Deep Learning Specialization

Course 2: Improving Deep Neural Networks

Week 1: Practical Aspects of Deep Learning

---

1. Activation Functions

Sigmoid Function

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

Properties:

- Output range: (0, 1)
- Smooth gradient
- Clear prediction (0 or 1)

Usage:

- Binary classification (output layer)
- Legacy networks (mostly replaced by ReLU in hidden layers)
- When probability interpretation is needed

Limitations:

- Vanishing gradient problem for very large/small inputs
- Outputs not zero-centered
- Computationally expensive (exponential function)

ReLU (Rectified Linear Unit)

Formula:
$\text{ReLU}(z) = \max(0, z)$

Properties:

- Output range: [0, ∞)
- Computationally efficient
- No vanishing gradient for positive values

Usage:

- Default choice for hidden layers
- Faster convergence than sigmoid/tanh
- Works well in most deep networks

Limitations:

- "Dying ReLU" problem (neurons can become inactive)
- Not zero-centered

Other Common Activation Functions

Leaky ReLU:
$f(z) = \max(\alpha z, z)$ where α is a small constant (e.g., 0.01)

Tanh (Hyperbolic Tangent):
$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Which simplifies to:
$\tanh(z) = \frac{\sinh(z)}{\cosh(z)}$

Softmax (for multi-class classification):
$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$

---

2. Dense Layer Implementation with For Loop

python

```python
# GRADED FUNCTION: my_dense

def my_dense(a_in, W, b, g):
    """
    Implements a dense layer computation using a for loop.

    Arguments:
    a_in -- input vector of shape (n,)
    W -- weight matrix of shape (n, units)
    b -- bias vector of shape (units,)
    g -- activation function

    Returns:
    a_out -- output vector of shape (units,)
    """
    units = W.shape[1]
    a_out = np.zeros(units)
    for j in range(units):
        z = np.dot(a_in, W[:, j]) + b[j]
        a_out[j] = g(z)
    return a_out
```

Explanation:

- The function computes a dense layer output one neuron at a time
- For each neuron (unit):
  1. Calculates weighted sum of inputs (z = w·x + b)
  2. Applies activation function g(z)
- Returns array of all neuron outputs

Test Case:

python

```python
x_tst = 0.1*np.arange(1,3,1).reshape(2,)  # (2,) input
W_tst = 0.1*np.arange(1,7,1).reshape(2,3) # (2,3) weights
b_tst = 0.1*np.arange(1,4,1).reshape(3,)  # (3,) biases
A_tst = my_dense(x_tst, W_tst, b_tst, sigmoid)
# Output: [0.54735762 0.57932425 0.61063923]
```

---

3. Three-Layer Neural Network Implementation

```python
def my_sequential(x, W1, b1, W2, b2, W3, b3):
    """
    Implements forward propagation for a 3-layer neural network.

    Arguments:
    x -- input features
    W1, W2, W3 -- weight matrices
    b1, b2, b3 -- bias vectors

    Returns:
    a3 -- output of the network
    """
    a1 = my_dense(x,  W1, b1, sigmoid)
    a2 = my_dense(a1, W2, b2, sigmoid)
    a3 = my_dense(a2, W3, b3, sigmoid)
    return a3
```

Explanation:

- Chains three dense layers together (input → hidden1 → hidden2 → output)
- Uses sigmoid activation throughout
- Forward propagation flow:
  - Layer 1: x → a1
  - Layer 2: a1 → a2
  - Layer 3: a2 → a3 (final output)

Network Architecture:

- Input layer: Features (dimensions depend on input)
- Hidden layer 1: sigmoid activation
- Hidden layer 2: sigmoid activation
- Output layer: sigmoid activation (for binary classification)

---

4. Vectorized Dense Layer Implementation

python

```python
def my_dense_v(A_in, W, b, g):
    """
    Implements a vectorized dense layer computation.

    Arguments:
    A_in -- input matrix of shape (m, n) where m = number of examples
    W -- weight matrix of shape (n, units)
    b -- bias vector of shape (1, units)
    g -- activation function

    Returns:
    A_out -- output matrix of shape (m, units)
    """
    Z = np.matmul(A_in, W) + b
    A_out = g(Z)
    return A_out
```

Explanation:

- Processes entire batch of examples at once using matrix operations
- Much faster than for-loop implementation thanks to NumPy's optimized C backend
- Broadcasting automatically handles adding bias to each example

Test Case:

python

```python
X_tst = 0.1*np.arange(1,9,1).reshape(4,2)  # (4,2) - 4 examples, 2 features
W_tst = 0.1*np.arange(1,7,1).reshape(2,3)  # (2,3) - 2 inputs, 3 outputs
b_tst = 0.1*np.arange(1,4,1).reshape(1,3)  # (1,3) - bias for 3 outputs
A_tst = my_dense_v(X_tst, W_tst, b_tst, sigmoid)
# Output:
# [[0.54735762 0.57932425 0.61063923]
#  [0.57199613 0.61301418 0.65248946]
#  [0.5962827  0.64565631 0.6921095 ]
#  [0.62010643 0.67699586 0.72908792]]
```

