# Binary Classification: Implementation

COMP 395 – Deep Learning

```
pip install torch matplotlib scikit-learn
```

# Recap: What We Derived

**Forward pass** (for one sample $\mathbf{x}$):

$$z = \mathbf{w}^T \mathbf{x} + b$$
$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

**Loss** (for one sample):

$$L = \frac{1}{2}(\hat{y} - y)^2$$

**Gradients** (for one sample):

$$\frac{\partial L}{\partial \mathbf{w}} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y}) \cdot \mathbf{x} \qquad \frac{\partial L}{\partial b} = (\hat{y} - y) \cdot \hat{y}(1 - \hat{y})$$

Today: turn these equations into working code!

# Step 1: Load the Data

```
import torch
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

# Load dataset
data = load_breast_cancer()
X = data.data       # Shape: (569, 30)
y = data.target     # Shape: (569,) -- 0 = malignant, 1 = benign

print(f'Features: {X.shape[1]}')
print(f'Samples: {X.shape[0]}')

# Split into train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

# Step 2: Normalize the Features

**Why normalize?**

- Features have very different scales (e.g., area vs. smoothness)
- Large values cause large gradients $\rightarrow$ unstable training
- Normalization puts all features on similar scales

```
# Compute mean and std from TRAINING data only
mean = X_train.mean(dim=0)
std = X_train.std(dim=0)

# Normalize both train and test using training statistics
X_train_norm = (X_train - mean) / std
X_test_norm = (X_test - mean) / std
```

**Important:** Never compute statistics from test data!

# Think-Pair-Share: Why Training Stats Only?

**Discuss with a partner (2 min):**

We normalize the test set using the mean and std from the *training* set, not from the test set itself.

1. Why would it be "cheating" to use test set statistics?
2. Imagine you deploy this model in a hospital. When a new patient comes in, you have *one* sample to classify. How would you normalize their features? What statistics would you use?
3. What could go wrong if training and test data have very different distributions?

# Step 3: Implement Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

```
def sigmoid(z):
    return ______________________  # Fill in!
```

**Hint:** Use `torch.exp()` for $e^{-z}$

**Test your implementation:**

```
print(sigmoid(torch.tensor(0.0)))      # Should be 0.5
print(sigmoid(torch.tensor(100.0)))    # Should be ~1.0
print(sigmoid(torch.tensor(-100.0)))   # Should be ~0.0
```

# Step 4: Implement Forward Pass

For a single sample (vector $\mathbf{x}$ of length $n$):

$$z = \mathbf{w}^T\mathbf{x} + b \qquad \hat{y} = \sigma(z)$$

```
def forward(x, w, b):

    x: (n,) feature vector for one sample
    w: (n,) weight vector
    b: scalar bias
    Returns: scalar prediction

    z = ______________________  # Dot product + bias
    y_hat = ______________________  # Apply sigmoid
    return y_hat
```

**Note:** `w @ x` or `torch.dot(w, x)` computes the dot product $\mathbf{w}^T\mathbf{x}$

# Step 5: Implement Loss

For a single sample:

$$L = \frac{1}{2}(\hat{y} - y)^2$$

```
def compute_loss(y, y_hat):

    y: scalar true label (0 or 1)
    y_hat: scalar prediction
    Returns: scalar loss

    loss = ______________________  # Fill in!
    return loss
```

**This is just squared error!** The $\frac{1}{2}$ makes the derivative cleaner.

# Step 6: Implement Gradients

For a single sample:

$$\frac{\partial L}{\partial \mathbf{w}} = \underbrace{(\hat{y} - y)}_{\text{error}} \cdot \underbrace{\hat{y}(1-\hat{y})}_{\text{sigmoid derivative}} \cdot \mathbf{x} \qquad \frac{\partial L}{\partial b} = (\hat{y} - y) \cdot \hat{y}(1-\hat{y})$$

