# CHƯƠNG 9: NHẬN DẠNG HÌNH ẢNH SỬ DỤNG MÁY HỌC 1

## 9.1. Nhận dạng số MNIST

### 9.1.1. Cài đặt Raspbian

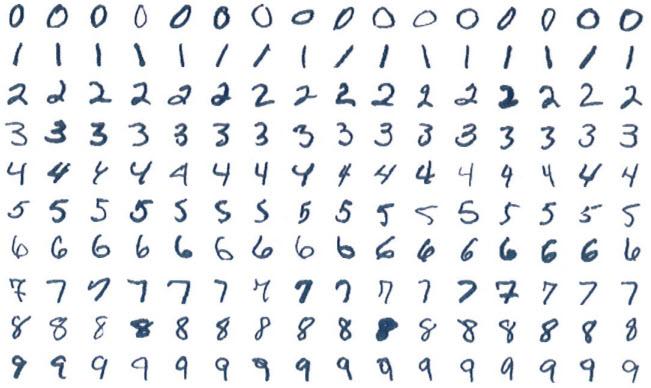
Dữ liệu **MNIST** là một tập dữ liệu phổ biến trong lĩnh vực học máy và thị giác máy tính. MNIST đại diện cho "**Modified National Institute of Standards and Technology**," là một bộ dữ liệu chứa các hình ảnh của chữ số viết tay từ 0 đến 9. Bộ dữ liệu này thường được sử dụng để phát triển và kiểm thử các mô hình học máy, đặc biệt là trong lĩnh vực nhận dạng chữ số.

Thông tin chi tiết về dữ liệu MNIST:

* **Kích thước ảnh**: Mỗi ảnh trong MNIST có kích thước 28x28 pixel.
* **Loại dữ liệu**: Mỗi ảnh là ảnh xám (grayscale), có nghĩa là mỗi pixel được biểu diễn bằng một giá trị từ 0 đến 255, thể hiện mức độ đen trắng.
* **Nhãn**: Mỗi ảnh được gắn kết với một nhãn tương ứng là chữ số từ 0 đến 9.

MNIST thường được sử dụng như một bài toán thử nghiệm cho các mô hình học máy đơn giản và phức tạp. Các nghiên cứu và dự án thường sử dụng MNIST để thử nghiệm và so sánh hiệu suất giữa các mô hình khác nhau.

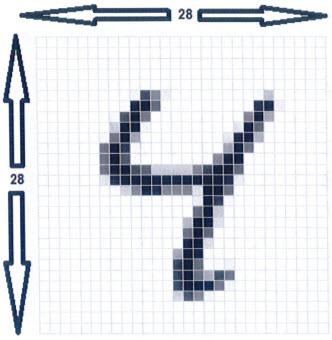
Tập dữ liệu MNIST là một tập hợp các số viết tay. Tập dữ liệu này được tạo bởi máy tính để tự động nhận dạng mã bưu chính ghi trên các mục bưu chính tại bưu điện. Cơ sở dữ liệu chứa một tập gồm 60.000 ví dụ và một bộ thử nghiệm gồm 10.000 ví dụ. Tập dữ liệu này là một cơ sở dữ liệu tuyệt vời cho những ai muốn tìm hiểu cách nhận biết các kỹ thuật và mẫu cho dữ liệu thực với nỗ lực tối thiểu về tiền xử lý và định dạng. Trong chương này, các nguyên tắc nhận dạng và phân lớp hình ảnh thông qua học cách phân loại các tập dữ liệu sẽ được tìm hiểu.



*Hình ảnh tập dữ liệu của MNIST*

### 9.1.2. Nhận dạng số MNIST bằng cách sử dụng máy học

Hình ảnh số của tập dữ liệu là hình ảnh đơn sắc 28 x 28 pixel. Nói cách khác, nó sử dụng 784 dữ liệu để nhận dạng số.



