Sheet 5

Julian Kusch, Jan Reckermann, Tim Weinreich

In [1]:
```
import os
import pandas as pd
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
```

1 The logistic sigmoid

a)

In [2]:
```
x = sp.symbols('x')
sigmoid = 1 / (1 + sp.exp(-x))
sigmoid_prime = sigmoid.diff(x)
sigmoid_prime
```

Out[2]:

$$\frac{e^{-x}}{\left(1 + e^{-x}\right)^2}$$

b)

In [3]:
```
x = sp.symbols('x')
tanh = (sp.exp(x) - sp.exp(-x)) / (sp.exp(x) + sp.exp(-x))
tanh_prime = sp.simplify(tanh.diff(x))
tanh_prime
```

Out[3]:

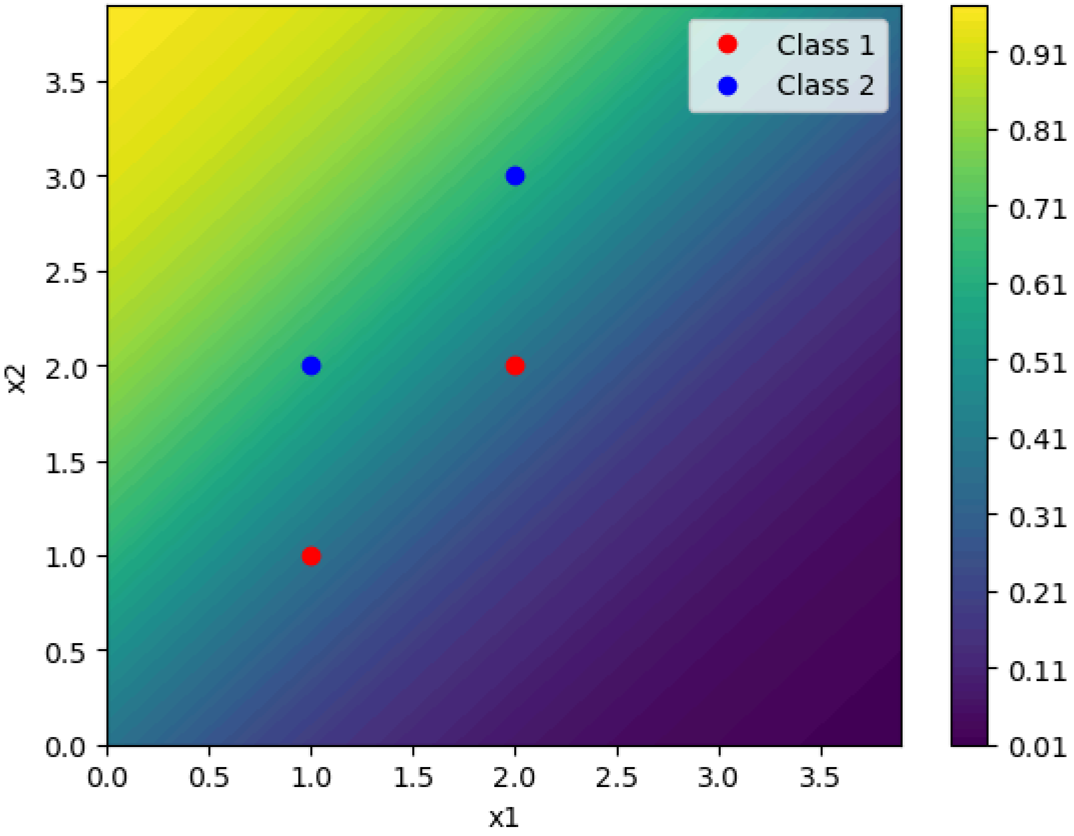
$$\frac{4e^{2x}}{e^{4x} + 2e^{2x} + 1}$$

with $2x = -x'$ and dropping the const $C = 4$ the two derivatives are the same. That shows that $\tanh(x)$ is a shifted and scaled version of $\sigma(x)$.

c)

In [4]:
```
x1 = np.array([1, 2])
x2 = np.array([1, 2])

y1 = np.array([1, 2])
y2 = np.array([2, 3])

def sigmoid(x1, x2):
    return 1 / (1 + np.exp(1*x1 - 1*x2 + 1/2))  # 1/2 is the bias and w1 = 1 and w2 = -1

a1 = np.arange(0,4,0.1)
a2 = np.arange(0,4,0.1)

values = np.zeros((len(a1), len(a2)))

for i,a in enumerate(a1):
    for j,b in enumerate(a2):
        values[i,j] = sigmoid(a, b)

plt.contourf(a1, a2, values.T, levels=100)
plt.colorbar()
plt.plot(x1, y1, 'ro', label='Class 1')
plt.plot(x2, y2, 'bo', label='Class 2')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
```

Out[4]:
```
<matplotlib.legend.Legend at 0x21f083923a0>
```



with $b = 1/2$ and $w^T = (1, -1)$ the activation seperates the two classes.

2 Logistic regression: an LLM lie detector

Download the data from https://heibox.uni-heidelberg.de/f/38bd3f2a9b7944248cc2/
Unzip it and place the lie_detection folder in the folder named `data` to get the following structure: "data/lie_detection/datasets" and "data/lie_detection/acts".

This is how you can load a dataset of LLM activations. Use a new Datamanager if you want to have a new dataset. Use the same data manager if you want to combine datasets.

In [5]:
```python
from lie_detection_utils import DataManager

path_to_datasets = "data/lie_detection/datasets"
path_to_acts = "data/lie_detection/acts"

# check if the datasets and activations are available
assert os.path.exists(path_to_datasets), "The path to the datasets does not exist."
assert os.path.exists(path_to_acts), "The path to the activations does not exist."

# these are the different datasets containing true and false factual statements about different topics
dataset_names = ["cities", "neg_cities", "sp_en_trans", "neg_sp_en_trans"]
dataset_name = dataset_names[0] # choose some dataset from the above datasets, index "0" loads the "cities" dataset for exampl

# the dataloader automatically loads the training data for us
dm = DataManager()
dm.add_dataset(dataset_name, "Llama3", "8B", "chat", layer=12, split=0.8, center=False,
                device='cpu', path_to_datasets=path_to_datasets, path_to_acts=path_to_acts)
acts_train, labels_train = dm.get('train') # train set
acts_test, labels_test = dm.get('val')
print(acts_train.shape, labels_train.shape)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[5], line 7
      4 path_to_acts = "data/lie_detection/acts"
      6 # check if the datasets and activations are available
----> 7 assert os.path.exists(path_to_datasets), "The path to the datasets does not exist."
      8 assert os.path.exists(path_to_acts), "The path to the activations does not exist."
     10 # these are the different datasets containing true and false factual statements about different topics

AssertionError: The path to the datasets does not exist.
```

In [ ]:
```python
# have a look at the statements that were fed to the LLM to produce the activations:
df = pd.read_csv(f"{path_to_datasets}/{dataset_name}.csv")
print(df.head(10))
```

3 Log-sum-exp and soft(arg)max

(a)

In [6]:
```
lambda_ = 1
sigma1 = np.array([1, 2, 3])
sigma2 = np.array([11, 12, 13])
sigma3 = np.array([10, 20, 30])

def softmax(x, l):
    return np.exp(l*x) / np.sum(np.exp(l*x), axis=0)

print(softmax(sigma1, lambda_))
print(softmax(sigma2, lambda_))
print(softmax(sigma3, lambda_))
```

```
[0.09003057 0.24472847 0.66524096]
[0.09003057 0.24472847 0.66524096]
[2.06106005e-09 4.53978686e-05 9.99954600e-01]
```

σ_1 and σ_2 yield identical results.

