PyTorch

Sagar Imambi, Kolla Bhanu Prakash, and G. R. Kanagachidambaresan

Python is favored for coding and working with deep learning and thus has a wide range of languages and libraries to look over, as given in Fig. 1.

Theano, one of the main deep learning structures, has stopped dynamic improvement. TensorFlow has devoured Keras altogether [2], elevating it to a top of the line Application Program Interface (AP)I. PyTorch is a scientific computing library, supplanted a large portion of the low-level code reused from the Lua-based Torch venture. Initially, PyTorch was created by Hugh Perkins as a Python wrapper.

It included help for ONNX, a seller unbiased [3] model portrayal, trade design, and a deferred execution "diagram mode" runtime called TorchScript. PyTorch is another deep learning library with the abilities of fast performance. Essentially, it is the Facebook answer for combine burn with Python.

1 The Significant Highlights of PyTorch

Simple Interface – PyTorch offers simple to utilize API; subsequently, it is viewed as extremely easy to work with and runs on Python. The [4] code execution in this structure is effortless.

Python use – This library is viewed as Pythonic, which smoothly incorporates with the Python core functions. In this way, it can use every one of the administrations and functionalities offered by Python [5]. PyTorch ensures local help for Python and utilization of its libraries.

S. Imambi · K. B. Prakash (✉)
KL Deemed to be University, Vijayawada, AP, India

G. R. Kanagachidambaresan
Vel Tech Rangarajan Dr Sagunthala R&D Institute of Science and Technology, Chennai, Tamil Nadu, India

© Springer Nature Switzerland AG 2021

K. B. Prakash, G. R. Kanagachidambaresan (eds.), *Programming with TensorFlow*, EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-57077-4_10

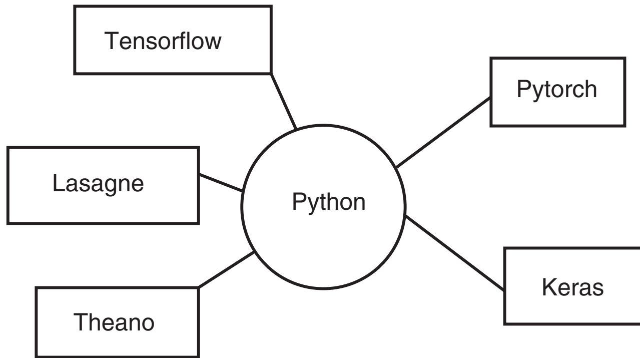

Fig. 1 The Python extension libraries

Dynamic computational charts – PyTorch gives a fantastic stage that offers dynamic computational diagrams. In this way, a client can transform them during runtime. It is a major highlight of PyTorch. This is exceptionally [6] valuable when an engineer has no clue about how a lot of memory is required for making a neural system model. They guarantee the diagram would develop progressively – at each purpose of code execution, the chart is worked along and can be controlled at runtime. So every part of the code executing graph was built and able to manipulate at runtime.

Facebook: It is effectively utilized in the improvement of Facebook for every last bit of its deep learning necessities in the stage. It is actively used in the development of Facebook and its subsidiary companies [7].

FAST: PyTorch is quick and feels local, henceforth guaranteeing simple coding and fast handling.

Compute Unified Device Architecture (CUDA): The help for CUDA guarantees that the code can run on the graphical processor in this way increasing the performance of the network.

2 Why We Prefer PyTorch

- It is simple to debug and comprehend the code.
- Has the same number of sort of layers as Torch (Unpool, Convolution (CONV) 1,2,3D, Long Short Term Memory networks (LSTM), Grus).

- A variety of loss functions are available.
- It can be considered as a numpy augmentation to GPUs.
- It is quicker than other libraries, as chainer and dynet.

3 Requirements for Implementing Deep Learning

Deep learning calculations are intended to vigorously rely upon very good quality machines as opposed to conventional Artificial Intelligence (AI) [8] calculations. Deep learning calculations play out a lot of network duplication tasks that require tremendous equipment support. To execute the PyTorch programs, we require a PC or laptop with a CUDA-competent graphical processing unit (GPU), GPU with 8GB of RAM (we recommend an NVIDIA GTX 1070 or better).

