Deep Learning I

Mitko Veta

Eindhoven University of Technology
Department of Biomedical Engineering

2024

Learning goals

- **Computational graphs:** Understand and represent linear and logistic regression models as computational graphs.
- **From logistic regression to neural networks**: Understand how logistic regression can be expanded into neural networks.
- **Building blocks of neural networks**: Identify and describe the basic components of neural networks—including inputs, outputs, hidden layers, and activation functions.
- **Implementation:** Apply the gained knowledge by implementing a simple neural network using PyTorch.

Material

- Chapters 10.1 and 10.2 form "*An introduction to statistical learning with applications in Python*, G. James, D. Witten, T. Hastie, R. Tibshirani, J. Taylor"
- Practicals part 2 doubles as a reader for the material covered today!
- PyTorch documentation and basics tutorial, https://pytorch.org/tutorials/beginner/basics/intro.html

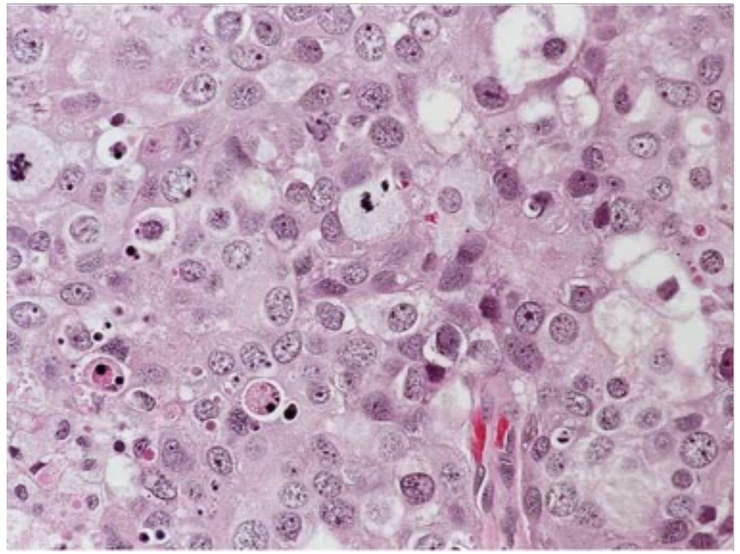
Overview

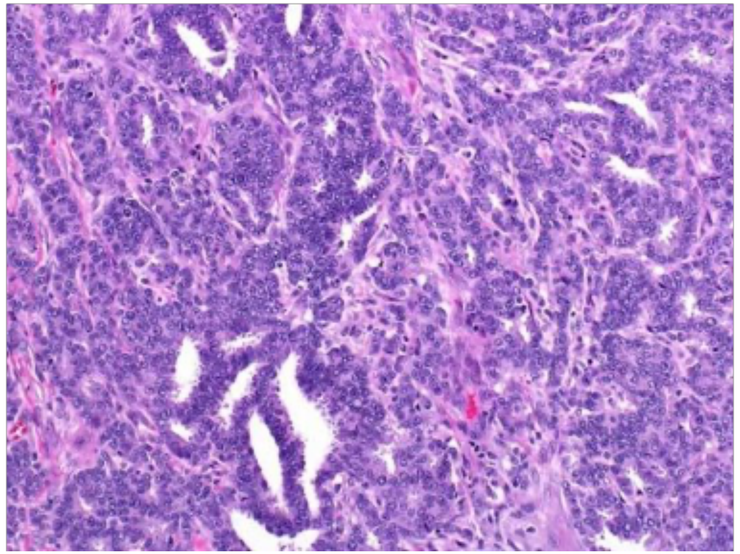
Topics covered in this lecture

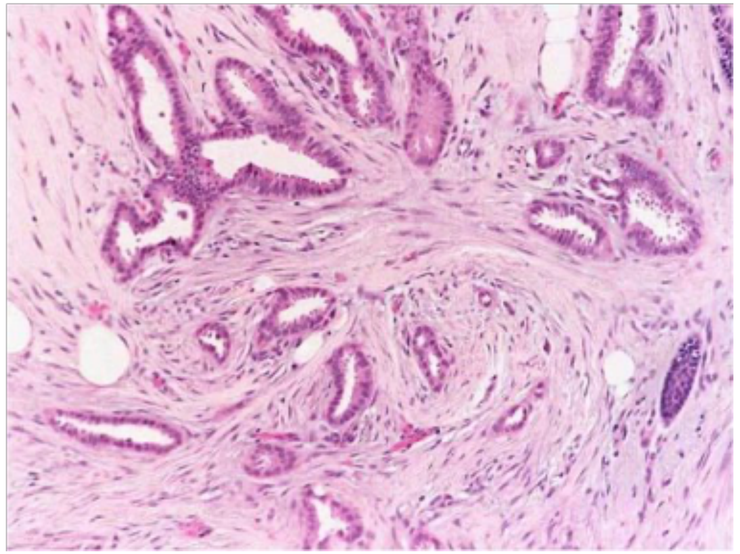
- A historical perspective of deep learning
- Linear and logistic regression as computational graphs
- From logistic regression to neural networks
- Layers as the basic building blocks of neural networks
- Implementation example in PyTorch of a simple neural network

How humans understand images?

From the three microscopic images of breast cancer given below, which one is the most “aggressive”/”dangerous for the patient”?







Pathology in a flash: 30-second training

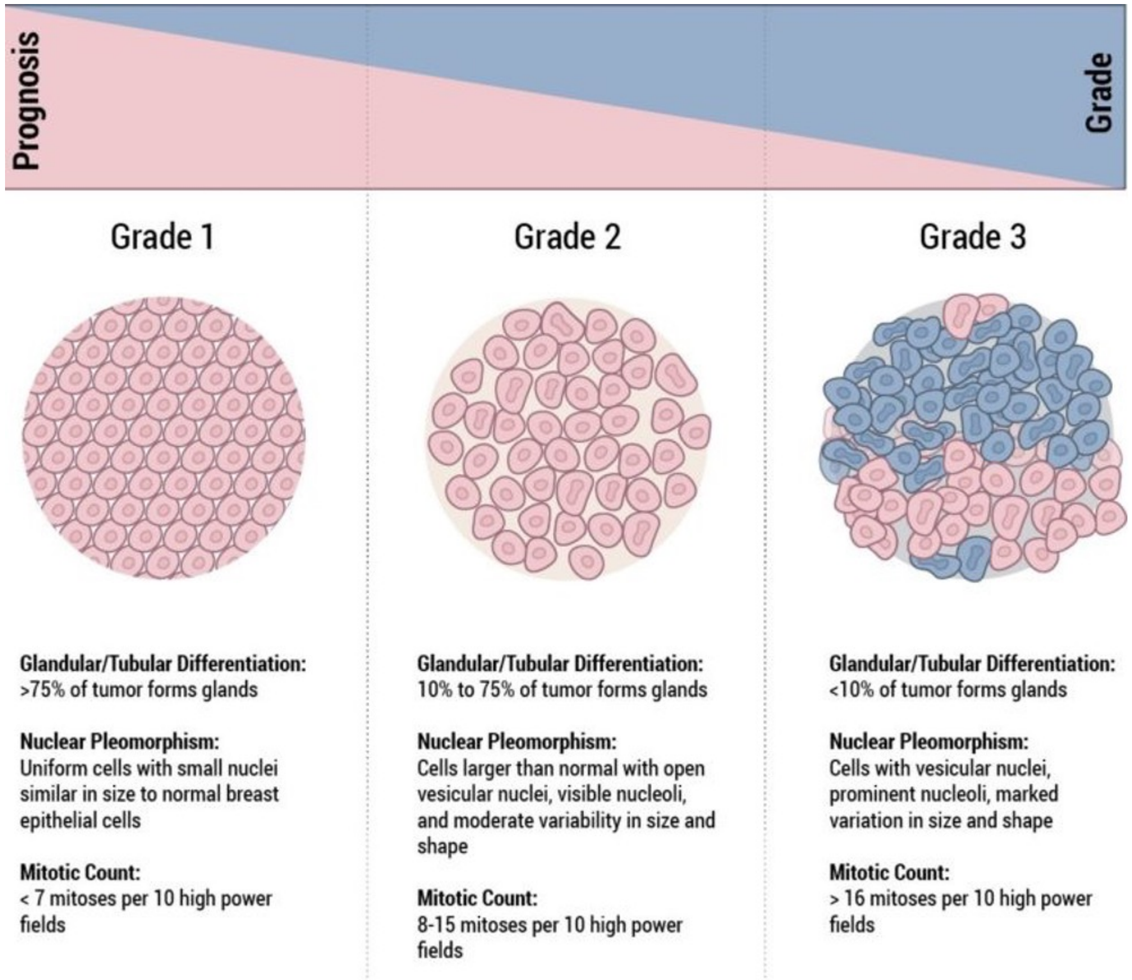
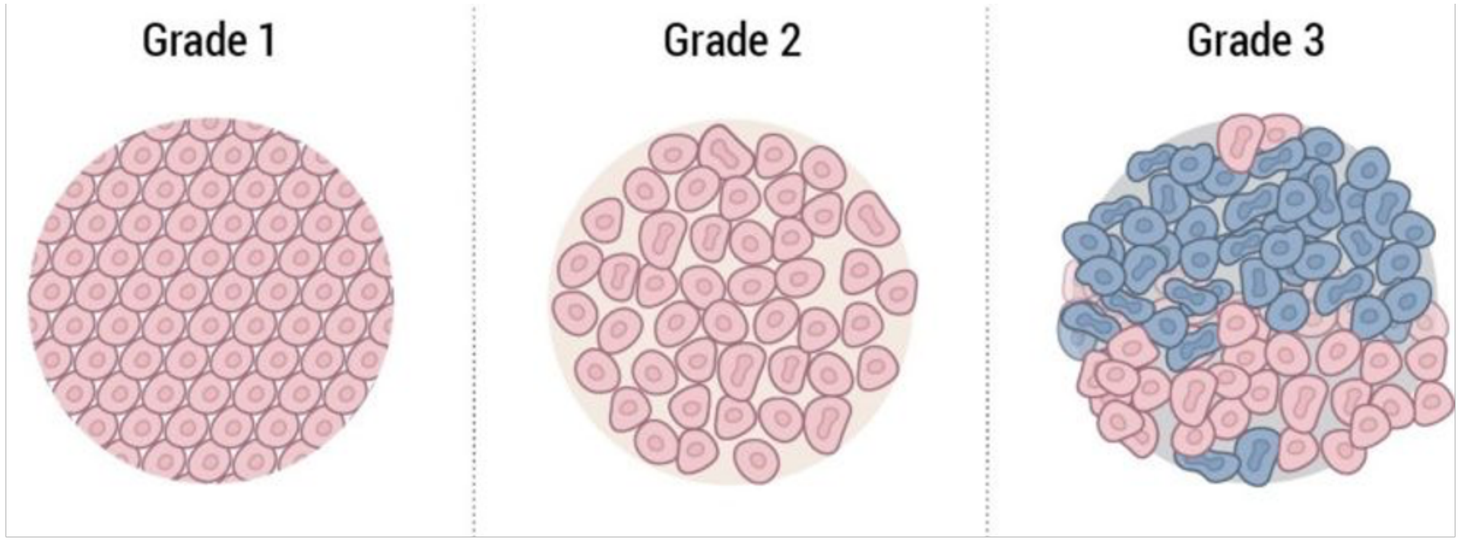
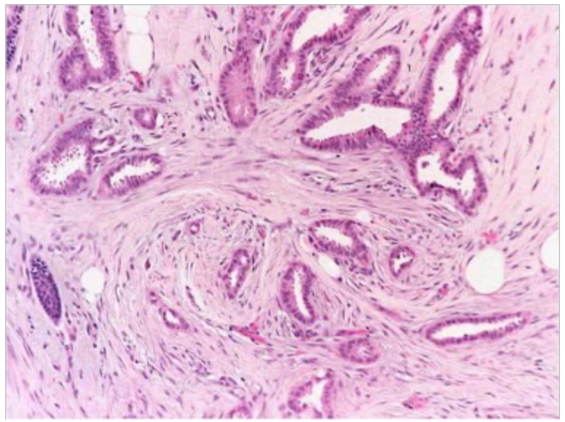
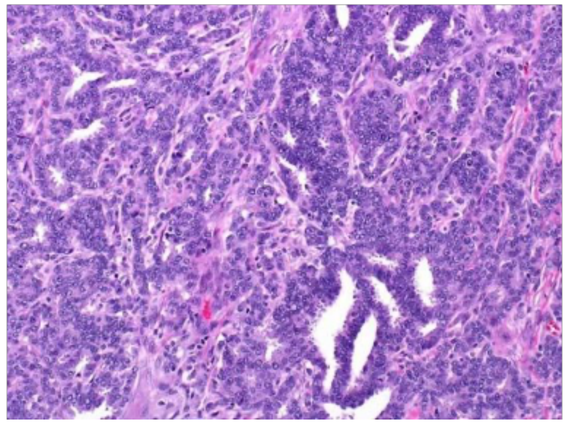


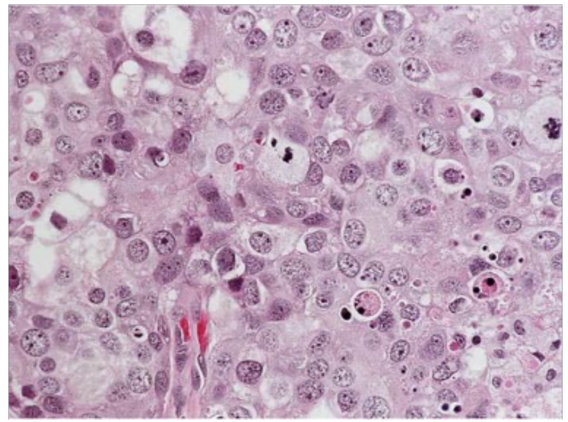
Figure from Rakha et al. 2010

The correct classification









Question

You are designing a machine learning system (e.g. using a linear regression model) with the goal of classifying breast cancer histopathology images as grade 1, 2 or 3.

What is the input to the logistic regression?

