@RAMCHANDRAPADWAL

Practical Guide to Theano for Data Science

A STEP-BY-STEP GUIDE

Table of Contents

@RAMCHANDRAPADWAL

CHAPTER N.1

Introduction to Theano

A Step-by-Step Guide

1.1 WHAT IS THEANO?

Theano is an open-source Python library developed at the Montreal Institute for Learning Algorithms (MILA) that allows efficient definition, optimization, and evaluation of mathematical expressions, especially matrix-valued ones. It is widely used for deep learning and other machine learning tasks due to its ability to perform symbolic computation, which is crucial for efficiently handling large datasets and complex models.

1.2 WHY USE THEANO FOR DATA SCIENCE?

There are several reasons why Theano is a popular choice for data scientists:

- **Symbolic Computation:** Theano uses symbolic variables and expressions, enabling it to optimize computations and handle large datasets efficiently.
- **Speed:** Theano automatically optimizes and compiles the computation graph to run efficiently on both CPU and GPU, making it ideal for deep learning tasks.
- **Flexibility:** Theano allows you to define and manipulate complex mathematical expressions with ease, making it suitable for implementing various machine learning algorithms.
- **Integration:** Theano seamlessly integrates with other libraries like NumPy, SciPy, and Pandas, enhancing its functionality and usability in data science workflows.

1.3 INSTALLING THEANO

Before getting started, make sure you have Python installed on your system. You can install Theano using pip:

```
pip install theano
```

Additionally, ensure you have the appropriate GPU drivers and libraries installed if you plan to use Theano with GPU acceleration.

@RAMCHANDRAPADWAL

CHAPTER N.2

Theano Basics

A Step-by-Step Guide

2.1 Symbolic Variables

In Theano, you work with symbolic variables that represent mathematical entities, such as scalars, vectors, or matrices. To create symbolic variables, use the theano.tensor module:

```
import theano.tensor as T

# Scalars
x = T.scalar('x')

# Vectors
y = T.vector('y')

# Matrices
z = T.matrix('z')
```

2.2 Computation Graph

Theano creates a computation graph to represent the mathematical operations you perform on the symbolic variables. The computation graph defines the relationships between different variables and the operations required to compute the final output.

```
import theano.tensor as T

# Symbolic variables
x = T.scalar('x')
y = T.scalar('y')

# Define a computation graph
result = x + y
```

2.3 Function Compilation

To perform computations on the symbolic variables, you need to compile **Theano functions**. The **theano.function** method compiles a function that can be later evaluated with specific inputs.

```
import theano.tensor as T
import theano

# Symbolic variables
x = T.scalar('x')
y = T.scalar('y')

# Define a computation graph
result = x + y

# Compile the function
addition_func = theano.function(inputs=[x, y], outputs=result)

# Evaluate the function
output = addition_func(2, 3)
print(output)  # Output: 5.0
```

2.4 Shared Variables

Shared variables are special types of symbolic variables that maintain their state between function calls. They are commonly used for model parameters during training.

```
import numpy as np
import theano.tensor as T
import theano

# Shared variable
weights = theano.shared(np.random.randn(5, 5), 'weights')

# Symbolic variable
x = T.vector('x')

# Define a computation graph
result = T.dot(x, weights)

# Compile the function
dot_product_func = theano.function(inputs=[x], outputs=result)

# Evaluate the function with different inputs
output1 = dot_product_func([1, 2, 3, 4, 5])
output2 = dot_product_func([5, 4, 3, 2, 1])
```

@RAMCHANDRAPADWAL

CHAPTER N.3

Linear Regression with Theano

A Step-by-Step Guide

3.1 Data Preparation

For linear regression, let's generate some random data and split it into training and testing sets.

```
import numpy as np
import theano.tensor as T
import theano

# Generate random data
np.random.seed(42)
X_train = np.random.rand(100, 1)
y_train = 3 * X_train + 2 + 0.2 * np.random.randn(100, 1)

X_test = np.random.rand(20, 1)
y_test = 3 * X_test + 2
```

3.2 Model Building

Next, we define the linear regression model using Theano.

```
import theano.tensor as T
import theano

# Symbolic variables
X = T.matrix('X')
y = T.matrix('y')

# Model parameters
W = theano.shared(np.random.randn(1), 'W')
b = theano.shared(np.random.randn(1), 'b')

# Model
y_pred = T.dot(X, W) + b
```

3.3 Training the Model

To train the model, we define a cost function and use gradient descent to minimize it.

```
# Cost function (mean squared error)
cost = T.mean((y_pred - y) ** 2)

# Gradient descent updates
learning_rate = 0.01
updates = [(W, W - learning_rate * T.grad(cost, W)),
           (b, b - learning_rate * T.grad(cost, b))]

# Compile the training function
train_func = theano.function(inputs=[X, y], outputs=cost, updates=updates)

# Training loop
num_epochs = 1000
for epoch in range(num_epochs):
    # Compute the cost and perform one update
    cost_val = train_func(X_train, y_train)
    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Cost: {cost_val:.4f}')
```