4 PyTorch Basic Components

PyTorch is known for having three levels of abstraction as given below:

- Tensor: N-dimensional array which runs on GPU.
- Variable: Node in computational graph. This stores data and gradient.
- Module: Neural network layer that will store state or learnable weights.

4.1 Tensor

PyTorch has an inside data structure, the tensor, a multi-dimensional group that offers various resemblances with numpy. From [9] that foundation, apparel overviews of features have been created that make it easy to prepare an endeavor for the activity or an assessment concerning another neural framework building organized and arranged.

Tensors give animating of logical assignments and PyTorch has packs for passed on getting ready, expert structures for capable data stacking, and an expansive library of typical significant learning limits.

Creating Tensors
After installing the packages, initialize the empty tensor and assign required values.

Ex:

```
#creates an empty matrix of size 5 × 3
a=torch.empty(5,3)
#Initilizematix with zeros
a=torch r.zeroes(5,3,dtype= torch.long)
#assign values to x
a=torch.tensor(5)
```

Data Type of Elements

PyTorch automatically decides the data type of the elements of the tensor when it is created; the data type applies to all the [10] elements of the tensor. Sometimes that can be overridden to convert it into another data type.

Ex:

```
b = torch.tensor([[3, 8, 9],[4-6]])
print(b.dtype)
# torch.int64
x= torch.tensor([[1,2.5,3],[5.3,5,6]])
print(x.dtype)
 # torch.float32
```

While initializing also we can define data type:

```
b = torch.tensor([[3, 8, 9],[4-6]], dtype=torch.int32)
print(b.dtype)
 # torch.int32
```

Creating Torch tensors from numpy array.

Ex:

```
a= np.ones(4)
b=torch.from_numpy(a)
```

Here *a* is numpy array and initialized with ones. When torch tensor is created using numpy array, they share underlying memory location.

Creating numpy array from torch tensors:

Ex:

```
a= torch.ones(4)
b=a.numpy()
```

4.2 Autograd Module

The most important thing PyTorch offers is to apply auto differentiation. We will see how it works in Fig. 2.

Basic operations are based on the training data set. Next we reply all the values of data to reduce the loss at every stage [11]. Then compute gradients. Gradients are computed by finding the negative slope and calculating the minima of the loss function. Automatic differentiation is a difficult and complicated process and that is easy through the autograd module. This module created the dynamic computational graphs as given in Fig. 3.

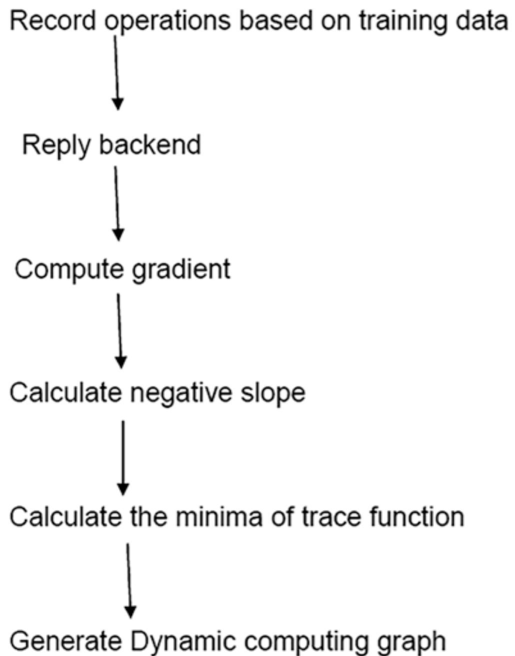

Fig. 2 Steps for autograd Module

```
.

# creating tensors
x = torch.tensor(1., requires grad=True)
w = torch.tensor(1.5, requires grad=True)
b = torch.tensor(3., requires grad=True)

# creating computational graph.
y = w * x + b    # y = 2 * x + 3

# calculating gradients.
y.backward()

# Print out the gradients.
print(x.grad)    # x.grad = 2
print(w.grad)    # w.grad = 1
print(b.grad)    # b.grad = 1
```

Fig. 3 Dynamic computational graphs

5 Implement the Neural Network Using PyTorch

Training a deep learning algorithm involves the following steps: Building a data pipeline, building network architecture, using [12] loss function to evaluate the architecture, and optimizing the weights of the network architecture using an optimization algorithm.

