Kuntal Ganguly

Learning Generative Adversarial Networks

Next-generation deep learning simplified

Packt

Table of Contents

Learning Generative Adversarial Networks

Learning Generative Adversarial Networks

First published: October 2017
Production reference: 1241017
Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.
ISBN 978-1-78839-641-7
www.packtpub.com

Credits

Author
Kuntal Ganguly
Reviewer
Max Strakhov
Commissioning Editor
Amey Varangaonkar
Acquisition Editor
Divya Poojari
Content Development Editor
Dattatraya More
Technical Editor
Jovita Alva
Copy Editor
Safis Editing
Project Coordinator
Shweta H Birwatkar
Proofreader
Safis Editing
Indexer
Rekha Nair
Graphics
Tania Dutta
Production Coordinator
Shantanu Zagade
Cover Work
Shantanu Zagade

About the Author

Kuntal Ganguly is a big data analytics engineer focused on building large-scale, data-driven systems using big data frameworks and machine learning. He has around 7 years experience of building big data and machine learning applications.

Kuntal provides solutions to cloud customers in building real-time analytics systems using managed cloud services and open source Hadoop ecosystem technologies such as Spark, Kafka, Storm, Solr, and so on, along with machine learning and deep learning frameworks. Kuntal enjoys hands-on software development and has single-handedly conceived, architected, developed, and deployed several large-scale distributed applications. He is a machine learning and deep learning practitioner and is very passionate about building intelligent applications. His LinkedIn profile is as follows: https://in.linkedin.com/in/kuntal-ganguly-59564088

I am grateful to my mother, Chitra Ganguly, and father, Gopal Ganguly, for their love, support, and for teaching me a lot about hard work—even the little I have absorbed has helped me immensely throughout my life. I would also like to thank all my friends, colleagues, and mentors that I've had over the years.

About the Reviewer

Max Strakhov is a research and software engineer with over 8 years experience in computer programming and over 4 years experience in machine learning. He has worked at Google and Yandex and is a cofounder and a CTO of AURA Devices LLC.
He is interested in deep learning and its applications in artificial intelligence, especially in generative modelling. He has a blog, monnoroch.github.io, where he shares posts about deep learning, software engineering, and all things related to technology.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <customercare@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1788396413.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

I believe that Data Science and Artificial Intelligence will give us superpowers.

Preface

The concepts and models (reviewed in this book) will help to build and effectively take deep networks from the arena of supervised task such as image classification, toward the unsupervised learning with creative power.

Using the basic generative network concepts, you'll learn to generate realistic images from unlabeled data, to synthesize images from textual descriptions, and to discover relationships across various domains for style transfer.

What this book covers

Chapter 1, *Introduction to Deep Learning*, speaks all about refreshing general concepts and terminology associated with deep learning in a simple way without too much math and equations. Also, it will show how deep learning network has evolved throughout the years and how they are making an inroad in the unsupervised domain with the emergence of generative models.

Chapter 2, *Unsupervised Learning with GAN*, shows how Generative Adversarial Networks work and speaks about the building blocks of GANs. It will show how deep learning networks can be used on semi-supervised domains, and how you can apply them to image generation and creativity. GANs are hard to train. This chapter looks at some techniques to improve the training/learning process.

Chapter 3, *Transfer Image Style Across Various Domains,* speaks about being very creative with simple but powerful CGAN and CycleGAN models. It explains the use of Conditional GAN to create images based on certain characteristics or conditions. This chapter also discusses how to overcome model collapse problems by stabilizing your network training using BEGAN. And finally, it covers transferring styles across different domains (apple to orange; horse to zebra) using CycleGAN.

Chapter 4, *Building Realistic Images from Your Text*, presents the latest approach of stacking Generative Adversarial Networks into multiple stages to decompose the problem of text to image synthesis into two more manageable subproblems with StackGAN. The chapter also shows how DiscoGAN successfully transfers styles across multiple domains to generate output images of handbags from the given input of shoe images or to perform gender transformations of celebrity images.

Chapter 5, *Using Various Generative Models to Generate Images*, introduces the concept of a pretrained model and discusses techniques for running deep learning and generative models over large distributed systems using Apache Spark. We will then enhance the resolution of low quality images using pretrained models with GAN. And finally, we will learn other varieties of generative models such as DeepDream and VAE for image generation and styling.

Chapter 6, *Taking Machine Learning to Production*, describes various approaches to deploying machine learning and deep learning-based intelligent applications to production both on data center and the cloud using microservice-based containerized or serverless techniques.

What you need for this book

All of the tools, libraries, and datasets used in this book are open source and available free of charge. Some cloud environments used in the book offer free trials for evaluation. With this book, and some adequate exposure to machine learning (or deep learning), the reader will be able to dive into the creative nature of deep learning through generative adversarial networks.

You will need to install Python and some additional Python packages using `pip` to effectively run the code samples presented in this book.