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

Feature extraction example: mitosis detection

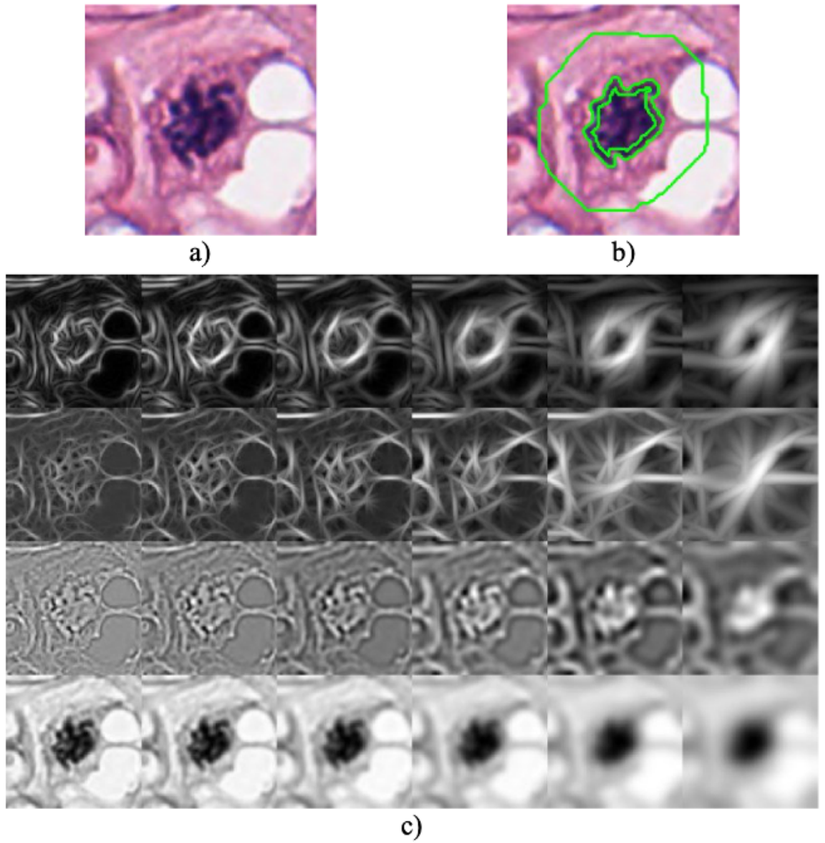
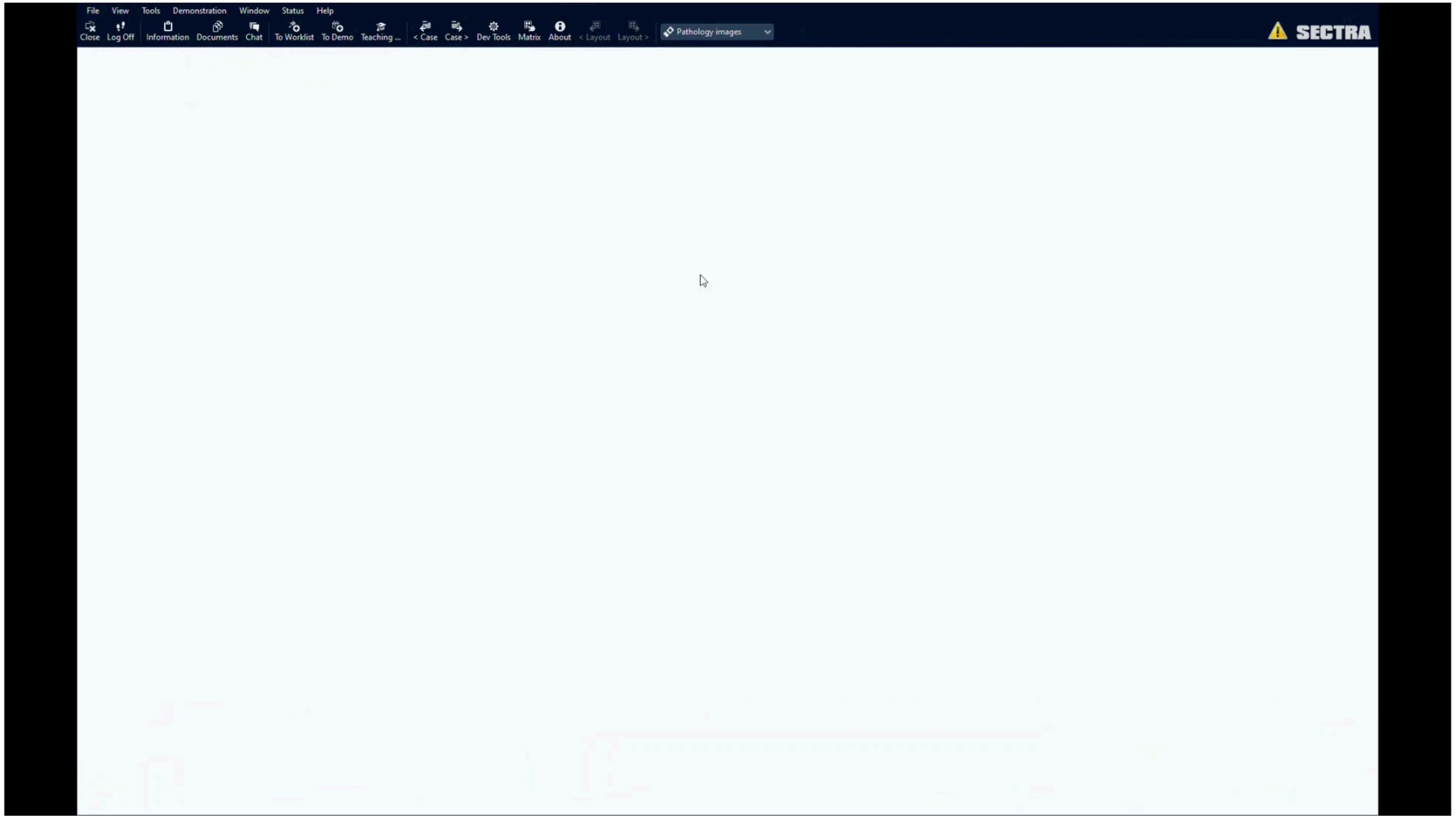


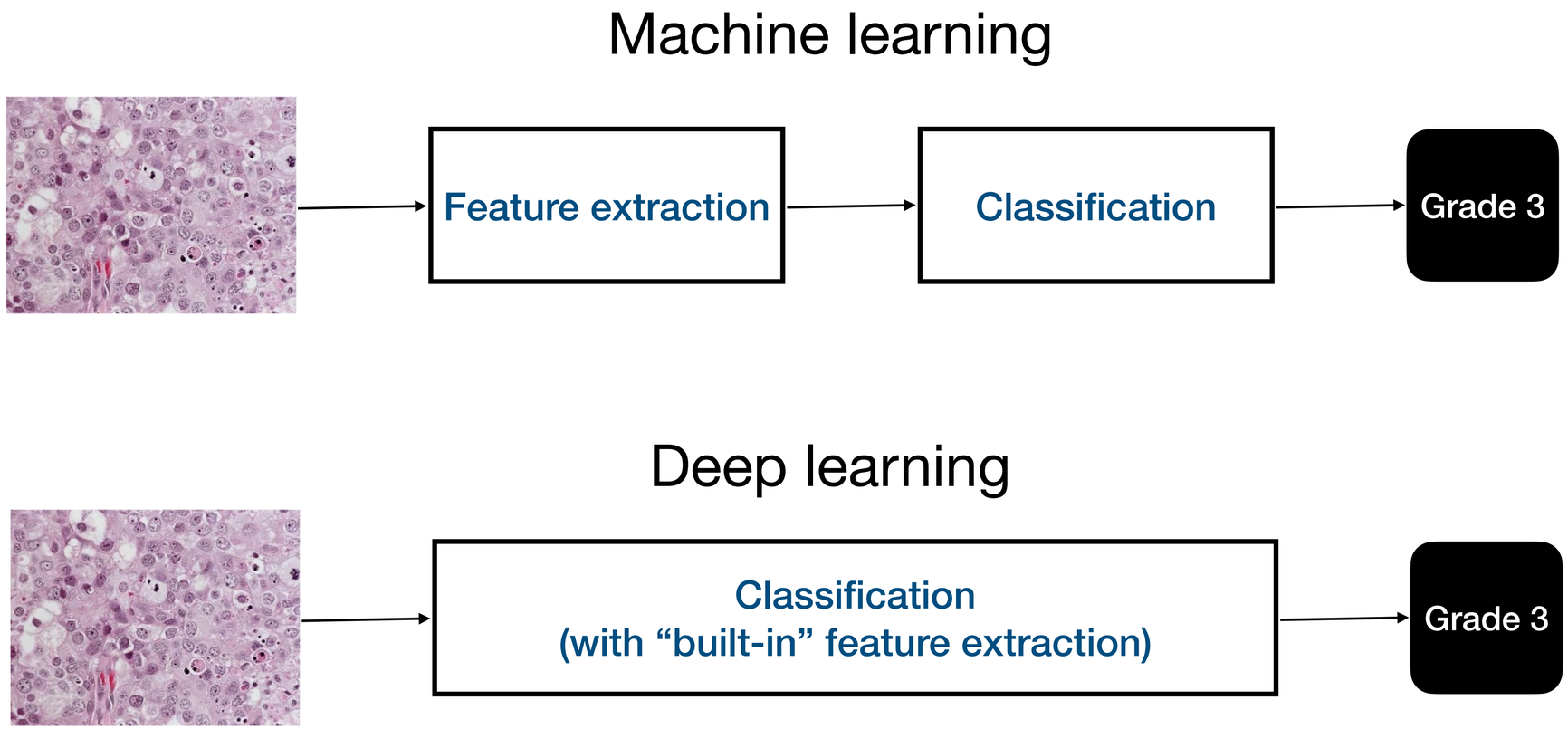
Figure 2. Feature extraction. a) Mitotic figure. b) The additional boundary and neighborhood regions. c) Outputs from the filter bank used for texture feature extraction. First row: output of edge filters, second row: output of bar filters, third row: output of Laplacian filters, fourth row: output of Gaussian filters.

Figure from Veta et al. 2013

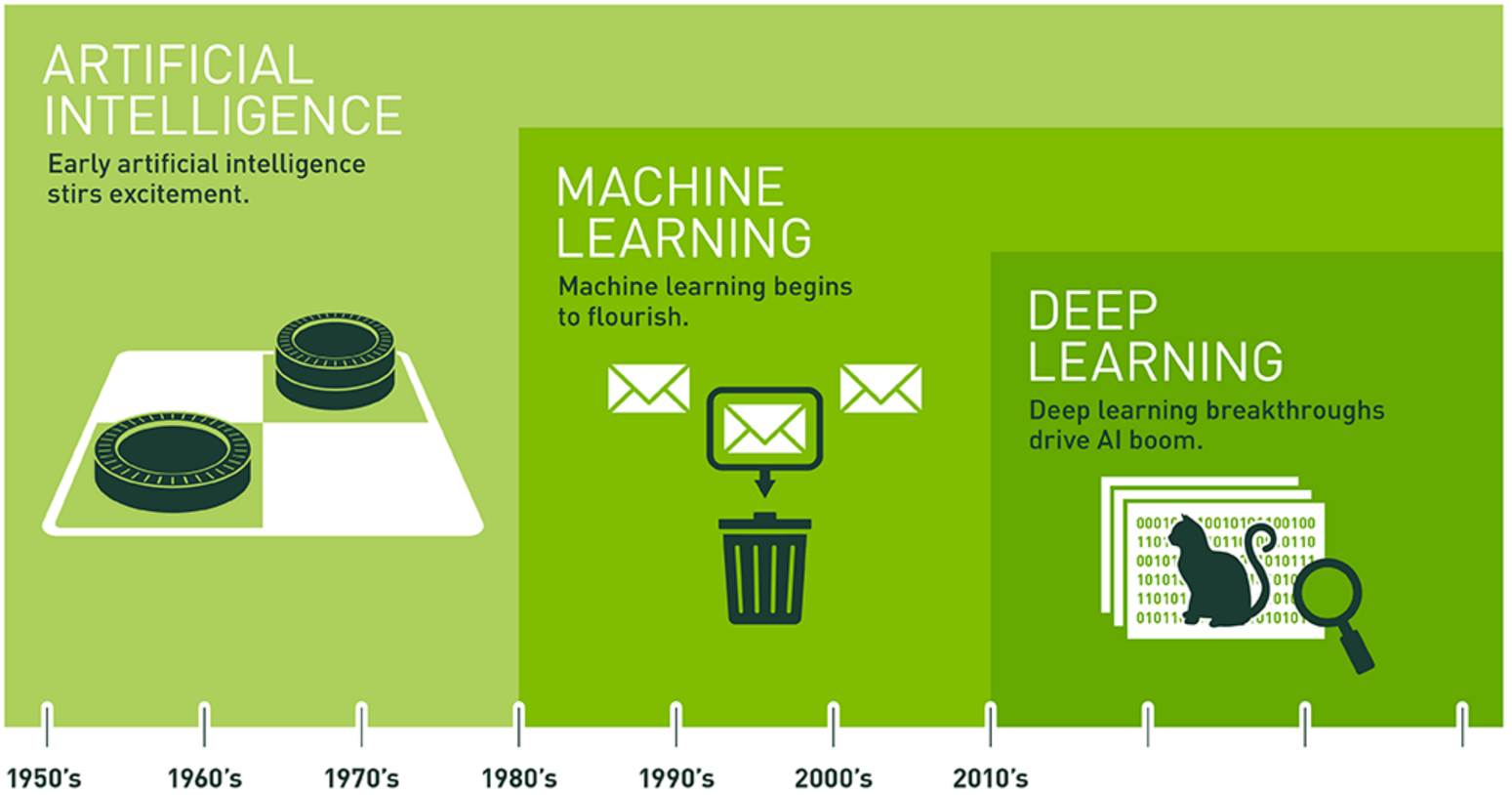
Mitosis detection implemented in the clinic



Feature extraction: handcrafted vs “built-in”



