Module 6: Feedforward Neural Networks

Overview

This module introduces neural networks—models that can learn non-linear patterns by composing simple operations in layers. Starting with the XOR problem that linear models cannot solve, we build intuition for why hidden layers are necessary. You will learn about activation functions, network architecture, forward propagation, and backpropagation. The module concludes with practical neural network implementation using TensorFlow/Keras.

---

1. The XOR Problem Revisited

Why Linear Models Fail

Recall from Module 5 that logistic regression cannot solve XOR: - (0,0) □ 0 - (1,1) □ 0
- (0,1) □ 1 - (1,0) □ 1

No straight line separates the two classes.

The Solution: Hidden Layers

By adding a hidden layer, we can: 1. Transform the inputs into a new representation 2. In this new space, classes become linearly separable 3. A final linear layer then classifies correctly

This is the fundamental insight of neural networks.

---

2. Network Architecture

Components

A feedforward neural network consists of: - **Input layer**: Receives features (not really a "layer" of computation) - **Hidden layers**: Transform representations - **Output layer**: Produces predictions

Layer Details

Each layer has: - **Units (neurons)**: Each computes a weighted sum plus bias - **Activation function**: Introduces non-linearity - **Weights and biases**: Learned parameters

Notation

For layer l:

```
z^(l) = W^(l) × a^(l-1) + b^(l)    (linear combination)
a^(l) = g(z^(l))                    (activation)
```

Where: - a^(l) = activations of layer l - W^(l) = weight matrix - b^(l) = bias vector - g = activation function

---

3. Activation Functions

Why Activations?

Without activation functions, stacking linear layers is pointless:

```
W2 × (W1 × x) = (W2 × W1) × x = W × x
```

Just another linear function! Activations introduce non-linearity.

ReLU (Rectified Linear Unit)

```
ReLU(z) = max(0, z)
```

Properties: - Simple and fast to compute - Derivative is 0 or 1 (no vanishing gradients for positive values) - Can "die" if inputs are always negative - Most commonly used in hidden layers

Sigmoid

```
sigmoid(z) = 1 / (1 + exp(-z))
```

Properties: - Output in (0, 1) - Good for output layer in binary classification - Vanishing gradients for large |z| - Rarely used in hidden layers anymore

Tanh

```
tanh(z) = (exp(z) - exp(-z)) / (exp(z) + exp(-z))
```

Properties: - Output in (-1, 1) - Zero-centered (unlike sigmoid) - Still has vanishing gradient issues - Sometimes used in LSTMs

Softmax

Used for multiclass output:

```
softmax(z)_i = exp(z_i) / Σ exp(z_j)
```

---

4. Forward Propagation

The Process

Forward propagation computes the network output given an input:

1. Input: x
2. Layer 1: a^(1) = g(W^(1) × x + b^(1))
3. Layer 2: a^(2) = g(W^(2) × a^(1) + b^(2))
4. …
5. Output: ŷ = a^(L)

Example: XOR Network

Network: 2 inputs □ 2 hidden units □ 1 output

```
Input: x = [0, 1]

Hidden layer (ReLU):
  z1 = w11*0 + w12*1 + b1
  z2 = w21*0 + w22*1 + b2
  a1 = ReLU(z1), a2 = ReLU(z2)

Output layer (sigmoid):
  z_out = w1*a1 + w2*a2 + b
  y_pred = sigmoid(z_out)
```

---

5. Loss Functions for Neural Networks

Binary Classification

Binary cross-entropy:

$$L = -[y \times \log(p) + (1-y) \times \log(1-p)]$$

Multiclass Classification

Categorical cross-entropy:

$$L = -\Sigma\ y_k \times \log(p_k)$$

Regression

Mean Squared Error:

$$L = (1/n) \times \Sigma(y - \hat{y})^2$$

---

6. Backpropagation

The Core Idea

Backpropagation computes gradients of the loss with respect to all parameters by applying the chain rule layer by layer, from output back to input.

Chain Rule Review

If L depends on z through a:

$$L/z = L/a \times a/z$$

Backprop Algorithm Sketch

1. **Forward pass**: Compute all activations, save intermediate values
2. **Compute output gradient**: ∂L/∂a^(L)
3. **For each layer (L to 1)**:
   - Compute ∂L/∂z^(l) = ∂L/∂a^(l) × g'(z^(l))
   - Compute ∂L/∂W^(l) = ∂L/∂z^(l) × (a^(l-1))□
   - Compute ∂L/∂b^(l) = ∂L/∂z^(l)
   - Propagate: ∂L/∂a^(l-1) = (W^(l))□ × ∂L/∂z^(l)

Automatic Differentiation

Modern frameworks (TensorFlow, PyTorch) compute gradients automatically. You define the forward pass; the framework handles backprop.

---

7. Optimizers

SGD with Momentum

Basic SGD can be slow. Momentum accelerates by accumulating velocity:

```
v =   × v − lr × gradient
  =   + v
```

Adam

Combines momentum with adaptive learning rates. Tracks: - First moment (mean of gradients) - Second moment (variance of gradients)

Adam is the default choice for most applications.

Learning Rate Schedules

Learning rate can change during training: - **Step decay**: Reduce by factor every N epochs - **Exponential decay**: lr = lr_0 × e^(-kt) - **Warm-up**: Start low, increase, then decay

---

8. Practical Considerations

Initialization

Random initialization must be done carefully: - Too small: Vanishing gradients - Too large: Exploding gradients

Common approaches: - **Xavier/Glorot**: For tanh/sigmoid - **He**: For ReLU

Batch Normalization

Normalizes layer inputs to stabilize training: - Reduces sensitivity to initialization - Allows higher learning rates - Acts as regularization

Dropout

Randomly sets activations to zero during training: - Prevents co-adaptation - Provides regularization - Typically 0.1-0.5 rate

---

9. Building Networks with Keras

Sequential API

```
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(input_dim,)),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(num_classes, activation='softmax')
])

model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.fit(X_train, y_train, epochs=100, validation_split=0.2)
```

Key Decisions

- Number of layers (depth)
- Units per layer (width)
- Activation functions
- Optimizer and learning rate
- Regularization (dropout, L2)

Key Takeaways

1. **Hidden layers** enable learning non-linear patterns
2. **Activation functions** (ReLU, sigmoid, tanh) introduce non-linearity
3. **Forward propagation** computes predictions layer by layer
4. **Backpropagation** computes gradients using the chain rule
5. **Adam** is a robust default optimizer

6. **Initialization and normalization** are crucial for stable training
7. **Dropout** provides regularization
8. **Keras** makes building networks straightforward

Connections to Future Modules

- **Module 10**: Convolutional networks for images
- **Module 11**: Advanced architectures (functional API)
- **Module 13**: Recurrent networks and attention

Further Reading

- [Links to be added]
- Deep Learning Book, Chapter 6: Deep Feedforward Networks
- TensorFlow/Keras documentation