Invariance under constant offsets:

Let $\sigma' = \sigma + c\mathbf{1}$ (adding c to all entries). Then:

$$\exp(\lambda\sigma'_k) = \exp(\lambda(\sigma_k + c)) = \exp(\lambda c)\exp(\lambda\sigma_k).$$

The normalization factor becomes:

$$\sum_{j=1}^{K}\exp(\lambda\sigma'_j) = \sum_{j=1}^{K}\exp(\lambda c)\exp(\lambda\sigma_j) = \exp(\lambda c)\sum_{j=1}^{K}\exp(\lambda\sigma_j).$$

Thus:

$$\frac{\exp(\lambda\sigma'_k)}{\sum_{j=1}^{K}\exp(\lambda\sigma'_j)} = \frac{\exp(\lambda\sigma_k)}{\sum_{j=1}^{K}\exp(\lambda\sigma_j)}.$$

Soft(arg)max is invariant under constant offsets.

Invariance under rescaling:

Let $\sigma' = a\sigma$, where $a > 0$. Then:

$$\exp(\lambda \sigma'_k) = \exp(\lambda(a\sigma_k)).$$

The normalization factor becomes:

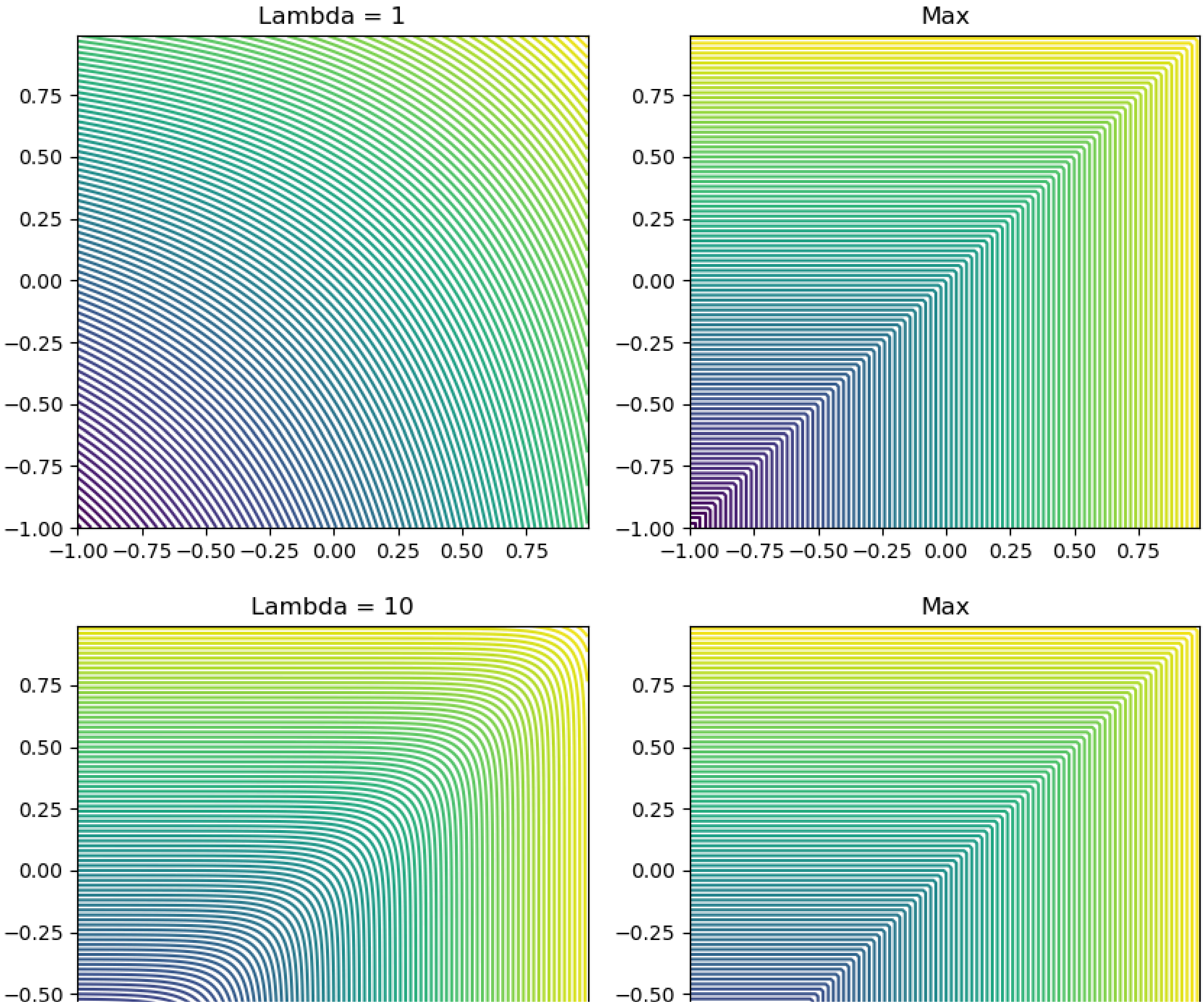
$$\sum_{j=1}^{K} \exp(\lambda \sigma'_j) = \sum_{j=1}^{K} \exp(\lambda(a\sigma_j)).$$

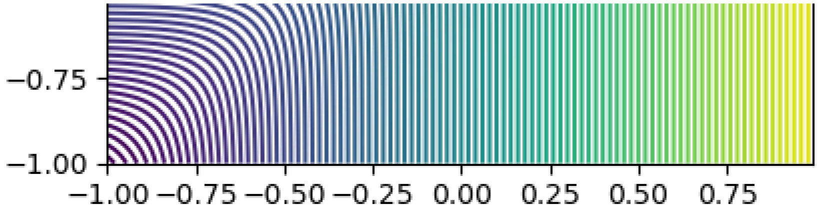
For general λ and a, the resulting probabilities will not remain unchanged. Hence, soft(arg)max is not invariant under rescaling.

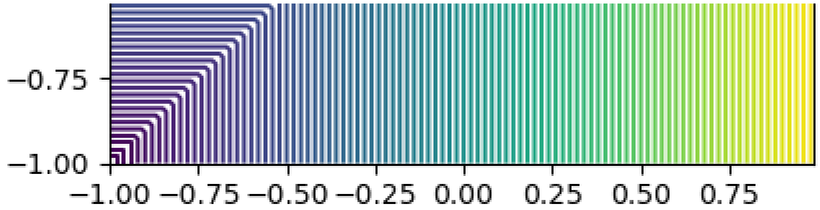
(b)