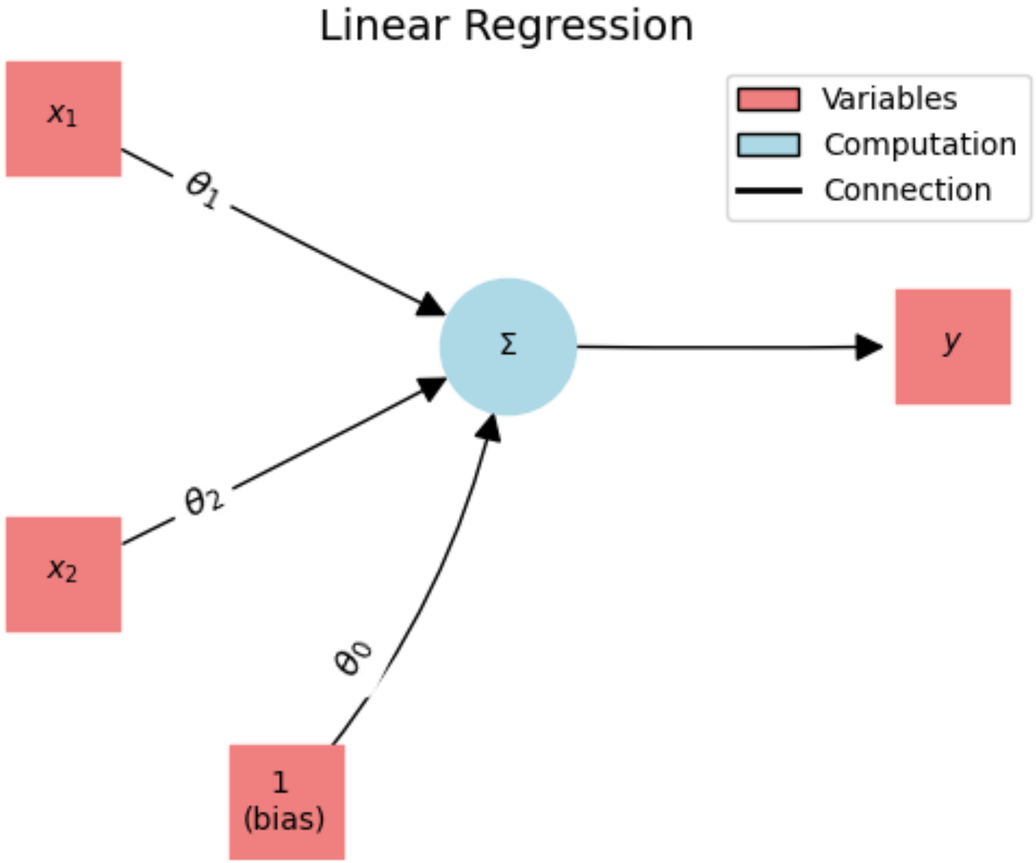
Deep learning: historical perspective



Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

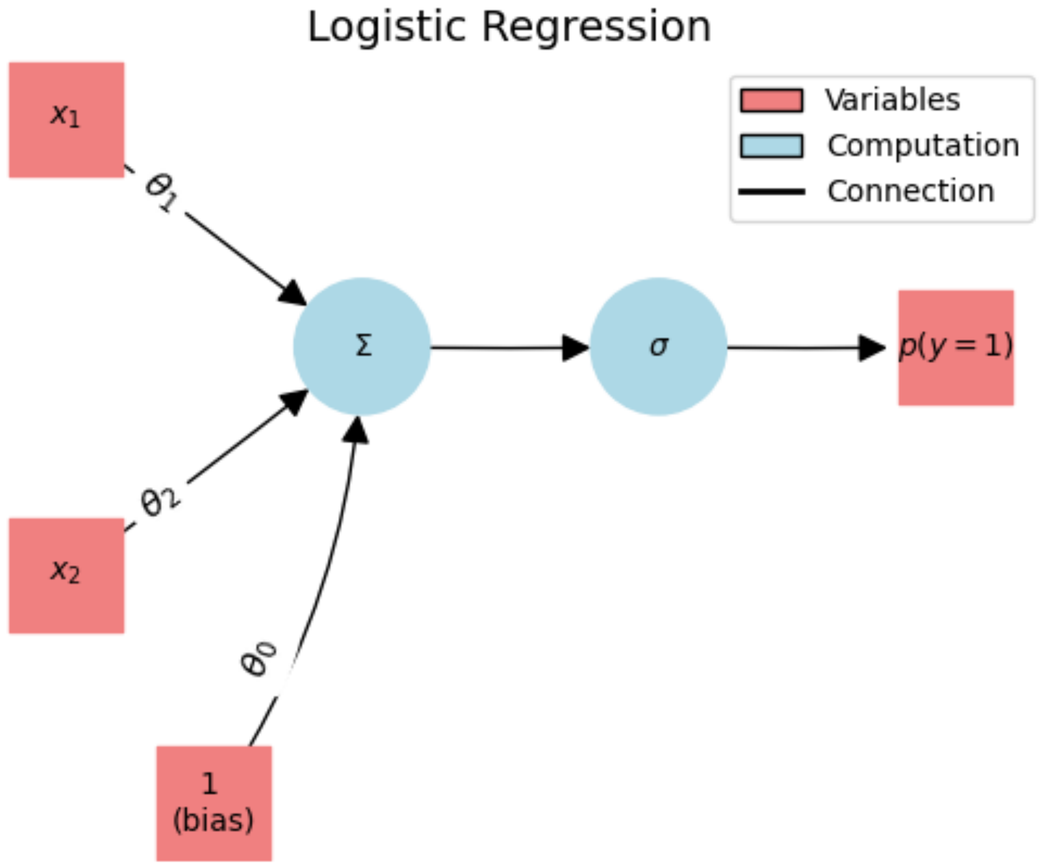
Figure from nvidia.com

Linear regression as computational graph



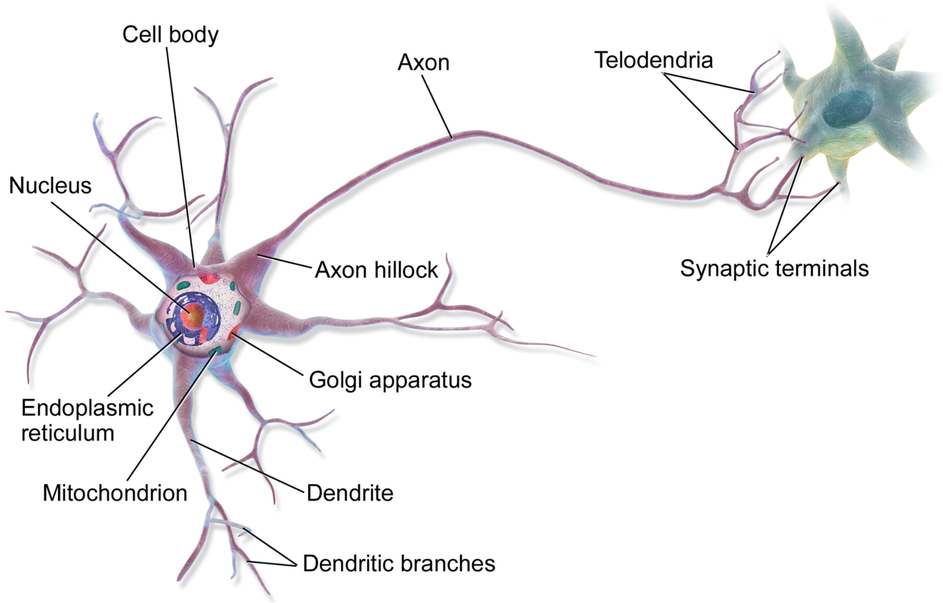
$$y = \theta_0 \cdot 1 + \theta_1 x_1 + \theta_2 x_2$$

Logistic regression as computational graph



$$p(y=1) = \sigma(\theta_0 \cdot 1 + \theta_1 x_1 + \theta_2 x_2) \qquad \sigma(a) = \frac{1}{1+e^{-a}}$$

Biological vs “artificial” neuron



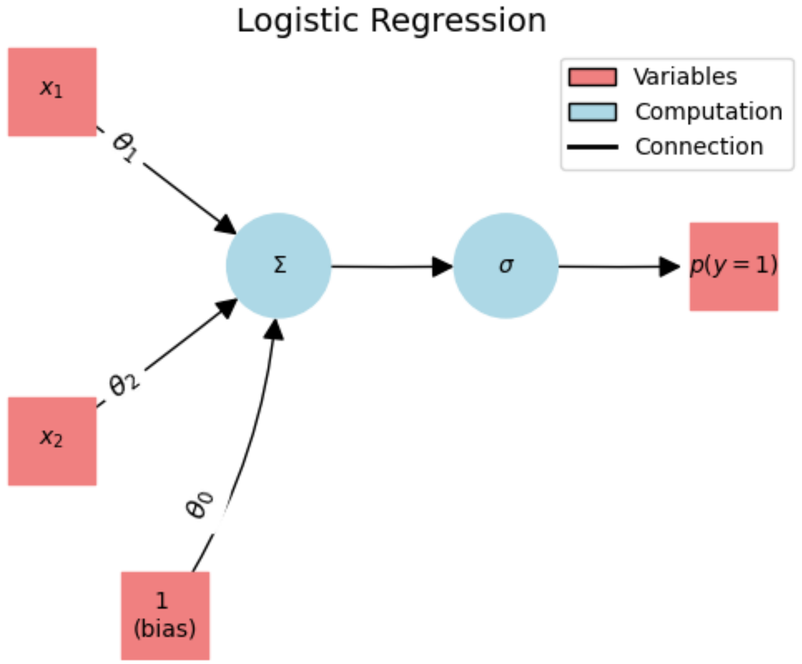


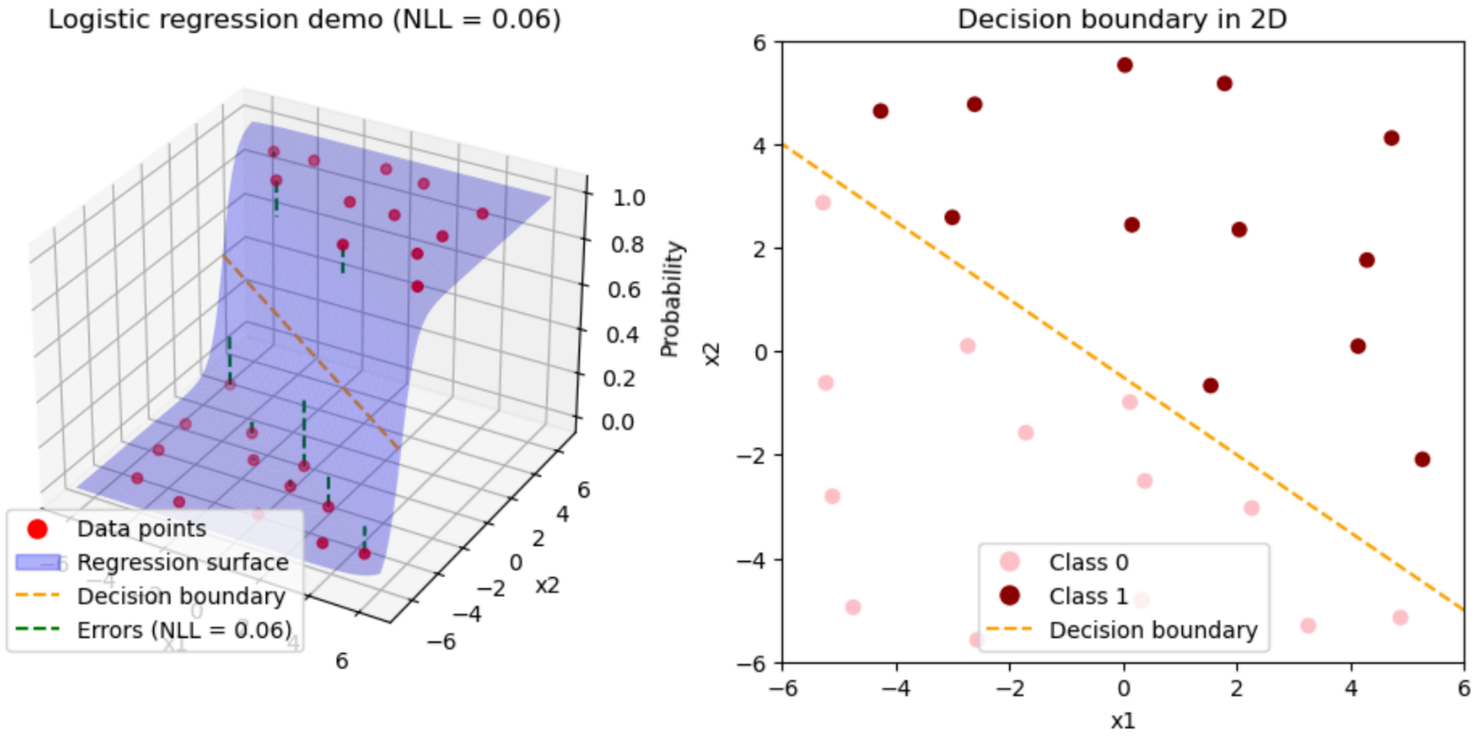
Figure from wikimedia.com

Question

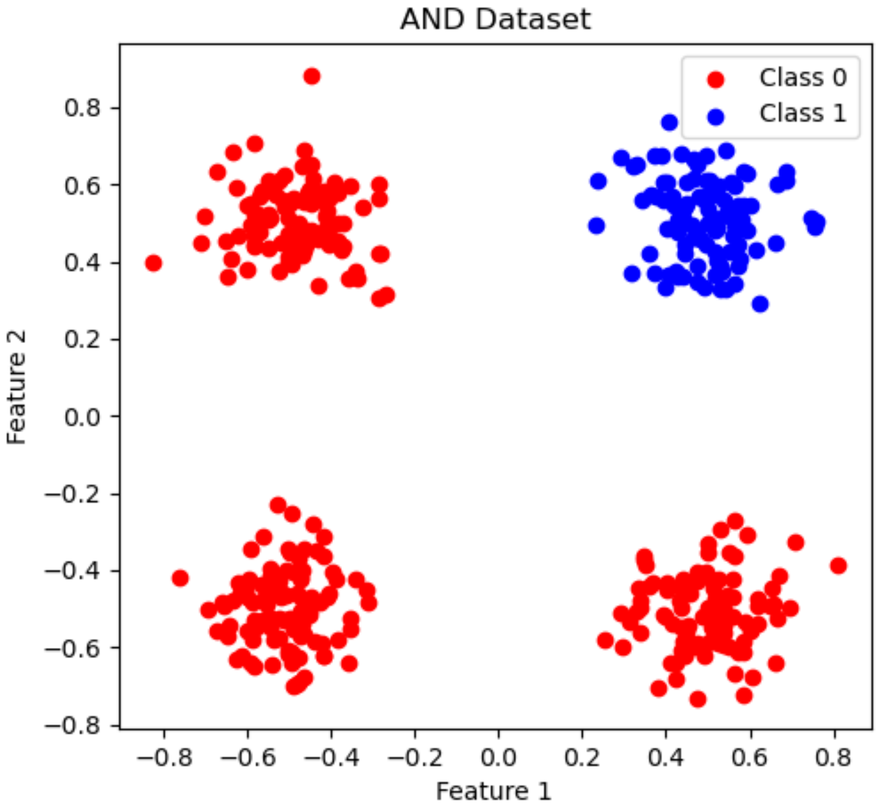
What type of decision boundary does the logistic regression classifier have?

