10
Transfer Learning

In this chapter, we will discuss the concept of Transfer Learning. The following are the topics that will be covered:

- Illustrating the use of a pretrained model
- Setting up the Transfer Learning model
- Building an image classification model
- Training a deep learning model on a GPU
- Comparing performance using CPU and GPU

Introduction

A lot of development has happened within the deep learning domain in recent years, to enhance algorithmic efficacy and computational efficiency across different domains such as text, images, audio, and video. However, when it comes to training on new datasets, machine learning usually rebuilds the model from scratch, as is done in traditional data science problem solving. This becomes challenging when a new big dataset need to be trained as it will require very high computation power a lot of and time to reach the desired model efficacy.

Transfer Learning is a mechanism to learn new scenarios from existing models. This approach is very useful to train on big datasets, not necessarily from a similar domain or problem statement. For example, researchers have shown examples of Transfer Learning where they have trained Transfer Learning for completely different problem scenarios, such as when a model built using classifications of cat and dog is used for classifying objects such as aeroplane vs automobile.

In terms of analogy, it's more about passing the learned relationship to new architecture in order to fine-tune weights. An example of how Transfer Learning is used is shown in the following figure:

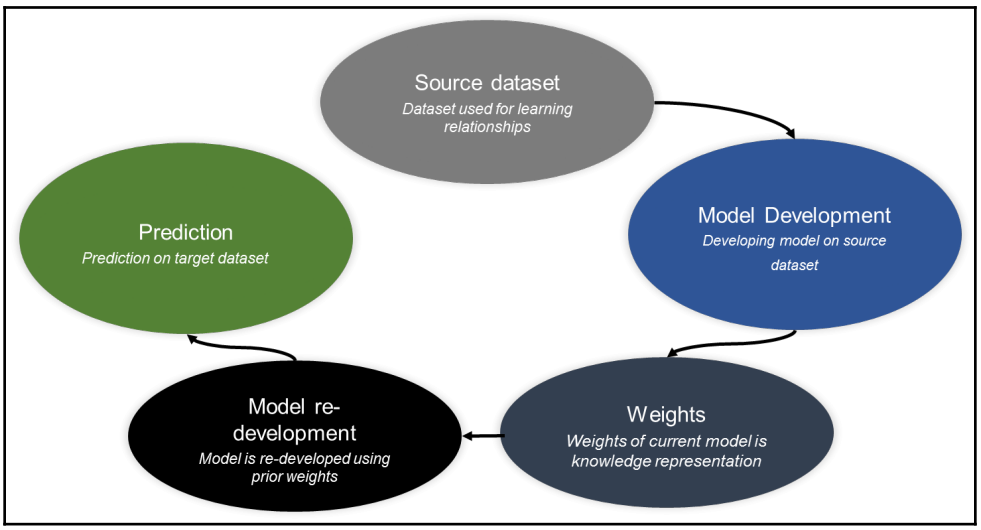


Illustration of Transfer Learning flow

The figure shows the steps of Transfer Learning, where the weights/architectures from a predeveloped deep learning model are reused to predict a new problem statement. Transfer Learning helps provide a good starting point for deep learning architectures. There are different open source projects going on in different domains, which facilitate Transfer Learning, for example, ImageNet (`http://image-net.org/index`) is an open source project for image classification where a lot of different architectures such as Alexnet, VGG16, and VGG19 have been developed. Similarly, in text mining, there is a Word2Vec representation of Google News trained using three billion running words.