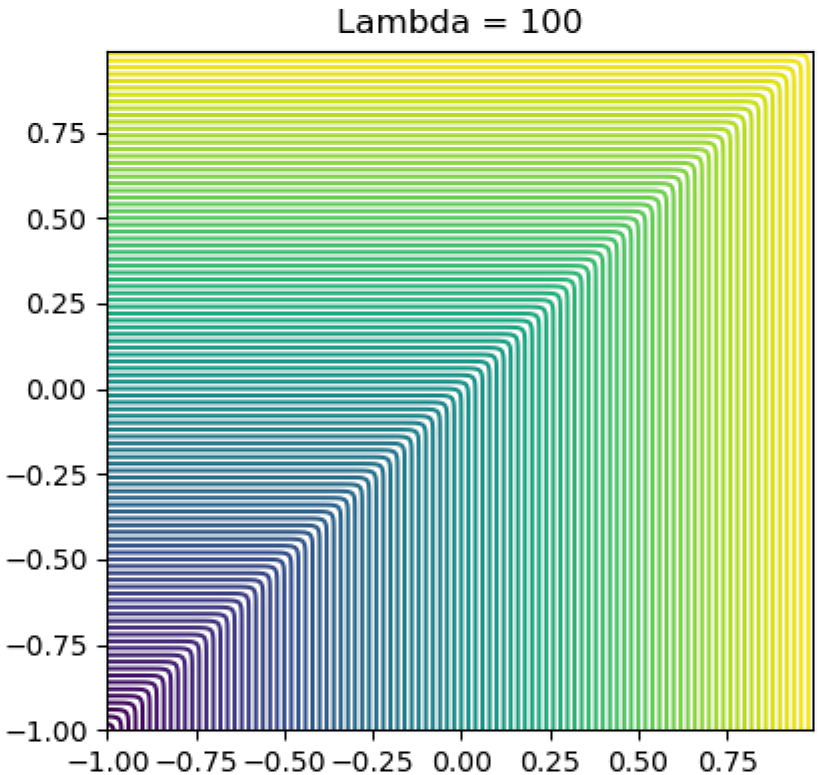
In [7]:
```
s1 = np.arange(-1, 1, 0.01)
s2 = np.arange(-1, 1, 0.01)

lambdas = [1, 10, 100]

def lse(x, l):
    return 1/l * np.log(np.sum(np.exp(l*x), axis=0))

values = np.zeros((len(s1), len(s2), len(lambdas)))
max_vals = np.zeros((len(s1), len(s2)))

for i,a in enumerate(s1):
    for j,b in enumerate(s2):
        max_vals[i,j] = np.max([a,b])
        for k,l in enumerate(lambdas):
            values[i,j,k] = lse(np.array([a, b]), l)

fig, ax = plt.subplots(3, 2, figsize=(10, 15))
for i in range(3):
    ax[i,0].contour(s1, s2, values[:,:,i].T, levels=100)
    ax[i,0].set_title(f"Lambda = {lambdas[i]}")
```

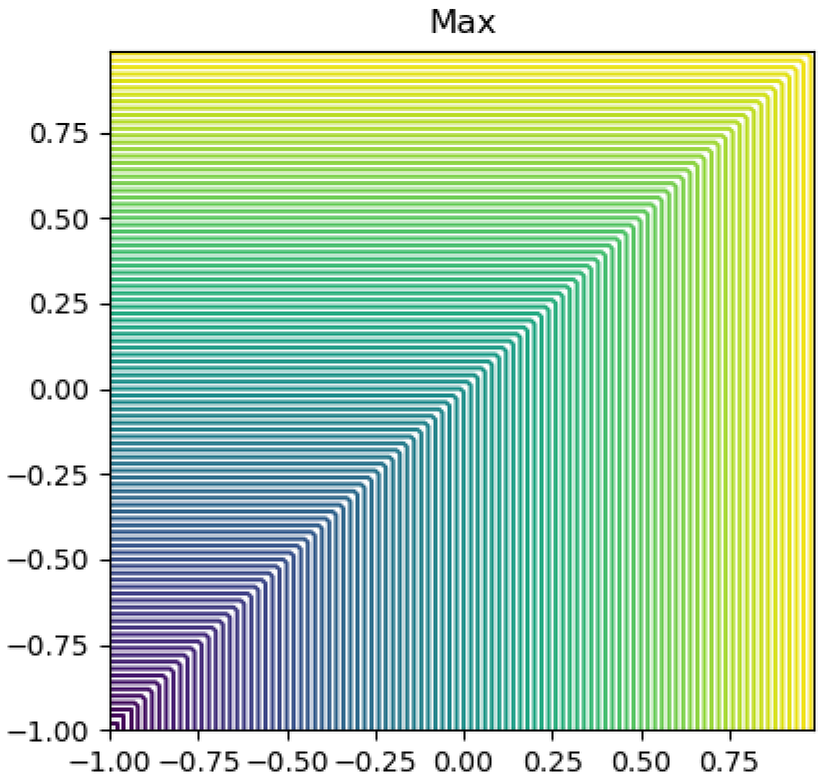
```
    ax[i,1].contour(s1, s2, max_vals.T, levels=100)
    ax[i,1].set_title("Max")
```