Who this book is for

The target audience of this book is machine learning experts and data scientists who want to explore the power of generative models to create realistic images from unlabeled raw data or noise.
Readers who possess adequate knowledge of machine learning and neural network, and have exposure to the Python programming language can use this book to learn how to synthesize images from text or discover relationships across similar domains automatically and explore unsupervised domains with deep learning based generative architecture.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.
A block of code for importing necessary packages and libraries modules is set as follows:

```
<div class="packt_code">
nsamples=6
Z_sample = sample_Z(nsamples, noise_dim)
y_sample = np.zeros(shape=[nsamples, num_labels])
y_sample[:, 7] = 1 # generating image based on label
samples = sess.run(G_sample, feed_dict={Z: Z_sample,
Y:y_sample})
</div>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

> **Note**
> Warnings or important notes appear in a box like this.

> **Tip**
> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e–mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e–mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Learning–Generative–Adversarial–Networks. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/LearningGenerati

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Introduction to Deep Learning

Deep Neural networks are currently capable of providing human level solutions to a variety of problems such as image recognition, speech recognition, machine translation, natural language processing, and many more.

In this chapter, we will look at how neural networks, a biologically-inspired architecture has evolved throughout the years. Then we will cover some of the important concepts and terminology related to deep learning as a refresher for the subsequent chapters. Finally we will understand the intuition behind the creative nature of deep networks through a generative model.

We will cover the following topics in this chapter:

- Evolution of deep learning
- Stochastic Gradient Descent, ReLU, learning rate, and so on
- Convolutional network, Recurrent Neural Network and LSTM
- Difference between discriminative and generative models

Evolution of deep learning

A lot of the important work on neural networks happened in the 80's and 90's, but back then computers were slow and datasets very tiny. The research didn't really find many applications in the real world. As a result, in the first decade of the 21st century neural networks have completely disappeared from the world of machine learning. It's only in the last few years, first in speech recognition around 2009, and then in computer vision around 2012, that neural networks made a big comeback (with LeNet, AlexNet, and so on). What changed?

Lots of data (big data) and cheap, fast GPU's. Today, neural networks are everywhere. So, if you're doing anything with data, analytics, or prediction, deep learning is definitely something that you want to get familiar with.

See the following figure:

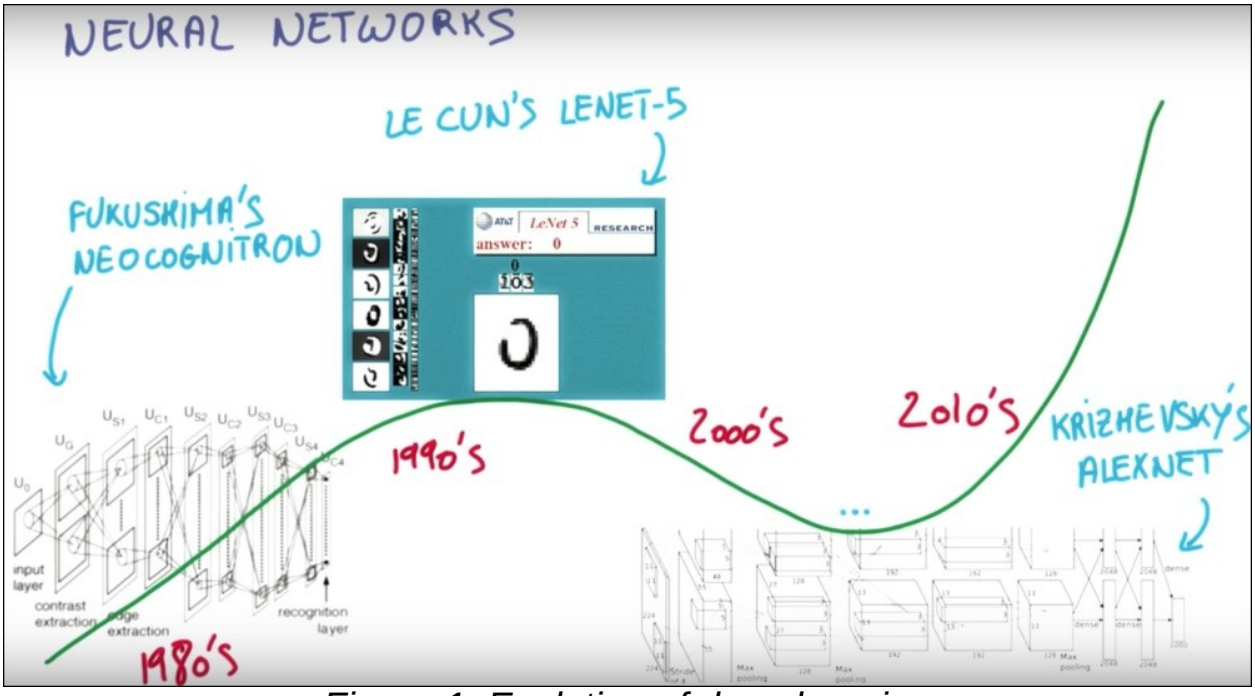


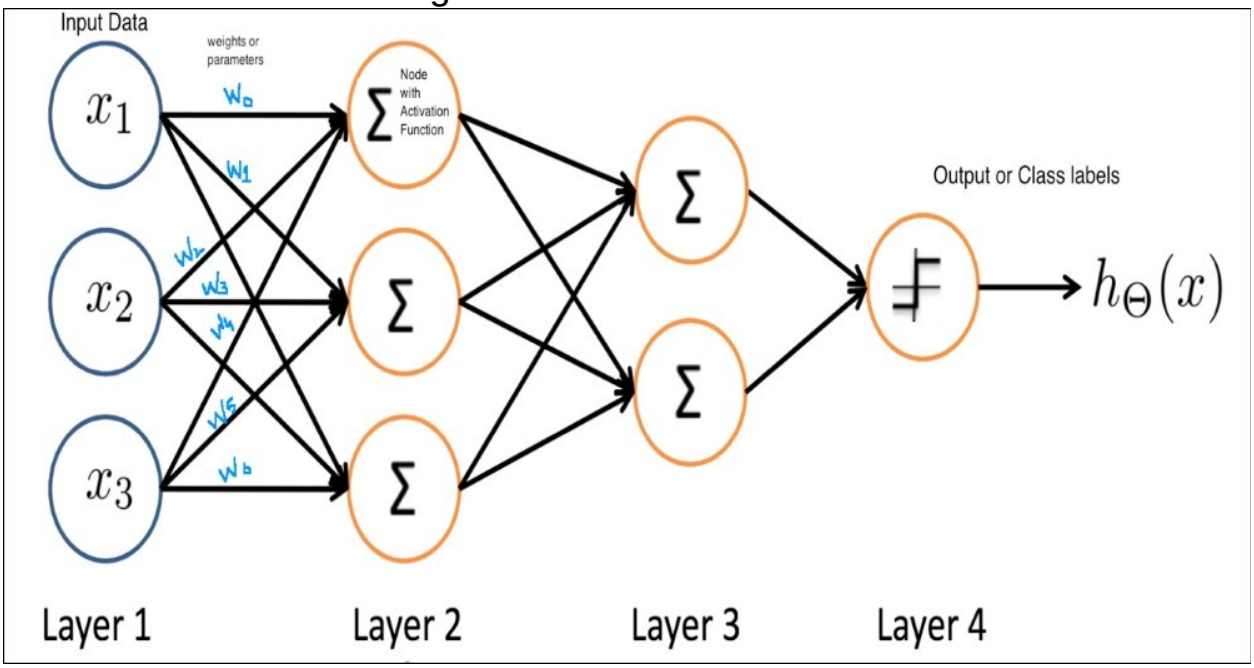
Figure-1: Evolution of deep learning

Deep learning is an exciting branch of machine learning that uses data, lots of data, to teach computers how to do things only humans were capable of before, such as recognizing what's in an image, what people are saying when they are talking on their phones, translating a document into another language, and helping robots explore the world and interact with it. Deep learning has emerged as a central tool to solve perception problems and it's state of the art with computer vision and speech recognition.

Today many companies have made deep learning a central part of their machine learning toolkit—Facebook, Baidu, Amazon, Microsoft, and Google are all using deep learning in their products because deep learning shines wherever there is lots of data and complex problems to solve.

Deep learning is the name we often use for "deep neural networks" composed of several layers. Each layer is made of nodes. The computation happens in the nodes, where it combines input data with a set of parameters or weights, that either amplify or dampen that input. These input-weight products are then summed and the sum is passed through the `activation` function, to determine to what extent the value should progress through the network to affect the final prediction, such as an act of classification. A layer consists of a row of nodes that that turn on or off as the input is fed through the network. The input of the first layer

becomes the input of the second layer and so on. Here's a diagram of what neural a network might look like:



Let's get familiarized with some deep neural network concepts and terminology.

Sigmoid activation

The sigmoid activation function used in neural networks has an output boundary of *(0, 1)*, and *α* is the offset parameter to set the value at which the sigmoid evaluates to 0.

