsilon^2}{M^2}\right).$$

Let $\varepsilon = \sqrt{\frac{M^2\ln(2/\delta)}{2N}}$, we can show that:

$$\Pr\left(\left|\frac{1}{N}\sum_{i=1}^N X_i - \mathbb{E}[X_i]\right| \leq \sqrt{\frac{M^2\ln(2/\delta)}{2N}}\right) \geq 1 - \delta.$$

(b)

For finite number of bounded training samples, it is always possible for ERM to approximate the correct expectation value, if the size of the training set is big enough. However, the decreasing speed of the absolute value maybe relatively slow (of order $O(1/\sqrt{n})$).


− 6
8
x_1
ϕ_1
1
−4
−4
y
x_2
ϕ_2
1
8
−0,5
−6

mle25_sheet03

May 24, 2025

1 Machine Learning Essentials SS25 - Exercise Sheet 3

1.1 Instructions

- `TODO`'s indicate where you need to complete the implementations.
- You may use external resources, but write your own solutions.
- Provide concise, but comprehensible comments to explain what your code does.
- Code that's unnecessarily extensive and/or not well commented will not be scored.

1.2 Exercise 2

1.3 Task 2

[9]:
```
import numpy as np
import matplotlib.pyplot as plt

# TODO: Define network parameters
W1 = np.array([[8, -4],
               [-4, 8]])
b1 = np.array([-6, -6])
w2 = np.array([1, 1])
b2 = -0.5

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

class XORNetwork:
    def __init__(self, W1, b1, w2, b2):
        self.W1 = W1
        self.b1 = b1
        self.w2 = w2
        self.b2 = b2

    # TODO: Implement the forward pass (& activation) of the two-layer network
    # Define sigmoid activation function
    def forward(self, x):
        """
        Forward pass through the network. Returns hidden layer activations and␣
 ↪output.
```

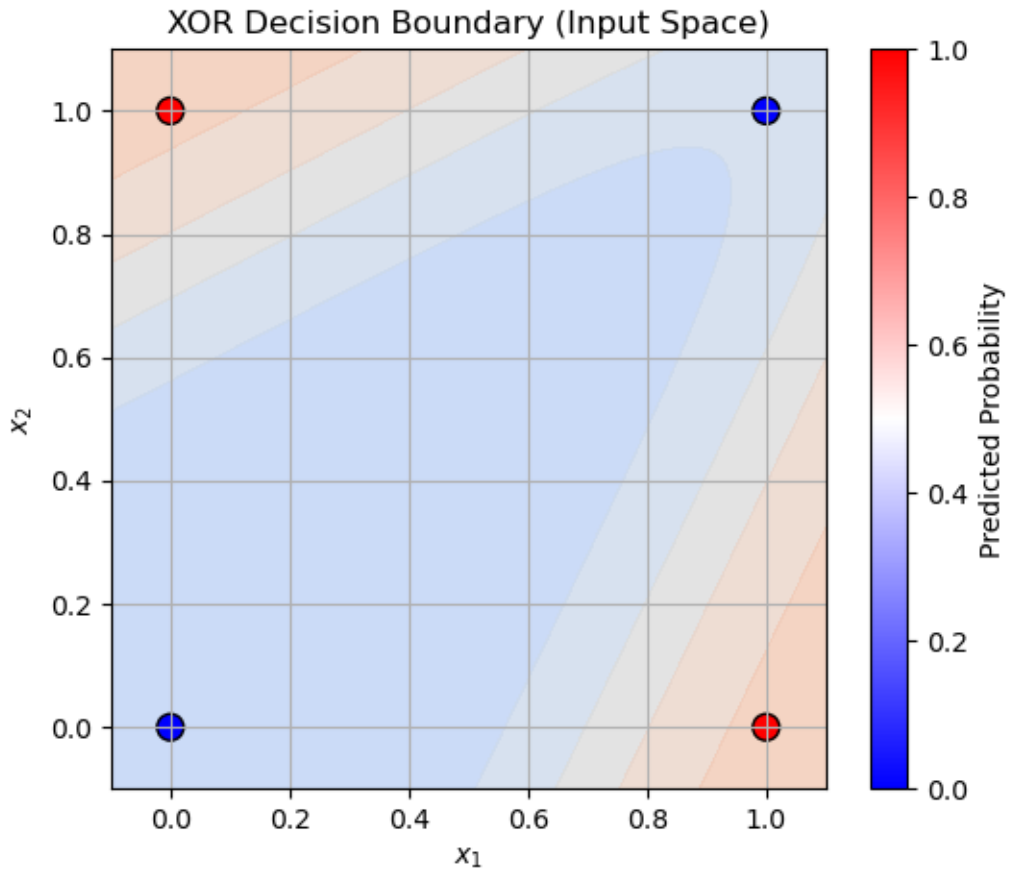
```
        """
        z1 = self.W1 @ x + self.b1
        phi = sigmoid(z1)
        z2 = np.dot(self.w2, phi) + self.b2
        output = sigmoid(z2)
        return phi, output
```