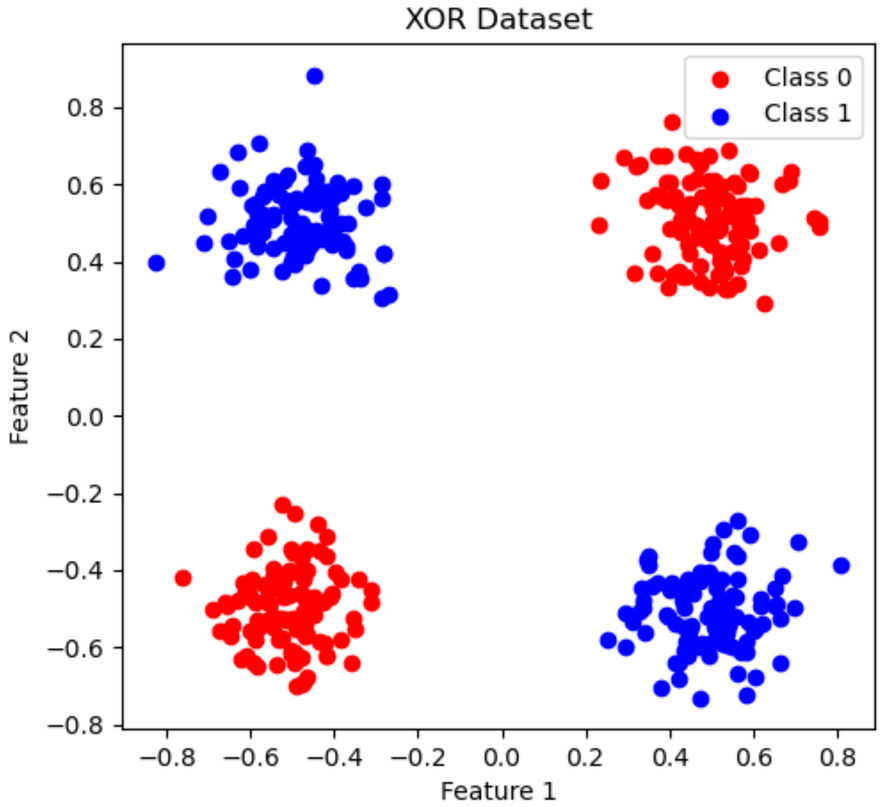
Linear decision boundary

From Part 1 of the practicals:



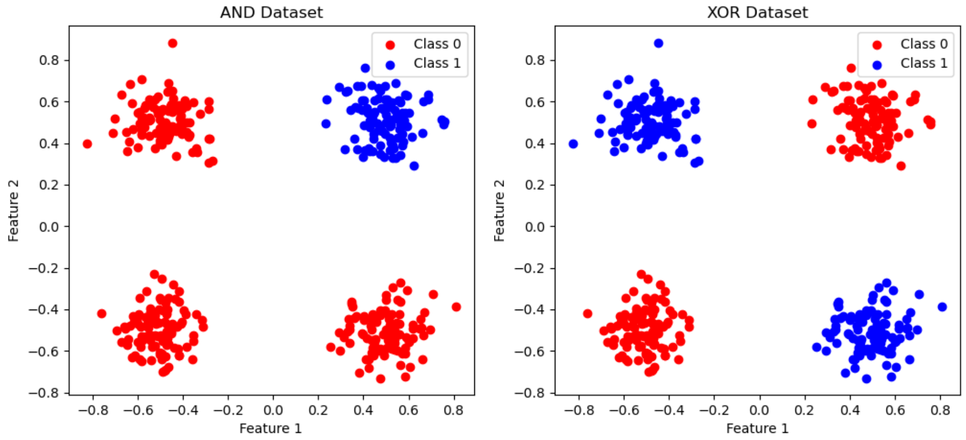
Two “toy” classification problems



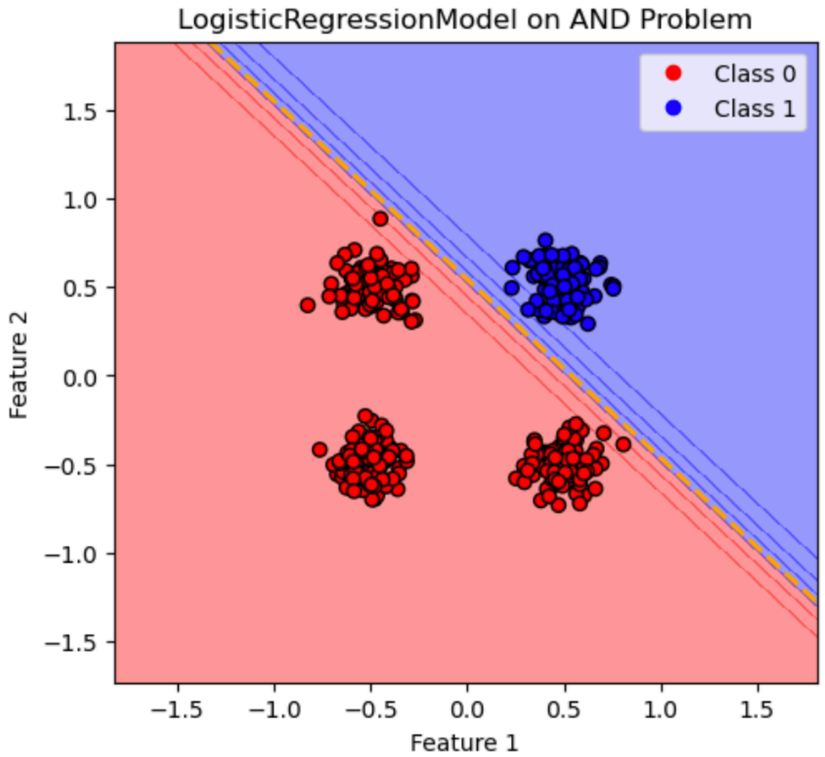


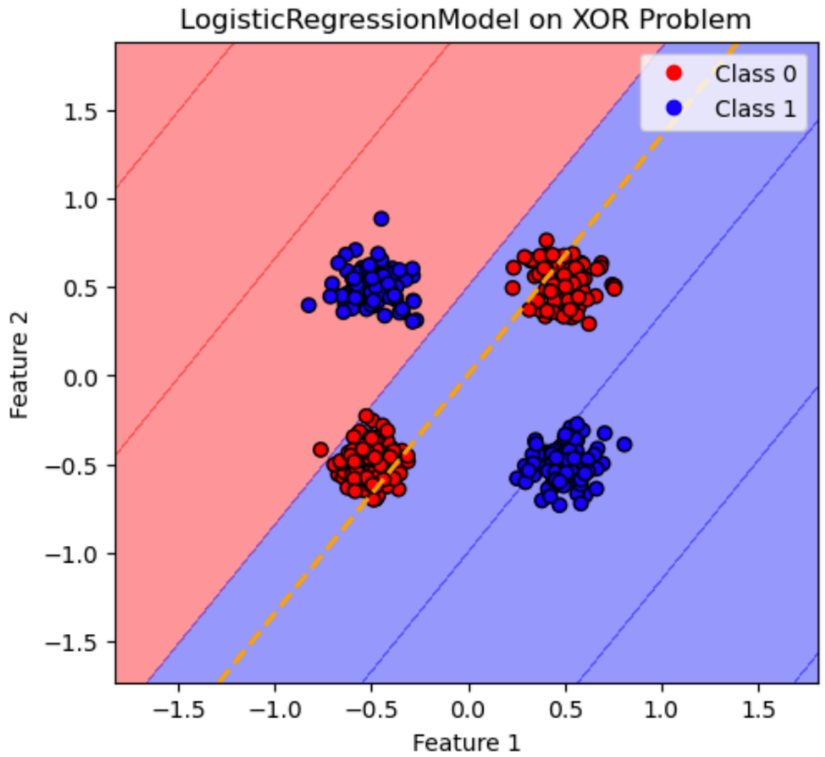
Question

Which problem is solvable by logistic regression?