Preparing a deep learning program includes the accompanying advances like building an information pipeline, building [13] system design, evaluating the engineering utilizing a loss function, and optimizing the weights of the network by an optimizing algorithm as given Fig. 4.

```
# step 1 import package and library
import torch

import torch.nn as nn

# step 2 creating data

x = torch.randn(batch_size, n_in)

y = torch.tensor([[1.0], [0.0], [0.0],

[1.0], [1.0], [1.0], [0.0], [0.0], [1.0], [1.0]])

# step 3 Create a model

model = nn.Sequential(nn.Linear(n_in, n_h),

  nn.ReLU(),

  nn.Linear(n_h, n_out),

  nn.Sigmoid())

criterion = torch.nn.MSELoss()

# step4 Construct the optimizer layer

optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

for epoch in range(50):

    y_pred = model(x)

#Step 5: Compute loss loss function

  loss = criterion(y_pred, y)

  print('epoch: ', epoch,' loss: ', loss.item())

  optimizer.zero_grad()

  loss.backward()

#Step 6. Run the autograde

    optimizer.step()
```

Fig. 4 Code snippet for optimizing algorithm

Neural network may be implemented simply by these steps:

Step 1: Import package and libraries.
Step 2: Input data.
Step 3: Construct NN using torch.nn package.
Step 4: Define all layers.
Step 5: Construct loss function.
Step 6: Run autograd.

The output generated of the above program is as given in Fig. 5.

6 Difference Between PyTorch and Tensorflow

Table 1 illustrates the classification of PyTorch and Tensorflow.

```
epoch: 0 loss: 0.2545787990093231
epoch: 1 loss: 0.2545052170753479
epoch: 2 loss: 0.254431813955307
epoch: 3 loss: 0.25435858964920044
epoch: 4 loss: 0.2542854845523834
epoch: 5 loss: 0.25421255826950073
epoch: 6 loss: 0.25413978099823
```

Fig. 5 Output screenshot of the above code

Table 1 PyTorch and Tensorflow classification

S.no.	PyTorch	Tensorflow
1	Dynamic computational graph	Static computational graph
2	Can make use of standard Python flow control	Not able to use
3	Supports Python debugging	Does not support
4	Dynamic inspection of variable and gradients	Not possible
5	Research oriented	Product oriented
6	Developed by Facebook group	Developed by Google group

7 PyTorch for Computer Vision

Computer vision is absolutely one of the fields that [14] has been generally affected by the appearance of profound learning, for an assortment of reasons. The requirement for characterizing or translating the substance of regular pictures existed, enormous datasets became accessible, and convolution layers were created that could be run rapidly on GPUs [15] with remarkable exactness. This joined with the inspiration of the Internet mammoths to comprehend pictures shot by a large number of clients through their cell phones and oversaw on said goliaths' platforms.

7.1 Image Classifier

Image classifier predicts data based [16] on an image set by constructing a neural network. Character/object recognition is generally an image processing technique where image data is imputed and explored by various libraries of Python and PyTorch.

Exploring Data
Standard Python package can be used to load data into numpy array. Then it can be converted into torch tensor. Image data is [17] converted using pillow, opencv. Audio data is interpreted using scipy and librosa and text data is by spacy and cython, etc.

The image prediction using the PyTorch networks is as given in Fig. 6.

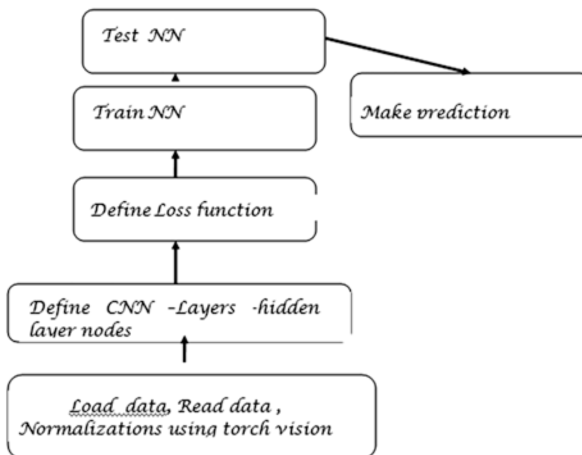

Fig. 6 Steps for image classifier

Data Loading
The first step in deep [18] learning is information loading and handling. PyTorch offers utilities for the identical in torch.utils.data. The crucial training in this module is Dataset and DataLoader. Dataset is built on the pinnacle of tensor data type and is used often for user defined datasets. DataLoader is used if you have a massive dataset and you need to load information from a Dataset in historical past in order that it is equipped and looking ahead to the schooling loop. We can also use torch. nn.DataParallel and torch.distributed if CUDA is available. Figure 7 elucidates the code snippet of data loading.