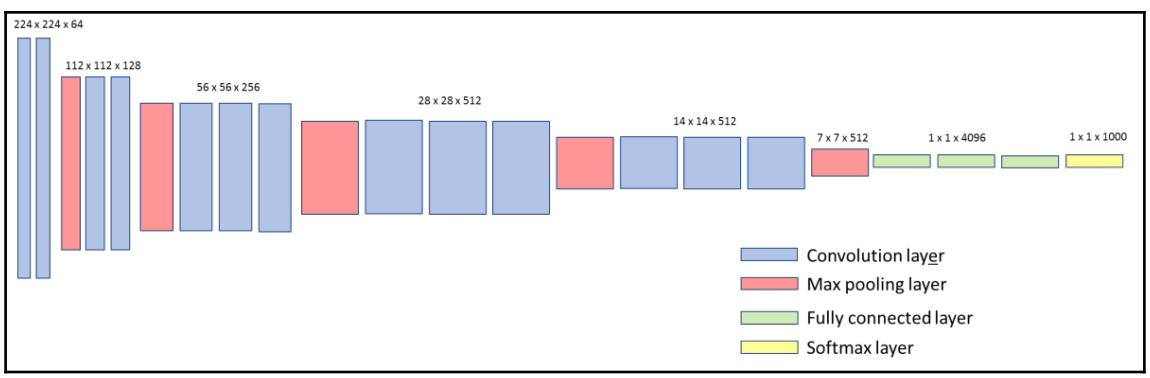
Details on word2vec can be found at `https://code.google.com/archive/p/word2vec/`.

Illustrating the use of a pretrained model

The current recipe will cover the set-up for using a pretrained model. We will use TensorFlow to demonstrate the recipe. The current recipe will use VGG16 architecture built using the ImageNet as dataset. The ImageNet is an open source image repository of images used for building image recognition algorithms. The database has more than 10 millions tagged images and more than 1 million images have bounding box to capture objects.

Lot of different deep learning architectures are developed using ImageNet dataset. Once of the popular one is VGG networks are convolution neural networks proposed by Zisserman and Simonyan (2014) and trained over ImageNet data with 1,000 classes. The current recipe will consider VGG16 variant of VGG architecture which is known for it's simplicity. The network uses input of 224 x 224 RGB image. The network utilizes 13 convolution layers with different width x height x depth. The maximum pooling layer is used to reduce volume size. The network uses 5 maxpooling layer. The output from convolution layer is passed through 3 fully connected layer. Outcome from fully connected layer go through softmax function to evaluate probability of 1000 classes.

The detailed architecture of VGG16 is shown in the following figure:



VGG16 architecture

Getting ready

The section covers required to use VGG16 pretrained model for classification.

1. Download VGG16 weights from `http://download.tensorflow.org/models/vgg_16_2016_08_28.tar.gz`. The file can be downloaded using the following script:

```
require(RCurl)
URL <-
'http://download.tensorflow.org/models/vgg_16_2016_08_28.tar.gz'
download.file(URL,destfile="vgg_16_2016_08_28.tar.gz",method="libcurl")
```

2. Install TensorFlow in Python.
3. Install R and the `tensorflow` package in R.
4. Download a sample image from `http://image-net.org/download-imageurls`.

How to do it...

The current section provides steps to use pretrained models:

1. Load `tensorflow` in R:

```
require(tensorflow)
```

2. Assign the `slim` library from TensorFlow:

```
slimobj = tf$contrib$slim
```

   The `slim` library in TensorFlow is used to maintain complex neural network models in terms of definition, training, and evaluation.

3. Reset graph in TensorFlow:

```
tf$reset_default_graph()
```

4. Define input images:

```
# Resizing the images
input.img= tf$placeholder(tf$float32, shape(NULL, NULL, NULL, 3))
scaled.img = tf$image$resize_images(input.img, shape(224,224))
```