Lambda = 1
Max
Lambda = 10
Max





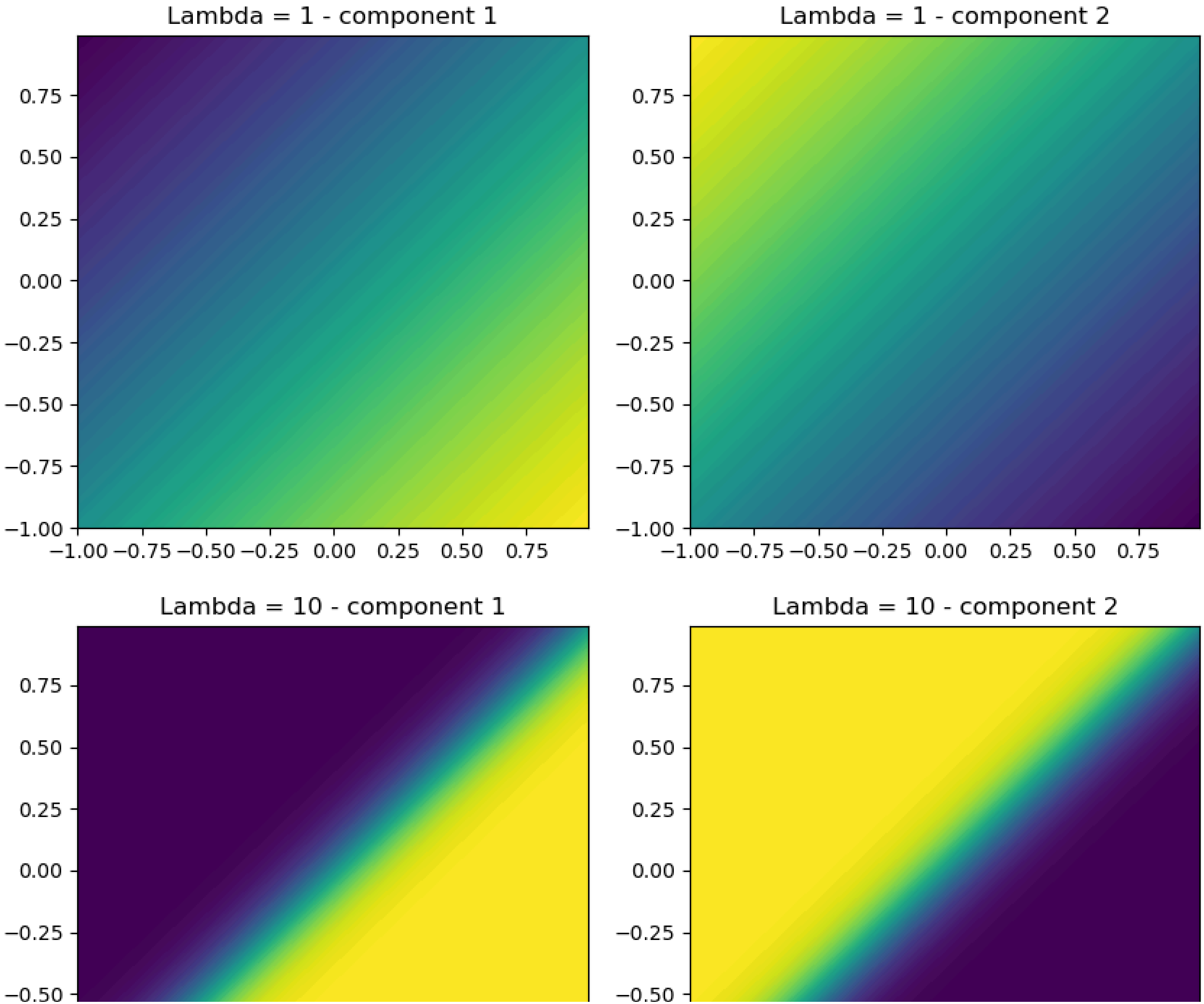


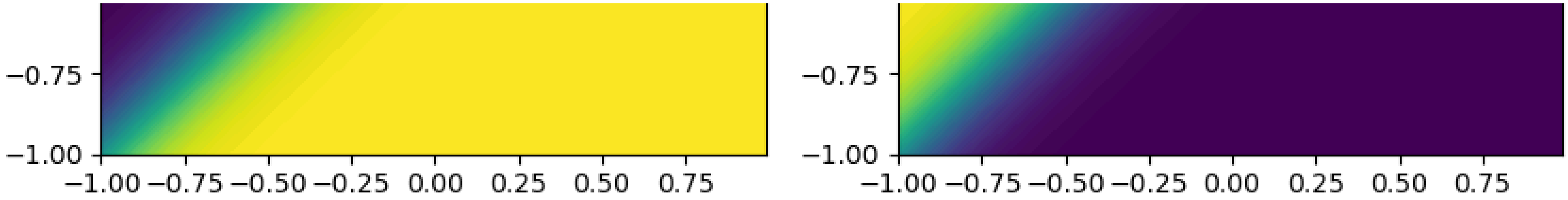


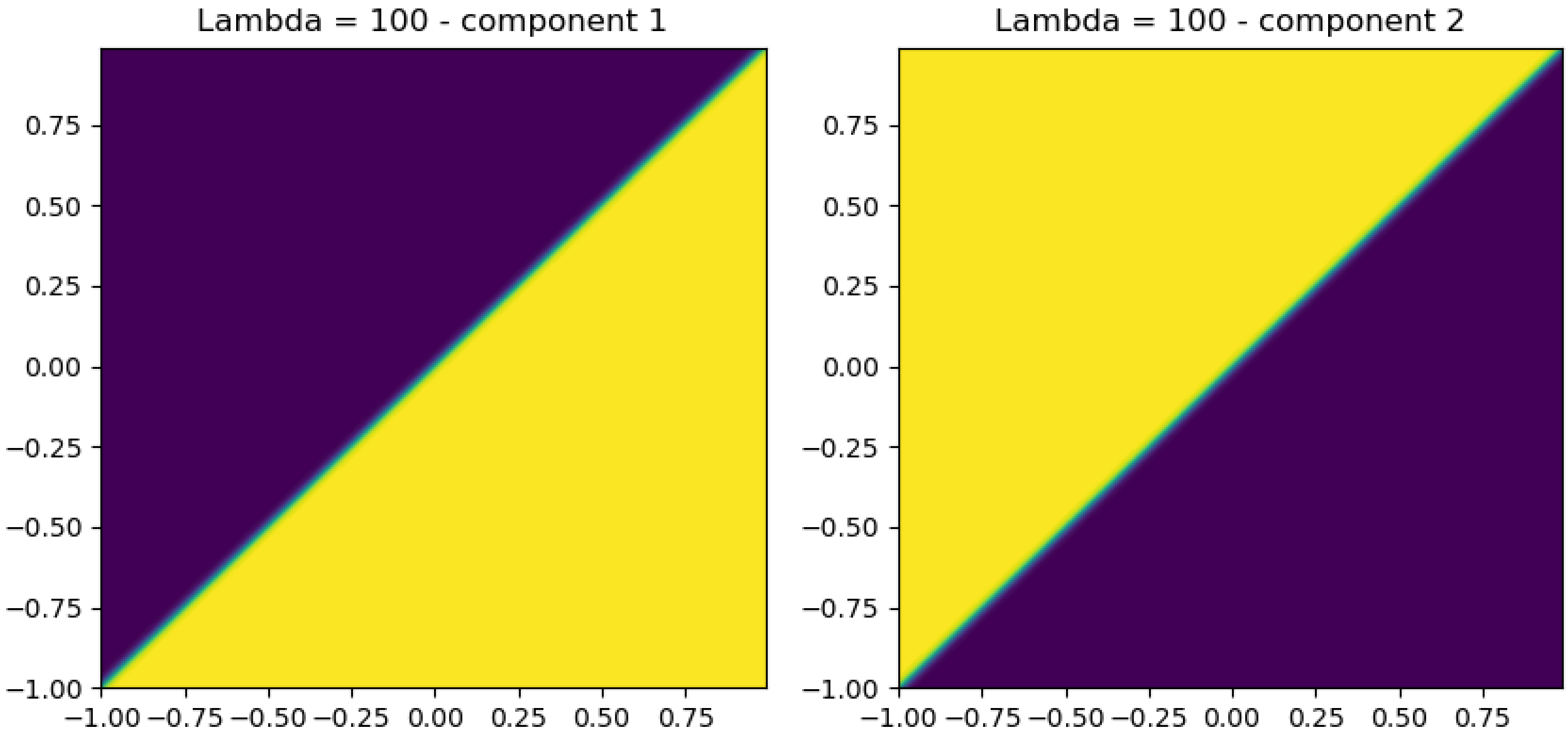
c)

In [9]:
```
values0 = np.zeros((len(s1), len(s2), len(lambdas)))
values1 = np.zeros((len(s1), len(s2), len(lambdas)))

for i,a in enumerate(s1):
    for j,b in enumerate(s2):
        for k,l in enumerate(lambdas):
```

```
            values0[i,j,k] = softmax(np.array([a, b]), l)[0]
            values1[i,j,k] = softmax(np.array([a, b]), l)[1]

fig, ax = plt.subplots(3, 2, figsize=(10, 15))
for i in range(3):
    ax[i,0].contourf(s1, s2, values0[:,:,i].T, levels=100)
    ax[i,0].set_title(f"Lambda = {lambdas[i]} - component 1")
    ax[i,1].contourf(s1, s2, values1[:,:,i].T, levels=100)
    ax[i,1].set_title(f"Lambda = {lambdas[i]} - component 2")
```