```
def compute_gradients(x, y, y_hat):

    x: (n,) input features for one sample
    y: scalar true label
    y_hat: scalar prediction
    Returns: dw (n,), db (scalar)

    error = ______________________
    sigmoid_deriv = ______________________
    delta = ______________________

    dw =
```

# Step 7: Training Loop

```
# Initialize parameters
n_features = X_train_norm.shape[1]
w = torch.zeros(n_features)
b = torch.tensor(0.0)

# Hyperparameters
alpha = 0.01
n_epochs = 100  # Number of passes through the data

# Training: loop over epochs and samples
for epoch in range(n_epochs):
    epoch_loss = 0.0

    for i in range(len(y_train)):
        x_i = X_train_norm[i]  # One sample
        y_i = y_train[i]       # Its label

        # Forward pass
```

# Step 8: Evaluate Accuracy

```
def predict(X, w, b, threshold=0.5):
    """Make predictions for multiple samples"""
    predictions = []
    for i in range(len(X)):
        y_hat = forward(X[i], w, b)
        predictions.append(1.0 if y_hat >= threshold else 0.0)
    return torch.tensor(predictions)

def accuracy(y_true, y_pred):
    """Compute classification accuracy"""
    return (y_true == y_pred).float().mean()

# Evaluate on training set
train_pred = predict(X_train_norm, w, b)
train_acc = accuracy(y_train, train_pred)
print(f'Training accuracy: {train_acc.item():.4f}')

# Evaluate on test set
```

# Visualization: Loss Curve (Boilerplate)

```
plt.figure(figsize=(10, 6))
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Over Time')
plt.grid(True)
plt.show()
```

**What should a healthy loss curve look like?**

- Decreasing over time
- Eventually flattens (converges)
- No wild oscillations

# Visualization: Feature Importance (Boilerplate)

```
# Plot the learned weights (convert to numpy for matplotlib)
plt.figure(figsize=(12, 6))
plt.bar(range(len(w)), w.numpy())
plt.xlabel('Feature Index')
plt.ylabel('Weight')
plt.title('Learned Feature Weights')
plt.xticks(range(len(w)), data.feature_names, rotation=90)
plt.tight_layout()
plt.show()
```

**Interpretation:**

- Large positive weight $\rightarrow$ feature increases P(benign)
- Large negative weight $\rightarrow$ feature increases P(malignant)

# Think-Pair-Share: What Did the Model Learn?

**After running your code, discuss with a partner (3 min):**

1. Look at your weight plot. Which features have the largest positive weights? Largest negative weights? Do these make medical sense? (You can print `data.feature_names` to see what each feature measures.)

2. Your model is a linear classifier: $z = \mathbf{w}^T\mathbf{x} + b$. This means the decision boundary is a *hyperplane* in 30-dimensional space. Why might a simple linear boundary work well for this problem?

3. If test accuracy is much lower than training accuracy, what might that indicate? What if they're both low?

# Experiments

Once your code is working:

1. **Try different learning rates:**
   - $\alpha = 0.001, 0.01, 0.1$
   - How does this affect convergence speed?
2. **Try different numbers of epochs:**
   - 10, 50, 100, 200 epochs
   - When does accuracy stop improving?

# Checkpoint

Before you leave, verify:

1. Your training loss decreases over epochs
2. You achieve $> 90\%$ test accuracy (should be $\sim$95-97%)
3. You can explain what each function does
4. You understand why normalization matters

**Looking ahead:**

- This is logistic regression – a 1-layer neural network!
- Adding more layers = deep learning
- The gradient computation gets more complex, but the principle (chain rule!) is the same

# Complete Template (1/2)

```
import torch
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

# Load and split data
data = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    data.data, data.target, test_size=0.2, random_state=42)

# Convert to tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)

# Normalize
mean, std = X_train.mean(dim=0), X_train.std(dim=0)
```

# Complete Template (2/2)

```
def compute_loss(y, y_hat):  # Single sample
    return ______

def compute_gradients(x, y, y_hat):  # Single sample
    error = ______
    sigmoid_deriv = ______
    delta = ______
    dw = ______
    db = ______
    return dw, db

# Initialize and train
w = torch.zeros(X_train_norm.shape[1])
b = torch.tensor(0.0)
alpha = 0.01

for epoch in range(100):
```