5. Redefine the VGG16 network:

```
# Define VGG16 network
library(magrittr)
VGG16.model<-function(slim, input.image){
  vgg16.network = slim$conv2d(input.image, 64, shape(3,3),
scope='vgg_16/conv1/conv1_1') %>%
    slim$conv2d(64, shape(3,3), scope='vgg_16/conv1/conv1_2')  %>%
    slim$max_pool2d( shape(2, 2), scope='vgg_16/pool1')  %>%
    slim$conv2d(128, shape(3,3), scope='vgg_16/conv2/conv2_1')  %>%
    slim$conv2d(128, shape(3,3), scope='vgg_16/conv2/conv2_2')  %>%
    slim$max_pool2d( shape(2, 2), scope='vgg_16/pool2')  %>%
    slim$conv2d(256, shape(3,3), scope='vgg_16/conv3/conv3_1')  %>%
    slim$conv2d(256, shape(3,3), scope='vgg_16/conv3/conv3_2')  %>%
    slim$conv2d(256, shape(3,3), scope='vgg_16/conv3/conv3_3')  %>%
    slim$max_pool2d(shape(2, 2), scope='vgg_16/pool3')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv4/conv4_1')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv4/conv4_2')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv4/conv4_3')  %>%
    slim$max_pool2d(shape(2, 2), scope='vgg_16/pool4')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv5/conv5_1')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv5/conv5_2')  %>%
    slim$conv2d(512, shape(3,3), scope='vgg_16/conv5/conv5_3')  %>%
    slim$max_pool2d(shape(2, 2), scope='vgg_16/pool5')  %>%
    slim$conv2d(4096, shape(7, 7), padding='VALID',
scope='vgg_16/fc6')  %>%
    slim$conv2d(4096, shape(1, 1), scope='vgg_16/fc7') %>%
    slim$conv2d(1000, shape(1, 1), scope='vgg_16/fc8')  %>%
    tf$squeeze(shape(1, 2), name='vgg_16/fc8/squeezed')
  return(vgg16.network)
}
```

6. The preceding function defines the network architecture used for the VGG16 network. The network can be assigned using the following script:

```
vgg16.network<-VGG16.model(slim, input.image = scaled.img)
```

7. Load the VGG16 weights `vgg_16_2016_08_28.tar.gz` downloaded in the *Getting started* section:

```
# Restore the weights
restorer = tf$train$Saver()
sess = tf$Session()
restorer$restore(sess, 'vgg_16.ckpt')
```

8. Download a sample test image. Let's download an example image from a `testImgURL` location as shown in following script:

```
# Evaluating using VGG16 network
testImgURL<-
"http://farm4.static.flickr.com/3155/2591264041_273abea408.jpg"
img.test<-tempfile()
download.file(testImgURL,img.test, mode="wb")
read.image <- readJPEG(img.test)
# Clean-up the temp file
file.remove(img.test)
```

The preceding script downloads the following image from URL mention in variable `testImgURL`. The following is the downloaded image:

Sample image used to evaluate imagenet

9. Determine the class using the VGG16 pretrained model:

```
## Evaluate
size = dim(read.image)
imgs = array(255*read.image, dim = c(1, size[1], size[2], size[3]))
VGG16_eval = sess$run(vgg16.network, dict(images = imgs))
probs = exp(VGG16_eval)/sum(exp(VGG16_eval))
```

The maximum probability achieved is 0.62 for class 672, which refers to the category--mountain bike, all-terrain bike, off-roader--in the VGG16 trained dataset.

Setting up the Transfer Learning model

The current recipe will cover Transfer Learning using the CIFAR-10 dataset. The previous recipe presented how to use a pretrained model. The current recipe will demonstrate how to use a pretrained model for different problem statements.

We will use another very good deep learning package, MXNET, to demonstrate the concept with another architecture, Inception. To simplify the computation, we will reduce the problem complexity from 10 classes to two classes: aeroplane and automobile. The recipe focuses on data preparation for Transfer Learning using Inception-BN.

Getting ready

The section prepares for the upcoming section for setting-up Transfer Learning model.

1. Download the CIFAR-10 dataset from `http://www.cs.toronto.edu/~kriz/cifar.html`. The `download.cifar.data` function from `Chapter 3`, *Convolution Neural Networks*, can be used to download the dataset.
2. Install the `imager` package:

```
install.packages("imager")
```

How to do it...

The current part of the recipe will provide a step-by-step guide to prepare the dataset for the Inception-BN pretrained model.