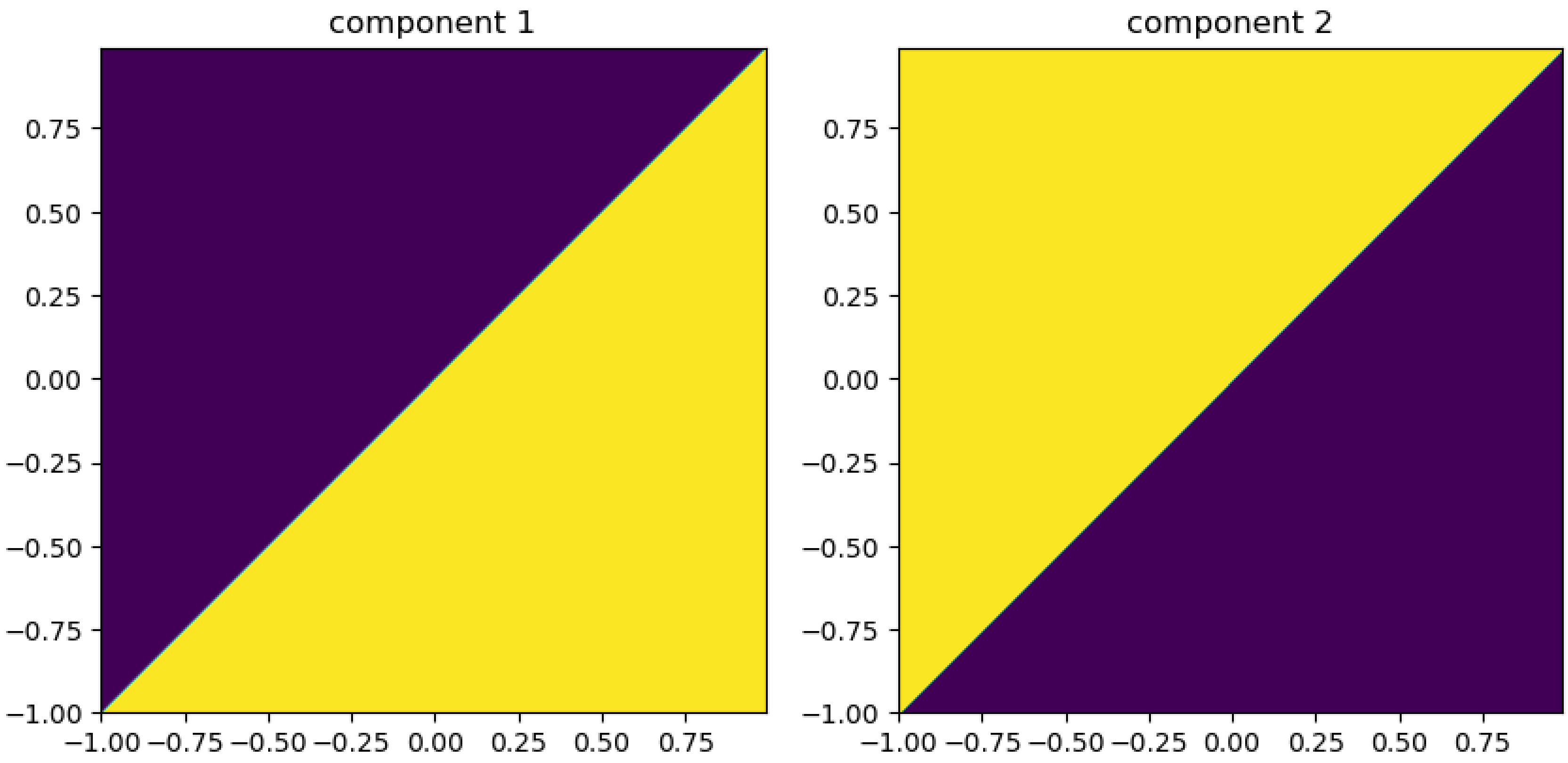

Lambda = 1 - component 1
Lambda = 1 - component 2
Lambda = 10 - component 1
Lambda = 10 - component 2
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75





In [10]:
```python
values0 = np.zeros((len(s1), len(s2)))
values1 = np.zeros((len(s1), len(s2)))

for i,a in enumerate(s1):
    for j,b in enumerate(s2):
        values0[i,j] = 1 if a > b else 0
        values1[i,j] = 0 if a > b else 1

fig, ax = plt.subplots(1, 2, figsize=(10, 4.7))
```

```
ax[0].contourf(s1, s2, values0.T, levels=100)
ax[0].set_title(f"component 1")
ax[1].contourf(s1, s2, values1.T, levels=100)
ax[1].set_title(f"component 2")
ax[0].set_aspect('equal')
ax[1].set_aspect('equal')
```



d)

$$\left(\nabla \frac{1}{\lambda} \log \sum_{j=1}^{k} e^{\lambda o_j}\right)_k = \frac{1}{\cancel{\lambda}} \frac{1}{\sum_{j=1}^{k} e^{\lambda o_j}} \cancel{\lambda}\, e^{\lambda o_k}$$

4 Linear regions of MLPs

a)

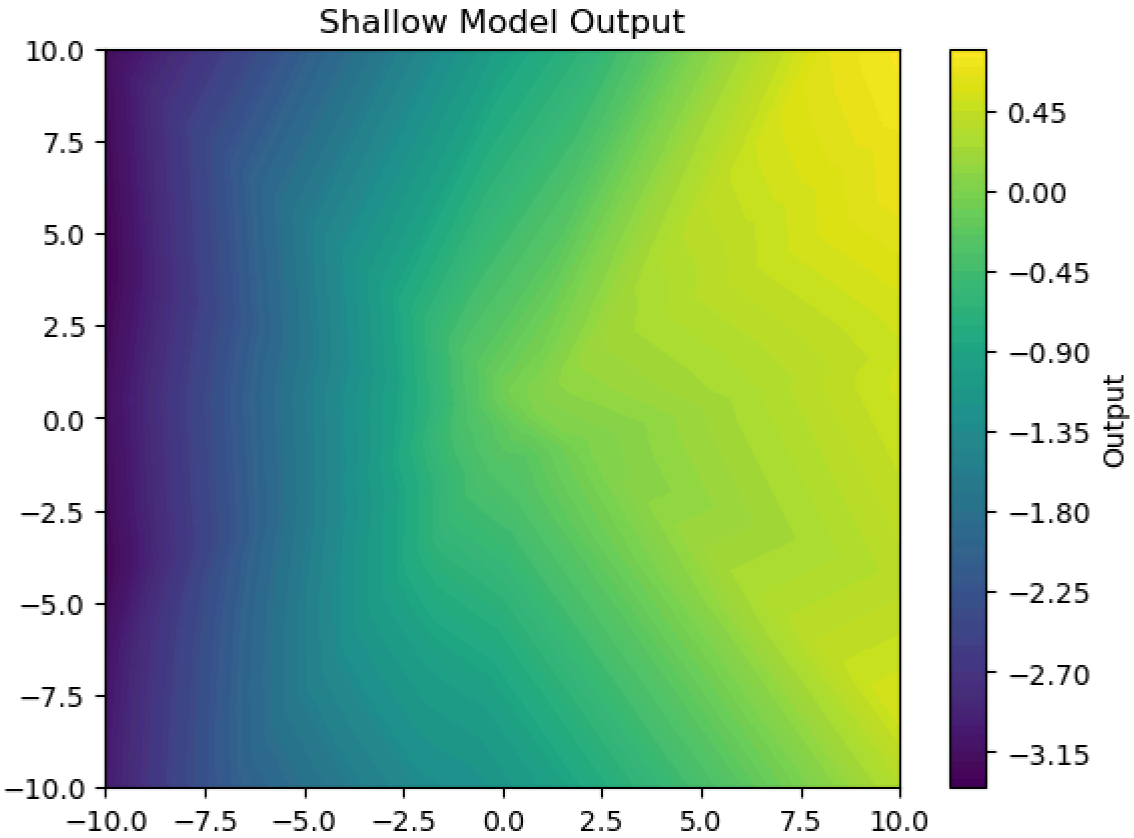
In [13]:
```
import torch
import torch.nn as nn

class ShallowMLP(nn.Module):
    def __init__(self):
        super(ShallowMLP, self).__init__()
        self.fc1 = nn.Linear(2, 20)  # Input to Hidden
        self.relu = nn.ReLU()       # Activation
        self.fc2 = nn.Linear(20, 1) # Hidden to Output

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Initialize the model
shallow_model = ShallowMLP()
print(f"Total parameters: {sum(p.numel() for p in shallow_model.parameters())}")
```

```
Total parameters: 81
```

b)