1.3.1 Task 3

[11]:
```
# TODO: Create XOR dataset, compute outputs & visualize decision boundary
net = XORNetwork(W1, b1, w2, b2)


inputs = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])
true_outputs = np.array([0, 1, 1, 0])


computed_outputs = []
hidden_activations = []

for x in inputs:
    phi, out = net.forward(x)
    computed_outputs.append(out)
    hidden_activations.append(phi)

computed_outputs = np.array(computed_outputs)
hidden_activations = np.array(hidden_activations)

for i, x in enumerate(inputs):
    print(f"Input: {x}, Network Output: {computed_outputs[i]:.3f}, True Output:␣
 ↪{true_outputs[i]}")

x1_vals = np.linspace(-0.1, 1.1, 200)
x2_vals = np.linspace(-0.1, 1.1, 200)
xx1, xx2 = np.meshgrid(x1_vals, x2_vals)
grid = np.c_[xx1.ravel(), xx2.ravel()]

Z = np.array([net.forward(x)[1] for x in grid])
Z = Z.reshape(xx1.shape)

# Plot
plt.figure(figsize=(6, 5))
plt.contourf(xx1, xx2, Z, levels=np.linspace(0, 1, 20), cmap="coolwarm",␣
 ↪alpha=0.8)
```

```
plt.scatter(inputs[:, 0], inputs[:, 1], c=true_outputs, cmap="bwr",␣
 ↪edgecolors='k', s=100)
plt.title("XOR Decision Boundary (Input Space)")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.grid(True)
plt.colorbar(label="Predicted Probability")
plt.show()
```

```
Input: [0 0], Network Output: 0.379, True Output: 0
Input: [0 1], Network Output: 0.594, True Output: 1
Input: [1 0], Network Output: 0.594, True Output: 1
Input: [1 1], Network Output: 0.435, True Output: 0
```