1. Load the dependent packages:

```
# Load packages
require(imager)
source("download_cifar_data.R")
The download_cifar_data consists of function to download and read
CIFAR10 dataset.
```

2. Read the downloaded CIFAR-10 dataset:

```
# Read Dataset and labels
DATA_PATH<-paste(SOURCE_PATH, "/Chapter 4/data/cifar-10-batches-
bin/", sep="")
labels <- read.table(paste(DATA_PATH, "batches.meta.txt", sep=""))
cifar_train <- read.cifar.data(filenames =
```

```
c("data_batch_1.bin","data_batch_2.bin","data_batch_3.bin","data_ba
tch_4.bin"))
```

3. Filter the dataset for aeroplane and automobile. This is an optional step and is done to reduce complexity later:

```
# Filter data for Aeroplane and Automobile with label 1 and 2,
respectively
Classes = c(1, 2)
images.rgb.train <- cifar_train$images.rgb
images.lab.train <- cifar_train$images.lab
ix<-images.lab.train%in%Classes
images.rgb.train<-images.rgb.train[ix]
images.lab.train<-images.lab.train[ix]
rm(cifar_train)
```

4. Transform to image. This step is required as the CIFAR-10 dataset is a 32 x 32 x 3 image, which is flattened to a 1024 x 3 format:

```
# Function to transform to image
transform.Image <- function(index, images.rgb) {
  # Convert each color layer into a matrix,
  # combine into an rgb object, and display as a plot
  img <- images.rgb[[index]]
  img.r.mat <- as.cimg(matrix(img$r, ncol=32, byrow = FALSE))
  img.g.mat <- as.cimg(matrix(img$g, ncol=32, byrow = FALSE))
  img.b.mat <- as.cimg(matrix(img$b, ncol=32, byrow = FALSE))

  # Bind the three channels into one image
  img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
  return(img.col.mat)
}
```

5. The next step involve padding images with zeros:

```
  # Function to pad image
  image.padding <- function(x) {
  img_width <- max(dim(x)[1:2])
  img_height <- min(dim(x)[1:2])
  pad.img <- pad(x, nPix = img_width - img_height,
                 axes = ifelse(dim(x)[1] < dim(x)[2], "x", "y"))
  return(pad.img)
}
```

6. Save the image to a specified folder:

```
# Save train images
MAX_IMAGE<-length(images.rgb.train)

# Write Aeroplane images to aero folder
sapply(1:MAX_IMAGE, FUN=function(x, images.rgb.train,
images.lab.train){
  if(images.lab.train[[x]]==1){
    img<-transform.Image(x, images.rgb.train)
    pad_img <- image.padding(img)
    res_img <- resize(pad_img, size_x = 224, size_y = 224)
    imager::save.image(res_img, paste("train/aero/aero", x,
".jpeg", sep=""))
  }
}, images.rgb.train=images.rgb.train,
images.lab.train=images.lab.train)


# Write Automobile images to auto folder
sapply(1:MAX_IMAGE, FUN=function(x, images.rgb.train,
images.lab.train){
  if(images.lab.train[[x]]==2){
    img<-transform.Image(x, images.rgb.train)
    pad_img <- image.padding(img)
    res_img <- resize(pad_img, size_x = 224, size_y = 224)
    imager::save.image(res_img, paste("train/auto/auto", x,
".jpeg", sep=""))
  }
}, images.rgb.train=images.rgb.train,
images.lab.train=images.lab.train)
```

The preceding script saves the aeroplane images into the `aero` folder and the automobile images in the `auto` folder.

7. Convert to the recording format `.rec` supported by MXNet. This conversion requires `im2rec.py` MXnet module from Python as conversion is not supported in R. However, it can be called from R once MXNet is installed in Python using the system command. The splitting of the dataset into train and test can be obtained using the following file:

```
System("python ~/mxnet/tools/im2rec.py --list True --recursive True
--train-ratio 0.90 cifar_224/pks.lst cifar_224/trainf/")
```

The preceding script will generate two list files: `pks.lst_train.lst` and `pks.lst_train.lst`. The splitting of train and validation is controlled by the `–train-ratio` parameter in the preceding script. The number of classes is based on the number of folders in the `trainf` directory. In this scenario, two classes are picked: automotive and aeroplane.

8. Convert the `*.rec` file for training and validation dataset:

```
# Creating .rec file from training sample list
System("python ~/mxnet/tools/im2rec.py --num-thread=4 --pass-
through=1 /home/prakash/deep\ learning/cifar_224/pks.lst_train.lst
/home/prakash/deep\ learning/cifar_224/trainf/")

# Creating .rec file from validation sample list
System("python ~/mxnet/tools/im2rec.py --num-thread=4 --pass-
through=1 /home/prakash/deep\ learning/cifar_224/pks.lst_val.lst
/home/prakash/deep\ learning/cifar_224/trainf/")
```

The preceding script will create the `pks.lst_train.rec` and `pks.lst_val.rec` files to be used in the next recipe to train the model using a pretrained model.