Defining layers and hidden nodes of network is given in Figs. 8 and 9.

```
imagedir = 'data/images
images = {x: datasets.ImageFolder(os.path.join(imagedir, x))
   for i in ['train', 'val']}
imageloaders = {i: torch.utils.data.DataLoader(images[i], batch size=4,
                                   shuffle=True, num workers=4)
 for i in ['train', 'val']}
Images size = {i: len(images [i]) for i in ['train', 'val']}
classlabels = images ['train'].classes

device = torch.device("cuda:0" if torch.cuda.is available() else "cpu")
```

Fig. 7 Code snippet of data loading

Fig. 8 Layer definition

```
self.linear_layers = Sequential(
    Linear(128 * 14 * 14, 512),
    ReLU(inplace=True),
    Dropout(),
    Linear(512, 256),
    ReLU(inplace=True),
    Dropout(),
    Linear(256,10),
    ReLU(inplace=True),
    Dropout(),
    Linear(10,2)
)
```

```
# Checking whether cuda is available or not
Cudab = torch.cuda.is available()
# Creating  model
model = SimoleNet(num classes=10)
#whenever  cuda is available, change the model to the GPU
if cudab:
    model.cuda()
#initilizing  optimizer and loss function
opt = Adam(model.parameters(), lr=0.001, weight decay=0.0001)
loss_function  = nn.CrossEntroovLoss()
```

Fig. 9 CUDA implementation of algorithm

```
Def validating()
 model.eval()
 test_acc = 0.0
 for x (imgs, cnames) in enumerate(test_loader):
     if cuda_avail:
         imgs = Variable(imgs.cuda())
        cnames = Variable(cnames.cuda())

     outputs = model(imgs)
     _, prediction = torch.max(outouts,data, 1)

     accuracy += torch.sum(prediction == cnames,data)
 # avg accuracy for 10000 images
 accuracy = accuracy / 10000
 return accuracy
```

Fig. 10 Code snippet for accuracy

Next we have to define optimizer and check whether CUDA is available. If it is then use GPU model.

The model was trained after providing information [19] like batch size and number of epochs. Then validating and testing, the model can be done (Fig. 10).

7.2 Image Augmentation in Less Data

We can utilize picture increase for profound learning in any setting – hackathons, industry ventures, etc. We will likewise construct a picture order model utilizing PyTorch to see how picture growth fits into the image.

Deep learning models as a rule require [20] a ton of information for preparing. As a rule, the more the information, the better the exhibition of the model. Be that as it may, obtaining monstrous measures of information accompanies its own difficulties. Not every person has the profound pockets of the enormous firms.

And the issue with an absence of information is [21] that our profound learning model will probably not take in the example or capacity from the information and henceforth it will probably not give a decent presentation on inconspicuous information.

Image augmentation is the way toward producing new pictures for preparing our profound learning model. These new pictures are created utilizing the current preparing pictures and consequently we do not need to gather them physically.

Various Image Augmentation Techniques

Image Rotation

Picture revolution is one of the most ordinarily utilized expansion procedures. It can enable our model to get hearty to the adjustments in the direction of items. Regardless of whether we pivot the picture, the data of the picture continues as before. A vehicle is a vehicle regardless of whether we see it from an alternate point

Shifting/Moving Images

There may be situations when the articles in the picture are not consummately focal adjusted. In these cases, picture move [22] can be utilized to add move invariance to the pictures. By moving the pictures, we can change the situation of the article in the picture and consequently give more assortments to the model. This will in the end lead to a progressively summed up model.

Flipping Images

Flipping is an augmentation of turn. It enables us to flip the picture in the left directly just as up-down bearing. We should perceive how we can execute flipping.