Quá trình trên được hiển thị dưới dạng toàn bộ nguồn chương trình như sau:

| import tensorflow.compat.v1 as tf  import matplotlib.pyplot as plt  import random  # Tắt Eager Execution (Eager mode)  tf.disable\_eager\_execution()  # Đặt giống ngẫu nhiên  tf.set\_random\_seed(777)  from tensorflow.examples.tutorials.mnist import input\_data  # Check out https://www.tensorflow.org/get started/mnist/beginners for more information about the mnist dataset  # Tải dữ liệu MNIST từ TensorFlow Datasets  mnist = tf.keras.datasets.mnist  (x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()  # Chuẩn hóa dữ liệu và chuyển đổi thành one-hot encoding  x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  y\_train = tf.one\_hot(y\_train, 10)  y\_test = tf.one\_hot(y\_test, 10)  # MNIST data image of shape 28 \* 28 = 784  X = tf.placeholder(tf.float32, [None, 784])  # 0-9 digits recognition = 10 classes  Y = tf.placeholder(tf.float32, [None, 10])  W = tf.Variable(tf.random.normal([784, 10]))  b = tf.Variable(tf.random.normal([10]))  # Hypothesis (using softmax)  hypothesis = tf.nn.softmax(tf.matmul(X, W) + b)  cost = tf.reduce\_mean(-tf.reduce\_sum(Y \* tf.math.log(hypothesis), axis=1))  train = tf.compat.v1.train.GradientDescentOptimizer(learning\_rate=0.1).minimize(cost)  # Test model  is\_correct = tf.equal(tf.argmax(hypothesis, 1), tf.argmax(Y, 1))  # Calculate accuracy  accuracy = tf.reduce\_mean(tf.cast(is\_correct, tf.float32))  # Create a TensorFlow session  with tf.compat.v1.Session() as sess:  # Initialize TensorFlow variables  sess.run(tf.compat.v1.global\_variables\_initializer())  # Training cycle  num\_epochs = 15  batch\_size = 100  num\_iterations = x\_train.shape[0] // batch\_size  for epoch in range(num\_epochs):  avg\_cost = 0  for i in range(num\_iterations):  batch\_xs = x\_train[i \* batch\_size : (i + 1) \* batch\_size].reshape(-1, 784)  batch\_ys = y\_train[i \* batch\_size : (i + 1) \* batch\_size].eval(session=tf.compat.v1.Session())  \_, cost\_val = sess.run([train, cost], feed\_dict={X: batch\_xs, Y: batch\_ys})  avg\_cost += cost\_val / num\_iterations  print("Epoch: {:04d}, Cost: {:.9f}".format(epoch + 1, avg\_cost))  print("Learning finished")  # Test the model using test sets  test\_accuracy = accuracy.eval(session=sess, feed\_dict={X: x\_test.reshape(-1, 784), Y: y\_test.eval(session=tf.compat.v1.Session())})  print("Accuracy: ", test\_accuracy)  # Get one and predict  r = random.randint(0, x\_test.shape[0] - 1)  print("Label: ", sess.run(tf.argmax(y\_test[r], 0)))  print("Prediction: ", sess.run(tf.argmax(hypothesis, 1), feed\_dict={X: x\_test[r].reshape(-1, 784)}))  plt.imshow(x\_test[r], cmap="gray")  plt.show() |
| --- |

Dưới đây là kết quả nhận được



Khi epoch chạy, chi phí giảm dần, và kết quả là, khoảng 89% độ chính xác được hiển thị. Nếu hình ảnh không xuất hiện, đặt con trỏ chuột vào mã nguồn chương trình và nhấn lại nút chạy.

### 9.1.3. Nhận dạng số MNIST bằng cách sử dụng Deep Learning

Lần này, chúng ta sẽ học cách sử dụng Deep Learning để tăng độ chính xác của nhận dạng số MNIST. Tăng độ chính xác có nghĩa là có thể nhận dạng hình ảnh chính xác hơn bằng cách sử dụng các phương pháp học máy tiên tiến khác nhau. Tỷ lệ nhận dạng có độ chính xác khoảng 88% được xác nhận bằng cách sử dụng phân lớp Softmax. Các phương pháp khác để giới thiệu đang sử dụng Mạng Neural (Mạng thần kinh nhân tạo), Relu, Xavier, Dropout, và Adam.

#### 9.1.3.1. Sử dụng mạng Neural

Phân lớp Softmax được xử lý trong một lớp, nhưng mạng lưới Neural sử dụng n lớp, thường được gọi là học sâu - Deep Learning. Nhiều lớp sử dụng một số W (Weight – trọng lượng, trọng số) và b (bias – độ lệch) thay vì một W và một b.

