

code.py

```
1 import tensorflow as tf
2 print(tf.__version__)
3
4 import os
5 import time
6
7 import numpy as np # linear algebra
8 import matplotlib.pyplot as plt
9 from tensorflow.keras.layers import Dense, LeakyReLU
10 from tensorflow.keras.optimizers import Adam, RMSprop
11 import warnings
12 warnings.filterwarnings('ignore')
13
14
15 # generate original training and test data
16 img_size = 28
17 n_classes = 10
18
19 #MNIST data image of shape 28*28=784
20 input_size = 784
21
22 # 0-9 digits recognition (labels)
23 output_size = 10
24
25 #-----
26 #option 1: load MNIST dataset
27 #from tensorflow.examples.tutorials.mnist import input_data
28 #mnist = input_data.read_data_sets("data/", one_hot=True)
29
30
31 #-----
32 #option 2: load MNIST dataset
33 print('\nLoading MNIST')
34 mnist = tf.keras.datasets.mnist
35 (x_train, y_train), (x_test, y_test) = mnist.load_data()
36
37 x_train = np.reshape(x_train, [-1, img_size*img_size])
38 x_train = x_train.astype(np.float32)/255
39
40 x_test = np.reshape(x_test, [-1, img_size*img_size])
41 x_test = x_test.astype(np.float32)/255
42
43 to_categorical = tf.keras.utils.to_categorical
44 y_train = to_categorical(y_train)
45 y_test = to_categorical(y_test)
46
47 print('\nSplitting data')
48
49 ind = np.random.permutation(x_train.shape[0])
50 x_train, y_train = x_train[ind], y_train[ind]
51
```

```

52 # 10% for validation
53 validationPct = 0.1
54 n = int(x_train.shape[0] * (1-validationPct))
55 x_valid = x_train[n:]
56 x_train = x_train[:n]
57 #
58 y_valid = y_train[n:]
59 y_train = y_train[:n]
60
61 train_num_examples = x_train.shape[0]
62 valid_num_examples = x_valid.shape[0]
63 test_num_examples = x_test.shape[0]
64
65 print(train_num_examples, valid_num_examples, test_num_examples)
66
67 # Global Parameters
68 #-----
69 # learning rate
70 learning_rate = 0.05
71
72 #training_epochs = 1000
73 #batch_size = 30
74
75 training_epochs = 100
76 batch_size = 50
77
78 display_step = 10
79
80 #Network Architecture
81 # -----
82 #
83 # Two hidden layers
84 #
85 #-----
86 # number of neurons in layer 1
87 n_hidden_1 = 200
88 # number of neurons in layer 2
89 n_hidden_2 = 300
90
91 #MNIST data image of shape 28*28=784
92 input_size = 784
93
94 # 0-9 digits recognition (labels)
95 output_size = 10
96
97 def loss_2(output, y):
98     """
99     Computes softmax cross entropy between logits and labels and returns the loss.
100
101     Input:
102         - output: the output (logits) of the inference function (shape: batch_size *
103         num_of_classes)
104         - y: true labels for the sample batch (shape: batch_size * num_of_classes)
105     Output:

```

```

105         - loss: the scalar loss value for the batch
106         """
107         # Computes softmax cross entropy between logits (output) and true labels (y)
108         xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=output, labels=y)
109
110         # Return the mean cross-entropy loss across the batch
111         loss = tf.reduce_mean(xentropy)
112
113         return loss
114
115     def evaluate(output, y):
116         """
117         Evaluates the accuracy on the validation set.
118         Input:
119             - output: prediction vector of the network for the validation set
120             - y: true value for the validation set
121         Output:
122             - accuracy: accuracy on the validation set (scalar between 0 and 1)
123         """
124         # Check if the predicted class equals the true class
125         correct_prediction = tf.equal(tf.argmax(output, 1), tf.argmax(y, 1))
126
127         # Compute accuracy as the mean of correct predictions
128         accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
129
130         # Log validation accuracy using TensorFlow summary (if needed)
131         with tf.summary.create_file_writer('./logs/validation').as_default():
132             tf.summary.scalar("validation_error", 1.0 - accuracy, step=0)
133
134         return accuracy
135
136     def build_model(architecture, neurons_per_layer):
137         model = tf.keras.Sequential()
138         input_shape = (input_size,)
139
140         for i, activation in enumerate(architecture):
141             if activation == 'leaky_relu':
142                 model.add(Dense(neurons_per_layer, input_shape=input_shape if i == 0 else None))
143                 model.add(LeakyReLU(negative_slope=0.01))
144             else:
145                 model.add(Dense(neurons_per_layer, activation=activation,
146                                input_shape=input_shape if i == 0 else None))
147
148         model.add(Dense(output_size))
149         return model
150
151     # Function to plot training history
152     def plot_training_history(history, architecture_name, neurons_per_layer, optimizer_name,
153                               epochs):
154         plt.figure(figsize=(12, 5))
155
156         # Add supertitle for the architecture configuration
157         plt.suptitle(f"Architecture: {architecture_name}, Neurons: {neurons_per_layer},
158                      Optimizer: {optimizer_name}")