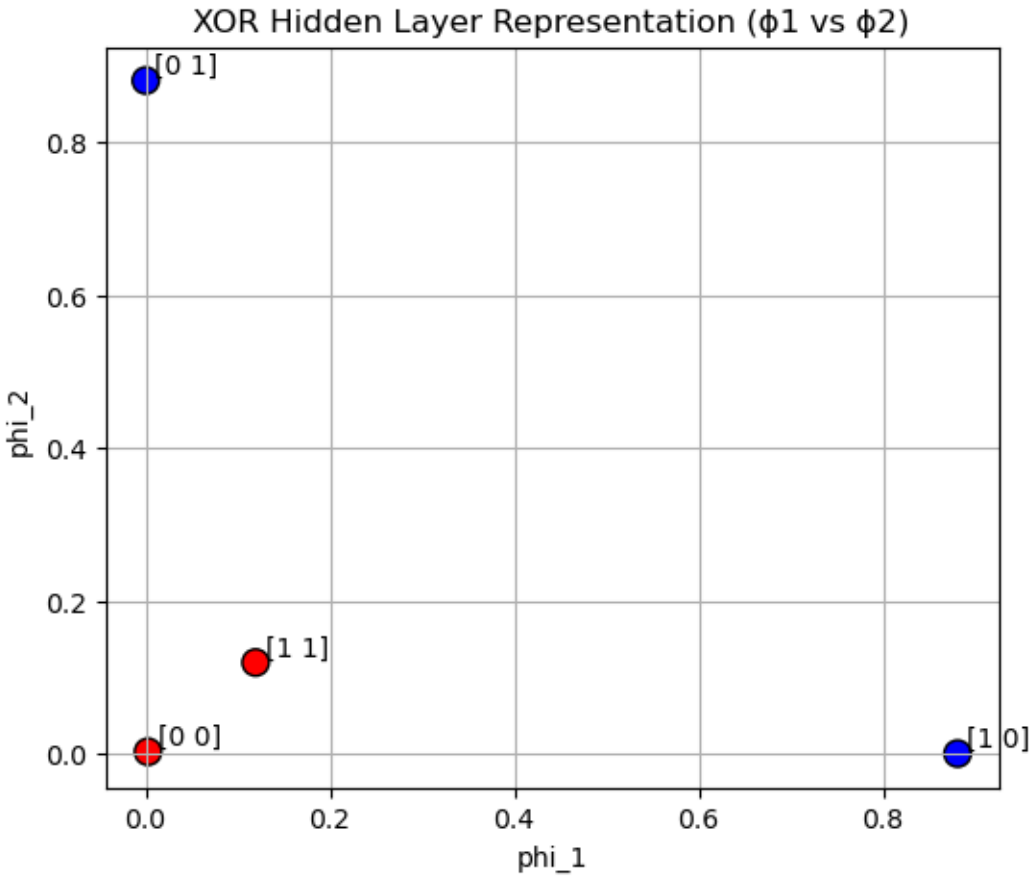
1.3.2 Task 5

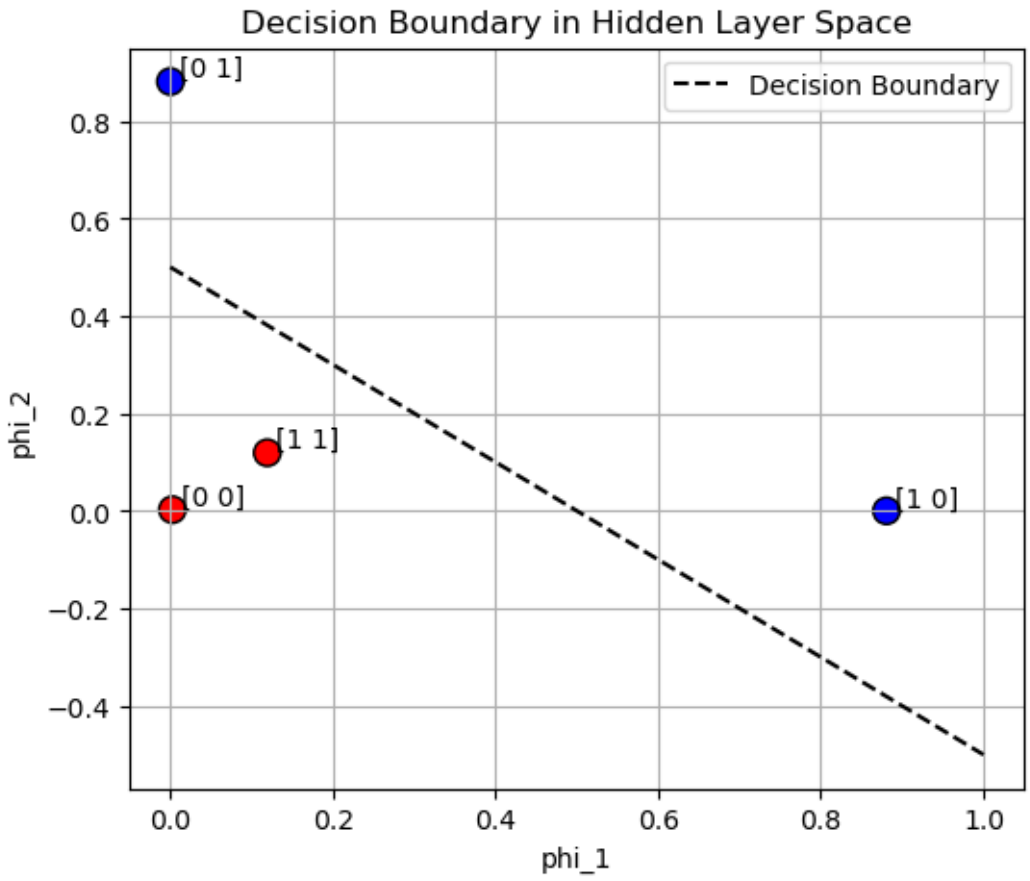
[16]:
```
# TODO: (a) Compute hidden layer activations
hidden_activations = []
```

```
for x in inputs:
    phi, _ = net.forward(x)
    hidden_activations.append(phi)

hidden_activations = np.array(hidden_activations)
print("Hidden Layer Activations ( 1,  2) for XOR Inputs:")
for i, phi in enumerate(hidden_activations):
    print(f"Input {inputs[i]} →   = {phi}")

# TODO: (b) Create scatter plot in the (phi1, phi2) plane, coloring points by␣
 ↪their label
plt.figure(figsize=(6, 5))
colors = ['red' if y == 0 else 'blue' for y in true_outputs]

plt.scatter(hidden_activations[:, 0], hidden_activations[:, 1],
            c=colors, edgecolors='k', s=100)

for i, txt in enumerate(inputs):
    plt.annotate(f"{txt}", (hidden_activations[i, 0] + 0.01,␣
 ↪hidden_activations[i, 1] + 0.01))

plt.xlabel("phi_1")
plt.ylabel("phi_2")
plt.title("XOR Hidden Layer Representation ( 1 vs  2)")
plt.grid(True)
plt.show()

# TODO: (c) Draw the output layer's decision boundary in the (phi1, phi2) space
plt.figure(figsize=(6, 5))
plt.scatter(hidden_activations[:, 0], hidden_activations[:, 1],
            c=colors, edgecolors='k', s=100)

# Add labels
for i, txt in enumerate(inputs):
    plt.annotate(f"{txt}", (hidden_activations[i, 0] + 0.01,␣
 ↪hidden_activations[i, 1] + 0.01))

# Decision boundary:  1 +  2 = 0.5
x_vals = np.linspace(0, 1, 100)