Trong mạng Neural, có một quy tắc phải được tuân theo bởi số lượng dữ liệu đầu vào và dữ liệu đầu ra. Ví dụ: nếu sử dụng 784 pixel dữ liệu làm đầu vào đầu tiên và nhận 10 dữ liệu số từ 0 đến 9 làm đầu ra cuối cùng, nếu đặt 256 dữ liệu đầu ra ở lớp giữa, thì nên nhập vào 256 dữ liệu đầu vào ở lớp tiếp theo. Có thể kiểm tra Weight của mã nguồn chương trình sau.

| W1 = tf.Variable(tf.compat.v1.random\_normal([784, 256]))  b1 = tf.Variable(tf.compat.v1.random\_normal([256]))  L1 = tf.nn.relu(tf.matmul(X, W1) + b1)  W2 = tf.Variable(tf.compat.v1.random\_normal([256, 256]))  b2 = tf.Variable(tf.compat.v1.random\_normal([256]))  L2 = tf.nn.relu(tf.matmul(L1, W2) + b2)  W3 = tf.Variable(tf.compat.v1.random\_normal([256, 10]))  b3 = tf.Variable(tf.compat.v1.random\_normal([10]))  hypothesis = tf.matmul(L2, W3) + b3 |
| --- |

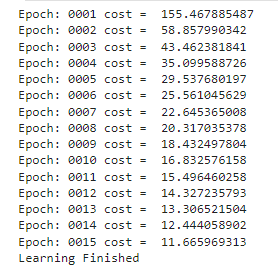
Hàm relu trong -

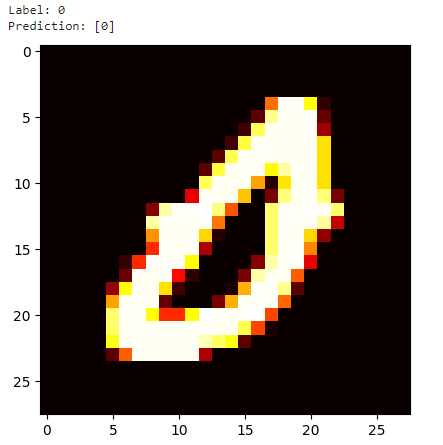
chương trình đã được thay thế để giải quyết vấn đề “Vanishing gradient” xảy ra khi sử dụng hàm Sigmoid. Vấn đề là độ dốc biến mất khi thêm nhiều lớp. Mạng Neural với nhiều lớp sử dụng ReLU thay vì sigmoids.

Mã nguồn chương trình hoàn chỉnh sử dụng mạng Neural với ba lớp được hiển thị như bên dưới.

| import tensorflow as tf  import random  import matplotlib.pyplot as plt  from tensorflow.keras.datasets import mnist  tf.random.set\_seed(777)  (x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()  x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  # parameters  learning\_rate = 0.001  training\_epochs = 15  batch\_size = 100  # Tắt tính toán tự động  tf.compat.v1.disable\_eager\_execution()  # input place holders  X = tf.compat.v1.placeholder(tf.float32, [None, 784])  Y = tf.compat.v1.placeholder(tf.float32, [None, 10])  # weights & bias for nn layers  W1 = tf.Variable(tf.compat.v1.random\_normal([784, 256]))  b1 = tf.Variable(tf.compat.v1.random\_normal([256]))  L1 = tf.nn.relu(tf.matmul(X, W1) + b1)  W2 = tf.Variable(tf.compat.v1.random\_normal([256, 256]))  b2 = tf.Variable(tf.compat.v1.random\_normal([256]))  L2 = tf.nn.relu(tf.matmul(L1, W2) + b2)  W3 = tf.Variable(tf.compat.v1.random\_normal([256, 10]))  b3 = tf.Variable(tf.compat.v1.random\_normal([10]))  hypothesis = tf.matmul(L2, W3) + b3  # define cost/loss & optimizer  cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(  logits=hypothesis, labels=Y  ))  optimizer = tf.compat.v1.train.GradientDescentOptimizer(learning\_rate=learning\_rate).minimize(cost)  # initialize  sess = tf.compat.v1.Session()  sess.run(tf.compat.v1.global\_variables\_initializer())  # train my model  for epoch in range(training\_epochs):  avg\_cost = 0  total\_batch = int(len(x\_train) / batch\_size)  for i in range(total\_batch):  start = i \* batch\_size  end = (i + 1) \* batch\_size  batch\_xs, batch\_ys = x\_train[start:end], y\_train[start:end]  batch\_xs = batch\_xs.reshape(-1, 784)  batch\_ys = tf.keras.utils.to\_categorical(batch\_ys, 10)  feed\_dict = {X: batch\_xs, Y: batch\_ys}  c, \_ = sess.run([cost, optimizer], feed\_dict=feed\_dict)  avg\_cost += c / total\_batch  print('Epoch:', '%04d' % (epoch + 1), 'cost = ', '{:.9f}'.format(avg\_cost))  print('Learning Finished')  # Kiểm thử mô hình và kiểm tra độ chính xác  correct\_prediction = tf.equal(tf.argmax(hypothesis, 1), tf.argmax(Y, 1))  accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  print('Accuracy:', sess.run(accuracy, feed\_dict={  X: x\_test.reshape(-1, 784),  Y: tf.keras.utils.to\_categorical(y\_test, 10)  }))  # Lấy một ví dụ và dự đoán  r = random.randint(0, len(x\_test) - 1)  print("Label:", y\_test[r])  print("Prediction:", sess.run(tf.argmax(hypothesis, 1), feed\_dict={X: x\_test[r].reshape(1, -1)}))  plt.imshow(x\_test[r], cmap="hot")  plt.show() |
| --- |