The sigmoid function often works fine for gradient descent as long as the input data *x* is kept within a limit. For large values of *x*, *y* is constant. Hence, the derivatives *dy/dx* (the gradient) equates to *0*, which is often termed as the **vanishing gradient** problem.

This is a problem because when the gradient is 0, multiplying it with the loss (actual value - predicted value) also gives us 0 and ultimately networks stop learning.

Rectified Linear Unit (ReLU)

A neural network can be built by combining some linear classifiers with some non-linear functions. The **Rectified Linear Unit** (**ReLU**) has become very popular in the last few years. It computes the function *f(x)=max(0,x)*. In other words, the activation is simply thresholded at zero.

Unfortunately, ReLU units can be fragile during training and can die, as a ReLU neuron could cause the weights to update in such a way that the

neuron will never activate on any datapoint again, and so the gradient flowing through the unit will forever be zero from that point on.
To overcome this problem, a leaky `ReLU` function will have a small negative slope (of 0.01, or so) instead of zero when *x<0*:

$$f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$$

where *αα* is a small constant.

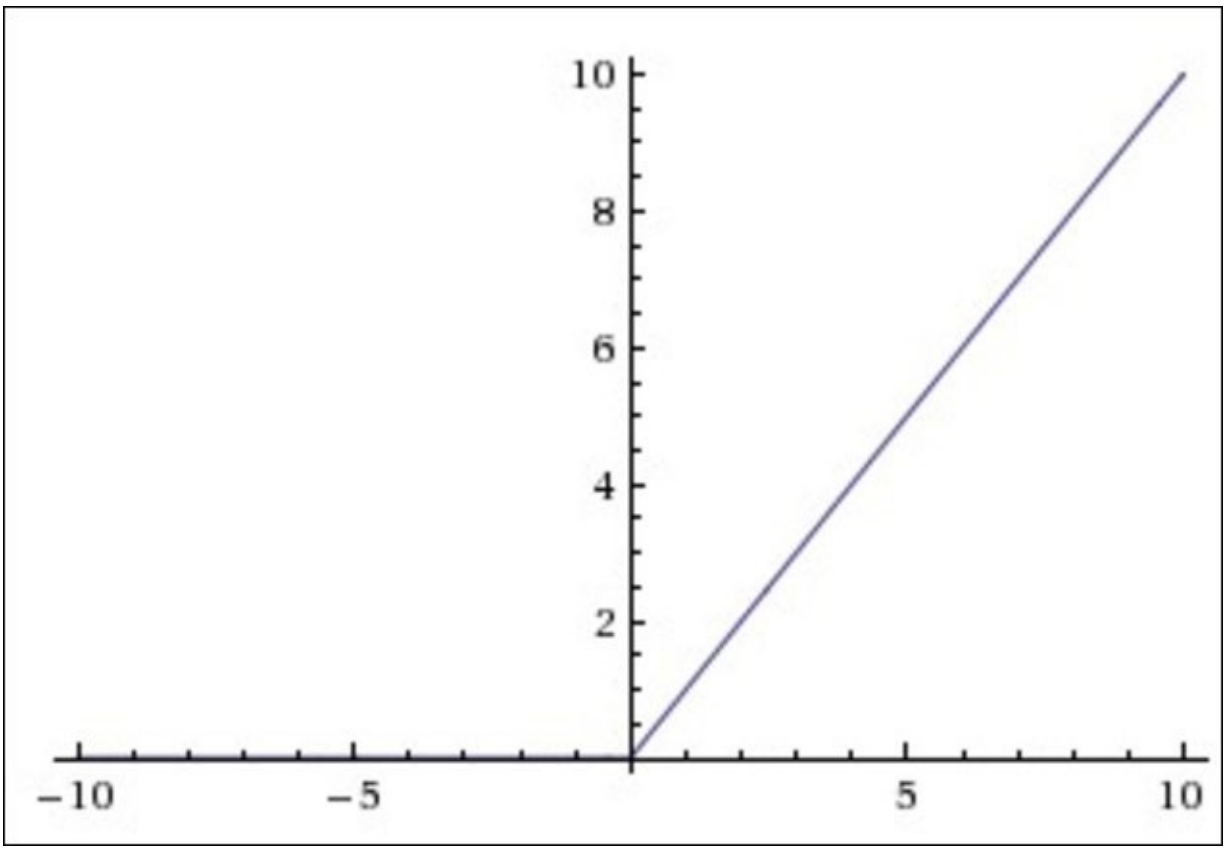


Figure-2: Rectified Linear Unit

Exponential Linear Unit (ELU)

The mean of ReLU activation is not zero and hence sometimes makes learning difficult for the network. The **Exponential Linear Unit** (**ELU)** is similar to ReLU activation function when the input *x* is positive, but for negative values it is a function bounded by a fixed value *-1*, for *α=1* (the hyperparameter *α* controls the value to which an ELU saturates for negative inputs). This behavior helps to push the mean activation of neurons closer to zero; that helps to learn representations that are more robust to noise.

Stochastic Gradient Descent (SGD)

Scaling batch gradient descent is cumbersome because it has to compute a lot if the dataset is big, and as a rule of thumb, if computing your loss takes *n* floating point operations, computing its gradient takes about three times that to compute.
But in practice we want to be able to train lots of data because on real problems we will always get more gains the more data we use. And because gradient descent is iterative and has to do that for many steps, that means that in order to update the parameters in a single step, it has to go through all the data samples and then do this iteration over the data tens or hundreds of times.
Instead of computing the loss over entire data samples for every step, we can compute the average loss for a very small random fraction of the training data. Think between 1 and 1000 training samples each time. This technique is called **Stochastic Gradient Descent** (**SGD**) and is at the core of deep learning. That's because SGD scales well with both data and model size.
SGD gets its reputation for being black magic as it has lots of hyper-parameters to play and tune such as initialization parameters, learning rate parameters, decay, and momentum, and you have to get them right.
AdaGrad is a simple modification of SGD, which implicitly does momentum and learning rate decay by itself. Using AdaGrad often makes learning less sensitive to hyper-parameters. But it often tends to be a little worse than precisely tuned SDG with momentum. It's still a very good option though, if you're just trying to get things to work:

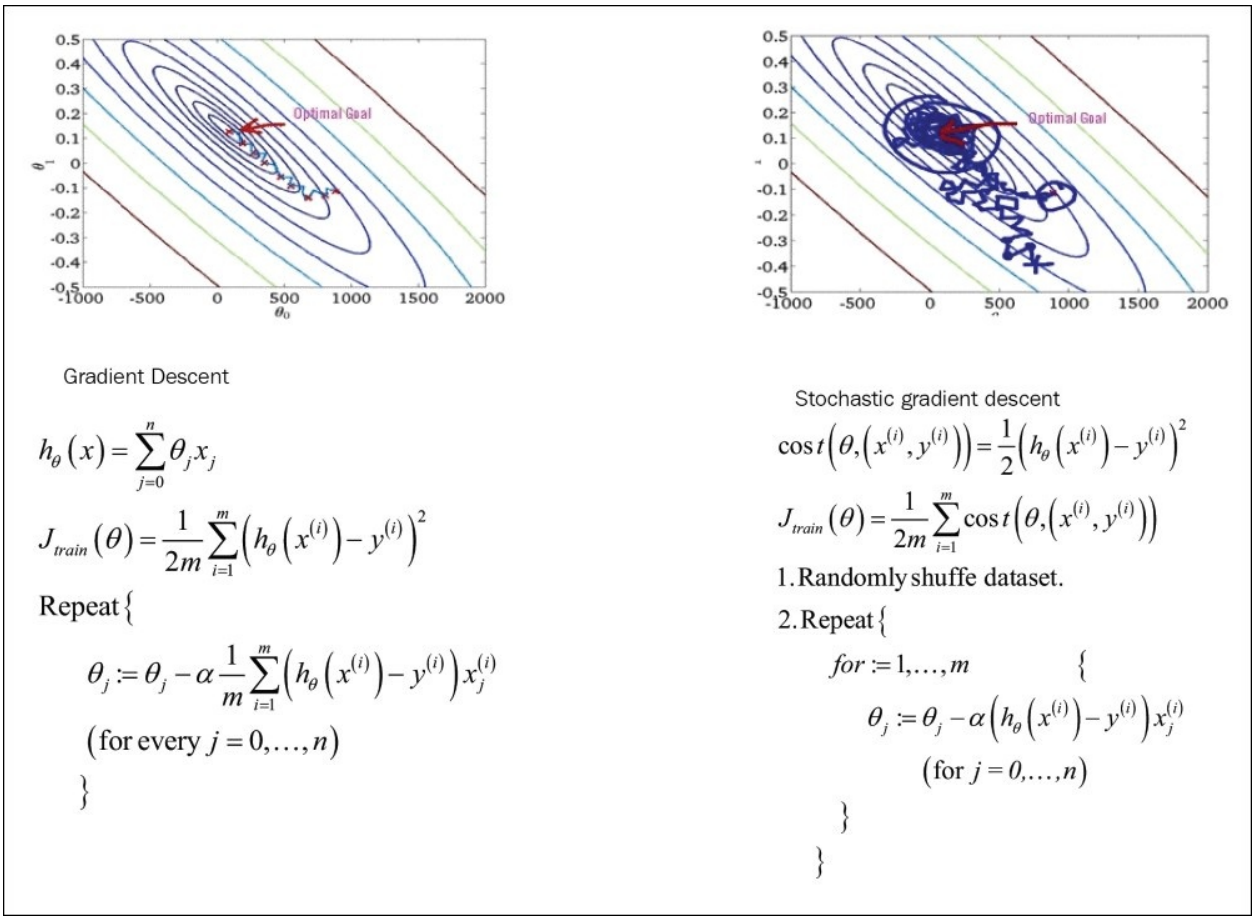


Figure-4a: Loss computation in batch gradient descent and SGD
Source: https://www.coursera.org/learn/machine-learning/lecture/DoRHJ/stochasticgradient- descent

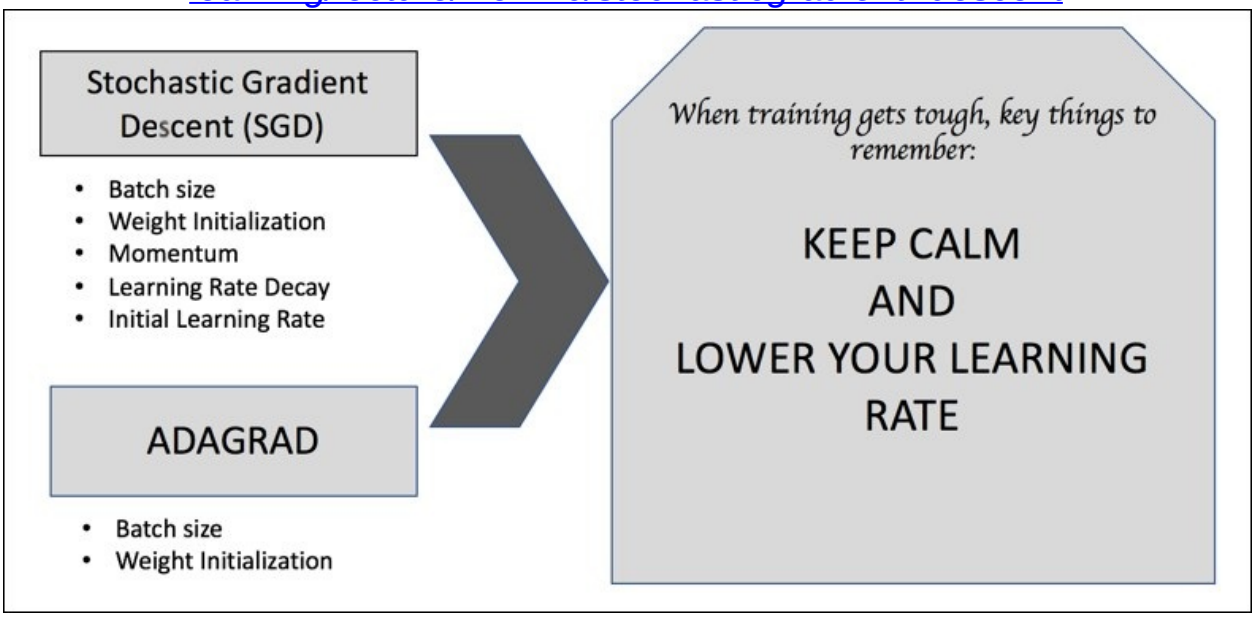


Figure 4b: Stochastic Gradient Descent and AdaGrad

You can notice from *Figure 4a* that in case of batch gradient descent the `loss`/`optimization` function is well minimized, whereas SGD calculates the loss by taking a random fraction of the data in each step and often oscillates around that point. In practice, it's not that bad and SGD often converges faster.

Learning rate tuning

The `loss` function of the neural network can be related to a surface, where the weights of the network represent each direction you can move in. Gradient descent provides the steps in the current direction of the slope, and the learning rate gives the length of each step you take. The learning rate helps the network to abandons old beliefs for new ones. Learning rate tuning can be very strange. For example, you might think that using a higher learning rate means you learn more or that you learn faster. That's just not true. In fact, you can often take a model, lower the learning rate, and get to a better model faster.

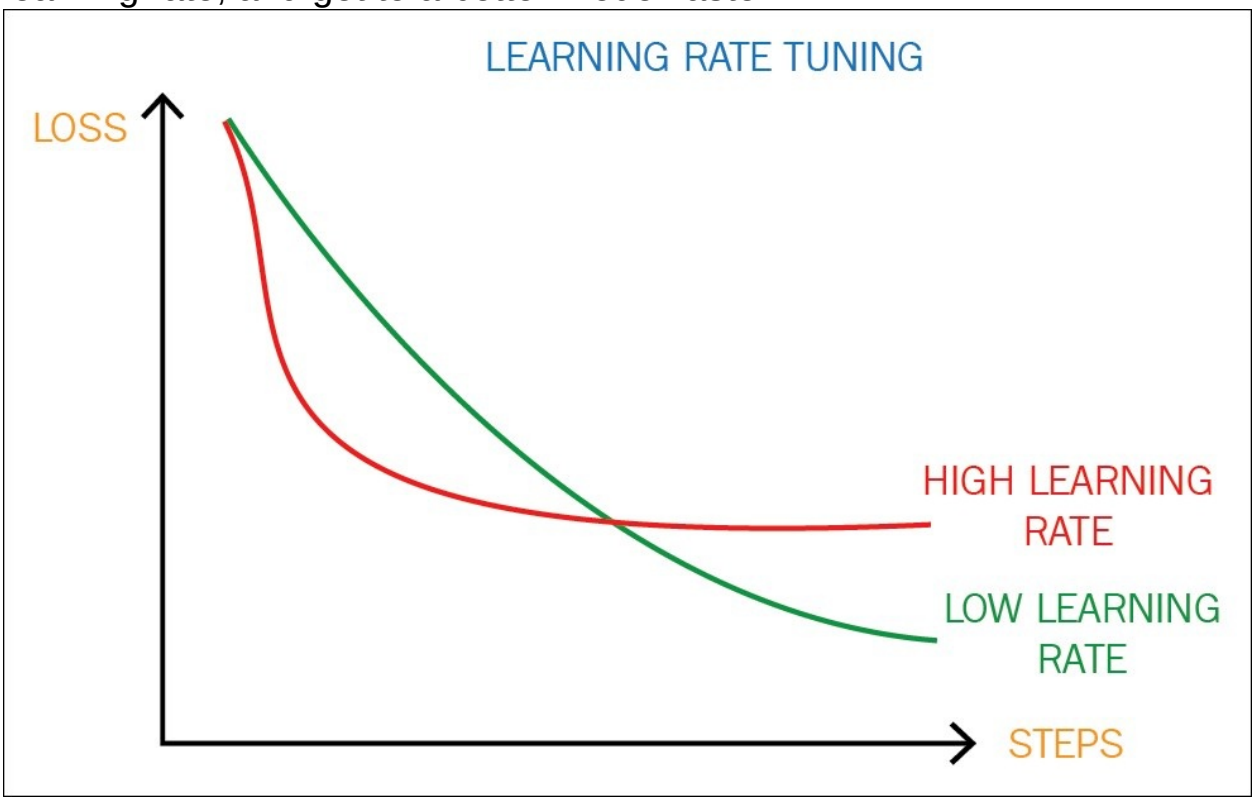


Figure-3: Learning rate

You might be tempted to look at the learning curve that shows the loss over time to see how quickly the network learns. Here the higher learning rate starts faster, but then it plateaus, whereas the lower learning rate keeps on going and gets better. It is a very familiar picture for anyone