---

5. Vectorized Sequential Model

```python
def my_sequential_v(X, W1, b1, W2, b2, W3, b3):
    """
    Implements vectorized forward propagation for a 3-layer neural network.

    Arguments:
    X -- input features matrix (m, n_x)
    W1, W2, W3 -- weight matrices
    b1, b2, b3 -- bias vectors

    Returns:
    A3 -- output of the network
    """
    A1 = my_dense_v(X,  W1, b1, sigmoid)
    A2 = my_dense_v(A1, W2, b2, sigmoid)
    A3 = my_dense_v(A2, W3, b3, sigmoid)
    return A3
```

Explanation:

- Fully vectorized version of the sequential model
- Efficiently processes entire batches of examples at once
- Each layer takes the entire batch output from previous layer
- Significant performance improvement for large datasets

Dimensions in a Typical Case:

- X: (m, n_x) - m examples with n_x features
- W1: (n_x, n_h1) - First hidden layer with n_h1 units
- A1: (m, n_h1) - Output of first hidden layer
- W2: (n_h1, n_h2) - Second hidden layer with n_h2 units
- A2: (m, n_h2) - Output of second hidden layer
- W3: (n_h2, n_y) - Output layer with n_y units
- A3: (m, n_y) - Final output

---

6. NumPy Broadcasting - Key Concept

Broadcasting is a powerful NumPy feature that allows operations between arrays of different shapes.

Key Broadcasting Rules:

1. Arrays are compatible for broadcasting if:
   - They have the same shape, or
   - One of the dimensions is 1
2. Broadcasting expands dimensions with size 1 to match the other array's size

Common Broadcasting Examples:

Example 1: Vector + Scalar

python

```
a = np.array([1, 2, 3])  # (3,)
b = 5                    # scalar
c = a + b                # (3,) + scalar = (3,)
# Result: [6, 7, 8]
```

Example 2: Matrix + Vector (column)

python

```
a = np.array([[1, 2, 3], [4, 5, 6]])  # (2,3)
b = np.array([10, 20, 30]).reshape(1, -1)  # (1,3)
c = a + b  # (2,3) + (1,3) = (2,3)
# Result: [[11, 22, 33], [14, 25, 36]]
```

Example 3: Matrix + Vector (row)

python

```
a = np.array([[1, 2, 3], [4, 5, 6]])  # (2,3)
b = np.array([10, 20]).reshape(-1, 1)  # (2,1)
c = a + b  # (2,3) + (2,1) = (2,3)
# Result: [[11, 12, 13], [24, 25, 26]]
```

Broadcasting in Neural Networks:

- Adding bias to entire batch of examples without looping
- Applying element-wise operations across batches
- Efficiently computing gradient updates

---

7. TensorFlow Implementation Comparison

Equivalent Dense Layer in TensorFlow:

```python
import tensorflow as tf

# Define a dense layer with 3 units and sigmoid activation
layer = tf.keras.layers.Dense(units=3, activation='sigmoid')

# Apply to input data
output = layer(input_data)
```

TensorFlow vs. Manual Implementation:

Advantages of TensorFlow:

- Automatic differentiation (backpropagation)
- Hardware acceleration (GPU/TPU support)
- Optimized for performance
- Built-in regularization, initialization methods
- Production-ready deployment options

When to Use Manual Implementation:

- Learning fundamentals
- Understanding internal mechanisms
- Custom algorithm development
- Debugging specific computation steps

Building a Complete Model in TensorFlow:

python

```python
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=25, activation='relu', input_shape=(n_x,)),
    tf.keras.layers.Dense(units=15, activation='relu'),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])

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

model.fit(X_train, y_train, epochs=10, batch_size=32)
```

---

8. Forward Propagation - Mathematical Formulation

Notation

Let's first establish our notation:

- L = total number of layers in the network
- $n^{[l]}$ = number of units in layer l
- $W^{[l]}$ = weights matrix for layer l of shape $(n^{[l]}, n^{[l-1]})$
- $b^{[l]}$ = bias vector for layer l of shape $(n^{[l]}, 1)$
- $Z^{[l]}$ = linear output for layer l
- $A^{[l]}$ = activation output for layer l
- $g^{[l]}$ = activation function for layer l
- m = number of training examples

General Formulation for a Single Example

For a neural network with L layers:

1. **Input**: x becomes $A^{[0]}$
2. **Forward Propagation**: For layer $l = 1$ to L: $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ $A^{[l]} = g^{[l]}(Z^{[l]})$
3. **Output**: $A^{[L]}$ is the network's final prediction