In [ ]:
```
n = 500

x = np.linspace(-10, 10, n)
y = np.linspace(-10, 10, n)

values = np.zeros((n, n))

for i in range(n):
    for j in range(n):
        values[i][j] = shallow_model.forward(torch.tensor([x[i],y[j]], dtype=torch.float32))
```

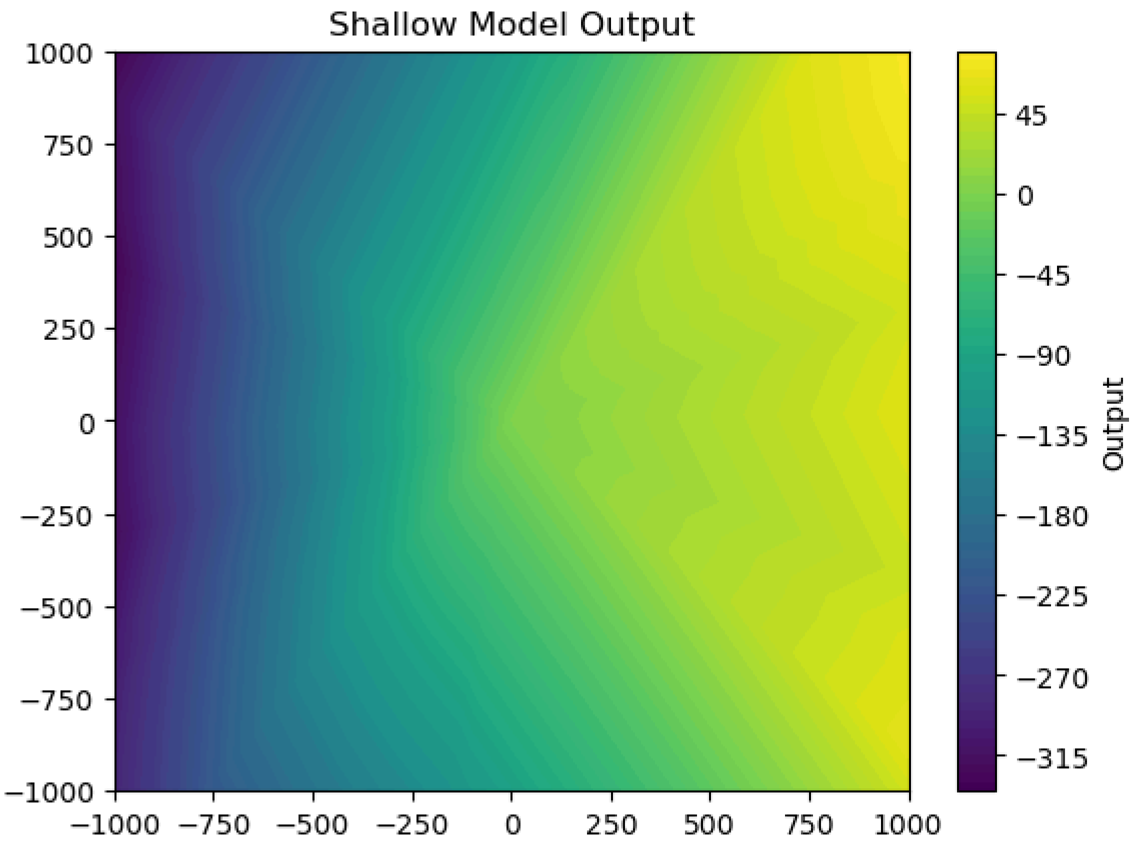
In [28]:
```
plt.contourf(x, y, values, levels=100)
plt.colorbar(label='Output')
plt.title("Shallow Model Output")
plt.show()
```



In [31]:
```
n = 500

x = np.linspace(-1000, 1000, n)
y = np.linspace(-1000, 1000, n)

values = np.zeros((n, n))

for i in range(n):
    for j in range(n):
        values[i][j] = shallow_model.forward(torch.tensor([x[i],y[j]], dtype=torch.float32))
```

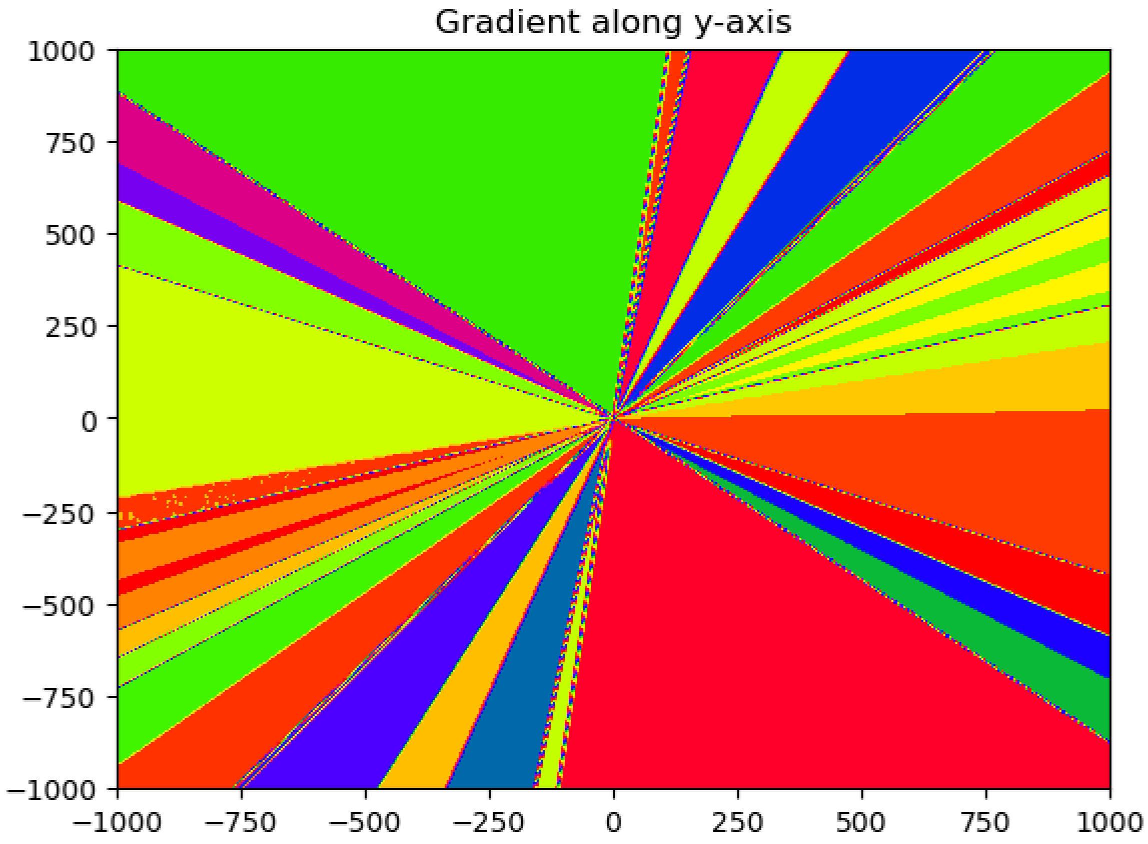
In [32]:
```
plt.contourf(x, y, values, levels=100)
plt.colorbar(label='Output')
```