Nếu chạy chương trình này, sẽ thấy các kết quả sau:





Có thể thấy rằng độ chính xác đã được cải thiện lên khoảng 91%.

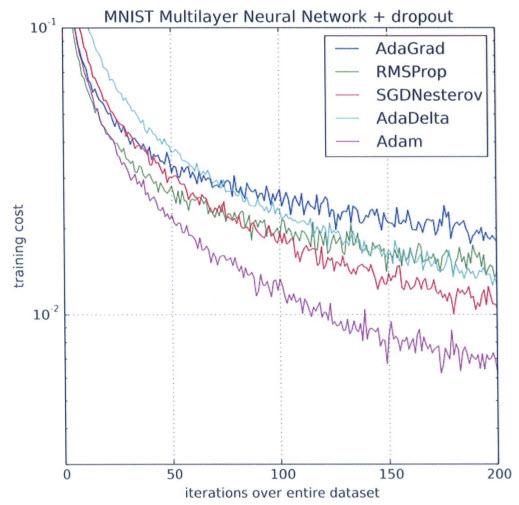
#### 9.1.3.2. Sử dụng Optimizer

Cho đến bây giờ, Optimizer được sử dụng phổ biến nhất để tối ưu hóa tổn thất là GradientDescent.

train = tf.train **GradientDescentOptimizer** {learning\_rate = 0.1) .minimize(cost)

Tuy nhiên, có nhiều loại Optimizer khác có sẵn. Sử dụng một trình Optimizertốt cũng là một cách tốt để cải thiện độ chính xác. Một trong những công cụ tối ưu hóa có thể áp dụng bằng TensorFlow là Adam.

Biểu đồ sau so sánh giá trị chi phí của một số Optimizer khi chúng chạy. Điểm mấu chốt là sự thay đổi về chi phí khi sử dụng Adam và có thể thấy rằng quá trình tối ưu hóa là nhanh nhất.



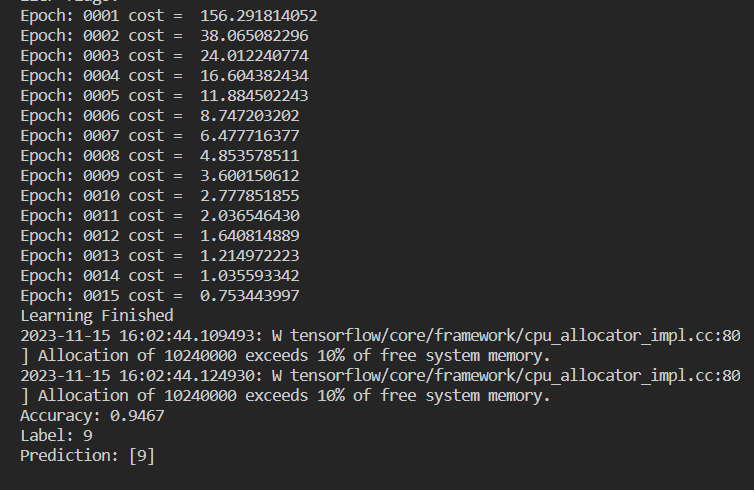
**Đào tạo mạng lưới Neural đa lớp trên hình ảnh MNIST**