who has trained neural networks. *Never trust how quickly you learn*.

Regularization

The first way to prevent over fitting is by looking at the performance under validation set, and stopping to train as soon as it stops improving. It's called early termination, and it's one way to prevent a neural network from over-optimizing on the training set. Another way is to apply regularization. Regularizing means applying artificial constraints on the network that implicitly reduce the number of free parameters while not making it more difficult to optimize.

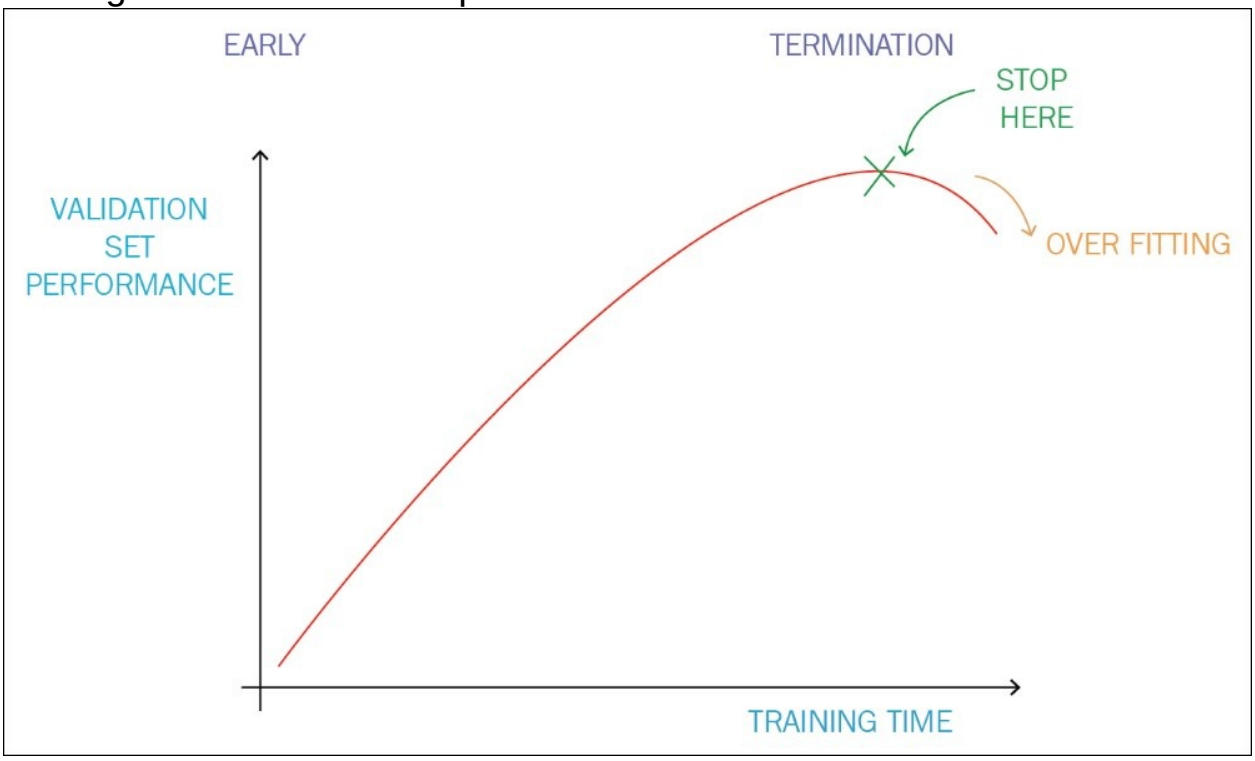


Figure 6a: Early termination

In the skinny jeans analogy as shown in *Figure 6b*, think stretch pants. They fit just as well, but because they're flexible, they don't make things harder to fit in. The stretch pants of deep learning are sometime called **L2 regularization**. The idea is to add another term to the loss, which penalizes large weights.

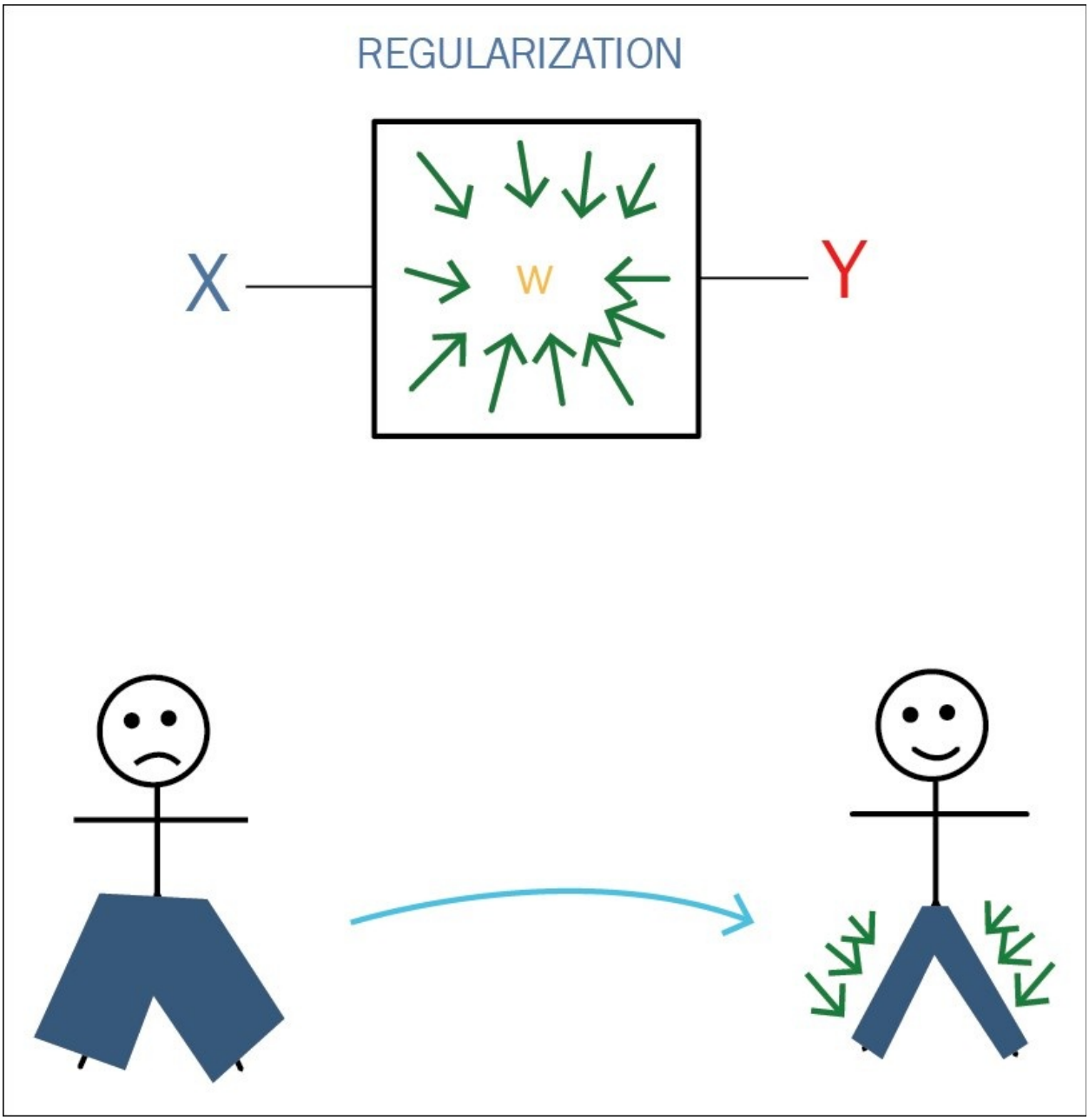


Figure 6b: Stretch pant analogy of deep learning

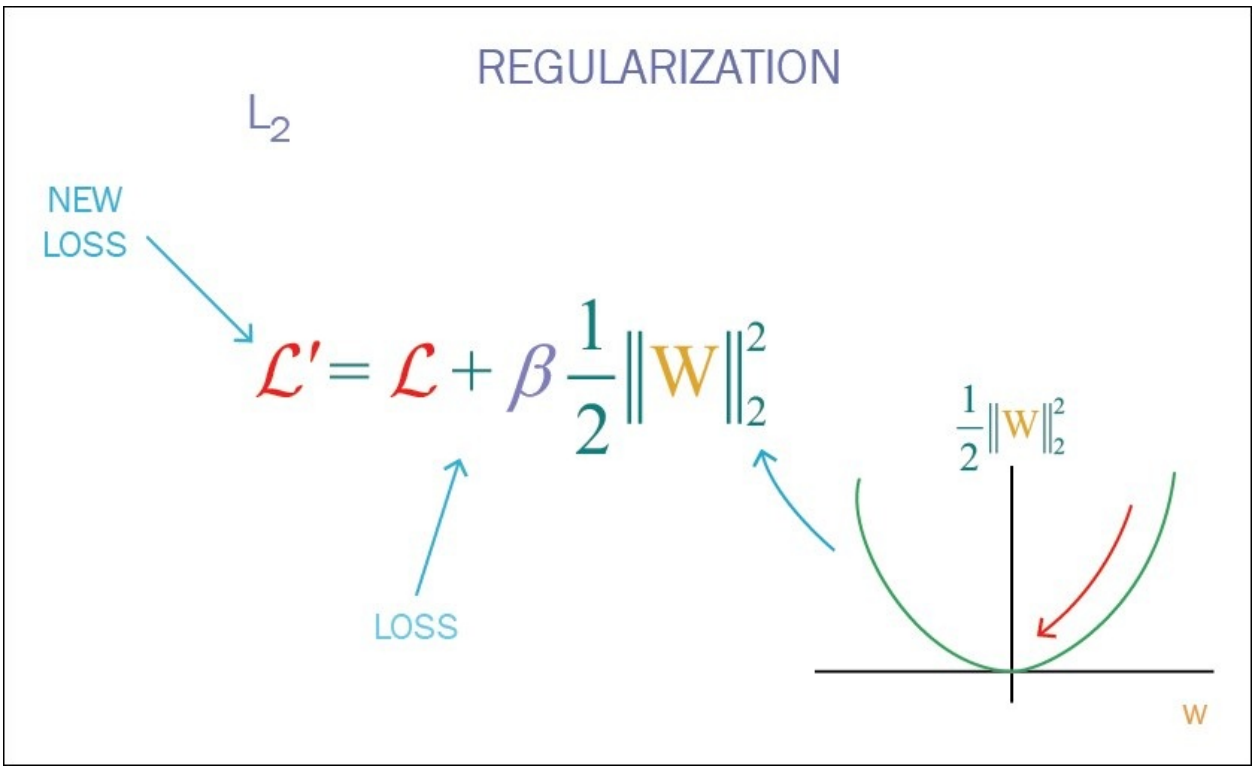


Figure 6c: L2 tegularization

Currently, in deep learning practice, the widely used approach for preventing overfitting is to feed lots of data into the deep network.

Shared weights and pooling

Let say an image has a cat in it and it doesn't really matter where the cat is in the image, as it's still an image with a cat. If the network has to learn about cats in the left corner and about cats in the right corner independently, that's a lot of work that it has to do. But objects and images are largely the same whether they're on the left or on the right of the picture. That's what's called **translation invariance**.

The way of achieving this in networks is called **weight sharing**. When networks know that two inputs can contain the same kind of information, then it can share the weights and train the weights jointly for those inputs. It is a very important idea. Statistical invariants are things that don't change on average across time or space, and are everywhere. For images, the idea of weight sharing will get us to study convolutional networks. For text and sequences in general, it will lead us to recurrent neural networks:

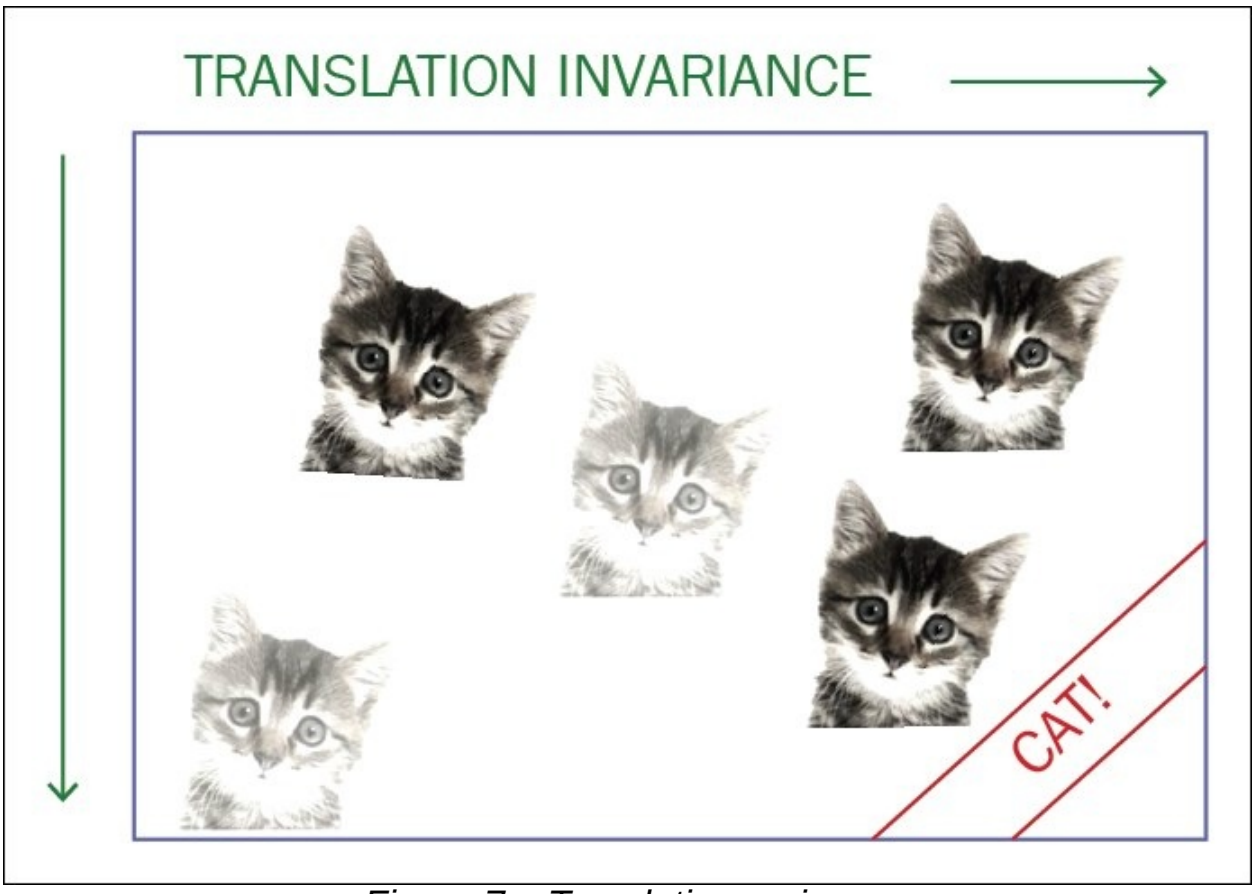


Figure 7a: Translation variance

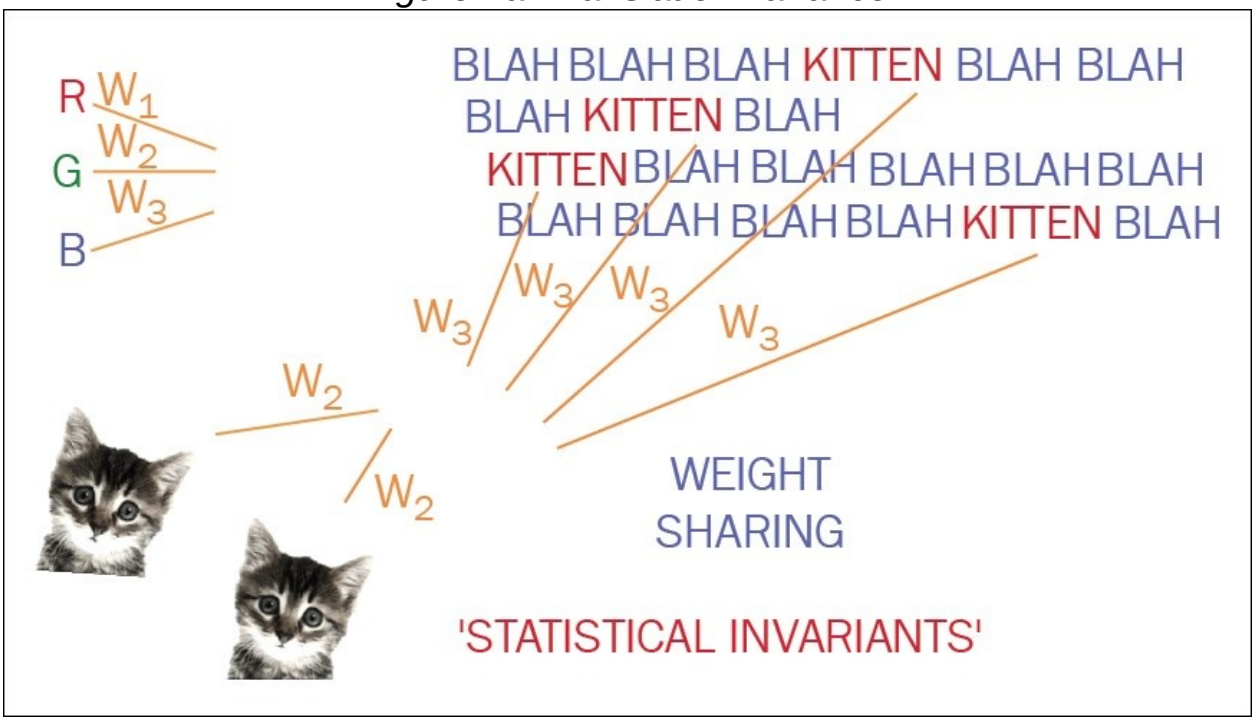


Figure 7b: Weight sharing

To reduce the spatial extent of the feature maps in the convolutional

pyramid, a very small stride could run and take all the convolutions in a neighborhood and combine them somehow. This is known as **pooling**. In max-pooling as shown in *Figure 7d*, at every point in the feature map, look at a small neighborhood around that point and compute the maximum of all the responses around it. There are some advantages to using max pooling. First, it doesn't add to your number of parameters. So, you don't risk an increasing over fitting. Second, it simply often yields more accurate models. However, since the convolutions that run below run at a lower stride, the model then becomes a lot more expensive to compute. Max-pooling extracts the most important feature, whereas average pooling sometimes can't extract good features because it takes all into account and results in an average value that may/may not be important for object detection-type tasks.

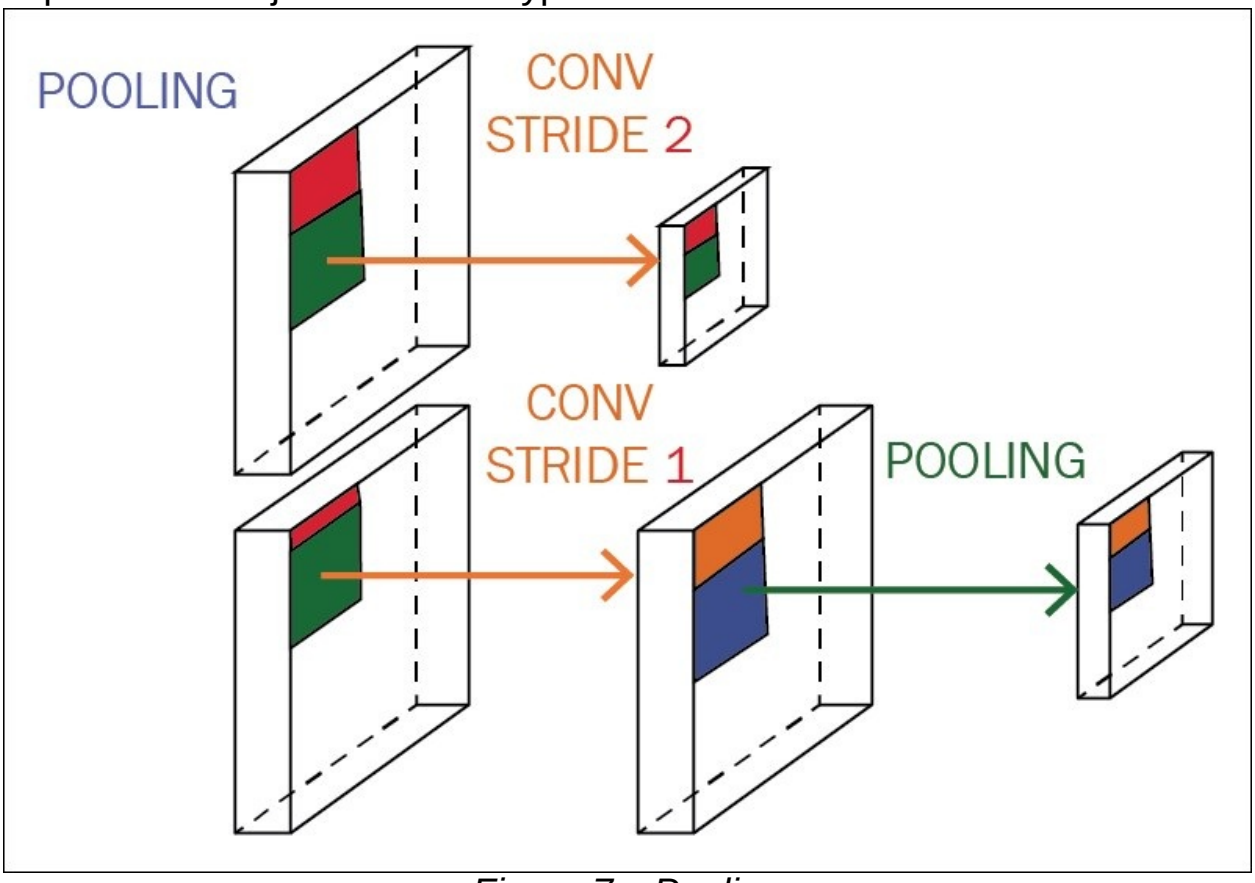


Figure 7c: Pooling

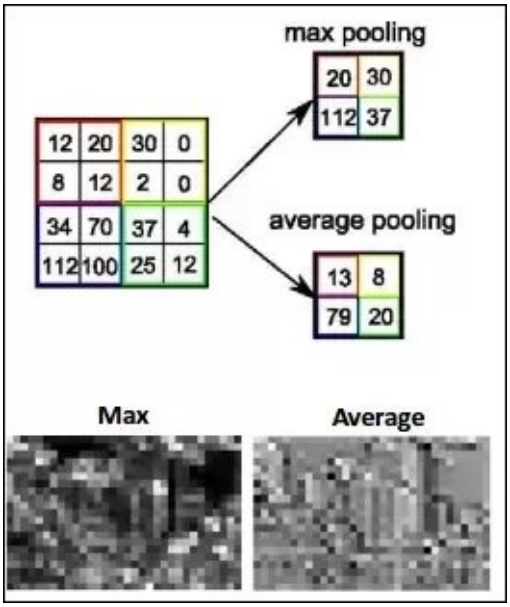


Figure 7d: Max and average pooling

Local receptive field

A simple way to encode the local structure is to connect a submatrix of adjacent input neurons into one single hidden neuron belonging to the next layer. That single hidden neuron represents one local receptive field. Let's consider CIFAR-10 images that have an input feature of size [32 x 32 x 3]. If the receptive field (or the filter size) is 4 x 4, then each neuron in the convolution layer will have weights to a [4 x 4 x 3] region in the input feature, for a total of 4*4*3 = 48 weights (and +1 bias parameter). The extent of the connectivity along the depth axis must be 3, since this is the depth (or number of channel: RGB) of the input feature.

Convolutional network (ConvNet)

Convolutional Networks (**ConvNets**) are neural networks that share their parameters/weights across space. An image can be represented as a flat pancake that has width, height, and depth or number of channel (for RGB: having red, green, and blue channel the depth is 3, whereas for grayscale the depth is 1).
Now let's slide a tiny neural network with *K* outputs across the image without changing the weights.

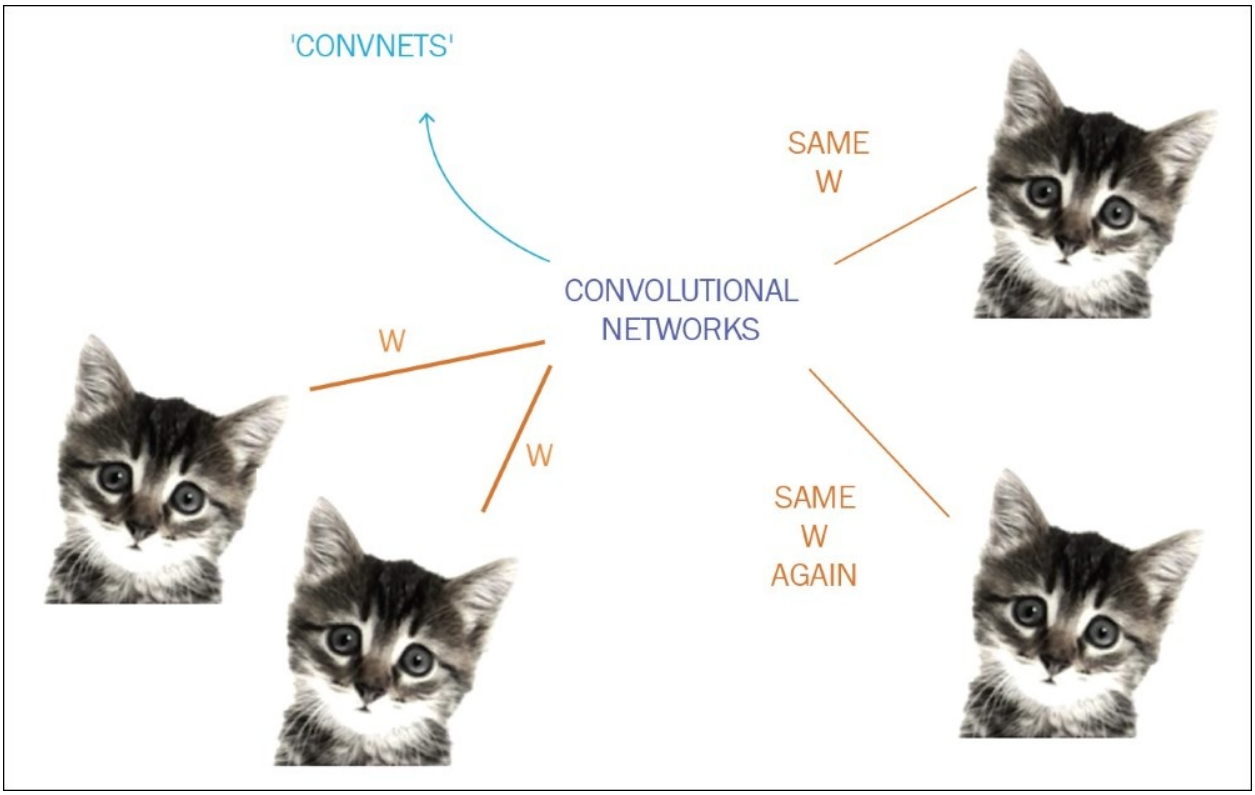


Figure 8a: Weight sharing across space

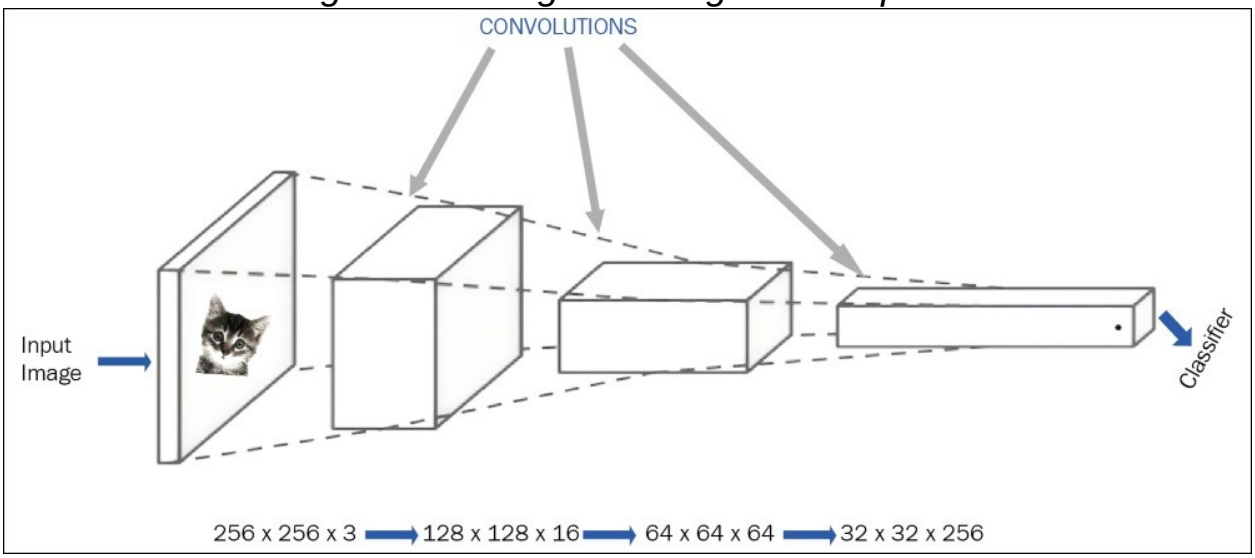
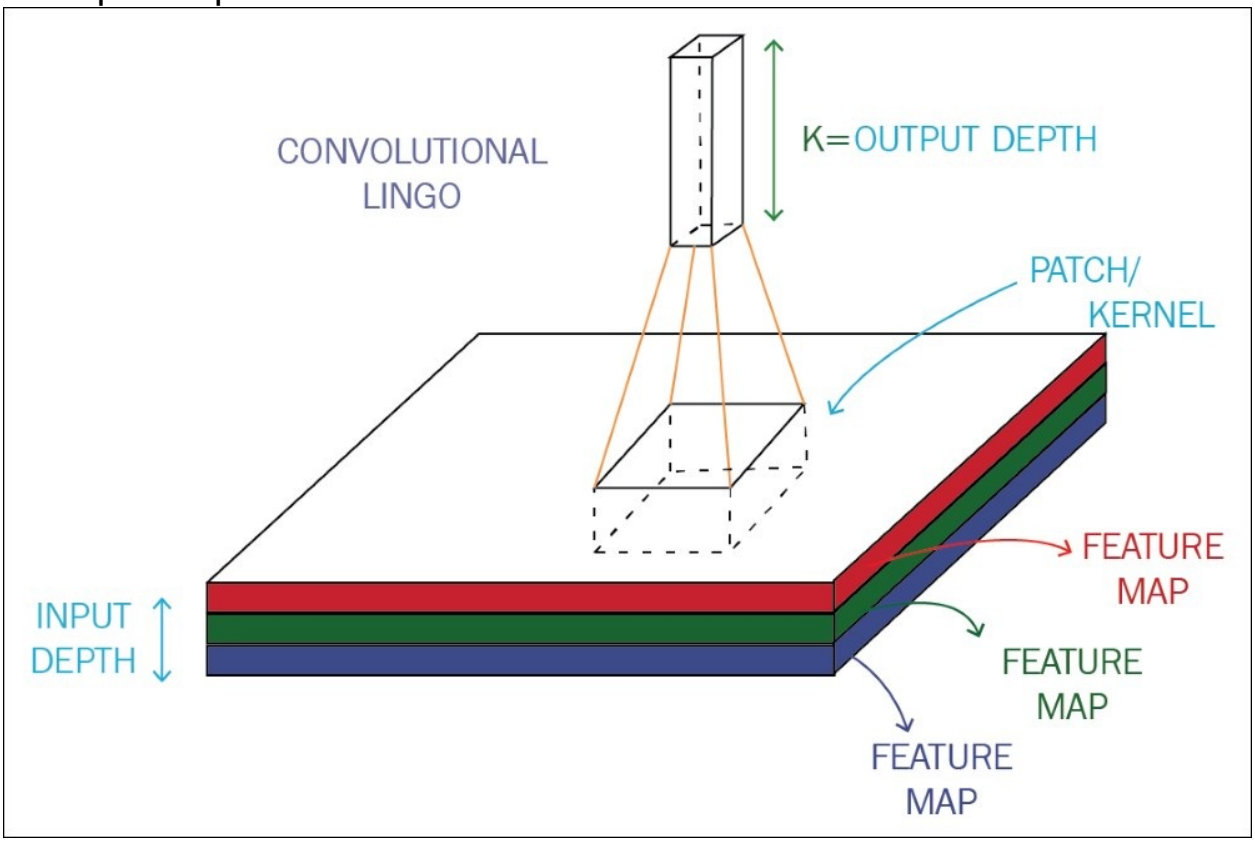


Figure 8b: Convolutional pyramid with layers of convolution

On the output, a different image will be drawn with different width, different height, and different depth (from just R, G, B color channels to *K* number of channels). This operation is known as convolution.