Building an image classification model

The recipe focuses on building an image classification model using Transfer Learning. It will utilize the dataset prepared in the previous recipes and use the Inception-BN architecture. The BN in Inception-BN stands for **batch normalization**. Details of the Inception model in computer vision can be found in Szegedy et al. (2015).

Getting ready

The section cover's the prerequisite to set-up a classification model using INCEPTION-BN pretrained model.

1. Convert images into `.rec` file for train and validation.
2. Download the Inception-BN architecture from `http://data.dmlc.ml/models/imagenet/inception-bn/`.
3. Install R and the `mxnet` package in R.

How to do it...

1. Load the `.rec` file as iterators. The following is the function to load the `.rec` data as iterators:

```
# Function to load data as iterators
data.iterator <- function(data.shape, train.data, val.data,
BATCHSIZE = 128) {
  # Load training data as iterator
  train <- mx.io.ImageRecordIter(
    path.imgrec = train.data,
    batch.size  = BATCHSIZE,
    data.shape  = data.shape,
    rand.crop   = TRUE,
    rand.mirror = TRUE)
  # Load validation data as iterator
  val <- mx.io.ImageRecordIter(
    path.imgrec = val.data,
    batch.size  = BATCHSIZE,
    data.shape  = data.shape,
    rand.crop   = FALSE,
    rand.mirror = FALSE
  )
  return(list(train = train, val = val))
}
```

   In the preceding function, `mx.io.ImageRecordIter` reads batches of images from the `RecordIO` (`.rec`) files.

2. Load data using the `data.iterator` function:

```
# Load dataset
data  <- data.iterator(data.shape = c(224, 224, 3),
                       train.data = "pks.lst_train.rec",
                       val.data  = "pks.lst_val.rec",
                       BATCHSIZE = 8)
train <- data$train
val   <- data$val
```

3. Load the Inception-BN pretrained model from the `Inception-BN` folder:

```
# Load Inception-BN model
inception_bn <- mx.model.load("Inception-BN", iteration = 126)
symbol <- inception_bn$symbol
The different layers of the model can be viewed using function
symbol$arguments
```

4. Get the layers of the Inception-BN model:

```
# Load model information
internals <- symbol$get.internals()
outputs <- internals$outputs
flatten <- internals$get.output(which(outputs == "flatten_output"))
```

5. Define a new layer to replace the `flatten_output` layer:

```
# Define new layer
new_fc <- mx.symbol.FullyConnected(data = flatten,
                                   num_hidden = 2,
                                   name = "fc1")
new_soft <- mx.symbol.SoftmaxOutput(data = new_fc,
                                    name = "softmax")
```

6. Initialize weights for the newly defined layer. To retrain the last layer, weight initialization is performed using the following script:

```
# Re-initialize the weights for new layer
arg_params_new <- mxnet:::mx.model.init.params(
  symbol = new_soft,
  input.shape = c(224, 224, 3, 8),
  output.shape = NULL,
  initializer = mxnet:::mx.init.uniform(0.2),
  ctx = mx.cpu(0)
)$arg.params
fc1_weights_new <- arg_params_new[["fc1_weight"]]
fc1_bias_new <- arg_params_new[["fc1_bias"]]
```