```

```

156 # Plot training and validation loss
157 plt.subplot(1, 2, 1)
158 plt.plot(history['loss'], label='Training Loss')
159 plt.plot(history['val_loss'], label='Validation Loss')
160 plt.xlabel('Epochs')
161 plt.ylabel('Loss')
162 plt.legend()
163 plt.title('Training and Validation Loss')
164 plt.xticks(np.arange(0, epochs, 1))
165
166 # Plot training and validation accuracy
167 plt.subplot(1, 2, 2)
168 plt.plot(history['accuracy'], label='Training Accuracy')
169 plt.plot(history['val_accuracy'], label='Validation Accuracy')
170 plt.xlabel('Epochs')
171 plt.ylabel('Accuracy')
172 plt.legend()
173 plt.title('Training and Validation Accuracy')
174 plt.xticks(np.arange(0, epochs, 1))
175
176 plt.show()
177
178 def visualize_error_surface(model, x_train, y_train, start_points):
179     """
180     Visualizes the error surface by interpolating between the starting weights and final
181     trained weights.
182
183     Parameters:
184     - model: Trained model to use for error surface visualization.
185     - x_train: Training data features.
186     - y_train: Training data labels.
187     - start_points: List of initial weights (random starting points) to interpolate from.
188     """
189     loss_function = tf.keras.losses.CategoricalCrossentropy()
190
191     # Get the final trained weights
192     final_weights = model.get_weights()
193
194     # Define the interpolation factor (alpha) values
195     alphas = np.linspace(0, 1, 100)
196
197     # Plot the error surface for each starting point
198     for idx, start_weights in enumerate(start_points):
199         # Store the losses along the interpolation path
200         losses = []
201
202         # Interpolate between the starting weights and final weights
203         for alpha in alphas:
204             # Compute interpolated weights
205             interpolated_weights = [(1 - alpha) * start + alpha * final
206                                     for start, final in zip(start_weights, final_weights)]
207
208             # Set interpolated weights in the model
209             model.set_weights(interpolated_weights)

```

```

209         # Compute loss for the interpolated model
210         y_pred = model(x_train)
211         loss = loss_function(y_train, y_pred).numpy()
212         losses.append(loss)
213
214     # Plot the losses for this interpolation path
215     plt.plot(alphas, losses, label=f"Start Point {idx + 1}")
216
217     plt.xlabel("Interpolation Factor (Alpha)")
218     plt.ylabel("Loss")
219     plt.title("Error Surface by Linear Interpolation")
220     plt.legend()
221     plt.show()
222
223 def training_testing(architecture_name, neurons_per_layer, optimizer_name):
224     start_time = time.time()
225
226     # Define inputs directly (no need for placeholders)
227     input_size = 784
228     output_size = 10
229     batch_size = 128
230     training_epochs = 20
231     display_step = 5
232
233     # Instantiate the model with the architecture parameters
234     model = build_model(architecture_name, neurons_per_layer)
235
236     # Define optimizer
237     if optimizer_name == 'Adam':
238         optimizer = tf.optimizers.Adam()
239     elif optimizer_name == 'RMSprop':
240         optimizer = tf.optimizers.RMSprop()
241
242     # Define the checkpoint manager
243     checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
244     checkpoint_manager = tf.train.CheckpointManager(checkpoint, './logs/multi_layer',
max_to_keep=5)
245
246     # Training loop
247     history = {'loss': [], 'accuracy': [], 'val_loss': [], 'val_accuracy': []}
248
249     for epoch in range(training_epochs):
250         avg_cost = 0.
251         total_batch = int((train_num_examples + batch_size - 1) / batch_size)
252
253         for i in range(total_batch):
254             start = i * batch_size
255             end = min(train_num_examples, start + batch_size)
256             minibatch_x = x_train[start:end]
257             minibatch_y = y_train[start:end]
258
259             # Define training step using GradientTape
260             with tf.GradientTape() as tape:
261                 output = model(minibatch_x)