A ConvNet is going to basically be a deep network with layers of convolutions that stack together to form a pyramid like structure. You can see from the preceding figure that the network takes an image as an input of dimension (width x height x depth) and then applys convolutions

progressively over it to reduce the spatial dimension while increasing the depth, which is roughly equivalent to its semantic complexity. Let's understand some of the common terminology in convent.
Each layer or depth in the image stack is called a feature map and patches or kernels are used for mapping three feature maps to *K* feature maps. A stride is the number of pixels that is shifted each time you move your filter. Depending on the type of padding a stride of 1 makes the output roughly the same size as the input. A stride of 2 makes it about half the size. In the case of valid padding, a sliding filter don't cross the edge of the image, whereas in same-padding it goes off the edge and is padded with zeros to make the output map size exactly the same size as the input map:



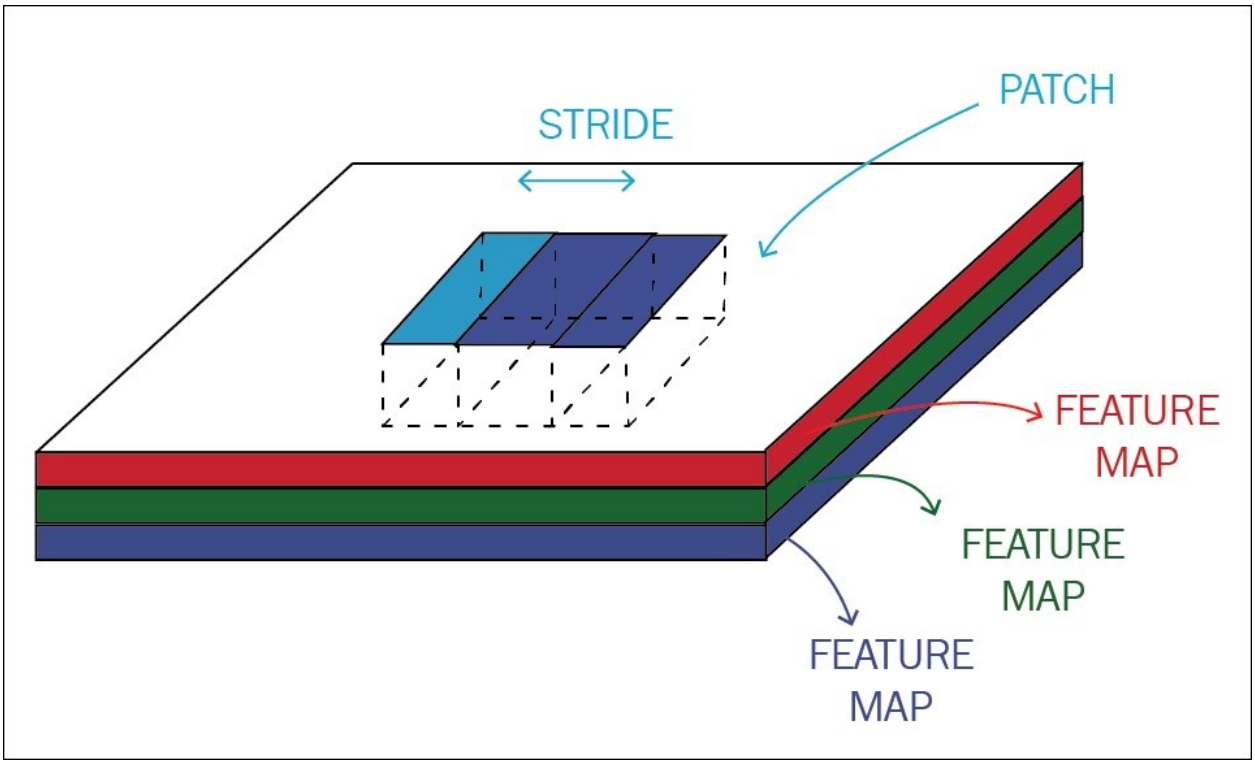


Figure 8c: Different terminology related to convolutional network

Deconvolution or transpose convolution

In the case of a computer vision application where the resolution of final output is required to be larger than the input, deconvolution/transposed convolution is the de-facto standard. This layer is used in very popular applications such as GAN, image super-resolution, surface depth estimation from image, optical flow estimation, and so on.
CNN in general performs down-sampling, that is, they produce output of a lower resolution than the input, whereas in deconvolution the layer up-samples the image to get the same resolution as the input image. Note since a naive up-sampling inadvertently loses details, a better option is to have a trainable up-sampling convolutional layer whose parameters will change during training.
Tensorflow method: `tf.nn.conv2d_transpose(value, filter, output_shape, strides, padding, name)`

Recurrent Neural Networks and LSTM

The key idea behind **Recurrent Neural Networks** (**RNN**) is to share parameters over time. Imagine that you have a sequence of events, and at each point in time you want to make a decision about what's happened so far in this sequence. If the sequence is reasonably stationary, you can use the same classifier at each point in time. That simplifies things a lot already. But since this is a sequence, you also want to take into account the past-everything that happened before that point.
RNN is going to have a single model responsible for summarizing the past and providing that information to your classifier. It basically ends up with a network that has a relatively simple repeating pattern, with part of the classifier connecting to the input at each time step and another part called the recurrent connection connecting you to the past at each step, as shown in the following figure:

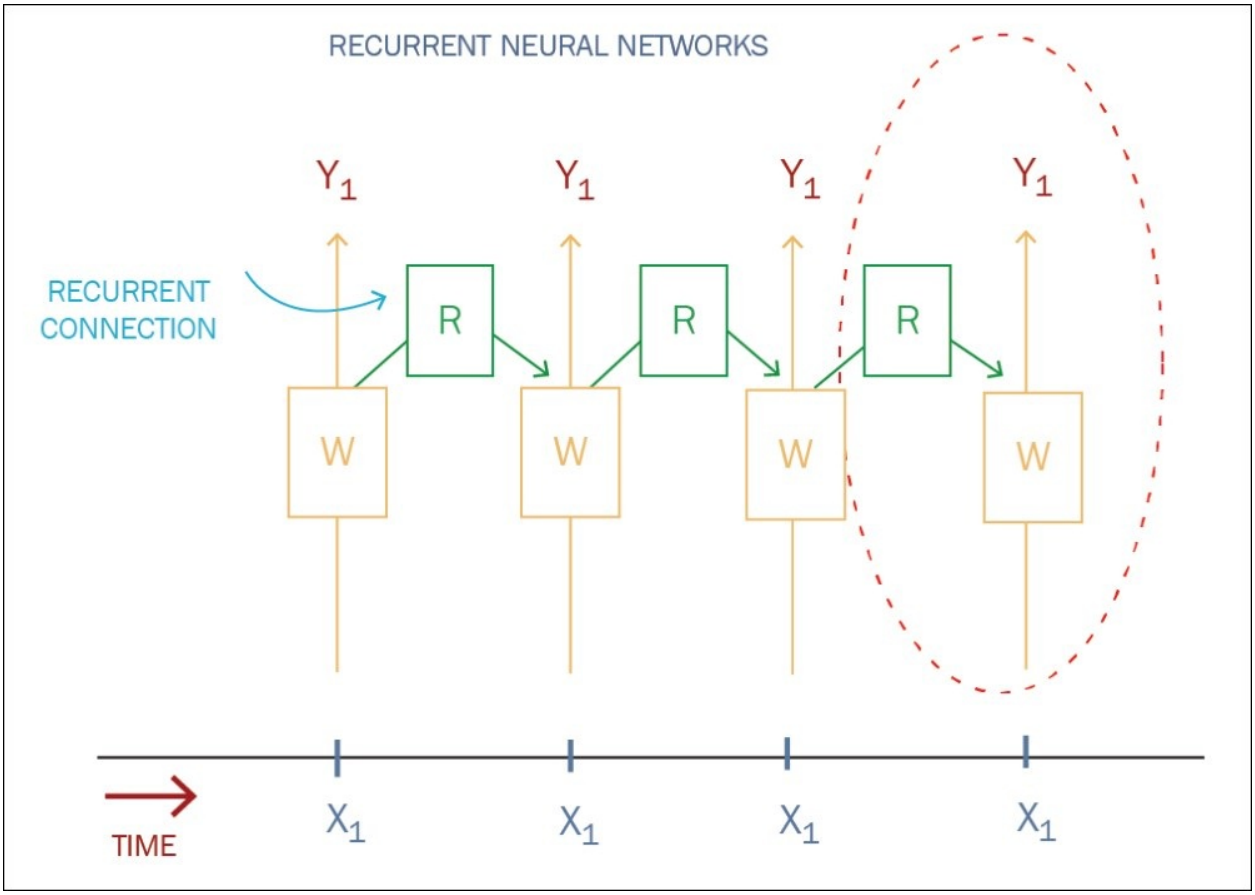


Figure 9a: Recurrent neural network

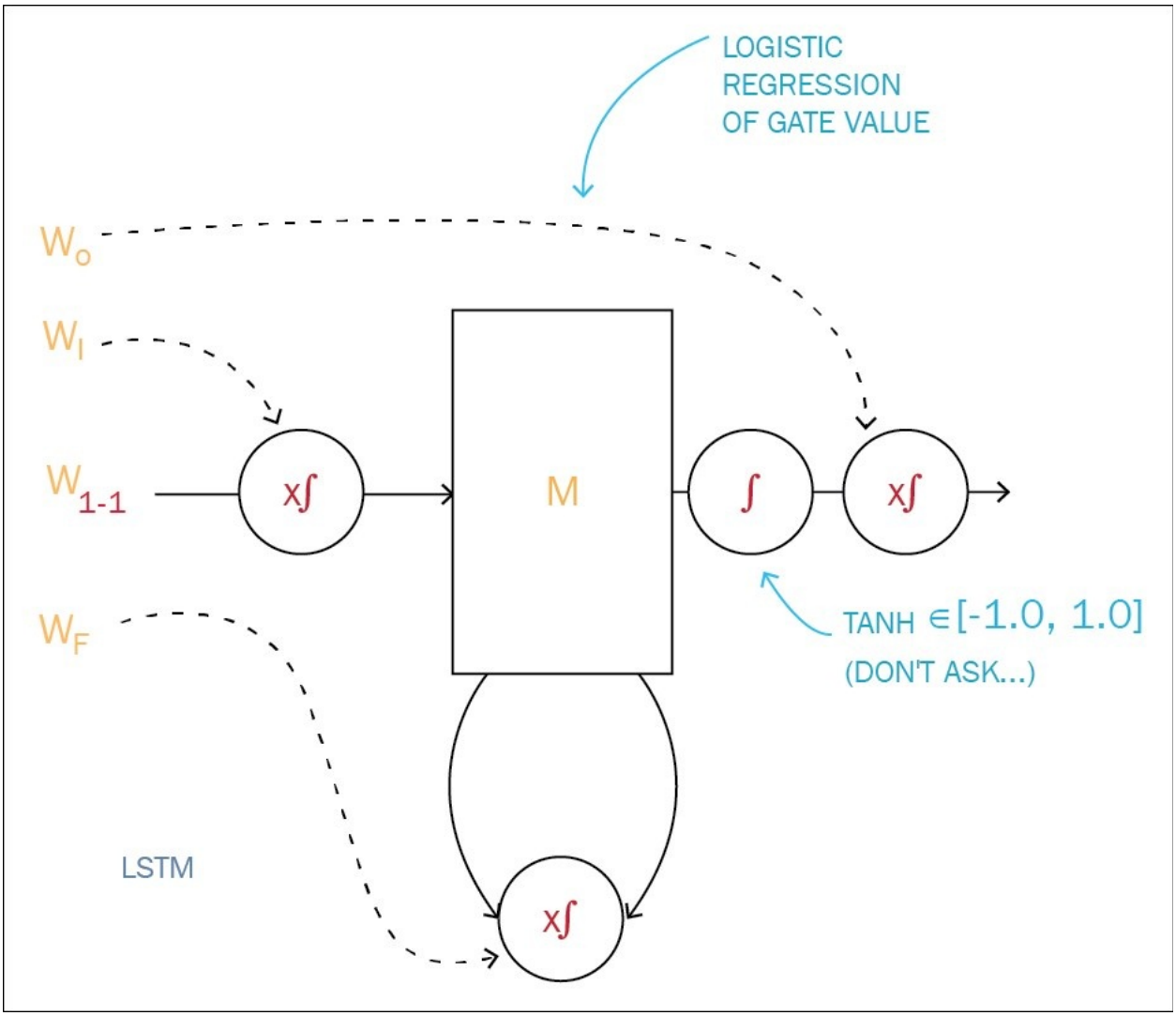


Figure-9b: Long short-term memory (LSTM)

LSTM stands for **long short-term memory**. Now, conceptually, a recurrent neural network consists of a repetition of simple little units like this, which take as an input the past, a new input, and produce a new prediction and connect to the future. Now, what's in the middle of that is typically a simple set of layers with some weights and linearities.
In LSTM as shown in *Figure 9b*, the gating values for each gate get controlled by a tiny logistic regression on the input parameters. Each of them has its own set of shared parameters. And there's an additional hyperbolic tension sprinkled to keep the outputs between -1 and 1. Also it's differentiable all the way, which means it can optimize the parameters very easily. All these little gates help the model keep its memory for longer when it needs to, and ignore things when it should.

Deep neural networks

The central idea of deep learning is to add more layers and make your model deeper. There are lots of good reasons to do that. One is parameter efficiency. You can typically get much more performance with fewer parameters by going deeper rather than wider.
Another one is that a lot of the natural phenomena that you might be interested in, tend to have a hierarchical structure, which deep models naturally capture. If you poke at a model for images, for example, and visualize what the model learns, you'll often find very simple things at the lowest layers, such as lines or edges.

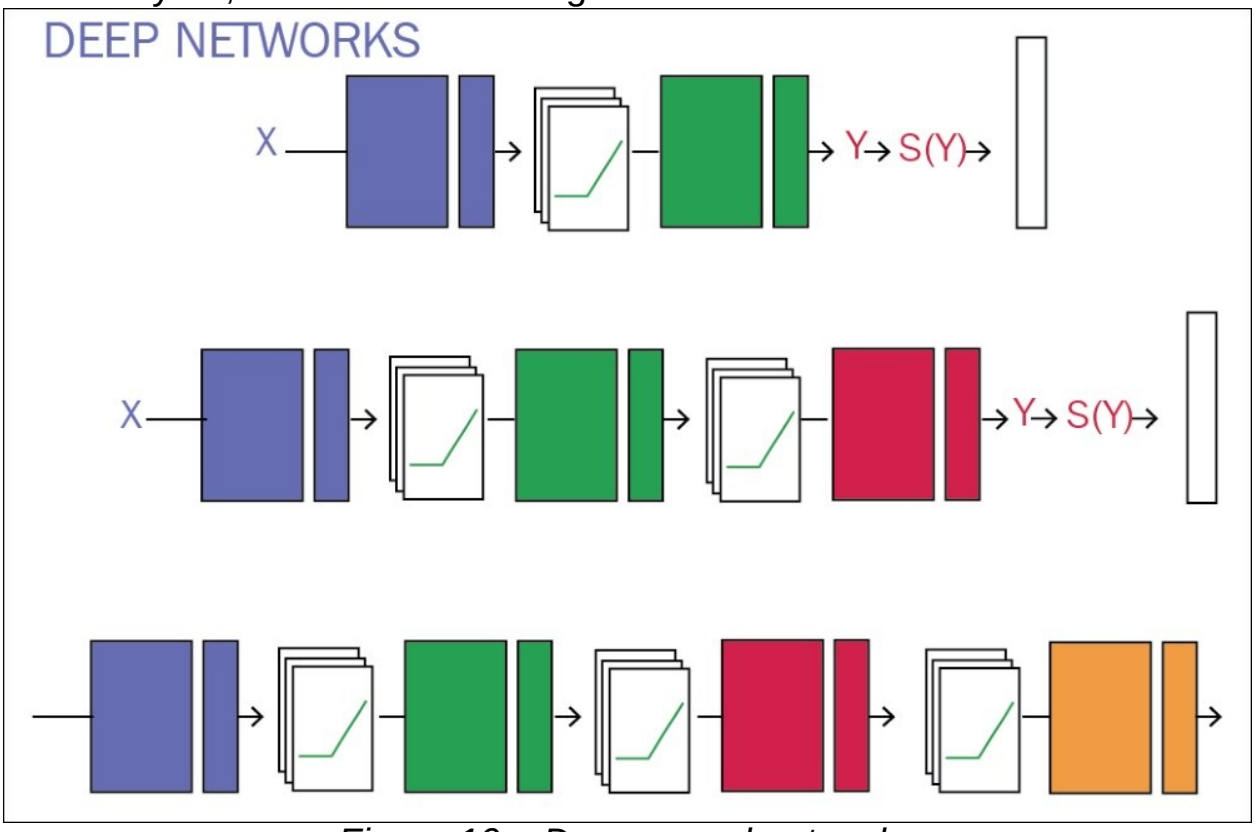


Figure 10a: Deep neural networks

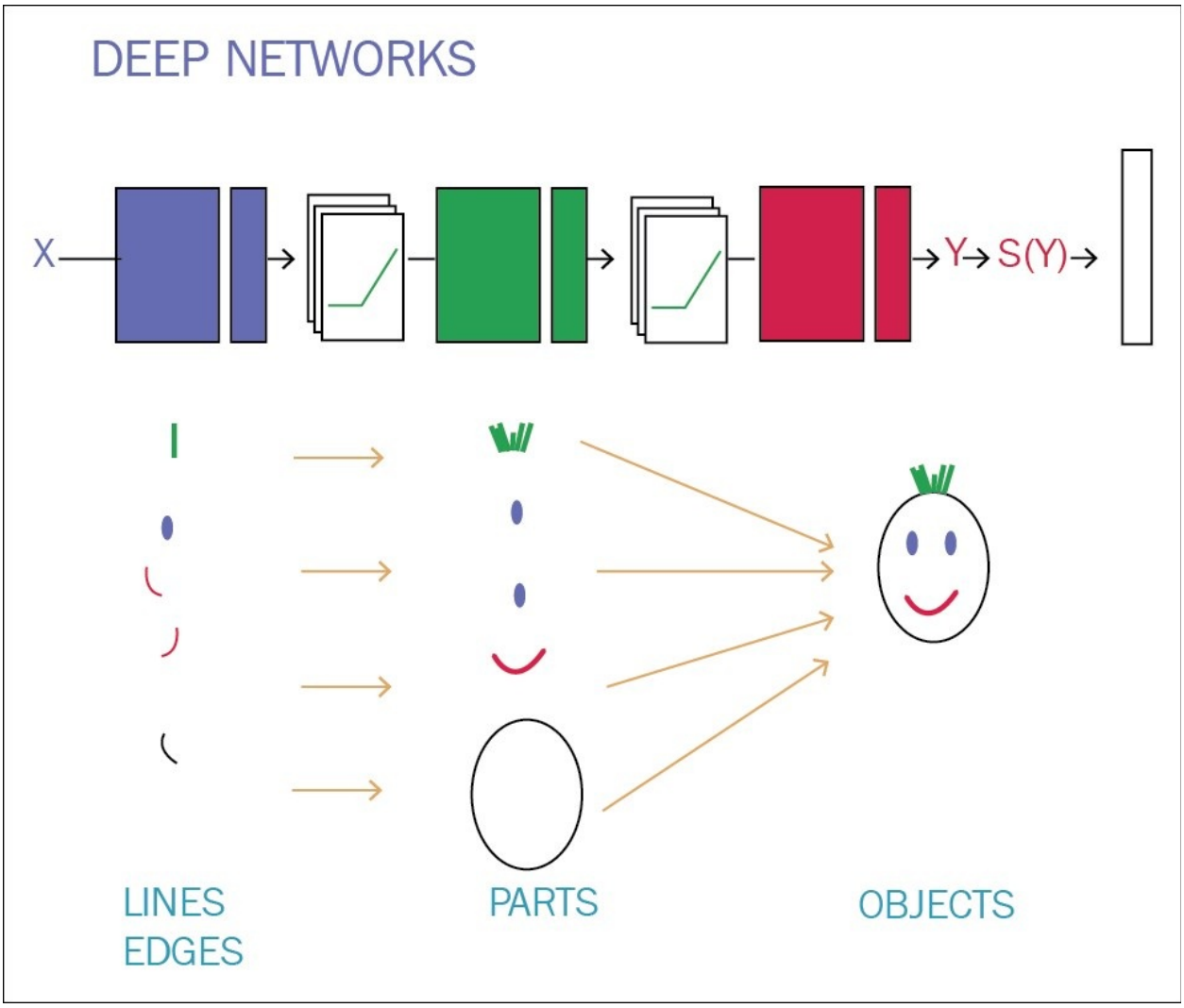


Figure 10b: Network layers capturing hierarchical structure of image