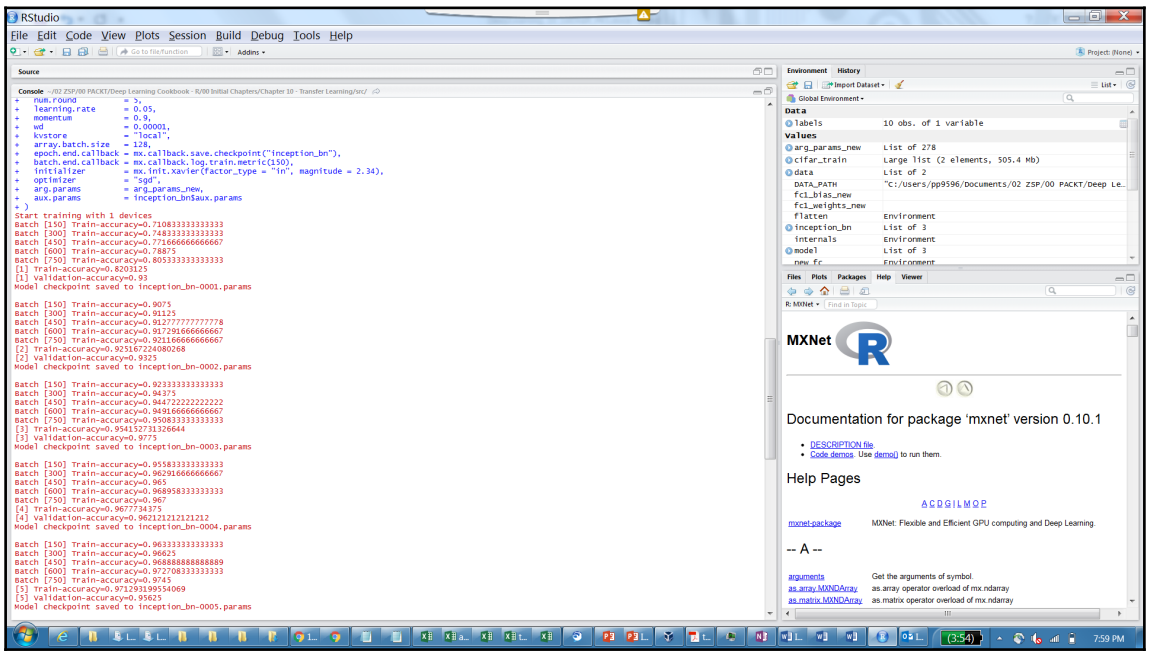
In the aforementioned layer, weights are assigned using uniform distribution between [*-0.2, 0.2*]. The `ctx` define the device on which the execution is to be performed.

7. Retrain the model:

```
# Mode re-train
model <- mx.model.FeedForward.create(
  symbol             = new_soft,
  X                  = train,
  eval.data          = val,
  ctx                = mx.cpu(0),
  eval.metric        = mx.metric.accuracy,
  num.round          = 5,
  learning.rate      = 0.05,
  momentum           = 0.85,
  wd                 = 0.00001,
```

```
    kvstore            = "local",
    array.batch.size   = 128,
    epoch.end.callback = mx.callback.save.checkpoint("inception_bn"),
    batch.end.callback = mx.callback.log.train.metric(150),
    initializer        = mx.init.Xavier(factor_type = "in", magnitude
  = 2.34),
    optimizer          = "sgd",
    arg.params         = arg_params_new,
    aux.params         = inception_bn$aux.params
  )
```

The preceding model is set to run on CPU with five rounds, using accuracy as the evaluation metric. The following screenshot shows the execution of the described model:



Output from Inception-BN model, trained using CIFAR-10 dataset

The trained model has produced a training accuracy of 0.97 and a validation accuracy of 0.95.