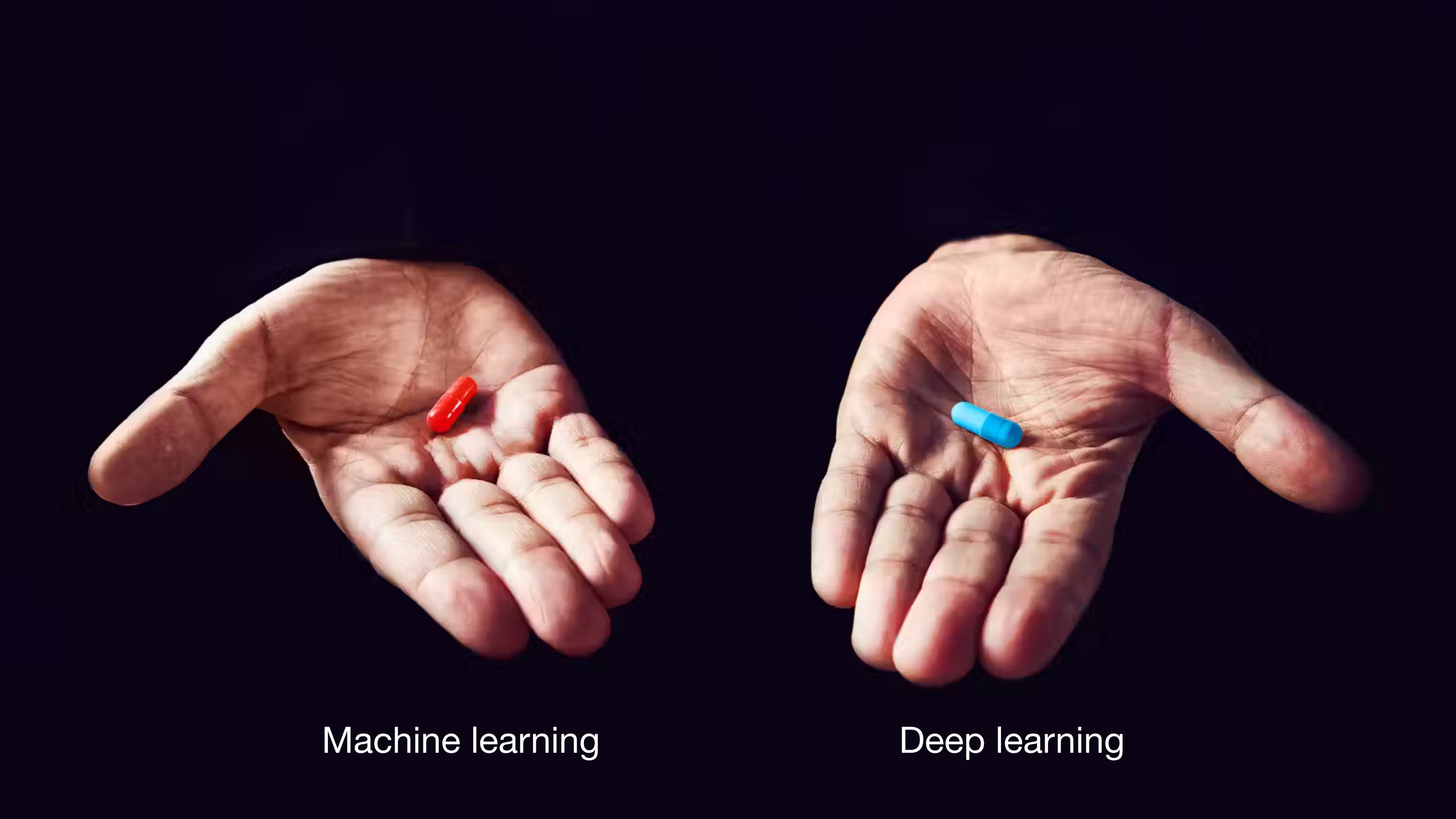
XOR is nor linearly separable



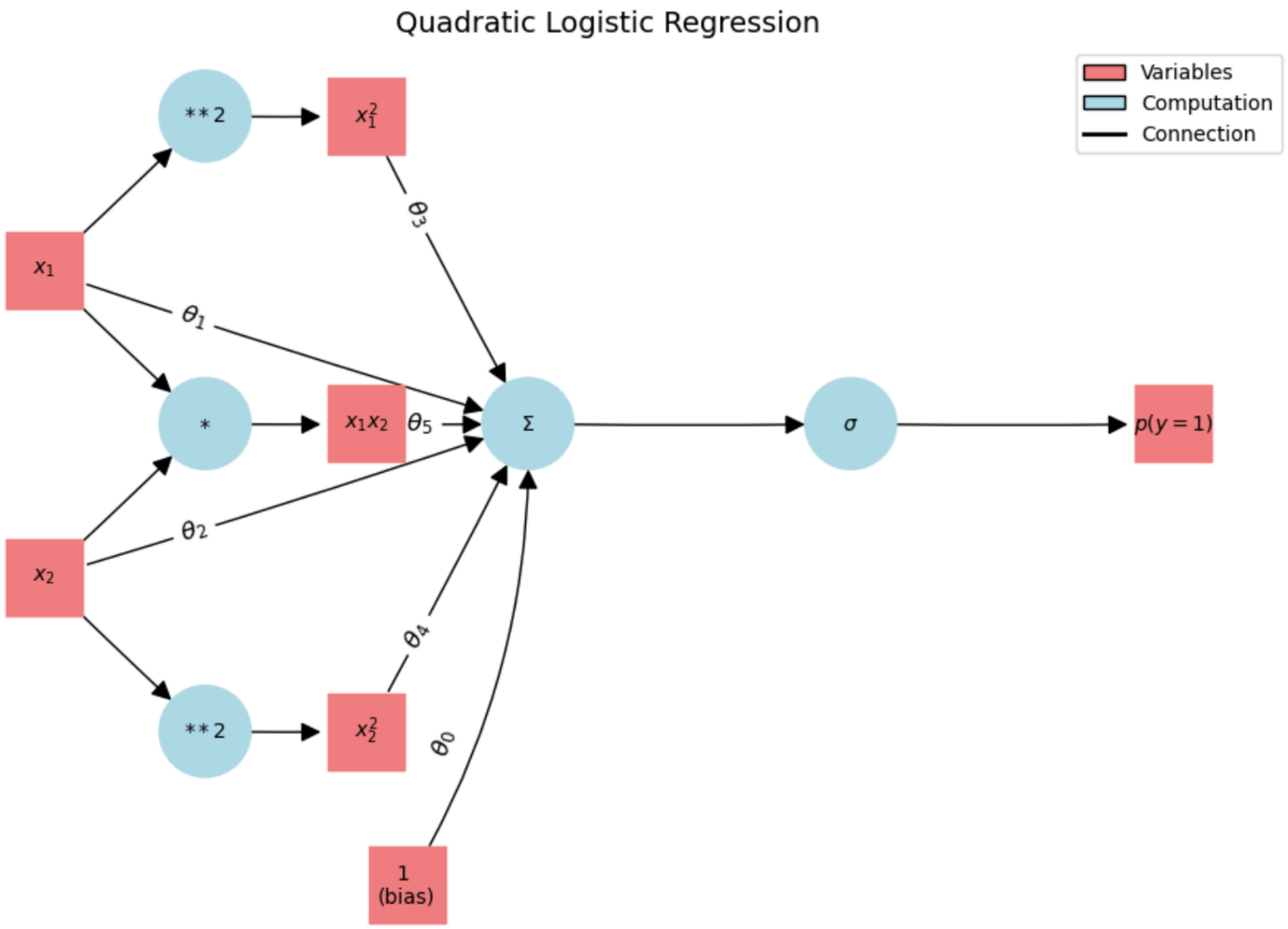


Question

How can you use logistic regression to solve the XOR problem?

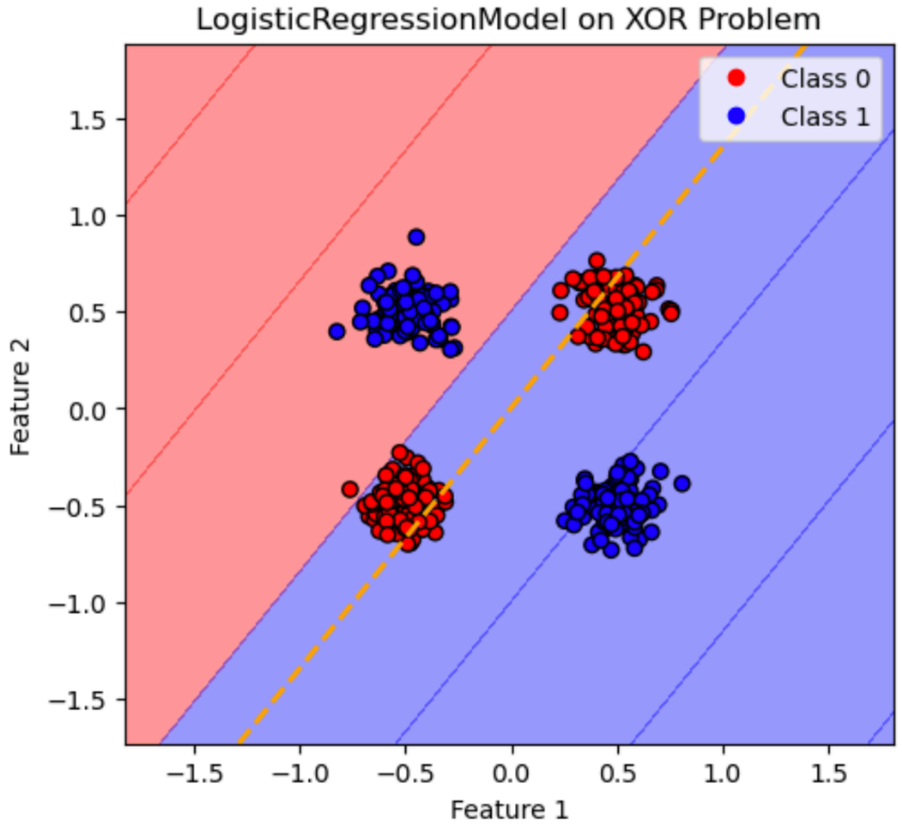

Machine learning
Deep learning

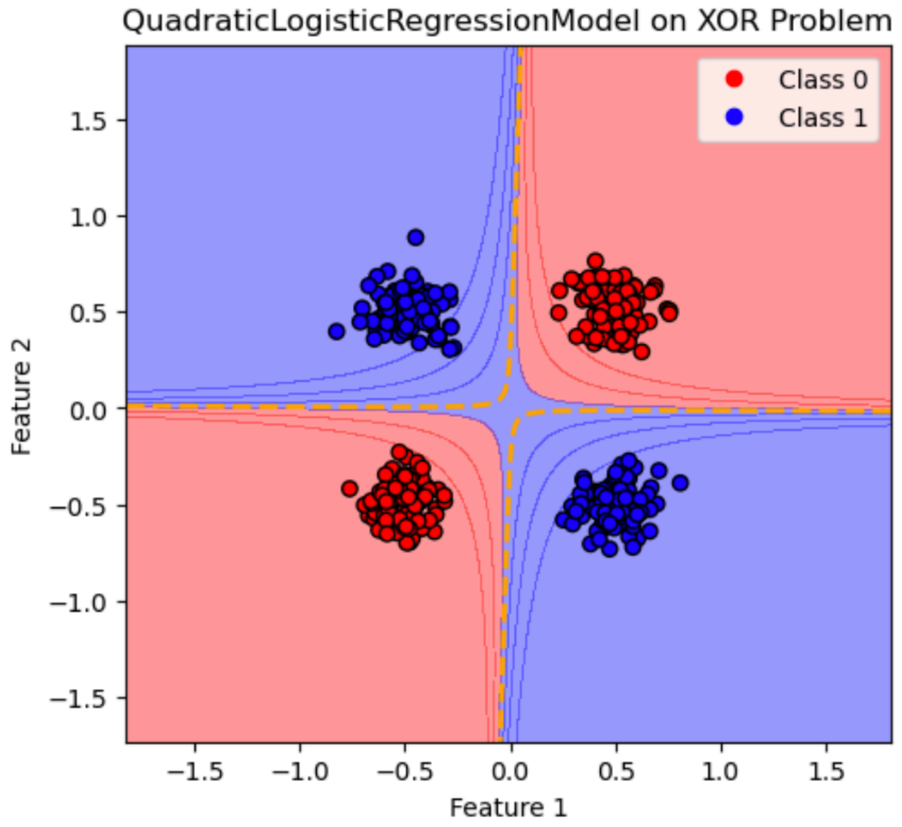
Solving XOR: the “classical” approach



Linear model, but quadratic decision boundary

Quadratic in the original 2D feature space, that is.





Feature engineering

- **Feature Engineering**: Creating or modifying features to improve model performance.
- In the XOR problem, quadratic features ($x_1, x_2, x_1^2, x_2^2, x_1 x_2$) help capture **non-linear patterns**.
- **Key benefit**: transforms raw data into a format that better reflects underlying patterns.
- Relies on **domain knowledge**: insights about the problem guide feature creation.

Challenges of feature engineering

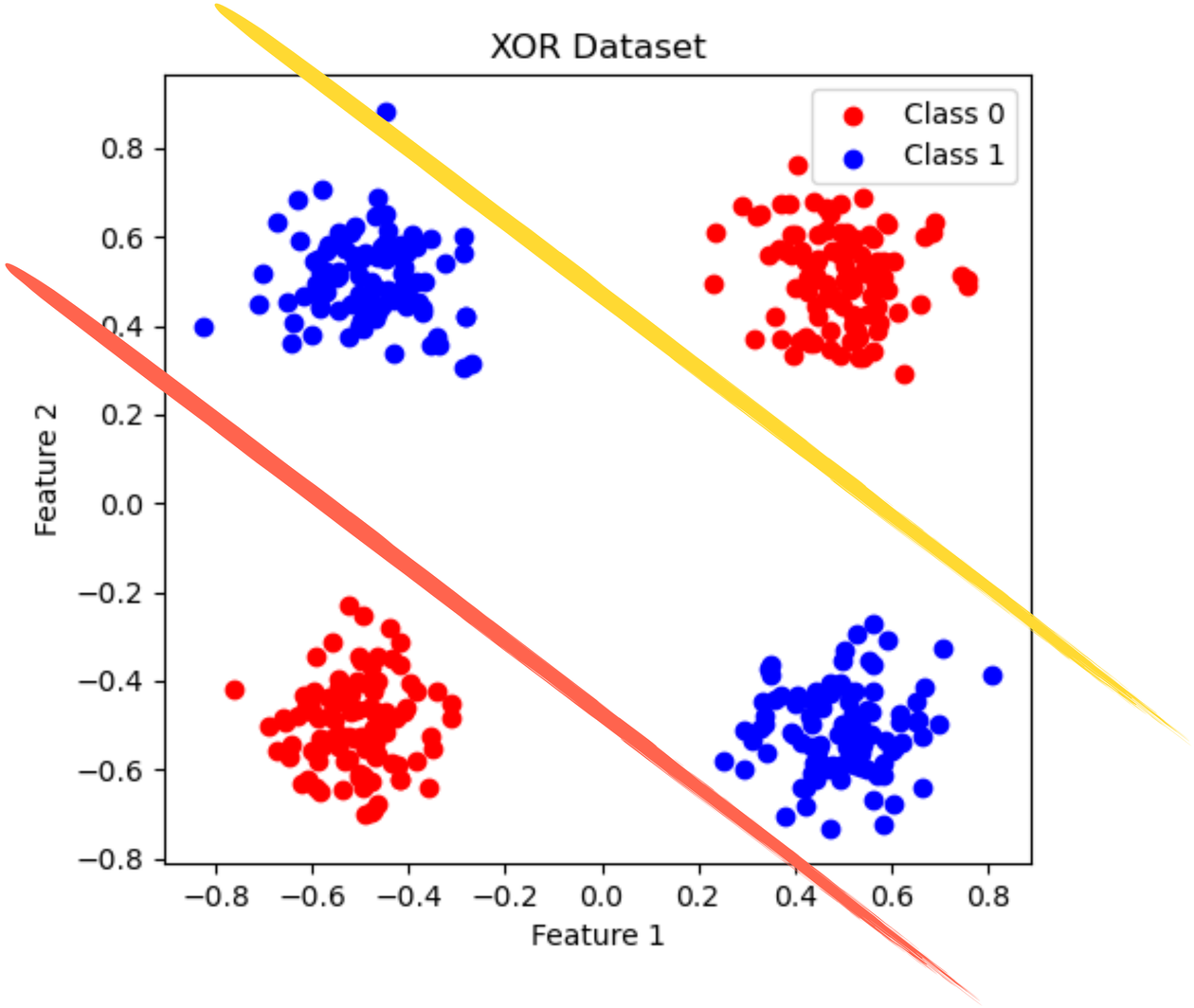
- **Time-consuming and dependent** on expert knowledge.
- Domain knowledge may be **incomplete**, missing valuable transformations.
- Capturing complex interactions is difficult with manual feature engineering.

- More automated techniques (e.g., deep learning) can learn features from raw data, reducing the need for manual feature engineering.

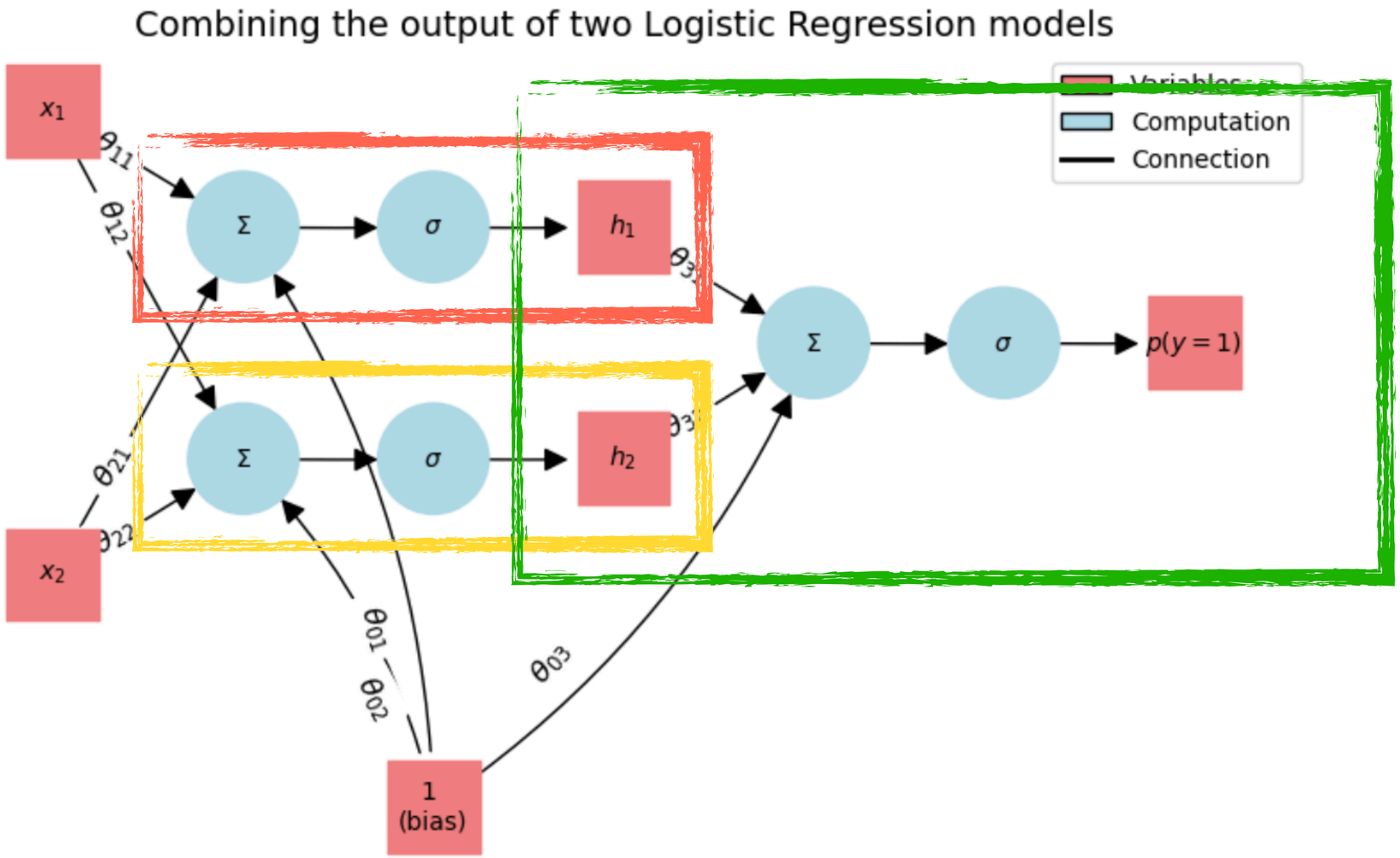
Question

How can you use two logistic regression models to solve the XOR problem?