After applying the various operations on the images, the data set is ready for the building model. The process is same as the image classification model as now the data set is having sufficient data.

8 Sequential Data Models

Natural language processing (NLP) provides endless opportunities for artificial intelligence problem solving, making products like Amazon Alexa and Google Translate possible. If you are a developer or data scientist new to NLP and deep learning, this hands-on guide will teach you how to use these approaches with PyTorch, a deep learning application built in Python.

Now, we have seen various feed-forward systems. That is, there is no situation any stretch of imagination keeps up by the system. This is probably not the conduct that we need. Grouping models are vital for NLP: They are models where there is a kind of reliance between your data sources over time. The traditional case of a grouping model is the hidden Markov model for grammatical feature labeling. Another model is the restrictive arbitrary field.

An intermittent neural system is a system that keeps up some sort of state. For instance, its yield could be utilized as a component of the following info, with the goal that data can propagate along as the system disregards the arrangement. On account of an LSTM, for every component in the succession, there is a comparing shrouded state, which on a fundamental level can contain data from subjective focuses prior in the arrangement. We can utilize the concealed state to foresee words in a language model, grammatical feature labels, and a bunch of different things.

8.1 LSTM in PyTorch

Note a few things before you get into the example. The LSTM at PyTorch finds all its inputs to be 3D tensors. The semantics of those tensors "axes" are essential. The first axis is the series itself, the second [23] one indexes the mini-batch instances, and the third one indexes the input elements. We have not discussed mini-batching, so let us just ignore that and assume that on the second axis we will always have only 1 dimension. If we want to run the model sequence over the phrase "The girl walks," our input should look as follows (Figs. 11 and 12):

[girl, is, walking]

Ex 2:
An LSTM for part-of-speech tagging.

Problem Definition Part-of-speech labeling is an outstanding assignment in natural language processing. It alludes to the way toward ordering words into their grammatical forms (otherwise called word classes or lexical classifications). This is an administered learning approach as LSTM is an extension of RNN where it shows the inputs of participants.

```
lstm = nn.LSTM(3, 3)  # Input dim is 3, output dim is 3
inputs = [torch.randn(1, 3) for _ in range(5)]  # make a sequence of length 5

# initialize the hidden state.
hidden = (torch.randn(1, 1, 3),
          torch.randn(1, 1, 3))
for i in inputs:
    # Step through the sequence one element at a time.
    # after each step, hidden contains the hidden state.
    out, hidden = lstm(i.view(1, 1, -1), hidden)

# alternatively, we can do the entire sequence all at once.
# the first value returned by LSTM is all of the hidden states throughout
# the sequence. the second is just the most recent hidden state
# (compare the last slice of "out" with "hidden" below, they are the same)
# The reason for this is that:
# "out" will give you access to all hidden states in the sequence
# "hidden" will allow you to continue the sequence and backpropagate,
# by passing it as an argument  to the lstm at a later time
# Add the extra 2nd dimension
inputs = torch.cat(inputs).view(len(inputs), 1, -1)
hidden = (torch.randn(1, 1, 3), torch.randn(1, 1, 3))  # clean out hidden state
out, hidden = lstm(inputs, hidden)
print(out)
print(hidden)
```

Fig. 11 LSTM implementation in PyTorch Part 1

```
tensor([[[-0.0187,  0.1713, -0.2944]],

        [[-0.3521,  0.1026, -0.2971]],

        [[-0.3191,  0.0781, -0.1957]],

        [[-0.1634,  0.0941, -0.1637]],

        [[-0.3368,  0.0959, -0.0538]]], grad_fn=<StackBackward>)
(tensor([[[-0.3368,  0.0959, -0.0538]]], grad_fn=<StackBackward>),
tensor([[[-0.9825,  0.4715, -0.0633]]], grad_fn=<StackBackward>))
```

Fig. 12 LSTM implementation in PyTorch Part 2

However, there is an additional second dimension with size 1.