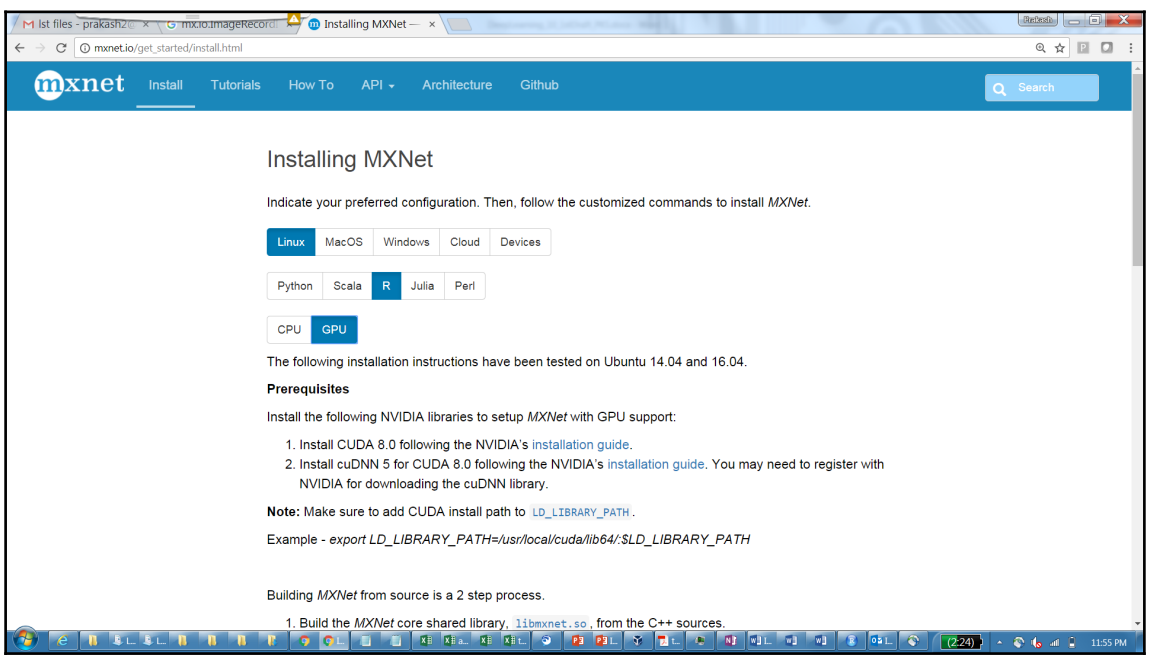
Training a deep learning model on a GPU

The **Graphical processing unit** (**GPU**) is hardware used for rendering images using a lot of cores. Pascal is the latest GPU micro architecture released by NVIDIA. The presence of hundreds of cores in GPU helps enhance the computation. This section provides the recipe for running a deep learning model using GPU.

Getting ready

This section provides dependencies required to run GPU and CPU:

1. The experiment performed in this recipe uses GPU hardware such as GTX 1070.
2. Install `mxnet` for GPU. To install `mxnet` for GPU for a specified machine, follow the installation instruction from `mxnet.io`. Select the requirement as shown in the screenshot, and follow the instructions:



Steps to get installation instruction for MXNet

How to do it...

Here is how you train a deep learning model on a GPU:

1. The Inception-BN-transferred learning recipe discussed in the previous section can be made to run on the GPU installed and the configured machine by changing the device settings as shown in the following script:

```
# Mode re-train
model <- mx.model.FeedForward.create(
  symbol             = new_soft,
  X                  = train,
  eval.data          = val,
  ctx                = mx.gpu(0),
  eval.metric        = mx.metric.accuracy,
  num.round          = 5,
  learning.rate      = 0.05,
  momentum           = 0.85,
  wd                 = 0.00001,
  kvstore            = "local",
  array.batch.size   = 128,
  epoch.end.callback = mx.callback.save.checkpoint("inception_bn"),
  batch.end.callback = mx.callback.log.train.metric(150),
  initializer        = mx.init.Xavier(factor_type = "in", magnitude
= 2.34),
  optimizer          = "sgd",
  arg.params         = arg_params_new,
  aux.params         = inception_bn$aux.params
)
```

In the aforementioned model, the device setting is changed from `mx.cpu` to `mx.gpu`. The CPU for five iteration tools takes ~2 hours of computational effort, whereas the same iteration is completed in ~15 min with GPU.

Comparing performance using CPU and GPU

One of the questions with device change is why so much improvement is observed when the device is switched from CPU to GPU. As the deep learning architecture involves a lot of matrix computations, GPUs help expedite these computations using a lot of parallel cores, which are usually used for image rendering.

The power of GPU has been utilized by a lot of algorithms to accelerate the execution. The following recipe provides some benchmarks of matrix computation using the `gpuR` package. The `gpuR` package is a general-purpose package for GPU computing in R.

Getting ready

The section covers requirement to set-up a comparison between GPU Vs CPU.

