Neural Networks for Handwritten Digit Recognition - Complete Lab Guide

Overview

This lab implements a neural network to recognize handwritten digits 0-9 using multiclass classification with softmax activation.

---

1. Package Imports and Setup

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import linear, relu, sigmoid
import matplotlib.pyplot as plt
plt.style.use('./deeplearning.mplstyle')
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
tf.autograph.set_verbosity(0)
from public_tests import *
from autils import *
from lab_utils_softmax import plt_softmax
np.set_printoptions(precision=2)
```

Explanation:

- **NumPy**: Fundamental package for numerical computations
- **TensorFlow**: Machine learning framework for building neural networks
- **Sequential**: Creates a linear stack of layers
- **Dense**: Fully connected neural network layer
- **Logging**: Suppresses TensorFlow warnings for cleaner output

---

2. ReLU Activation Function

Theory

ReLU (Rectified Linear Unit) is defined as:

```
f(x) = max(0, x)
```

Mathematical Formula:

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

Where:

- `z` is the input to the activation function
- `a` is the output (activation)

Code Implementation:

```python
plt_act_trio()  # Visualizes ReLU, sigmoid, and linear activations
```

Why ReLU?

- **Non-linear**: Enables the network to learn complex patterns
- **Computationally efficient**: Simple max operation
- **Solves vanishing gradient**: Gradient is either 0 or 1
- **Sparsity**: Many neurons output 0, making the network sparse

Expected Output:

```
[Visualization showing ReLU function - linear for positive values, zero for negative]
```

---

3. Softmax Function

Theory

Softmax converts a vector of real numbers into a probability distribution.

Mathematical Formula:

```
a_j = e^(z_j) / Σ(k=0 to N-1) e^(z_k)
```

Where:

- `z_j` is the j-th input
- `a_j` is the j-th output probability
- `N` is the number of classes

Code Implementation:

python

```python
def my_softmax(z):
    """ Softmax converts a vector of values to a probability distribution.
    Args:
      z (ndarray (N,))  : input data, N features
    Returns:
      a (ndarray (N,))  : softmax of z
    """
    z_max = np.max(z)              # for numerical stability
    exp_z = np.exp(z - z_max)      # subtract max value
    sum_exp_z = np.sum(exp_z)      # sum of exponentials
    a = exp_z / sum_exp_z          # probability distribution
    return a

# Test the implementation
z = np.array([1., 2., 3., 4.])
a = my_softmax(z)
atf = tf.nn.softmax(z)
print(f"my_softmax(z):         {a}")
print(f"tensorflow softmax(z): {atf}")
```

Expected Output:

```
my_softmax(z):         [0.03 0.09 0.24 0.64]
tensorflow softmax(z): [0.03 0.09 0.24 0.64]
All tests passed.
```

Line-by-Line Explanation:

1. `z_max = np.max(z)`: Find maximum value for numerical stability
2. `exp_z = np.exp(z - z_max)`: Compute exponentials (subtracting max prevents overflow)
3. `sum_exp_z = np.sum(exp_z)`: Sum all exponentials (denominator)
4. `a = exp_z / sum_exp_z`: Normalize to get probabilities

Numerical Stability: Subtracting the maximum value prevents overflow when computing exponentials of large numbers.

---

4. Dataset Loading and Exploration

Loading Data:

python

```
X, y = load_data()
```

Data Exploration:

python

```
print('The first element of X is: ', X[0])
print('The first element of y is: ', y[0,0])
print('The last element of y is: ', y[-1,0])
print('The shape of X is: ' + str(X.shape))
print('The shape of y is: ' + str(y.shape))
```

Expected Output:

```
The first element of X is:  [0.00e+00 0.00e+00 0.00e+00 ... (400 pixel values)]
The first element of y is:  0
The last element of y is:  9
The shape of X is: (5000, 400)
The shape of y is: (5000, 1)
```

Data Structure:

- **X**: 5000 examples × 400 features (20×20 pixel images flattened)
- **y**: 5000 labels (digits 0-9)
- Each image is 20×20 grayscale pixels "unrolled" into a 400-dimensional vector

Data Visualization:

python

```python
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

m, n = X.shape
fig, axes = plt.subplots(8,8, figsize=(5,5))
fig.tight_layout(pad=0.13,rect=[0, 0.03, 1, 0.91])

for i,ax in enumerate(axes.flat):
    # Select random indices
    random_index = np.random.randint(m)