A unique integral was assigned to each term (and tag). We measure a set of unique words (and tags), then turn them into a list, and index them into a dictionary [24]. The word vocabulary and the tag vocabulary are those dictionaries. We will also add a special padding value for the sequences (more on that later), and another one for unknown vocabulary words (Fig. 13).

```
def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)


training_data = [
    ("The dog ate the apple".split(), ["DET", "NN", "V", "DET", "NN"]),
    ("Everybody read that book".split(), ["NN", "V", "DET", "NN"])
]
word_to_ix = {}
for sent, tags in training_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)
tag_to_ix = {"DET": 0, "NN": 1, "V": 2}

# These will usually be more like 32 or 64 dimensional.
# We will keep them small, so we can see how the weights change as we train.
EMBEDDING_DIM = 6
HIDDEN_DIM = 6
```

Fig. 13 PyTorch implementation on vocabulary words

```
class LSTMTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs, and outputs hidden states
        # with dimensionality hidden_dim.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, _ = self.lstm(embeds.view(len(sentence), 1, -1))
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

Fig. 14 Code implementation of LSTM tagger

In preprocessing the sequence is generated. And words are generated from the sentences using nltk library. The tokens are indexed and assigned the corresponding index as given in Fig. 14.

```
model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_ix), len(tag_to_ix))
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# See what the scores are before training
# Note that element i,j of the output is the score for tag j for word i.
# Here we don't need to train, so the code is wrapped in torch.no_grad()
with torch.no_grad():
    inputs = prepare_sequence(training_data[0][0], word_to_ix)
    tag_scores = model(inputs)
    print(tag_scores)
```

Fig. 15 Code implementation of parameter tuning

```
for epoch in range(300):  # again, normally you would NOT do 300 epochs, it is toy
data
    for sentence, tags in training_data:
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is, turn them into
        # Tensors of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = prepare_sequence(tags, tag_to_ix)

        # Step 3. Run our forward pass.
        tag_scores = model(sentence_in)

        # Step 4. Compute the loss, gradients, and update the parameters by
        #  calling optimizer.step()
        loss = loss_function(tag_scores, targets)
        loss.backward()
        optimizer.step()
```

Fig. 16 Epoch initialization and model fit

The model was built using the parameter tuned (Fig. 15).
The process is repeated until we get the best fit model (Figs. 16 and 17).

9 Summary

- Deep network models robotically learn to associate inputs and favored outputs from examples.
- Libraries like PyTorch can help you construct and educate neural network fashions efficiently.