1. Use GPU hardware installed such as GTX 1070.
2. CUDA toolkit installation using URL `https://developer.nvidia.com/cuda-downloads`.
3. Install the `gpuR` package:

```
install.packages("gpuR")
```

4. Test `gpuR`:

```
library(gpuR)
# verify you have valid GPUs
detectGPUs()
```

How to do it...

Let's get started by loading the packages:

1. Load the package, and set the precision to `float` (by default, the GPU precision is set to a single digit):

```
library("gpuR")
options(gpuR.default.type = "float")
```

2. Matrix assignment to GPU:

```
# Assigning a matrix to GPU
A<-matrix(rnorm(1000), nrow=10)

vcl1 = vclMatrix(A)
```

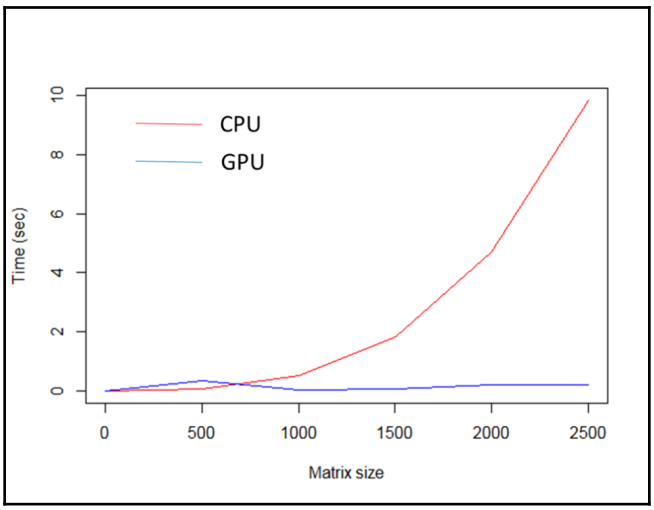
The output of the preceding command will contain details of the object. An illustration is shown in the following script:

```
> vcl1
An object of class "fvclMatrix"
Slot "address":
<pointer: 0x000000001822e180>

Slot ".context_index":
[1] 1

Slot ".platform_index":
[1] 1

Slot ".platform":
[1] "Intel(R) OpenCL"

Slot ".device_index":
[1] 1

Slot ".device":
[1] "Intel(R) HD Graphics 530"
```

3. Let's consider the evaluation of CPU vs GPU. As most of the deep learning will be using GPUs for matrix computation, the performance is evaluated by matrix multiplication using the following script:

```
# CPU vs GPU performance
DF <- data.frame()
evalSeq<-seq(1,2501,500)
for (dimpower in evalSeq){
  print(dimpower)
  Mat1 = matrix(rnorm(dimpower^2), nrow=dimpower)
  Mat2 = matrix(rnorm(dimpower^2), nrow=dimpower)
  now <- Sys.time()
  Matfin = Mat1%*%Mat2
  cpu <- Sys.time()-now
  now <- Sys.time()
  vcl1 = vclMatrix(Mat1)
  vcl2 = vclMatrix(Mat2)
  vclC = vcl1 %*% vcl2
  gpu <- Sys.time()-now
  DF <- rbind(DF,c(nrow(Mat1), cpu, gpu))
}
DF<-data.frame(DF)
colnames(DF) <- c("nrow", "CPU_time", "gpu_time")
```

The preceding script computes the matrix multiplication using CPU and GPU; the time is stored for different dimensions of the matrix. The output from the preceding script is shown in the following diagram:



Comparison between CPU and GPU

The graph shows that the computation effort required by CPUs increases exponentially with the CPU. Thus, GPUs help expedite it drastically.

There's more...

The GPUs are new arena in machine learning computing, and a lot of packages have been developed in R to access GPUs while keeping you in a familiar R environment such as `gputools`, `gmatrix`, and `gpuR`. The other algorithms are also developed and implemented while accessing GPUs to enhance their computational power such as `RPUSVM`, which implements SVM using GPUs. Thus, the topic requires a lot of creativity with some exploration to deploy algorithms while utilizes full capability of hardware.

See also

To learn more on parallel computing using R, go through *Mastering Parallel Programming with R* by Simon R. Chapple et al. (2016).