**Diederik P. Kingma, 2015**

Dưới đây là cách sử dụng Optimizer Adam trong TensorFlow.

| cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(  logits=hypothesis, labels=Y  ))  optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=learning\_rate).minimize(cost) |
| --- |

Trước đó, trong một chương trình sử dụng Mạng Neural, hãy sửa đổi phần Optimizer để xem liệu có sự cải thiện nào về độ chính xác hay không. Có thể tìm thấy các kết quả sau bằng cách chạy chương trình đã sửa đổi:



Có thể thấy rằng độ chính xác đã được cải thiện lên khoảng 94%.

#### 9.1.3.3. Sử dụng Xavier initializer

ReLU được giới thiệu như một cách để giải quyết vấn đề “Vanishing gradient” gây ra bởi số lượng lớp ngày càng tăng. Có một cách khác để giải quyết vấn đề, đó là khởi tạo tốt. Giá trị khởi tạo của trọng số được sử dụng cho đến nay là giá trị ngẫu nhiên trong khoảng từ 0 đến 1. Nếu giá trị ban đầu là 0, độ dốc trở thành 0 và độ dốc biến mất. Một trong những cách để thực hiện khởi tạo tốt để giải quyết vấn đề là khởi tạo Xavier. Khởi tạo Xavier là một phương pháp đưa ra một giá trị ban đầu tỷ lệ với số lượng đầu vào và đầu ra của lớp.

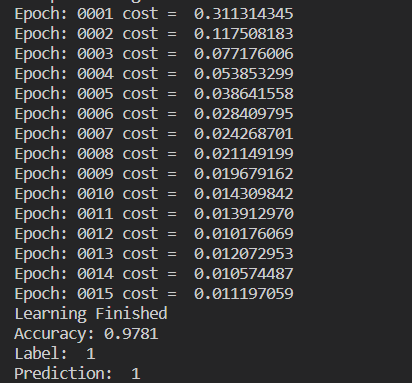
Có thể kiểm tra những điều đó trong chương trình sau:



Mã nguồn chương trình hoàn chỉnh được tạo bằng cách thêm phần khởi tạo trong chương trình bằng cách sử dụng mạng Neural như sau:

| import tensorflow as tf  import random  import matplotlib.pyplot as plt  from tensorflow.keras.datasets import mnist  tf.compat.v1.disable\_eager\_execution()  tf.compat.v1.set\_random\_seed(777) # reproducibility  (x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()  x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  # Flatten the images from 28x28 to 784  x\_train = x\_train.reshape((-1, 784))  x\_test = x\_test.reshape((-1, 784))  # Convert labels to one-hot encoding  y\_train = tf.keras.utils.to\_categorical(y\_train, 10)  y\_test = tf.keras.utils.to\_categorical(y\_test, 10)  # Parameters  learning\_rate = 0.001  training\_epochs = 15  batch\_size = 100  # Input placeholders  X = tf.compat.v1.placeholder(tf.float32, [None, 784])  Y = tf.compat.v1.placeholder(tf.float32, [None, 10])  # Weights and biases for NN layers with Xavier initialization  initializer = tf.initializers.GlorotUniform()  W1 = tf.Variable(initializer(shape=[784, 256]))  b1 = tf.Variable(tf.random.normal([256]))  L1 = tf.nn.relu(tf.matmul(X, W1) + b1)  W2 = tf.Variable(initializer(shape=[256, 256]))  b2 = tf.Variable(tf.random.normal([256]))  L2 = tf.nn.relu(tf.matmul(L1, W2) + b2)  W3 = tf.Variable(initializer(shape=[256, 10]))  b3 = tf.Variable(tf.random.normal([10]))  hypothesis = tf.matmul(L2, W3) + b3  # Define cost/loss and optimizer  cost = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=hypothesis, labels=Y))  optimizer = tf.compat.v1.train.AdamOptimizer(learning\_rate=learning\_rate).minimize(cost)  # Initialize  sess = tf.compat.v1.Session()  sess.run(tf.compat.v1.global\_variables\_initializer())  # Train the model  for epoch in range(training\_epochs):  avg\_cost = 0  total\_batch = int(len(x\_train) / batch\_size)  for i in range(total\_batch):  start = i \* batch\_size  end = (i + 1) \* batch\_size  batch\_xs, batch\_ys = x\_train[start:end], y\_train[start:end]  feed\_dict = {X: batch\_xs, Y: batch\_ys}  c, \_ = sess.run([cost, optimizer], feed\_dict=feed\_dict)  avg\_cost += c / total\_batch  print('Epoch:', '%04d' % (epoch + 1), 'cost = ', '{:.9f}'.format(avg\_cost))  print('Learning Finished')  # Test the model and check accuracy  correct\_prediction = tf.equal(tf.argmax(hypothesis, 1), tf.argmax(Y, 1))  accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  print('Accuracy:', sess.run(accuracy, feed\_dict={X: x\_test, Y: y\_test}))  # Get a random example and print label and prediction  r = random.randint(0, len(x\_test) - 1)  label = y\_test[r]  prediction = sess.run(tf.argmax(hypothesis, 1), feed\_dict={X: x\_test[r].reshape(1, -1)})  print("Label: ", sess.run(tf.argmax(label, 0)))  print("Prediction: ", prediction[0])  # Display the image  plt.imshow(x\_test[r].reshape(28, 28), cmap="gray")  plt.show() |
| --- |