Solving XOR with two linear decision boundaries

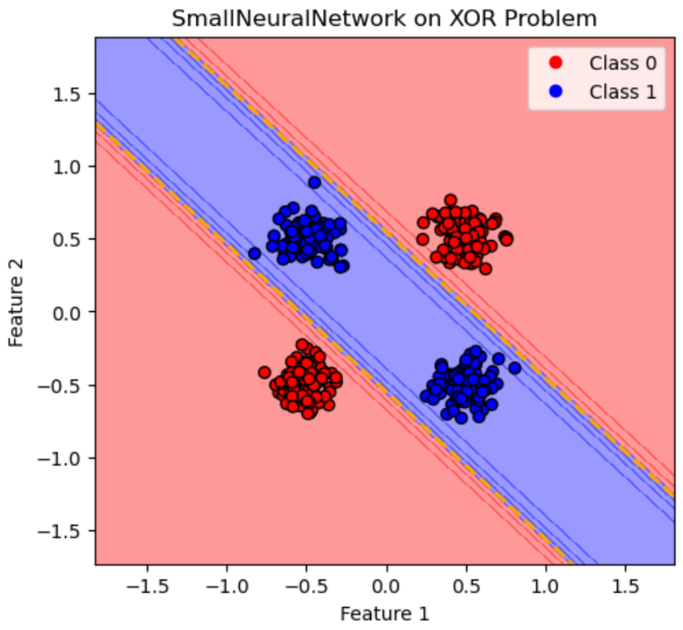


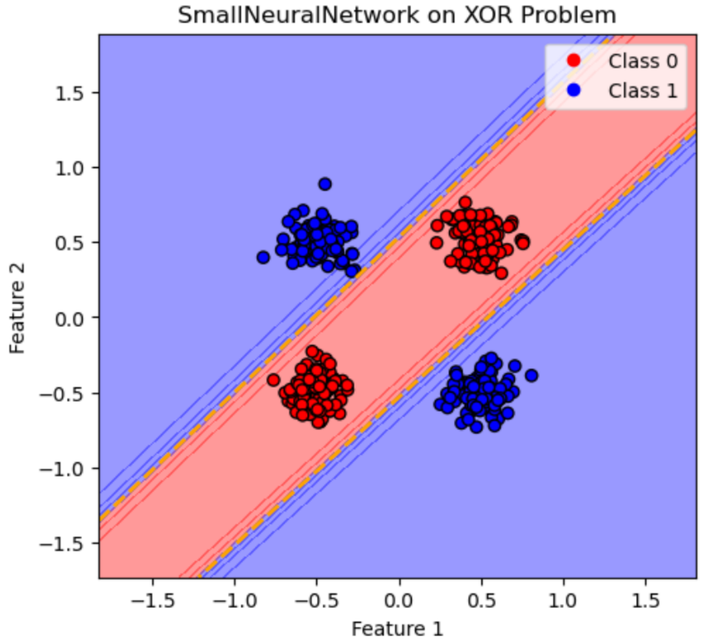
It takes two to XOR



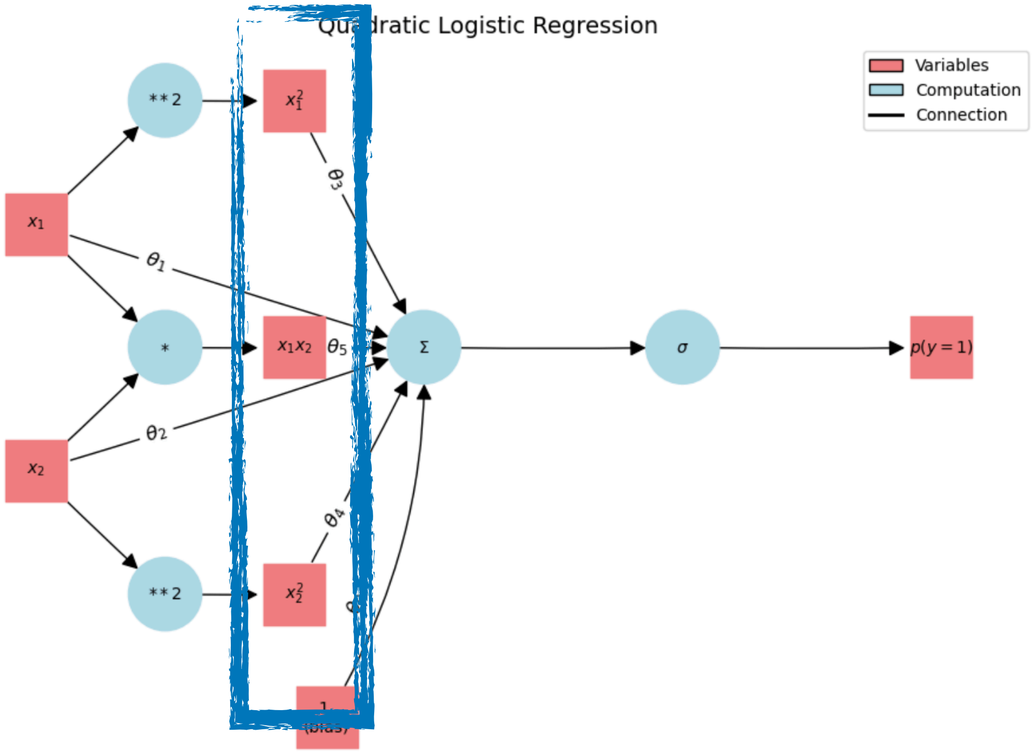
Question

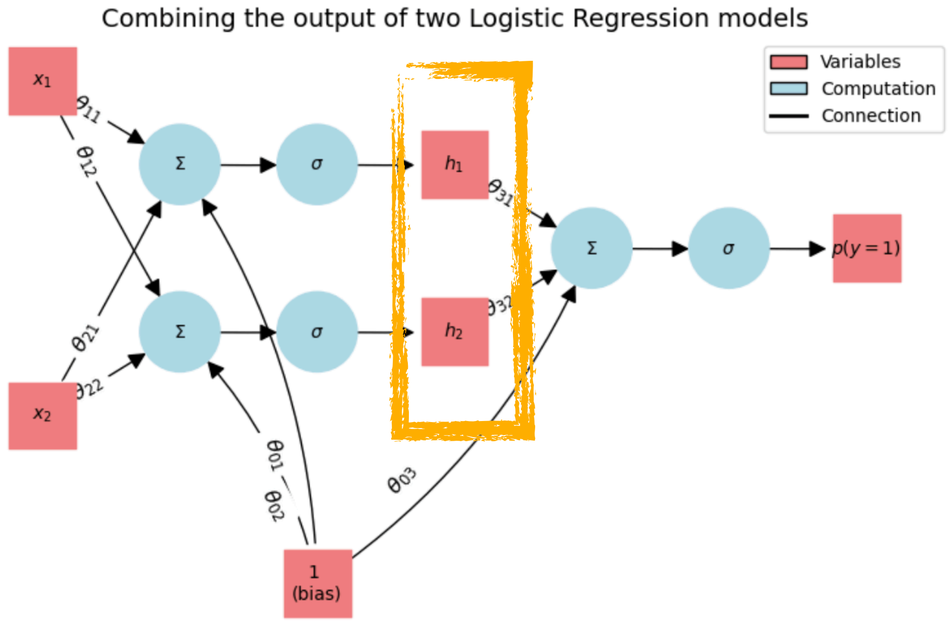
If you run the optimisation procedure (in the practicals) for the same dataset multiple times you might end up with different solutions. Is there a correct one?