    # Reshape the flattened image back to 20x20
    X_random_reshaped = X[random_index].reshape((20,20)).T

    # Display the image
    ax.imshow(X_random_reshaped, cmap='gray')

    # Display the label above the image
    ax.set_title(y[random_index,0])
    ax.set_axis_off()
    fig.suptitle("Label, image", fontsize=14)
```

Expected Output:

[8×8 grid of handwritten digit images with their corresponding labels displayed above each image]

---

5. Neural Network Architecture

Network Structure:

```
Input Layer: 400 units (20×20 pixels)
    ↓
Hidden Layer 1: 25 units (ReLU activation)
    ↓
Hidden Layer 2: 15 units (ReLU activation)
    ↓
Output Layer: 10 units (Linear activation)
    ↓
Softmax (applied during loss calculation)
```

Parameter Dimensions:

- **Layer 1**: W_1 shape = (400, 25), b_1 shape = (25,)
- **Layer 2**: W_2 shape = (25, 15), b_2 shape = (15,)
- **Layer 3**: W_3 shape = (15, 10), b_3 shape = (10,)

Total Parameters: 10,575

---

6. Model Implementation

python

```python
tf.random.set_seed(1234)  # for consistent results
model = Sequential([
    tf.keras.Input(shape=(400,)),                    # Input layer
    Dense(25, activation='relu', name="L1"),         # First hidden layer
    Dense(15, activation='relu', name="L2"),         # Second hidden layer
    Dense(10, activation='linear', name="L3")        # Output layer
], name = "my_model")

model.summary()
```

Expected Output:

```
Model: "my_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 L1 (Dense)                  (None, 25)                10025

 L2 (Dense)                  (None, 15)                390

 L3 (Dense)                  (None, 10)                160

=================================================================
Total params: 10,575
Trainable params: 10,575
Non-trainable params: 0
_________________________________________________________________
```

Weight Verification:

python

```
[layer1, layer2, layer3] = model.layers

W1,b1 = layer1.get_weights()
W2,b2 = layer2.get_weights()
W3,b3 = layer3.get_weights()

print(f"W1 shape = {W1.shape}, b1 shape = {b1.shape}")
print(f"W2 shape = {W2.shape}, b2 shape = {b2.shape}")
print(f"W3 shape = {W3.shape}, b3 shape = {b3.shape}")
```

Expected Output:

```
W1 shape = (400, 25), b1 shape = (25,)
W2 shape = (25, 15), b2 shape = (15,)
W3 shape = (15, 10), b3 shape = (10,)
```

---

7. Model Compilation and Training

Compilation:

```python
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
)
```

Components Explained:

SparseCategoricalCrossentropy Loss:

Formula:

$$\text{Loss} = -\Sigma\ y\_true * \log(y\_pred)$$

- `from_logits=True`: Indicates that output layer produces raw logits (not probabilities)
- **Sparse**: Labels are integers (0, 1, 2, ..., 9) not one-hot encoded

Adam Optimizer:

Adam (Adaptive Moment Estimation) combines momentum and adaptive learning rates:

Formulas:

$$m\_t = \beta_1 * m\_\{t-1\} + (1 - \beta_1) * g\_t$$
$$v\_t = \beta_2 * v\_\{t-1\} + (1 - \beta_2) * g\_t^2$$
$$\hat{m}\_t = m\_t / (1 - \beta_1 \hat{}\ t)$$
$$\hat{v}\_t = v\_t / (1 - \beta_2 \hat{}\ t)$$
$$\theta\_\{t+1\} = \theta\_t - \alpha * \hat{m}\_t / (\sqrt{\hat{v}\_t} + \varepsilon)$$

Where:

- g_t = gradient at time t
- m_t = first moment estimate (momentum)
- v_t = second moment estimate (adaptive learning rate)
- α = learning rate (0.001)
- β_1 = 0.9, β_2 = 0.999, ε = 1e-8

Training:

```python
history = model.fit(X, y, epochs=40)
```

Expected Output:

```
Epoch 1/40
157/157 [==============================] - 0s 1ms/step - loss: 2.2770
Epoch 2/40
157/157 [==============================] - 0s 1ms/step - loss: 1.8823
...
Epoch 40/40
157/157 [==============================] - 0s 1ms/step - loss: 0.0891
```

Training Process:

- **Epochs**: Complete passes through the dataset (40 times)
- **Batches**: Data split into batches of 32 examples (157 batches total)
- **Loss**: Decreases over time, indicating learning

Loss Visualization:

```python
plot_loss_tf(history)
```

Expected Output:

```
[Graph showing decreasing loss over 40 epochs]
```

---

8. Making Predictions

Single Prediction:

python

```python
image_of_two = X[1015]
display_digit(image_of_two)

prediction = model.predict(image_of_two.reshape(1,400))
print(f" predicting a Two: \n{prediction}")
print(f" Largest Prediction index: {np.argmax(prediction)}")
```

Expected Output:

```
predicting a Two:
[[ -7.99   -2.23    0.77   -2.41  -11.66  -11.15   -9.53   -3.36   -4.42   -7.17]]
Largest Prediction index: 2
```

Converting to Probabilities:

python

```python
prediction_p = tf.nn.softmax(prediction)
print(f" predicting a Two. Probability vector: \n{prediction_p}")
print(f"Total of predictions: {np.sum(prediction_p):0.3f}")

yhat = np.argmax(prediction_p)
print(f"np.argmax(prediction_p): {yhat}")
```

Expected Output:

```
predicting a Two. Probability vector:
[[1.42e-04 4.49e-02 8.98e-01 3.76e-02 3.61e-06 5.97e-06 3.03e-05 1.44e-02
  5.03e-03 3.22e-04]]
Total of predictions: 1.000
np.argmax(prediction_p): 2
```

Explanation:

- **Raw logits**: Output from linear layer (can be any real number)
- **Softmax**: Converts logits to probabilities (sum to 1.0)
- **argmax**: Returns index of highest probability (predicted class)

---

9. Model Evaluation

Visual Comparison:

python

```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

m, n = X.shape
fig, axes = plt.subplots(8,8, figsize=(5,5))
fig.tight_layout(pad=0.13,rect=[0, 0.03, 1, 0.91])

for i,ax in enumerate(axes.flat):
    # Select random indices
    random_index = np.random.randint(m)

    # Reshape and display image
    X_random_reshaped = X[random_index].reshape((20,20)).T
    ax.imshow(X_random_reshaped, cmap='gray')

    # Make prediction
    prediction = model.predict(X[random_index].reshape(1,400))
    prediction_p = tf.nn.softmax(prediction)
    yhat = np.argmax(prediction_p)

    # Display true label and prediction
    ax.set_title(f"{y[random_index,0]},{yhat}",fontsize=10)
    ax.set_axis_off()
fig.suptitle("Label, yhat", fontsize=14)
plt.show()
```

Expected Output:

```
[8×8 grid showing images with format "true_label,predicted_label" above each image]
```

Error Analysis:

python

```
print(f"{display_errors(model,X,y)} errors out of {len(X)} images")
```

Expected Output:

```
15 errors out of 5000 images
```

Accuracy: (5000 - 15) / 5000 = 99.7%

---

10. Key Concepts Summary

ReLU Activation:

- **Formula**: f(x) = max(0, x)
- **Purpose**: Introduces non-linearity, prevents vanishing gradients
- **Advantage**: Computationally efficient, sparse activation

Softmax Function:

- **Formula**: softmax(z_i) = e^(z_i) / Σ(e^(z_j))
- **Purpose**: Converts logits to probability distribution
- **Properties**: Output sums to 1, all values between 0 and 1

Adam Optimizer:

- **Purpose**: Adaptive learning rate optimization
- **Advantage**: Combines momentum with per-parameter learning rates
- **Default parameters**: β_1=0.9, β_2=0.999, ε=1e-8

Neural Network Architecture:

- **Input**: 400 features (20×20 pixels)
- **Hidden layers**: 25 and 15 units with ReLU
- **Output**: 10 units (one per digit class)
- **Total parameters**: 10,575

Training Process:

- **Loss function**: Sparse Categorical Crossentropy
- **Optimization**: Adam with learning rate 0.001
- **Epochs**: 40 complete passes through data
- **Final accuracy**: 99.7% on training data

This neural network successfully learns to classify handwritten digits with high accuracy using a relatively simple architecture and modern optimization techniques.