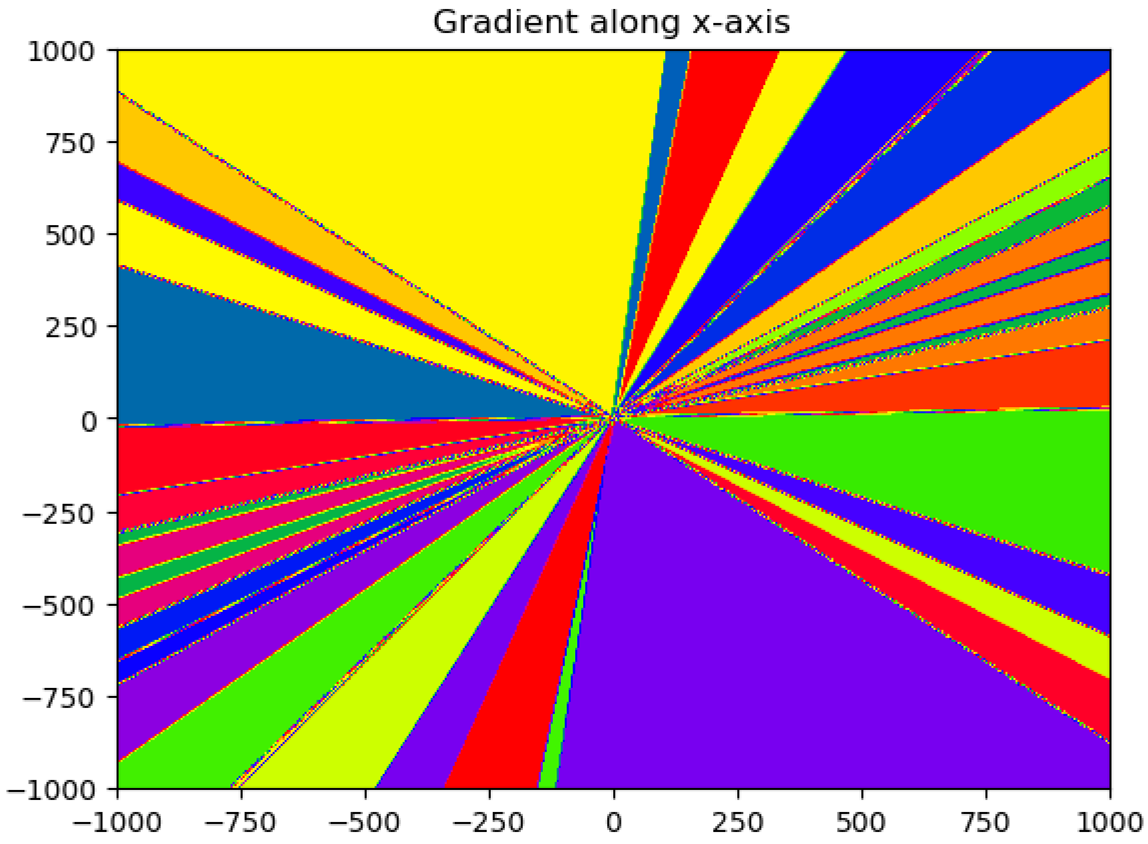
```
plt.title("Shallow Model Output")
plt.show()
```



c)

In [ ]:
```
x_grad, y_grad = np.gradient(values)

plt.contourf(x, y, y_grad, levels=100, cmap='prism')
plt.title("Gradient along y-axis")
plt.show()
plt.contourf(x, y, x_grad, levels=100, cmap='prism')
plt.title("Gradient along x-axis")
plt.show()
```


Gradient along y-axis
1000
750
500
250
0
−250
−500
−750
−1000
−1000
−750
−500
−250
0
250
500
750
1000



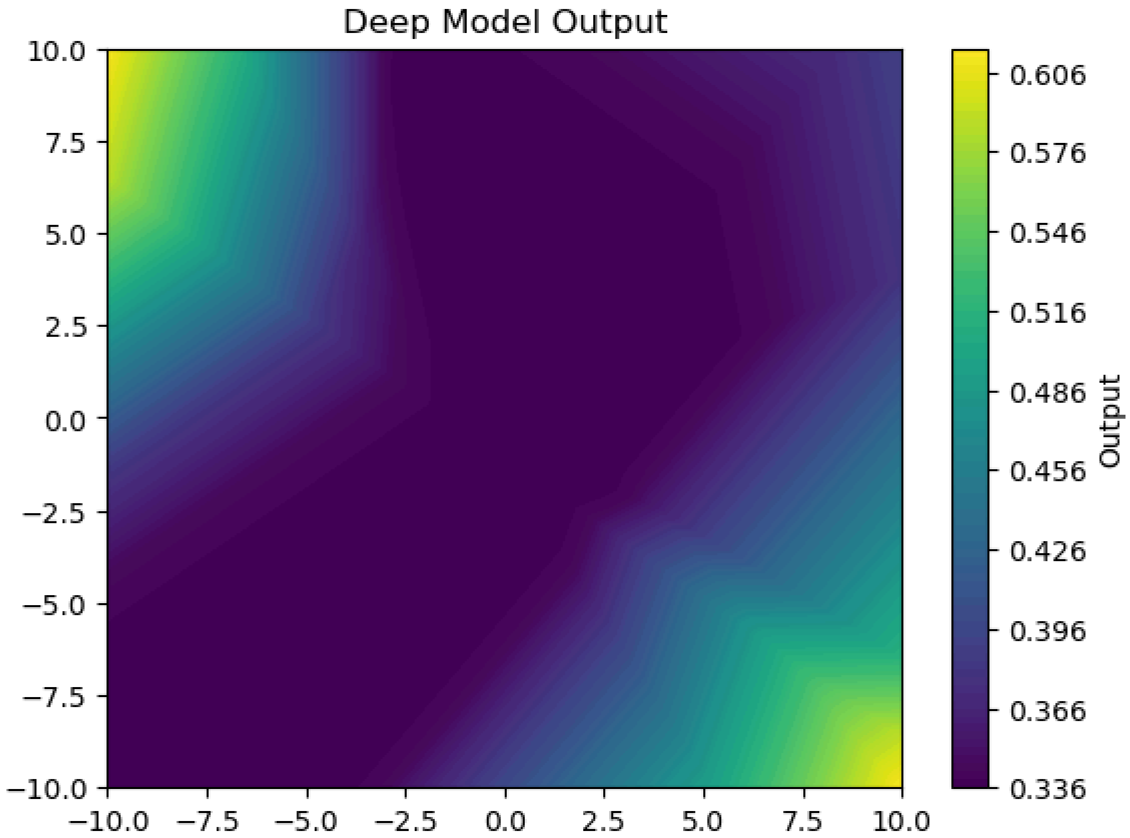
d)

In [45]:
```
class DeepMLP(nn.Module):
    def __init__(self):
        super(DeepMLP, self).__init__()
        self.fc1 = nn.Linear(2, 5)  # Input to Hidden
        self.fc2 = nn.Linear(5, 5)  # Hidden
        self.fc3 = nn.Linear(5, 5)  # Hidden
        self.fc4 = nn.Linear(5, 5)  # Hidden
        self.relu = nn.ReLU()       # Activation
        self.fc5 = nn.Linear(5, 1)  # Hidden to Output

    def forward(self, x):
```