Feature engineering vs learned features

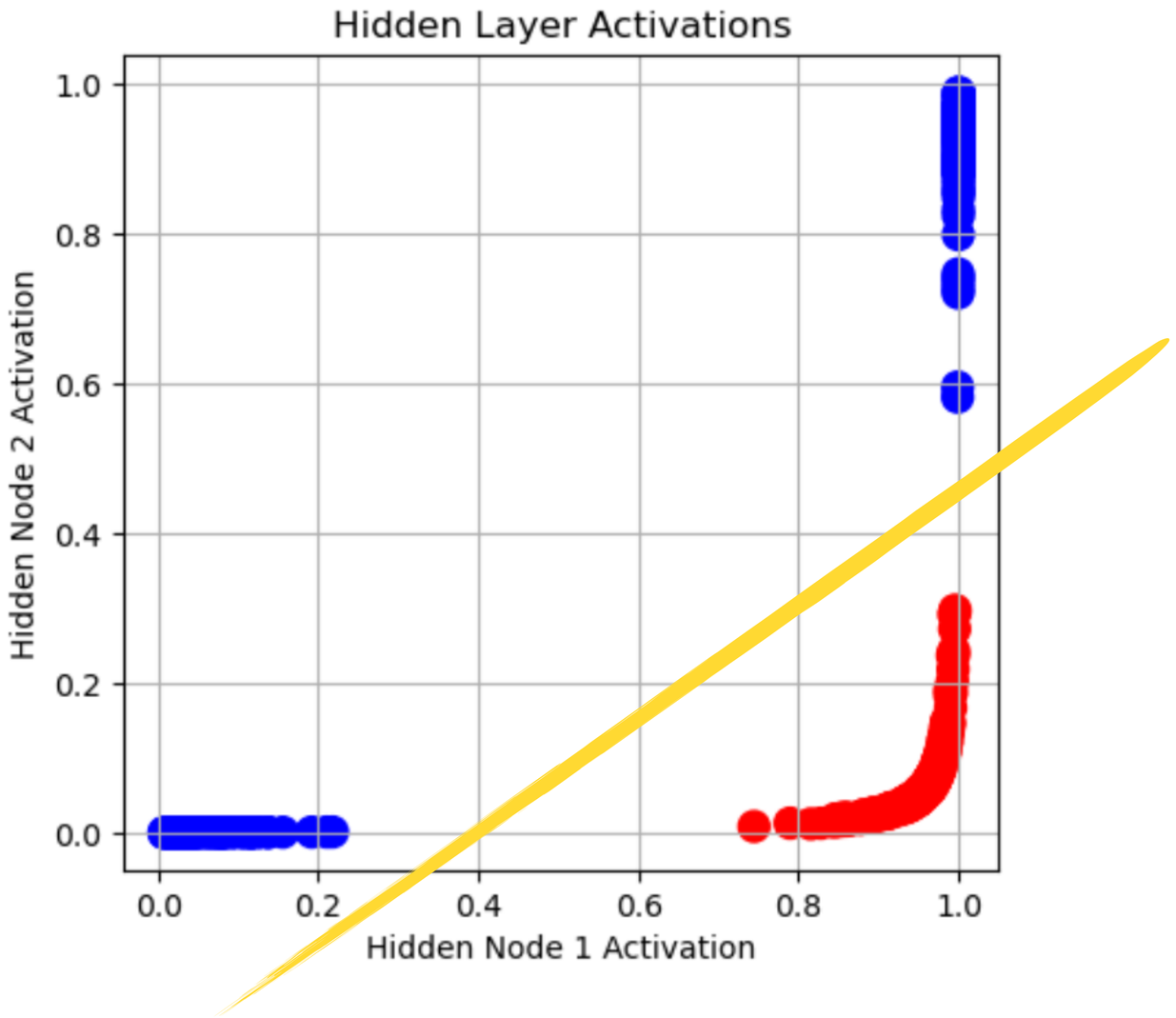




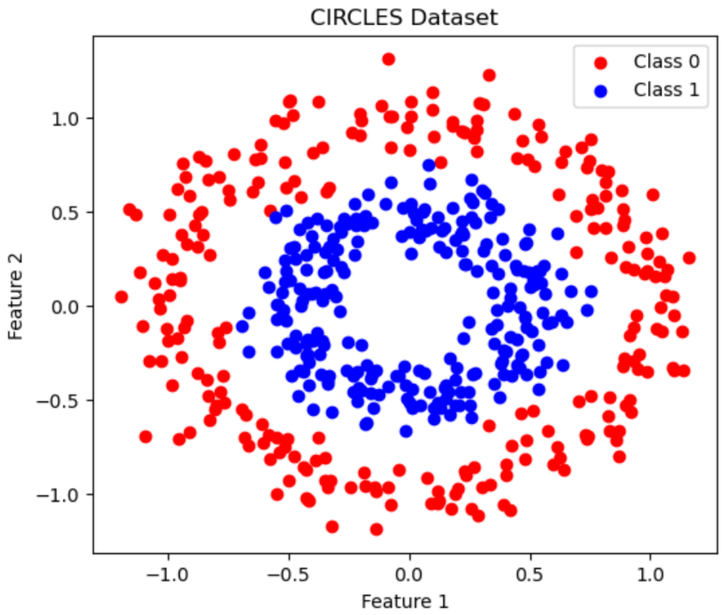
Manual feature representation

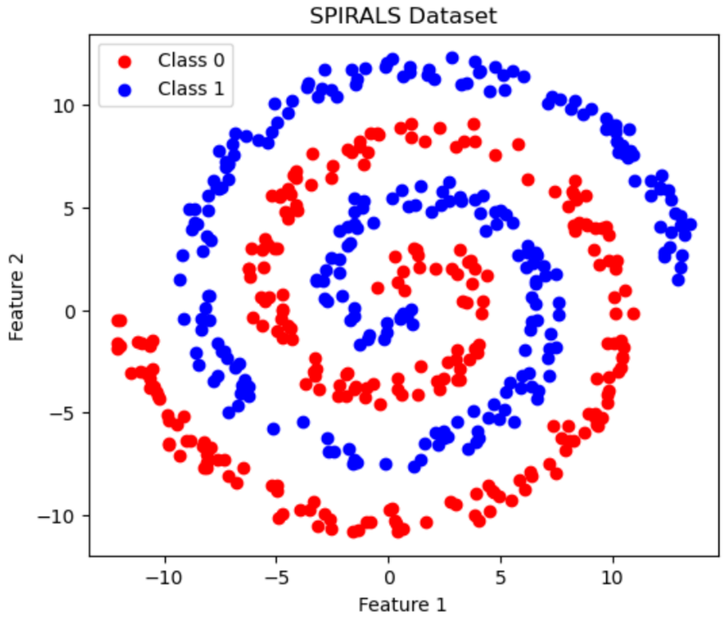
Learned feature representation

Visualising the learned feature representation



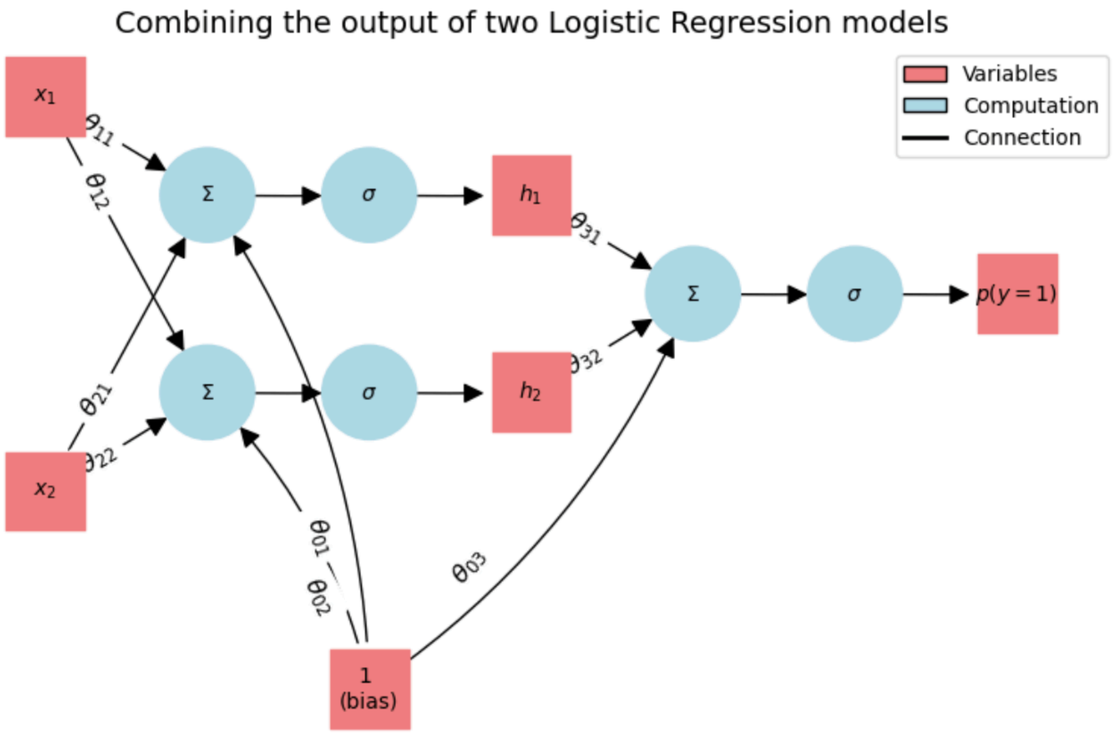
In the practicals…

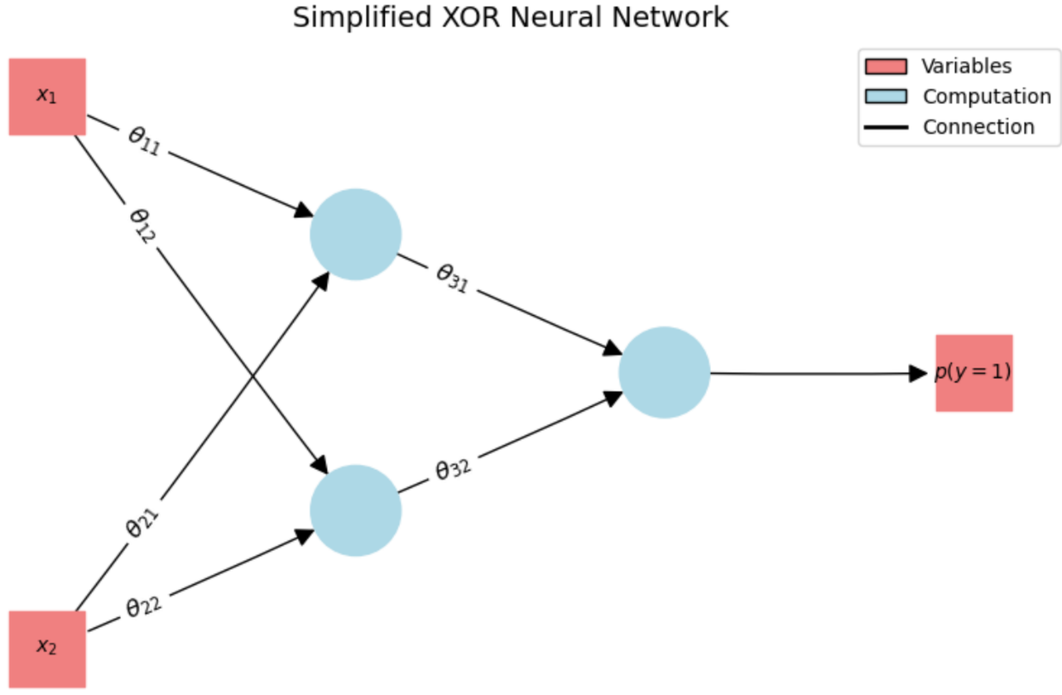




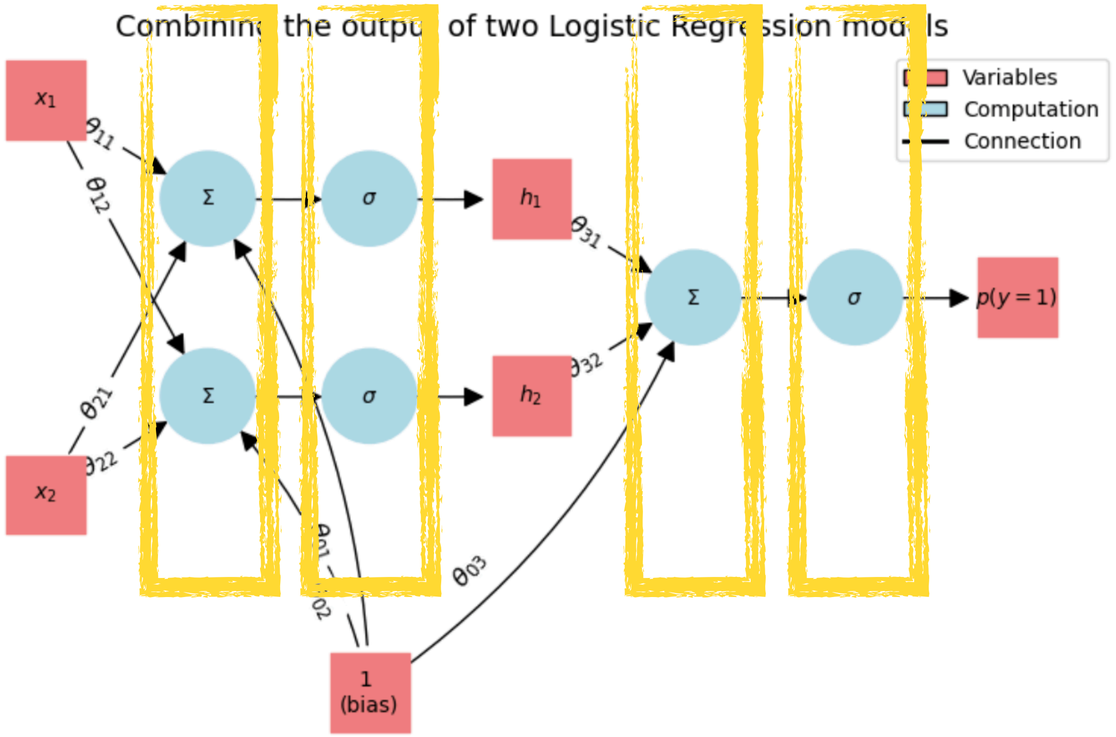
You can also experiment in Tensorflow Playground: http://playground.tensorflow.org