3.4 Making Predictions

Finally, let's use the trained model to make predictions on the test set.

```
# Compile the prediction function
predict_func = theano.function(inputs=[X], outputs=y_pred)

# Make predictions
y_pred_test = predict_func(X_test)
```

@RAMCHANDRAPADWAL

CHAPTER N.4

Logistic Regression with Theano

A Step-by-Step Guide

4.1 Data Preparation

For logistic regression, we'll generate some random data and create binary labels.

```
import numpy as np
import theano.tensor as T
import theano

# Generate random data
np.random.seed(42)
X_train = np.random.rand(100, 2)
y_train = (X_train[:, 0] + X_train[:, 1] > 1).astype(int)

X_test = np.random.rand(20, 2)
y_test = (X_test[:, 0] + X_test[:, 1] > 1).astype(int)
```

4.2 Model Building

Next, we define the logistic regression model using Theano.

```
import theano.tensor as T
import theano

# Symbolic variables
X = T.matrix('X')
y = T.vector('y')

# Model parameters
W = theano.shared(np.random.randn(2), 'W')
b = theano.shared(np.random.randn(), 'b')

# Model
logits = T.dot(X, W) + b
y_pred = T.nnet.sigmoid(logits)
```

4.3 Training the Model

To train the model, we define the cross-entropy loss function and use gradient descent to minimize it.

```
# Cross-entropy loss
cost = T.mean(T.nnet.binary_crossentropy(y_pred, y))

# Gradient descent updates
learning_rate = 0.1
updates = [(W, W - learning_rate * T.grad(cost, W)),
           (b, b - learning_rate * T.grad(cost, b))]

# Compile the training function
train_func = theano.function(inputs=[X, y], outputs=cost, updates=updates)

# Training loop
num_epochs = 1000
for epoch in range(num_epochs):
    # Compute the cost and perform one update
    cost_val = train_func(X_train, y_train)
    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Cost: {cost_val:.4f}')
```

4.4 Evaluating the Model

Finally, let's evaluate the trained logistic regression model on the test set.

```
# Compile the prediction function
predict_func = theano.function(inputs=[X], outputs=y_pred)

# Make predictions
y_pred_test = predict_func(X_test)

# Convert probabilities to binary predictions
y_pred_binary = (y_pred_test > 0.5).astype(int)

# Calculate accuracy
accuracy = np.mean(y_pred_binary == y_test)
print(f'Test Accuracy: {accuracy:.2%}')
```

@RAMCHANDRAPADWAL

CHAPTER N.5

Neural Networks with Theano

A Step-by-Step Guide

5.1 Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron (MLP) is a basic neural network architecture with one or more hidden layers. Let's build a simple MLP using Theano.

```
import numpy as np
import theano.tensor as T
import theano

# Generate random data
np.random.seed(42)
X_train = np.random.rand(100, 2)
y_train = (X_train[:, 0] + X_train[:, 1] > 1).astype(int)

X_test = np.random.rand(20, 2)
y_test = (X_test[:, 0] + X_test[:, 1] > 1).astype(int)
```

5.2 Data Preparation

Before building the MLP model, it's essential to preprocess the data.

```
# Normalize the data
X_mean, X_std = X_train.mean(axis=0), X_train.std(axis=0)
X_train = (X_train - X_mean) / X_std
X_test = (X_test - X_mean) / X_std
```

5.3 Model Building

Next, let's define the MLP model architecture using Theano.

```
import theano.tensor as T
import theano

# Symbolic variables
X = T.matrix('X')
y = T.vector('y')

# Model parameters
num_input = 2
num_hidden = 10
num_output = 1

W_hidden = theano.shared(np.random.randn(num_input, num_hidden), 'W_hidden')
b_hidden = theano.shared(np.random.randn(num_hidden), 'b_hidden')

W_output = theano.shared(np.random.randn(num_hidden, num_output), 'W_output')
b_output = theano.shared(np.random.randn(num_output), 'b_output')

# Model
hidden_layer = T.nnet.relu(T.dot(X, W_hidden) + b_hidden)
y_pred = T.nnet.sigmoid(T.dot(hidden_layer, W_output) + b_output)
```

5.4 Training the Model

To train the MLP, we'll use the cross-entropy loss function and perform gradient descent.

```
# Cross-entropy loss
cost = T.mean(T.nnet.binary_crossentropy(y_pred.flatten(), y))

# Gradient descent updates
learning_rate = 0.1
updates = [(W_hidden, W_hidden - learning_rate * T.grad(cost, W_hidden)),
           (b_hidden, b_hidden - learning_rate * T.grad(cost, b_hidden)),
           (W_output, W_output - learning_rate * T.grad(cost, W_output)),
           (b_output, b_output - learning_rate * T.grad(cost, b_output))]

# Compile the training function
train_func = theano.function(inputs=[X, y], outputs=cost, updates=updates)

# Training loop
num_epochs = 1000
for epoch in range(num_epochs):
    # Compute the cost and perform one update
    cost_val = train_func(X_train, y_train)
    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Cost: {cost_val:.4f}')
```