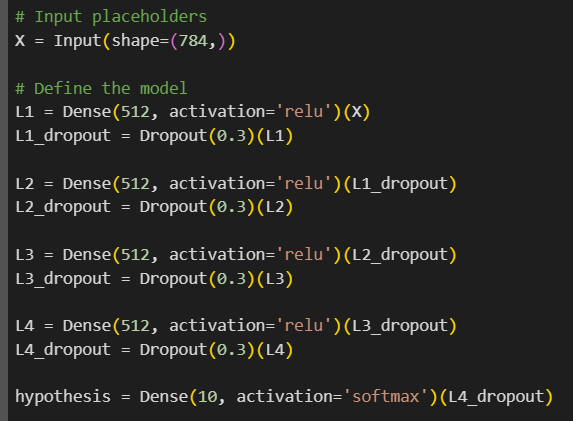
Nếu chạy chương trình này, sẽ thấy kết quả sau:



Có thể thấy rằng độ chính xác được cải thiện đến khoảng 97%. Ngoài ra, giá trị chi phí rất thấp từ epoch đầu tiên. Điều này xác nhận rằng giá trị ban đầu được thiết lập tốt.

#### 9.1.3.4. Sử dụng Dropout

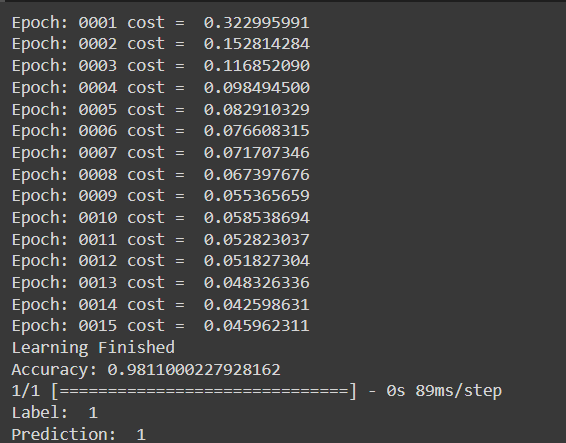
Khi số lớp mạng tăng lên, dữ liệu cần học được tinh chỉnh và đào tạo quá kỹ, do đó, khi dữ liệu thử nghiệm mới xuất hiện, độ chính xác giảm mạnh. Nó gọi là sự quá khớp. Một cách để giải quyết điều này là sử dụng Dropout. Dropout là một cách để chặn một phần của mạng và đào tạo để mạng không bị quá tải.