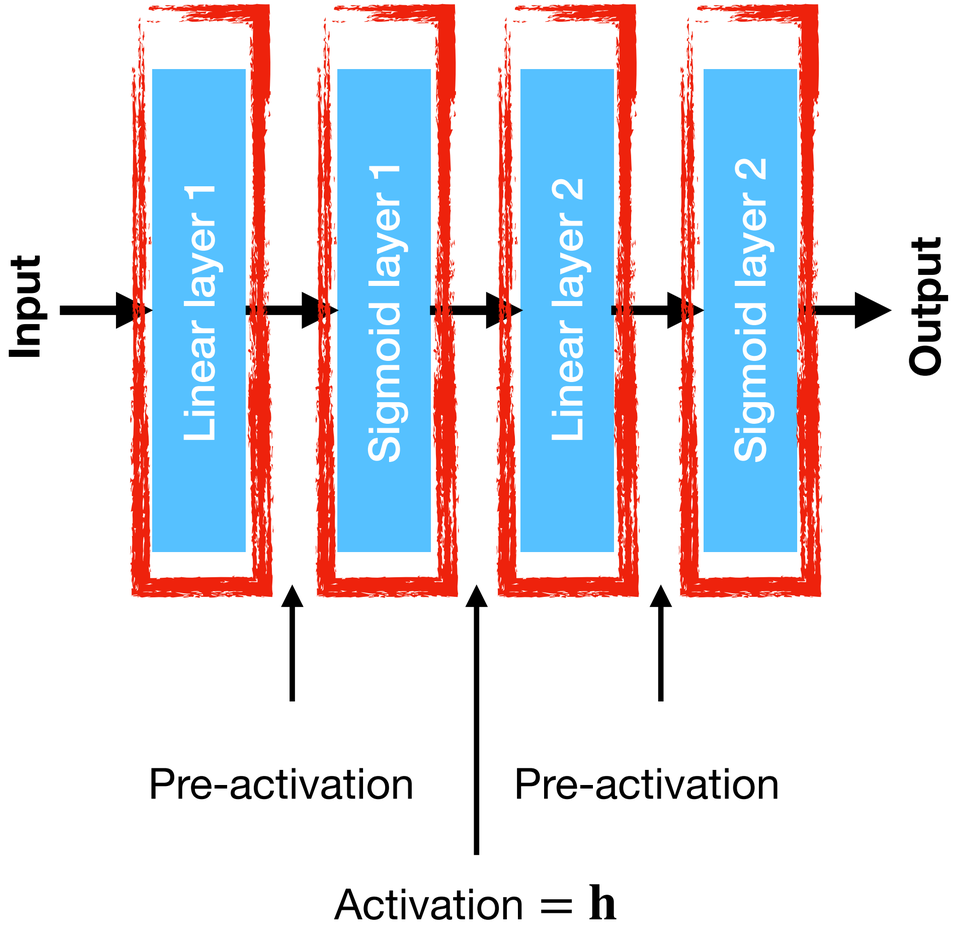
Neurons as the basic building blocks





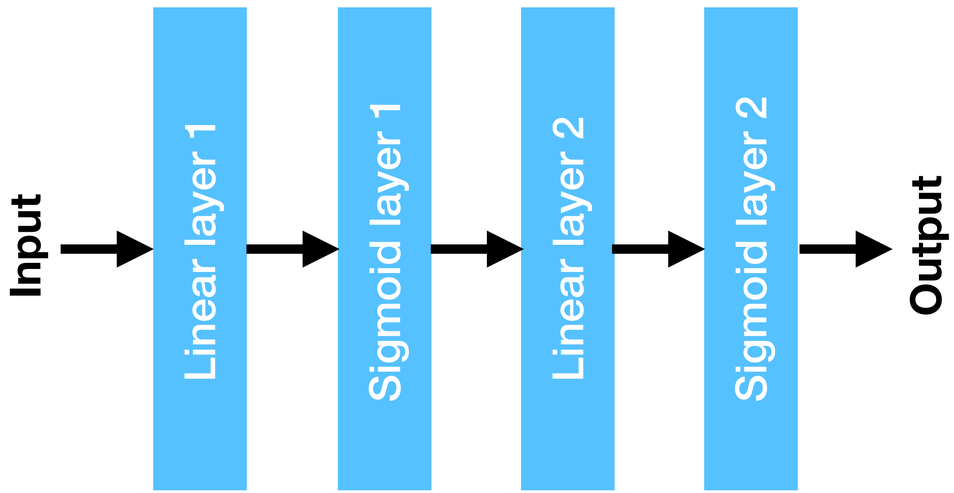
Layers as the basic building blocks

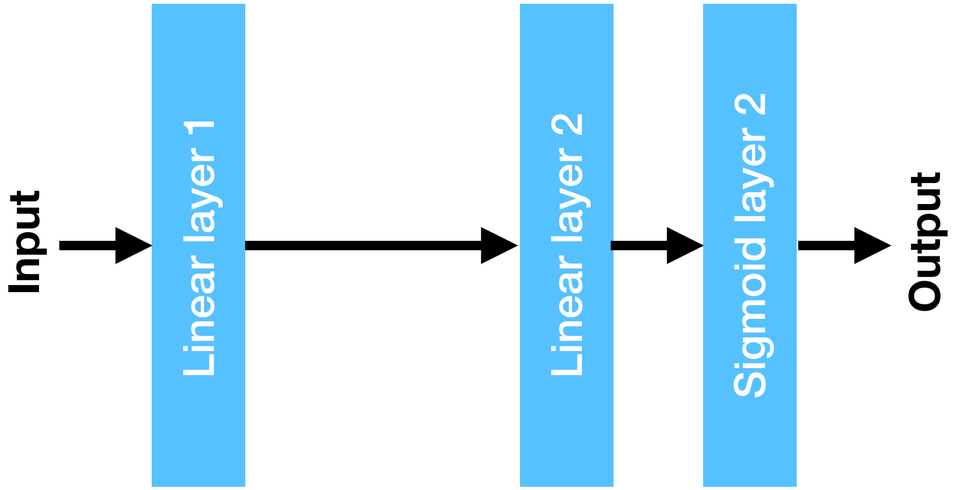




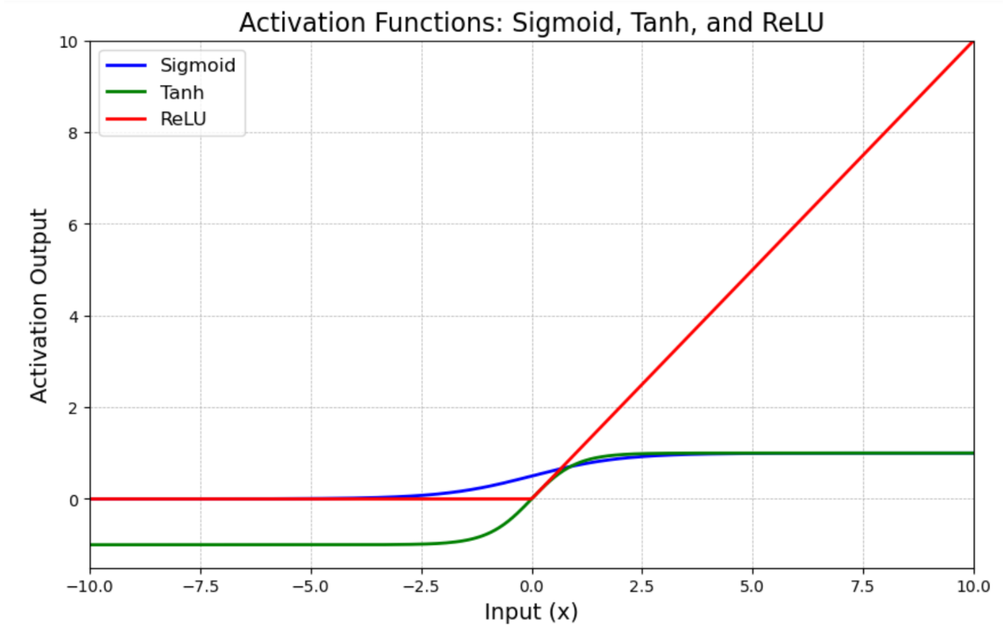
Question

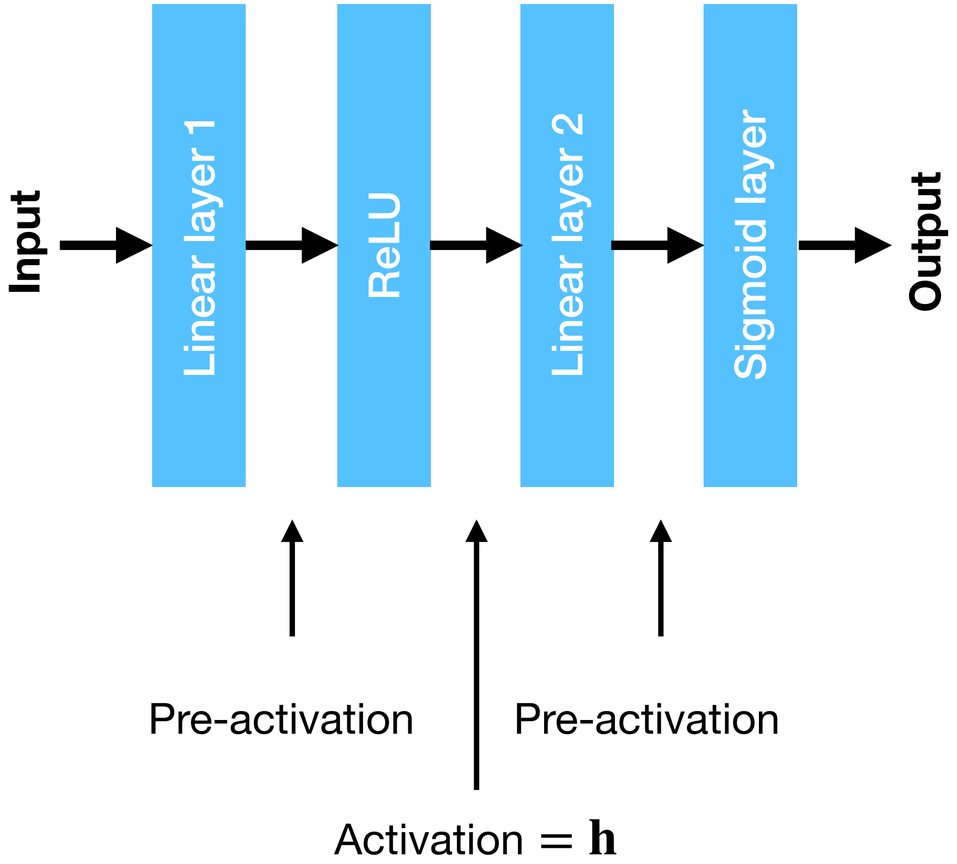
What would happen if we remove all nonlinearities (except for the output) from a neural network?





Layers as the basic building blocks





The vanishing gradient problem

- When using sigmoid non-linearity, the gradient of the loss w.r.t. to the very early layers will become very small
  - This happens during backpropagation (more on this next week!)
- Rectified linear units (ReLUs) solve this issue as they are non-saturating, thus keep the gradients large

PyTorch: quick introduction

Mitko Veta

Eindhoven University of Technology
Department of Biomedical Engineering

2024

Based on the PyTorch tutorial
https://pytorch.org/tutorials/beginner/basics/intro.html

Tensors

```
[2]  import torch
     import numpy as np
```

Initializing a Tensor

Directly from data

```
[3]  data = [[1, 2],[3, 4]]
     x_data = torch.tensor(data)
```

From a NumPy array

```
[4]  np_array = np.array(data)
     x_np = torch.from_numpy(np_array)
```

From another tensor:

The new tensor retains the properties (shape, datatype) of the argument tensor, unless explicitly overridden.

```
[5]  x_ones = torch.ones_like(x_data) # retains the properties of x_data
     print(f"Ones Tensor: \n {x_ones} \n")

     x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
     print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.9881, 0.9428],
        [0.9476, 0.1002]])
```

Attributes of tensors

Tensor attributes describe their shape, datatype, and the device on which they are stored.

[7] 0s

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Operations on tensors

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to("cuda")
```

Standard numpy-like indexing and slicing:

```
[9] tensor = torch.ones(4, 4)
    print(f"First row: {tensor[0]}")
    print(f"First column: {tensor[:, 0]}")
    print(f"Last column: {tensor[..., -1]}")
    tensor[:,1] = 0
    print(tensor)
```

```
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Operations on tensors

Joining tensors You can use `torch.cat` to concatenate a sequence of tensors along a given dimension. See also torch.stack, another tensor joining operator that is subtly different from `torch.cat`.

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

```
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

Arithmetic operations

```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will have the same value
# ``tensor.T`` returns the transpose of a tensor
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)


# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Datasets

```
[1]  import torch
     from torch.utils.data import Dataset
     from torchvision import datasets
     from torchvision.transforms import ToTensor
     import matplotlib.pyplot as plt


     training_data = datasets.FashionMNIST(
         root="data",
         train=True,
         download=True,
         transform=ToTensor()
     )

     test_data = datasets.FashionMNIST(
         root="data",
         train=False,
         download=True,
         transform=ToTensor()
     )
```

25s

This holds only for built-in datasets, for custom datasets the procedure is somewhat more complex.

In the practicals, we will use built-in datasets from another package (MedMNIST).

Data loaders

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

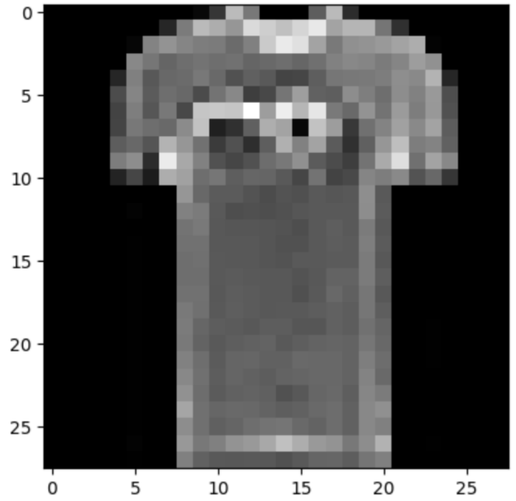
[2]
```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Double-click (or enter) to edit

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
```



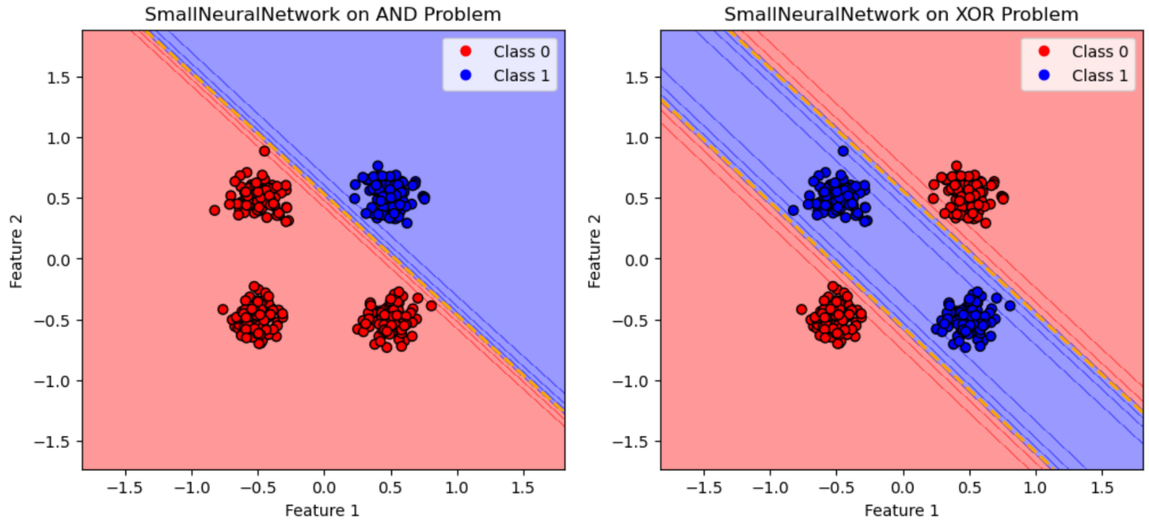

```
Label: 0
```

torch.nn

[40]:
```
class SmallNeuralNetwork(nn.Module):
    def __init__(self):
        super(SmallNeuralNetwork, self).__init__()
        self.linear_1 = nn.Linear(2, 2)
        self.hidden = nn.Sigmoid()
        self.linear_2 = nn.Linear(2, 1)
        self.output = nn.Sigmoid()

    def forward(self, x): # x is the input layer
        x = self.linear_1(x)
        h = self.hidden(x)
        logits = self.linear_2(h)
        p = self.output(logits)
        return p

demo_datasets.run(SmallNeuralNetwork());
```



torch.nn

```
[3]  class NeuralNetwork(nn.Module):
         def __init__(self):
             super().__init__()
             self.flatten = nn.Flatten()
             self.linear_relu_stack = nn.Sequential(
                 nn.Linear(28*28, 512),
                 nn.ReLU(),
                 nn.Linear(512, 512),
                 nn.ReLU(),
                 nn.Linear(512, 10),
             )

         def forward(self, x):
             x = self.flatten(x)
             logits = self.linear_relu_stack(x)
             return logits
```

Complete list of layers can be found in the documentation: https://pytorch.org/docs/stable/nn.html

Loss function and optimiser

```
[ ]: criterion = nn.BCELoss()
     optimizer = optim.SGD(model.parameters(), lr=0.005, momentum=0.95)
```

BCE stands for binary cross-entropy,
which is specific case of the negative log likelihood.

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_i'))$$

$$\text{NLL}(\beta_0, \beta_1) = -\left(\sum_{i:y_i=1} \log p(x_i) + \sum_{i':y_{i'}=0} \log \left(1 - p(x_{i'})\right) \right)$$

Training function

```python
# training function
def train(model, loader, criterion, optimizer):
    model.train()
    running_loss = 0.0
    for images, labels in loader:
        # convert labels to float for BCELoss
        labels = labels.float()

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    return running_loss / len(loader)
```

More on `backward()` in the next lecture!

Evaluation function

```python
# evaluation function
def evaluate(model, loader):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in loader:
            # convert labels to float for BCELoss
            labels = labels.float()

            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            # apply threshold to get binary predictions
            predicted = (outputs > 0.5).float()
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    avg_loss = running_loss / len(loader)
    return avg_loss, accuracy
```

The training loop

```python
# training loop
num_epochs = 50
train_losses = []
val_losses = []
val_accuracies = []

best_val_loss = float('inf')

for epoch in range(num_epochs):
    train_loss = train(model, train_loader, criterion, optimizer)
    val_loss, val_accuracy = evaluate(model, val_loader)

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

    print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f},
        Validation loss: {val_loss:.4f}, Validation accuracy: {val_accuracy:.2f}%')

    # Check for the best validation loss
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        # Save the model
        torch.save(model.state_dict(), 'best_model.pth')
        print(f'New best model saved at epoch {epoch+1} with validation loss {val_loss:.4f}')
```

Questions?