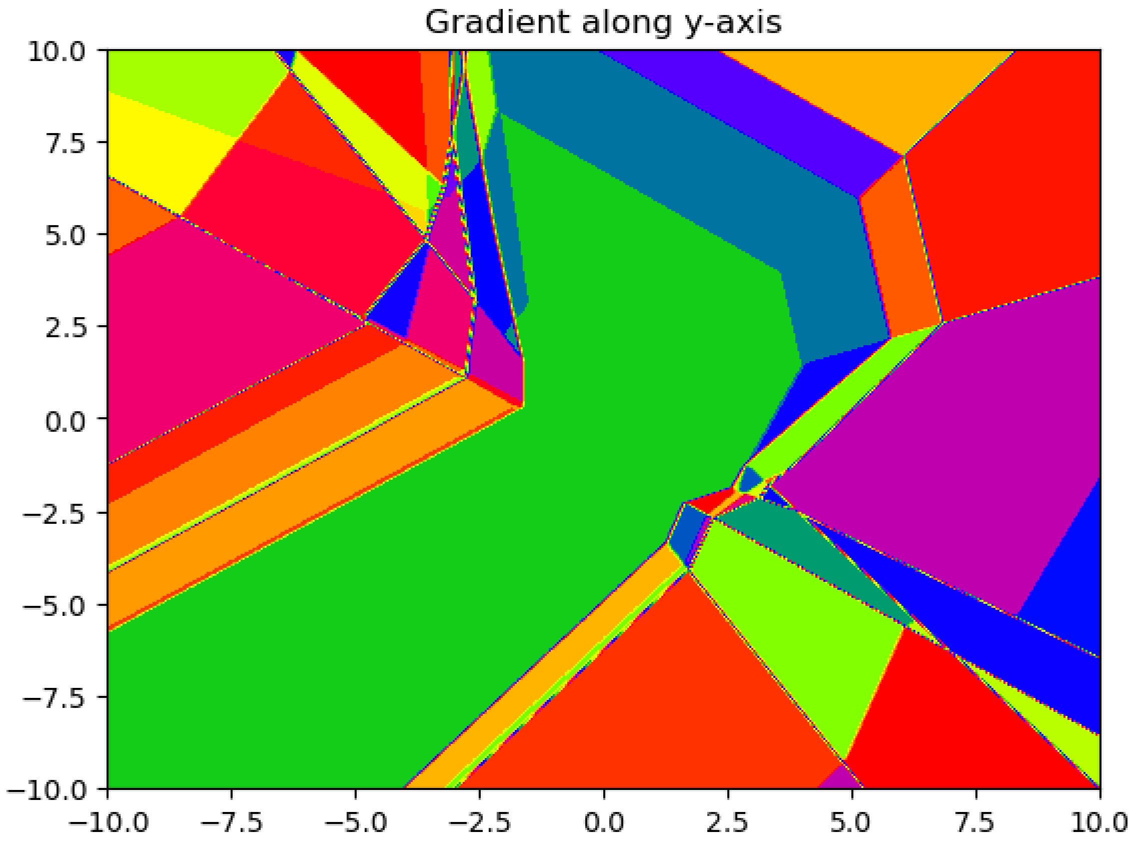
```
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        x = self.fc5(x)
        return x

# Initialize the model
deep_model = DeepMLP()
print(f"Total parameters: {sum(p.numel() for p in deep_model.parameters())}")
```

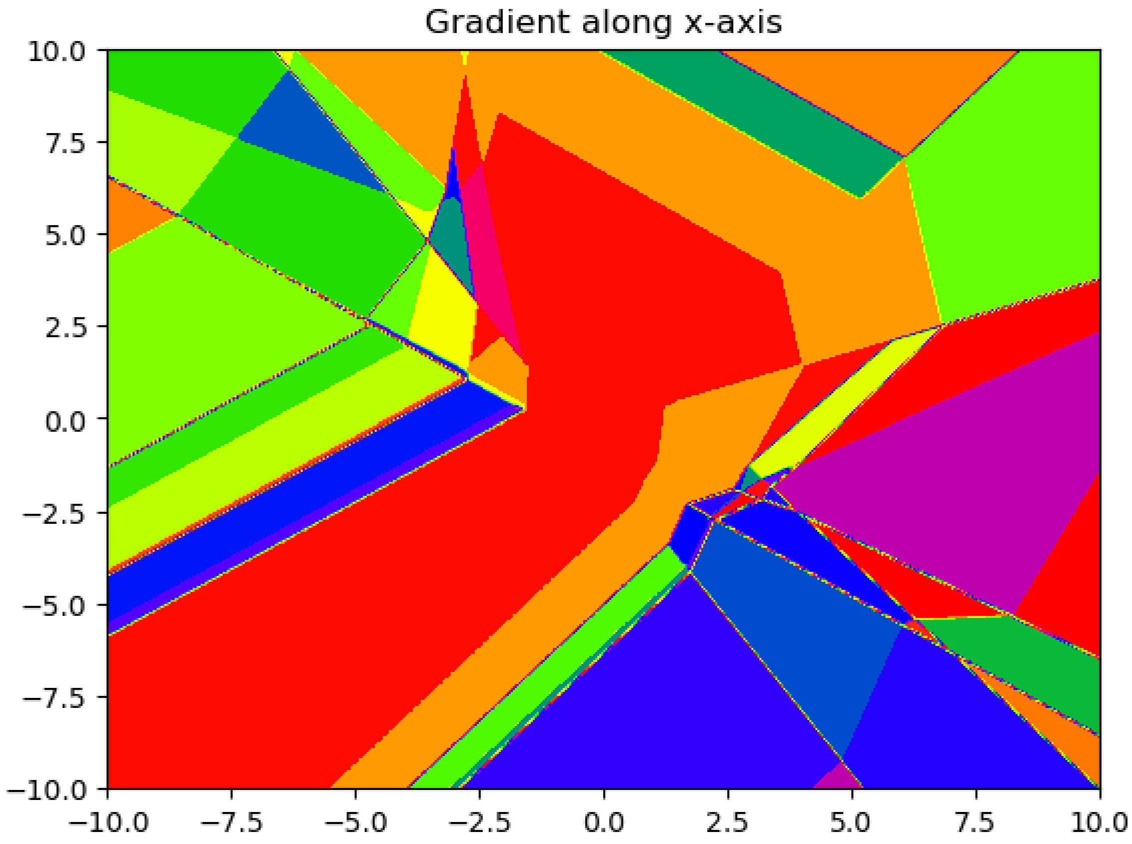
```
Total parameters: 111
```

In [46]:
```
n = 500

x = np.linspace(-10, 10, n)
y = np.linspace(-10, 10, n)

values = np.zeros((n, n))

for i in range(n):
    for j in range(n):
        values[i][j] = deep_model.forward(torch.tensor([x[i],y[j]], dtype=torch.float32))

plt.contourf(x, y, values, levels=100)
plt.colorbar(label='Output')
plt.title("Deep Model Output")
plt.show()
```



In [ ]:
```
x_grad, y_grad = np.gradient(values)

plt.contourf(x, y, y_grad, levels=100, cmap='prism')
plt.title("Gradient along y-axis")
plt.show()
plt.contourf(x, y, x_grad, levels=100, cmap='prism')
plt.title("Gradient along x-axis")
plt.show()
```


Gradient along y-axis
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Gradient along x-axis
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0