y_vals = 0.5 - x_vals
plt.plot(x_vals, y_vals, 'k--', label='Decision Boundary')

plt.xlabel("phi_1")
plt.ylabel("phi_2")
plt.title("Decision Boundary in Hidden Layer Space")
plt.grid(True)
plt.legend()
```

```
plt.show()
```

```
Hidden Layer Activations ( 1,  2) for XOR Inputs:
Input [0 0] →   = [0.00247262 0.00247262]
Input [0 1] →   = [4.53978687e-05 8.80797078e-01]
Input [1 0] →   = [8.80797078e-01 4.53978687e-05]
Input [1 1] →   = [0.11920292 0.11920292]
```





TODO: (d) Briefly discuss the result

The original 2D space is not linearly seperable, but after the non linear transformation using the sigmoid function the points become linearly separable, which can be seen using the visualization. Each hidden neuron calculates a weighted sum based on the inputs and then applies the sigmoid function. This bends the input space in a non linear way, so that the output neuron is able to draw a straight line through the data.

2.4.

If the first Layer is $\tilde{z}^{(1)} = W^{(1)}x + b^{(1)}$ for $\tilde{z}^{(0)} = x$

than the second layer is:

$$\tilde{z}^{(2)} = W^{(2)}(W^{(1)}x + b^{(1)}) + b^2$$

$$= \underbrace{W^{(2)}W^{(1)}}_{W^*}x + \underbrace{W^{(2)}b^{(1)} + b^2}_{b^* \text{ (some bias)}}$$

which has the same form as the first layer

$$W^*x + b^*$$

If all functions in the network are linear, the whole network collapses into a single linear model and a linear model cannot separate the XOR-problem