```

```

262         cost = loss_2(output, minibatch_y)
263
264         # Compute gradients and apply them
265         gradients = tape.gradient(cost, model.trainable_variables)
266         optimizer.apply_gradients(zip(gradients, model.trainable_variables))
267
268         avg_cost += cost.numpy() / total_batch
269
270         # Append metrics for plotting
271         history['loss'].append(avg_cost)
272         accuracy = evaluate(model(x_train), y_train)
273         history['accuracy'].append(accuracy)
274
275         val_loss = loss_2(model(x_valid), y_valid).numpy()
276         val_accuracy = evaluate(model(x_valid), y_valid)
277         history['val_loss'].append(val_loss)
278         history['val_accuracy'].append(val_accuracy)
279
280         if (epoch+1) % display_step == 0:
281             print(f"Epoch: {(epoch+1):2d}, cost={avg_cost:.7f}, Validation Error={1-
accuracy:.7f}, Training Accuracy={accuracy}, Validation Accuracy={val_accuracy}")
282             checkpoint_manager.save()
283
284         # Final test accuracy
285         accuracy = evaluate(model(x_test), y_test)
286         print("Test Accuracy:", accuracy)
287
288         elapsed_time = time.time() - start_time
289         print(f'Execution time (seconds) was {elapsed_time:.3f}')
290
291         # Call the plotting function to visualize training history
292         plot_training_history(history, architecture_name, neurons_per_layer, optimizer_name,
training_epochs)
293
294         # Generate a few random starting points for error surface visualization
295         start_points = [ [np.random.normal(size=w.shape) for w in model.get_weights()] for _ in
range(3)]
296
297         # Visualize error surface by linear interpolation
298         visualize_error_surface(model, x_train, y_train, start_points)
299
300 def training_testing_diff_lr(architecture_name, neurons_per_layer, optimizer_name,
learning_rates):
301     results = [] # Store validation accuracy and elapsed time for each learning rate
302
303     # Define inputs
304     input_size = 784
305     output_size = 10
306     batch_size = 128
307     training_epochs = 20
308     display_step = 5
309
310     # Loop through different learning rates
311     for lr in learning_rates:

```

```

312     print(f"\nTraining with learning rate: {lr}")
313
314     # Start timer for this learning rate
315     start_time = time.time()
316
317     # Instantiate the model with the architecture parameters
318     model = build_model(architecture_name, neurons_per_layer)
319
320     # Define optimizer with the specific learning rate
321     if optimizer_name == 'Adam':
322         optimizer = tf.optimizers.Adam(learning_rate=lr)
323     elif optimizer_name == 'RMSprop':
324         optimizer = tf.optimizers.RMSprop(learning_rate=lr)
325
326     # Define the checkpoint manager
327     checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
328     checkpoint_manager = tf.train.CheckpointManager(checkpoint, './logs/multi_layer',
max_to_keep=5)
329
330     # Training loop
331     history = {'loss': [], 'accuracy': [], 'val_loss': [], 'val_accuracy': []}
332
333     for epoch in range(training_epochs):
334         avg_cost = 0.
335         total_batch = int((train_num_examples + batch_size - 1) / batch_size)
336
337         for i in range(total_batch):
338             start = i * batch_size
339             end = min(train_num_examples, start + batch_size)
340             minibatch_x = x_train[start:end]
341             minibatch_y = y_train[start:end]
342
343             # Define training step using GradientTape
344             with tf.GradientTape() as tape:
345                 output = model(minibatch_x)
346                 cost = loss_2(output, minibatch_y)
347
348             # Compute gradients and apply them
349             gradients = tape.gradient(cost, model.trainable_variables)
350             optimizer.apply_gradients(zip(gradients, model.trainable_variables))
351
352             avg_cost += cost.numpy() / total_batch
353
354         # Append metrics for plotting
355         history['loss'].append(avg_cost)
356         accuracy = evaluate(model(x_train), y_train)
357         history['accuracy'].append(accuracy)
358
359         val_loss = loss_2(model(x_valid), y_valid).numpy()
360         val_accuracy = evaluate(model(x_valid), y_valid)
361         history['val_loss'].append(val_loss)
362         history['val_accuracy'].append(val_accuracy)
363
364         if (epoch+1) % display_step == 0:

```

```

365         print(f"Epoch: {(epoch+1):2d}, cost={avg_cost:.7f}, Validation Error={1-
accuracy:.7f}, Training Accuracy={accuracy}, Validation Accuracy={val_accuracy}")
366         checkpoint_manager.save()
367
368     # Final test accuracy
369     test_accuracy = evaluate(model(x_test), y_test)
370     print("Test Accuracy:", test_accuracy)
371
372     # Record elapsed time
373     elapsed_time = time.time() - start_time
374     print(f'Execution time (seconds) was {elapsed_time:.3f}')
375
376     # Store results for this learning rate
377     results.append({
378         'learning_rate': lr,
379         'final_val_accuracy': val_accuracy,
380         'test_accuracy': test_accuracy,
381         'time': elapsed_time
382     })
383
384     # Display summary of results for all learning rates
385     for result in results:
386         print(f"Learning Rate: {result['learning_rate']}, Final Validation Accuracy:
{result['final_val_accuracy']}, Test Accuracy: {result['test_accuracy']}, Time Taken:
{result['time']} seconds")
387
388     # Return results for further analysis if needed
389     return results
390
391 if __name__ == '__main__':
392     architectures = {
393         "1_tanh_2_sigmoid_3_leaky_relu": ['tanh', 'sigmoid', 'leaky_relu'],
394         "1_tanh_2_sigmoid_3_sigmoid_4_relu": ['tanh', 'sigmoid', 'sigmoid', 'relu'],
395         "3_layers_sigmoid": ['sigmoid', 'sigmoid', 'sigmoid'],
396         "3_layers_leaky_relu": ['leaky_relu', 'leaky_relu', 'leaky_relu'],
397         "4_layers_tanh": ['tanh', 'tanh', 'tanh', 'tanh'],
398         "4_layers_leaky_relu": ['leaky_relu', 'leaky_relu', 'leaky_relu', 'leaky_relu']}
399
400     experiment_configurations = [
401         ("1_tanh_2_sigmoid_3_leaky_relu", 50, "Adam"),
402         ("1_tanh_2_sigmoid_3_leaky_relu", 50, "RMSprop"),
403         ("1_tanh_2_sigmoid_3_leaky_relu", 100, "Adam"),
404         ("1_tanh_2_sigmoid_3_leaky_relu", 100, "RMSprop"),
405         ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 50, "Adam"),
406         ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 50, "RMSprop"),
407         ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 100, "Adam"),
408         ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 100, "RMSprop"),
409         ("3_layers_sigmoid", 50, "Adam"),
410         ("3_layers_sigmoid", 50, "RMSprop"),
411         ("3_layers_sigmoid", 100, "Adam"),
412         ("3_layers_sigmoid", 100, "RMSprop"),
413         ("3_layers_leaky_relu", 50, "Adam"),
414         ("3_layers_leaky_relu", 50, "RMSprop"),
415         ("3_layers_leaky_relu", 100, "Adam"),

```



```
416     ("3_layers_leaky_relu", 100, "RMSprop"),
417     ("4_layers_tanh", 50, "Adam"),
418     ("4_layers_tanh", 50, "RMSprop"),
419     ("4_layers_tanh", 100, "Adam"),
420     ("4_layers_tanh", 100, "RMSprop"),
421     ("4_layers_leaky_relu", 50, "Adam"),
422     ("4_layers_leaky_relu", 50, "RMSprop"),
423     ("4_layers_leaky_relu", 100, "Adam"),
424     ("4_layers_leaky_relu", 100, "RMSprop")]
425
426     for configuration in experiment_configurations:
427         architecture_name, neurons_per_layer, optimizer_name = configuration
428         training_testing(architectures[architecture_name], neurons_per_layer,
429                           optimizer_name)
429
430     learning_rates = [0.0001, 0.001, 0.01, 0.1, 0.5]
431     results = training_testing_diff_lr(['tanh', 'sigmoid', 'leaky_relu'], 50, "Adam",
432                                       learning_rates)
```