Example: Three-Layer Network (Single Example)

For a neural network with one input layer, two hidden layers, and one output layer:

Layer 1 (First Hidden Layer):
$Z^{[1]} = W^{[1]}X + b^{[1]}$
$A^{[1]} = g^{[1]}(Z^{[1]})$

Where:

- X is the input features of shape $(n^{[0]}, 1)$
- $W^{[1]}$ is of shape $(n^{[1]}, n^{[0]})$
- $b^{[1]}$ is of shape $(n^{[1]}, 1)$
- $Z^{[1]}$ and $A^{[1]}$ are of shape $(n^{[1]}, 1)$

Layer 2 (Second Hidden Layer):
$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$
$A^{[2]} = g^{[2]}(Z^{[2]})$

Where:

- $W^{[2]}$ is of shape $(n^{[2]}, n^{[1]})$
- $b^{[2]}$ is of shape $(n^{[2]}, 1)$
- $Z^{[2]}$ and $A^{[2]}$ are of shape $(n^{[2]}, 1)$

Layer 3 (Output Layer):
$Z^{[3]} = W^{[3]}A^{[2]} + b^{[3]}$
$A^{[3]} = g^{[3]}(Z^{[3]})$

Where:

- $W^{[3]}$ is of shape $(n^{[3]}, n^{[2]})$
- $b^{[3]}$ is of shape $(n^{[3]}, 1)$
- $Z^{[3]}$ and $A^{[3]}$ are of shape $(n^{[3]}, 1)$

Vectorized Implementation (Multiple Examples)

For m examples:

- X is of shape $(n^{[0]}, m)$
- $Z^{[l]}$ and $A^{[l]}$ are of shape $(n^{[l]}, m)$

The forward propagation equations remain the same, but now process all examples at once:
$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$
$A^{[l]} = g^{[l]}(Z^{[l]})$

Where $b^{[l]}$ is broadcast to add to each example.

---

9. Practical Tips for Neural Network Implementation

Initialization Strategies:

- Random initialization (avoids symmetry breaking)
- He initialization for ReLU: $W^{[l]} = \text{randn}(n^{[l-1]}, n^{[l]}) \times \sqrt{\frac{2}{n^{[l-1]}}}$
- Xavier initialization for sigmoid/tanh: $W^{[l]} = \text{randn}(n^{[l-1]}, n^{[l]}) \times \sqrt{\frac{1}{n^{[l-1]}}}$

Normalization:

- Standardize input features (zero mean, unit variance): $x_{\text{norm}} = \frac{x - \mu}{\sigma}$
- Batch normalization for internal layers: $\hat{z}^{(i)} = \frac{z^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ $\tilde{z}^{(i)} = \gamma \hat{z}^{(i)} + \beta$
- Input normalization speeds up training significantly

Avoid Vanishing/Exploding Gradients:

- Use ReLU activation (for hidden layers)
- Proper weight initialization
- Gradient clipping when necessary: $\text{if } ||\nabla J|| > \text{threshold} : \nabla J \leftarrow \frac{\text{threshold}}{||\nabla J||} \nabla J$

Debugging Strategies:

- Verify layer dimensions
- Start with smaller networks
- Use a learning rate finder: $\alpha = \alpha_0 \times \beta^t$
- Track training/validation metrics

---

10. Looking Ahead: Backpropagation

The next week's materials will cover:

- Backpropagation algorithm
- Computing gradients efficiently
- Implementing gradient descent
- Optimization algorithms (momentum, RMSprop, Adam)
- Learning rate scheduling

Key Backpropagation Concepts:

- Chain rule for computing derivatives: $\frac{dJ}{dW^{[l]}} = \frac{dJ}{dZ^{[l]}} \frac{dZ^{[l]}}{dW^{[l]}}$
- Gradient flow through the network
- Updating weights to minimize loss: $W^{[l]} := W^{[l]} - \alpha \frac{dJ}{dW^{[l]}}$

Summary

In Week 1, you've learned:

1. Different activation functions and their properties
2. How to implement a neural network layer from scratch
3. Vectorization techniques for efficient computation
4. Building a complete neural network with forward propagation
5. NumPy broadcasting for neural network operations
6. Comparison between manual implementation and TensorFlow

These fundamental skills provide the foundation for understanding how deep learning models are built, optimized, and deployed.

Practice Exercises

1. Implement a dense layer with tanh activation
2. Create a 4-layer neural network with mixed activations (ReLU for hidden, sigmoid for output)
3. Vectorize a network that processes multiple examples at once
4. Compare the performance of for-loop vs. vectorized implementations
5. Implement a neural network for a simple classification problem

---

References

1. Coursera Deep Learning Specialization by Andrew Ng
2. Deep Learning by Goodfellow, Bengio, and Courville
3. Neural Networks and Deep Learning by Michael Nielsen