```
tensor([[-1.1389, -1.2024, -0.9693],
        [-1.1065, -1.2200, -0.9834],
        [-1.1286, -1.2093, -0.9726],
        [-1.1190, -1.1960, -0.9916],
        [-1.0137, -1.2642, -1.0366]])
tensor([[-0.0462, -4.0106, -3.6096],
        [-4.8205, -0.0286, -3.9045],
        [-3.7876, -4.1355, -0.0394],
        [-0.0185, -4.7874, -4.6013],
        [-5.7881, -0.0186, -4.1778]])
```

Fig. 17 Output matrix value

- PyTorch minimizes cognitive overhead, while focusing on flexibility and speed. It additionally defaults to immediate execution for operations.
- TorchScript is a pre-compiled deferred execution mode that can be invoked from Cpp.
- PyTorch gives some of software libraries to facilitate deep studying projects.
- PyTorch is used in a variety of deep learning applications like object detection, image analysis, and sequence modeling.

References

1. Deep Learning for Computer Vision: Expert techniques to train advanced neural networks using TensorFlow and Keras. [Authors: RajalingappaaShanmugamani]
2. Deep Learning in Python: Master Data Science and Machine Learning with Modern Neural Networks written in Python, Theano, and TensorFlow. [Authors: LazyProgrammer]
3. Deep learning quick reference: useful hacks for training and optimizing deep neural networks with TensorFlow and Keras. [Authors: Bernico, Mike]
4. Deep Learning with TensorFlow: Explore neural networks with Python [Authors: Giancarlo Zaccone, Md. RezaulKarim, Ahmed Menshawy]
5. Erdmann M, Glombitza J, Walz D. A deep learning-based reconstruction of cosmic ray-induced air showers. AstropartPhys 2018;97:46–53. doi:https://doi.org/10.1016/j.astropartphys.2017.10.006, URL http://www.sciencedirect.com/science/article/pii/S0927650517302219.
6. Feng Q, Lin TTY. The analysis of VERITAS muon images using convolutional neural networks, in: Proceedings of the International Astronomical Union, vol. 12, 2016.
7. Goodfellow I, Bengio Y, Courville A, Bengio Y (2016) Deep learning, vol 1. MIT Press, Cambridge
8. Grandison T, Sloman M (2000) A survey of trust in internet applications. IEEE CommunSurv Tutor 3(4):2–16

9. Guo X, Ipek E, Soyata T (2010) Resistive computation: avoiding the power wall with low-leakage, STT-MRAM based computing. In: ACM SIGARCH computer architecture news, vol 38. ACM, pp 371–382
10. Hands-on unsupervised learning with Python: implement machine learning and deep learning models using Scikit-Learn, TensorFlow, and more [Authors: Bonaccorso, Giuseppe]
11. Holch TL, Shilon I, Büchele M, Fischer T, Funk S, Groeger N, Jankowsky D, Lohse T, Schwanke U, Wagner P. Probing convolutional neural networks forevent reconstruction in γ -ray astronomy with Cherenkov telescopes, in:PoS ICRC2017, The Fluorescence detector Array of Single-pixel Telescopes:Contributions to the 35th International Cosmic Ray Conference (ICRC2017), p. 795, https://doi.org/10.22323/1.301.0795, arXiv:1711.06298.
12. Huennefeld M. Deep learning in physics exemplified by the reconstruction of muon-neutrino events in IceCube, in: PoS ICRC2017, The Fluorescence detector Array of Single-pixel Telescopes: Contributions to the 35th International Cosmic Ray Conference (ICRC 2017), p. 1057, https://doi.org/10.22323/1.301.1057.
13. Hurst S (1969) An introduction to threshold logic: a survey of present theory and practice. Radio Electron Eng 37(6):339–351
14. Intelligent Projects Using Python: 9 real-world AI projects leveraging machine learning and deep learning with TensorFlow and Keras. [Authors: SantanuPattanayak]
15. Internet of Things for Industry 4.0, EAI, Springer, Editors, G. R. Kanagachidambaresan, R. Anand, E. Balasubramanian and V. Mahima, Springer.
16. Jeong H, Shi L (2018) Memristor devices for neural networks. J Phys D: ApplPhys 52(2):023003
17. Krestinskaya O, Dolzhikova I, James AP (2018) Hierarchical temporal memory using memristor networks: a survey. IEEE Trans Emerg Top ComputIntell 2(5):380–395. doi:https://doi.org/10.1109/TETCI.2018.2838124
18. LeCun Y, Bengio Y, Hinton G. Deep learning. Nature 2015;521(7553):436–44. doi:https://doi.org/10.1038/nature14539.
19. Mangano S, Delgado C, Bernardos M, Lallena M, Vzquez JJR. Extracting gamma-ray information from images with convolutional neural networkmethods on simulated cherenkov telescope array data. In: ANNPR 2018. LNAI, vol. 11081, 2018, p. 243–54. doi:https://doi.org/10.1007/978-3-319-99978-4, arXiv:1810.00592.
20. Mastering TensorFlow 1.x: Advanced machine learning and deep learning concepts using TensorFlow 1.x and Keras. [Author: Armando Fandango]
21. Practical Deep Learning for Cloud, Mobile, and Edge: Real-World AI & Computer-Vision Projects Using Python, Keras&TensorFlow [Authors: AnirudhKoul, Siddha Ganju, MeherKasam]
22. Python Deep Learning: Exploring deep learning techniques, neural network architectures and GANs with PyTorch, Keras and TensorFlow. [Authors: Ivan Vasilev, Daniel Slater, GianmarioSpacagna, Peter Roelants, Valentino Zocca]
23. Shilon I, Kraus M, Büchele M, Egberts K, Fischer T, HolchTL, Lohse T, Schwanke U, Steppa C, Funk S. Application of deep learning methods to analysis of imaging atmospheric cherenkov telescopes data. AstropartPhys 2019; 105: 44–53. doi:https://doi.org/10.1016/j.astropartphys.2018.10.003, URL http://www.sciencedirect.com/science/article/pii/S0927650518301178
24. TensorFlow 1.x Deep Learning Cookbook: Over 90 unique recipes to solve artificial-intelligence driven problems with Python. [Authors: Antonio Gulli, AmitaKapoor]