5.5 Evaluating the Model

Finally, let's evaluate the trained MLP model on the test set.

```
# Compile the prediction function
predict_func = theano.function(inputs=[X], outputs=y_pred)

# Make predictions
y_pred_test = predict_func(X_test)

# Convert probabilities to binary predictions
y_pred_binary = (y_pred_test > 0.5).astype(int)

# Calculate accuracy
accuracy = np.mean(y_pred_binary == y_test)
print(f'Test Accuracy: {accuracy:.2%}')
```

@RAMCHANDRAPADWAL

CHAPTER N.6

Convolutional Neural Networks (CNN) with Theano

A Step-by-Step Guide

6.1 Data Preparation

For CNN, we'll use the popular MNIST dataset.

```
import numpy as np
import theano.tensor as T
import theano
import keras
from keras.datasets import mnist

# Load MNIST data
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize and reshape the data
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)

# Convert labels to one-hot encoding
y_train_onehot = keras.utils.to_categorical(y_train, 10)
y_test_onehot = keras.utils.to_categorical(y_test, 10)
```

6.2 Model Building

Next, let's define the CNN model architecture using Theano.

```
import theano.tensor as T
import theano

# Symbolic variables
X = T.tensor4('X')
y = T.matrix('y')

# Model
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

6.3 Training the Model

To train the CNN, we'll use the **fit** method.

```
# Training
batch_size = 128
num_epochs = 10

model.fit(X_train, y_train_onehot, batch_size=batch_size, epochs=num_epochs, verbose=1)
```

6.4 Evaluating the Model

Finally, let's evaluate the trained CNN model on the test set.

```
# Evaluation
test_loss, test_accuracy = model.evaluate(X_test, y_test_onehot, verbose=0)
print(f'Test Accuracy: {test_accuracy:.2%}')
```

@RAMCHANDRAPADWAL

CHAPTER N.7

Recurrent Neural Networks (RNN) with Theano

A Step-by-Step Guide

7.1 Data Preparation

For RNN, we'll use a synthetic time series dataset.

```
import numpy as np
import theano.tensor as T
import theano

# Generate synthetic time series data
np.random.seed(42)
time_steps = 100
X_train = np.random.rand(1000, time_steps, 1)
y_train = np.sin(np.sum(X_train, axis=1))

X_test = np.random.rand(200, time_steps, 1)
y_test = np.sin(np.sum(X_test, axis=1))
```

7.2 Model Building

Next, let's define the RNN model architecture using Theano.

```
import theano.tensor as T
import theano

# Symbolic variables
X = T.tensor3('X')
y = T.matrix('y')

# Model
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense

model = Sequential()
model.add(SimpleRNN(32, input_shape=(time_steps, 1)))
model.add(Dense(1))

# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_squared_error'])
```

7.3 Training the Model

To train the RNN, we'll use the **fit** method.

```
# Training
batch_size = 32
num_epochs = 10

model.fit(X_train, y_train, batch_size=batch_size, epochs=num_epochs, verbose=1)
```

7.4 Evaluating the Model

Finally, let's evaluate the trained RNN model on the test set.

```
# Evaluation
test_loss, test_mse = model.evaluate(X_test, y_test, verbose=0)
print(f'Test MSE: {test_mse:.4f}')
```

@RAMCHANDRAPADWAL

CHAPTER N.8

Theano Optimization Techniques

A Step-by-Step Guide

8.1 Automatic Differentiation

Theano's symbolic computation enables automatic differentiation. It automatically computes gradients of the cost function with respect to the model parameters, simplifying the implementation of gradient-based optimization algorithms.

8.2 Gradient Clipping

Gradient clipping is a technique used to prevent exploding gradients during training. By setting a maximum gradient value, we can control the magnitude of updates, ensuring a stable optimization process.

8.3 Batch Normalization

Batch normalization is another optimization technique that normalizes the activations of each layer to have zero mean and unit variance. This helps stabilize training, enables higher learning rates, and often leads to faster convergence.

@RAMCHANDRAPADWAL

CHAPTER N.9

Deploying Theano Models

A Step-by-Step Guide

9.1 Pickling Models

To save a trained Theano model, you can use the **cPickle** module or the built-in pickle module in Python.

```
import pickle

# Save the trained model
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)
```

9.2 Using Saved Models

To use a saved model for prediction, you can load it using pickle and then perform inference as usual.

```
import pickle

# Load the trained model
with open('model.pkl', 'rb') as f:
    loaded_model = pickle.load(f)

# Perform inference
predictions = loaded_model.predict(X_test)
```

Conclusion

This practical guide provides an overview of using Theano for data science tasks. We covered the basics of Theano, implemented linear regression, logistic regression, neural networks (MLP, CNN, RNN), and optimization techniques. Additionally, we learned how to save and load trained models for deployment. Theano's symbolic computation and seamless integration with Python libraries make it a powerful tool for data scientists interested in deep learning and other machine learning tasks. As you delve deeper into Theano and its various applications, you will discover its versatility and efficiency in handling complex data science projects. Happy coding!