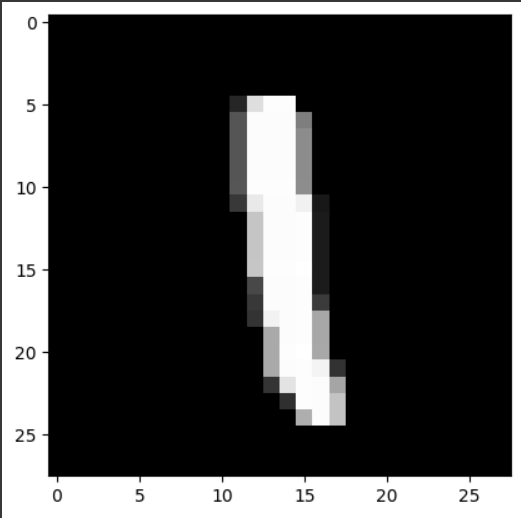


Toàn bộ mã nguồn chương trình trước đó đã mở rộng lớp bằng Xavier đã thêm Dropout như sau:

| import tensorflow as tf  from tensorflow.keras.layers import Input, Dense, Dropout  from tensorflow.keras.models import Model  from tensorflow.keras.datasets import mnist  import random  import matplotlib.pyplot as plt  tf.random.set\_seed(777) # Reproducibility  (x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()  # Normalize pixel values to be between 0 and 1  x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  # Flatten the images  x\_train\_flat = x\_train.reshape((x\_train.shape[0], -1))  x\_test\_flat = x\_test.reshape((x\_test.shape[0], -1))  # Convert labels to one-hot encoding  y\_train = tf.one\_hot(y\_train, 10)  y\_test = tf.one\_hot(y\_test, 10)  # Parameters  learning\_rate = 0.001  training\_epochs = 15  batch\_size = 100  # Input placeholders  X = Input(shape=(784,))  # Define the model  L1 = Dense(512, activation='relu')(X)  L1\_dropout = Dropout(0.3)(L1)  L2 = Dense(512, activation='relu')(L1\_dropout)  L2\_dropout = Dropout(0.3)(L2)  L3 = Dense(512, activation='relu')(L2\_dropout)  L3\_dropout = Dropout(0.3)(L3)  L4 = Dense(512, activation='relu')(L3\_dropout)  L4\_dropout = Dropout(0.3)(L4)  hypothesis = Dense(10, activation='softmax')(L4\_dropout)  # Build the model  model = Model(inputs=X, outputs=hypothesis)  # Compile the model  optimizer = tf.keras.optimizers.Adam(learning\_rate=learning\_rate)  model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])  # Custom training loop  for epoch in range(training\_epochs):  avg\_cost = 0  total\_batch = int(len(x\_train) / batch\_size)  for i in range(total\_batch):  start = i \* batch\_size  end = (i + 1) \* batch\_size  batch\_xs, batch\_ys = x\_train\_flat[start:end], y\_train[start:end]  with tf.GradientTape() as tape:  logits = model(batch\_xs, training=True)  loss = tf.keras.losses.categorical\_crossentropy(batch\_ys, logits)  avg\_cost += tf.reduce\_mean(loss) / total\_batch  gradients = tape.gradient(loss, model.trainable\_variables)  optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))  print('Epoch:', '{:04d}'.format(epoch + 1), 'cost = ', '{:.9f}'.format(avg\_cost))  print('Learning Finished')  # Test the model and check accuracy  accuracy = model.evaluate(x\_test\_flat, y\_test, batch\_size=batch\_size, verbose=0)[1]  print('Accuracy:', accuracy)  # Get a random example and print label and prediction  r = random.randint(0, len(x\_test) - 1)  label = y\_test[r]  prediction = model.predict(x\_test\_flat[r].reshape(1, -1))  predicted\_label = tf.argmax(prediction, 1).numpy()[0]  print("Label: ", tf.argmax(label, 0).numpy())  print("Prediction: ", predicted\_label)  # Display the image  plt.imshow(x\_test[r].reshape(28, 28), cmap="gray")  plt.show() |
| --- |

Nếu chạy chương trình này, sẽ thấy kết quả sau:





Có thể mất một chút thời gian tùy thuộc vào hiệu suất của PC cho đến khi kết quả được đưa ra và độ chính xác được cải thiện lên khoảng 98%. Sự khác biệt khoảng 1% có vẻ hơi nhỏ hơn so với kết quả trước đó, nhưng lưu ý rằng việc cải thiện từ 97% lên 98% là rất khó, và đến 99% thì khó hơn nữa.