A very typical architecture for a ConvNet is a few layers alternating convolutions and max pooling, followed by a few fully connected layers at the top. The first famous model to use this architecture was LeNet-5 designed by Yann Lecun for character recognition back in 1998. Modern convolutional networks such as AlexNet, which famously won the competitive ImageNet object recognition challenge in 2012, use a very similar architecture with a few wrinkles. Another notable form of pooling is average pooling. Instead of taking the max, just take an average over the window of pixels around a specific location.

Discriminative versus generative models

A discriminative model learns the conditional probability distribution *p(y|x)* which could be interpreted as the *probability of y given x*. A discriminative classifier learns by observing data. It makes fewer assumptions on the

distributions, but depends heavily on the quality of the data. The distribution *p(y|x)* simply classifies a given example x directly into a label *y*. For example, in logistic regression all we have to do is to learn weights and bias that would minimize the squared loss.
Whereas a generative model learns the joint probability distribution *p(x,y)*, where *x* is the input data and *y* is the label that you want to classify. A generative model can generate more samples by itself artificially, based on assumptions about the distribution of data. For example, in the Naive Bayes' model, we can learn *p(x)* from data, also *p(y)*, the prior class probabilities, and we can also learn *p(x|y)* from the data using say maximum likelihood.
Once we have *p(x)*, *p(y)* and *p(x|y)*, *p(x, y)* is not difficult to find out. Now using Bayes' rule, we can replace the *p(y|x)* with *(p(x|y)p(y))/p(x)*. And since we are just interested in the *arg max*, the denominator can be removed, as that will be the same for every *y*:

$$\arg\max \mathrm{p}(\mathrm{x} \mid \mathrm{y})\mathrm{p}(\mathrm{y})$$

This is the equation we use in generative models, as *p(x, y) = p(x | y) p(y)*, which explicitly models the actual distribution of each class.
In practice, the discriminative models generally outperform generative models in classification tasks, but the generative model shines over discriminative models in creativity/generation tasks.

Summary

So far you have refreshed various concepts related to deep learning and also learned how deep networks have evolved from the arena of supervised tasks of classifying an image, recognizing voice, text, and so on, towards the creative power through generative model. In the next chapter we will see how deep learning can be used for performing wonderful creativity tasks in the unsupervised domain using **Generative Adversarial Networks** (**GANs**).

Chapter 2. Unsupervised Learning with GAN

Recently, with the progress of generative models, neural networks can not only recognize images but they can be used to generate audio and realistic images as well.

In this chapter, we will deep dive into the creative nature of deep learning through the latest state of the art algorithm of **Generative Adversarial Network**, commonly known as **GAN**. You will learn through hands-on examples to use the generative ability of the neural networks in generating realistic images from various real-world datasets (such as `MNIST` and `CIFAR`). Also, you will understand how to overcome the major challenge of unsupervised learning with deep networks using semi-supervised approach and apply it to your own problem domain. In the final section of this chapter, you will learn some of the training obstacles followed by practical tips and tricks of working with GAN models.

We will cover the following topics in this chapter:

- What is GAN? its application, tips, and tricks
- Explaining the concept of GAN through two-layer neural network image generation with TensorFlow
- Image generation with **Deep Convolutional GAN** (**DCGAN**) using Keras
- Implementation of semi-supervised learning using TensorFlow

Automating human tasks with deep neural networks

In the last few years, there has been an explosion of deep neural networks that can perform image classification, voice recognition, and understanding natural language with good accuracy.

The current state of the art algorithms within the field of deep neural network are able to learn highly complex models of the patterns inherent in a set of data. While the capabilities are impressive, human beings are capable of doing much more than just image recognition or understanding what people are talking about and automating those tasks through machines seems far-fetched.

Let us see some use cases where we need human creativity (at least as

of now):

- Training an artificial author that can write an article and explain data science concepts to a community in a very simplistic manner by learning from past articles from Wikipedia
- Creating an artificial painter that can paint like any famous artist by learning from his/her past collections

Do you believe that machines are capable of accomplishing these tasks? To your surprise the answer is "YES".
Of course, these are difficult tasks to automate, but GANs have started making some of these tasks possible.
Yann LeCun, a prominent figure in the deep learning domain (Director of Facebook AI) said that:

Generative Adversarial Network (GANs), and the variations that are now being proposed is the most interesting idea in the last 10 years in Machine Learning.

If you feel intimidated by the name GAN, don't worry! You will master this technique and apply it to real-world problems yourself by the end of this book.

The purpose of GAN

Some generative models are able to generate samples from model distribution. GANs are an example of generative models. GAN focuses primarily on generating samples from distribution.
You might be wondering why generative models are worth studying, especially generative models that are only capable of generating data rather than providing an estimate of the density function.
Some of the reasons to study generative models are as follows:

- Sampling (or generation) is straightforward
- Training doesn't involve maximum likelihood estimation
- Robust to overfitting since the generator never sees the training data
- GANs are good at capturing the modes of distribution

An analogy from the real world

Let's consider the real-world relationship between a money counterfeiting criminal and the police. Let's enumerate the objective of the criminal and the police in terms of money:

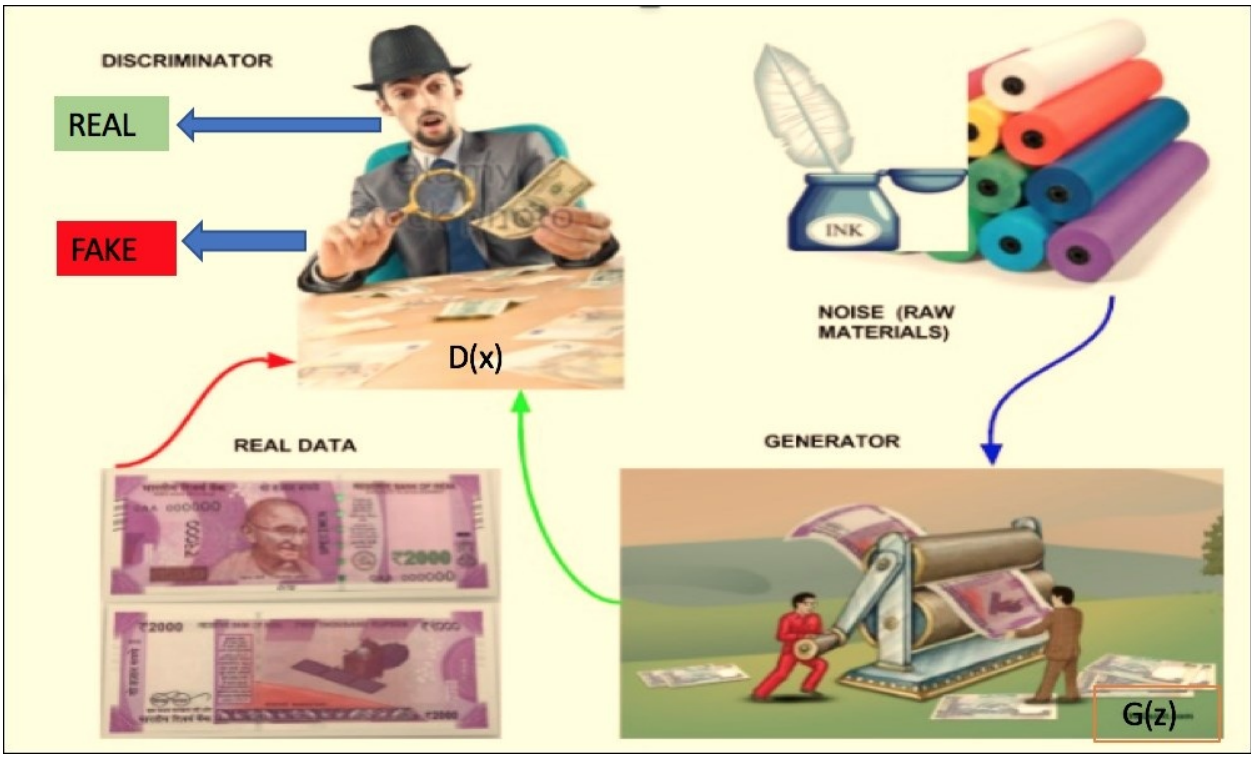


Figure1a: GAN real world analogy

- To become a successful money counterfeiter, the criminal needs to fool the police so that the police can't tell the difference between the counterfeit/fake money and real money
- As a paragon of justice, the police want to detect fake money as effectively as possible

This can be modeled as a minimax game in game theory. This phenomenon is called **adversarial process**. GAN, introduced by Ian Goodfellow in 2014 at *arXiv: 1406.2661*, is a special case of an adversarial process where two neural networks compete against each other. The first network generates data and the second network tries to find the difference between the real data and the fake data generated by the first network. The second network will output a scalar [0, 1], which represents a probability of real data.

The building blocks of GAN

In GAN, the first network is called generator and is often represented as *G(z)* and the second network is called discriminator and is often represented as *D(x)*:

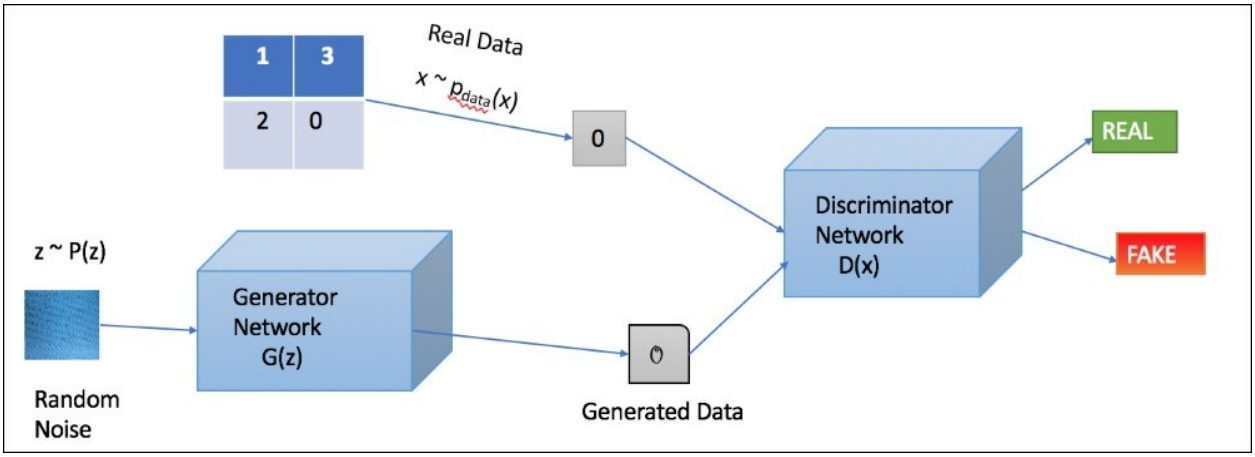


Figure 1b: Generative adversarial network

At the equilibrium point, which is the optimal point in the minimax game, the first network will model the real data and the second network will output a probability of 0.5 as the output of the first network = real data:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{x\sim p_z(z)}\left[\log\left(1-D\left(G(z)\right)\right)\right]$$

Sometimes the two networks eventually reach equilibrium, but this is not always guaranteed and the two networks can continue learning for a long time. An example of learning with both generator and discriminator loss is shown in the following figure:

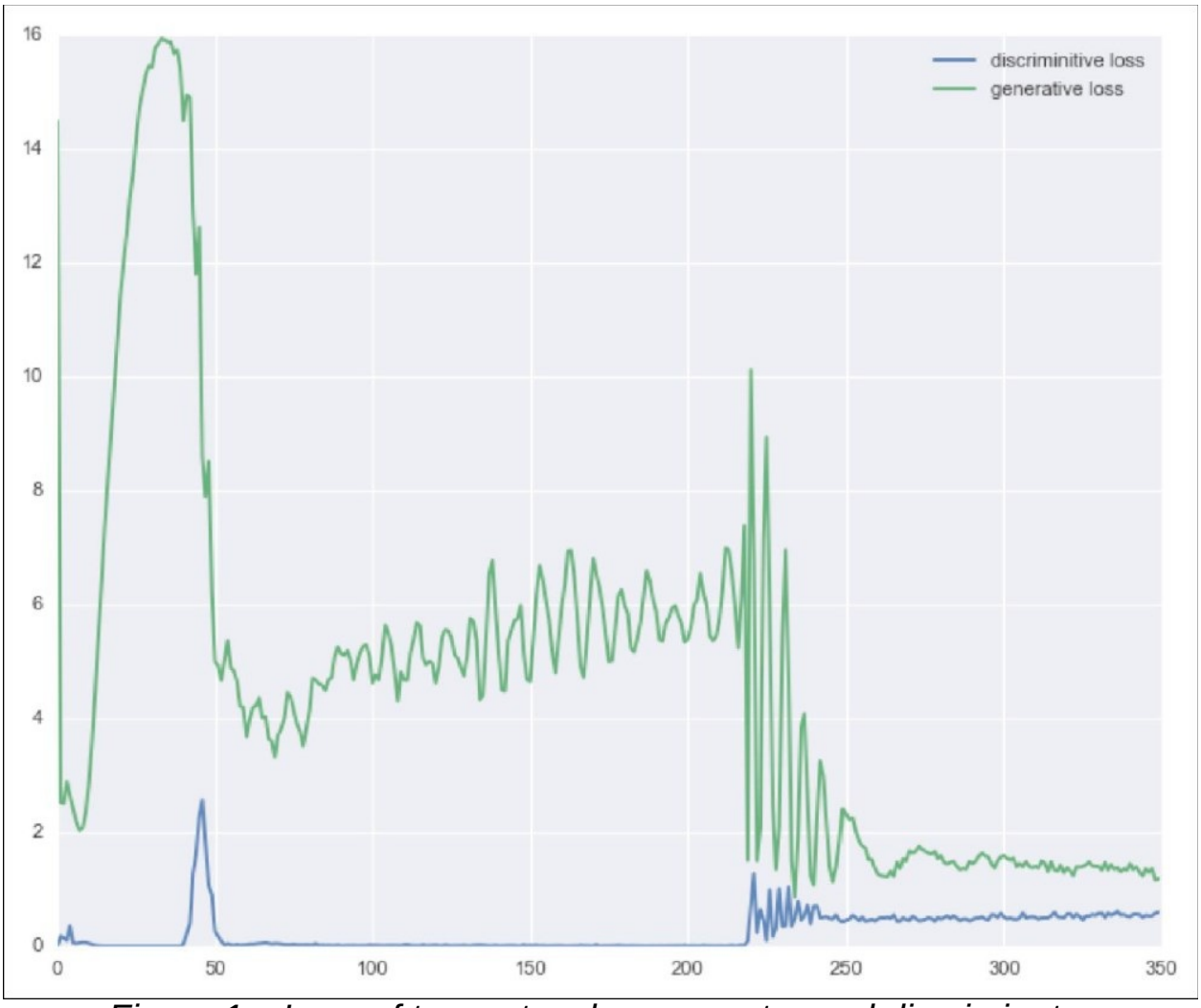


Figure 1c: Loss of two networks, generator and discriminator

Generator

The generator network takes as input random noise and tries to generate a sample of data. In the preceding figure, we can see that generator *G(z)* takes an input *z* from probability distribution *p(z)* and generates data that is then fed into a discriminator network *D(x)*.

Discriminator

The discriminator network takes input either from the real data or from the generator's generated data and tries to predict whether the input is real or generated. It takes an input *x* from real data distribution $P_{data}(x)$ and then solves a binary classification problem giving output in the scalar range 0 to 1.

GANs are gaining lot of popularity because of their ability to tackle the important challenge of unsupervised learning, since the amount of

available unlabeled data is much larger than the amount of labeled data. Another reason for their popularity is that GANs are able to generate the most realistic images among generative models. Although this is subjective, it is an opinion shared by most practitioners.

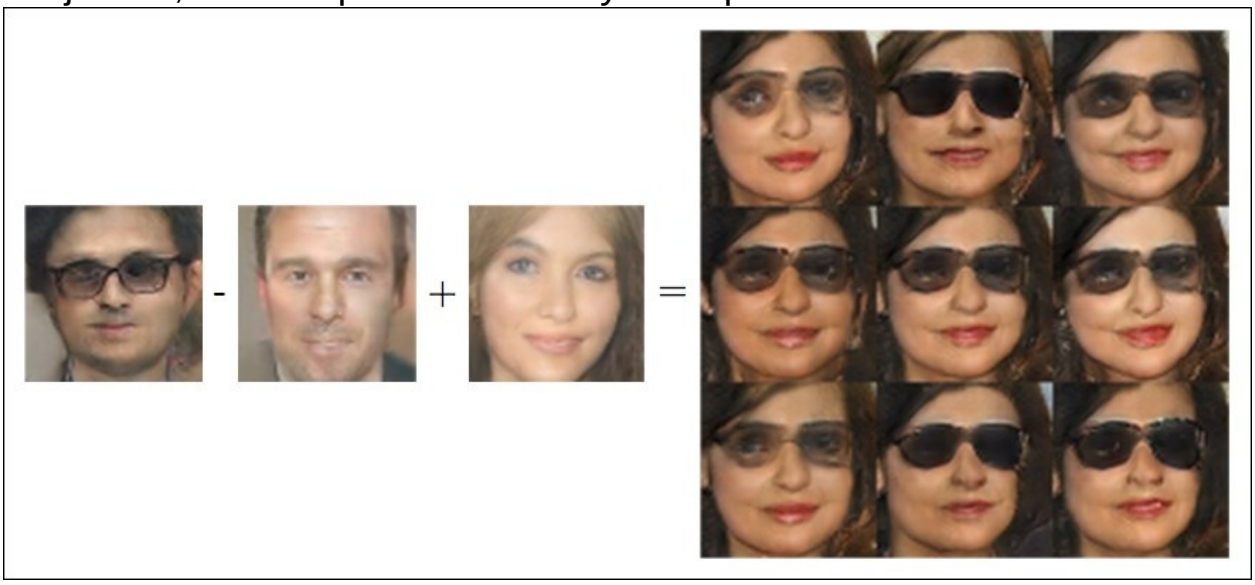


Figure-1d: Vector arithmetic in GANs

Beside this, GAN is often very expressive: it can perform arithmetic operations in the latent space, that is the space of the z vectors, and translate into corresponding operations in feature space. As shown in *Figure 1d*, if you take the representation of a man with glasses in latent space, subtract the `neutral man` vector and add back the `neutral woman` vector, you end up with a picture of a woman with glasses in feature space. This is truly amazing.

Implementation of GAN

As per the definition of GAN, we basically require two networks, be it a sophisticated network such as ConvNet or a simple two-layer neural network. Let's use a simple two-layer neural network with the `MNIST` dataset using TensorFlow for implementation purposes. `MNIST` is a dataset of handwritten digits where each image is gray scale of dimension 28x28 pixel:

```
# Random noise setting for Generator
Z = tf.placeholder(tf.float32, shape=[None, 100],
name='Z')

#Generator parameter settings
G_W1 = tf.Variable(xavier_init([100, 128]),
name='G_W1')
G_b1 = tf.Variable(tf.zeros(shape=[128]), name='G_b1')
G_W2 = tf.Variable(xavier_init([128, 784]),
name='G_W2')
G_b2 = tf.Variable(tf.zeros(shape=[784]), name='G_b2')
theta_G = [G_W1, G_W2, G_b1, G_b2]


# Generator Network
def generator(z):
    G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
    G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
    G_prob = tf.nn.sigmoid(G_log_prob)

    return G_prob
```

The `generator(z)` takes as input a 100-dimensional vector from a random distribution (in this case we are using uniform distribution) and returns a 786-dimensional vector, which is a `MNIST` image (28x28). The `z` here is the prior for the *G(z)*. In this way, it learns a mapping between the prior space to p_{data} (real data distribution):

```
#Input Image MNIST setting for Discriminator
[28x28=784]
X = tf.placeholder(tf.float32, shape=[None, 784],
name='X')

#Discriminator parameter settings
D_W1 = tf.Variable(xavier_init([784, 128]),
name='D_W1')
D_b1 = tf.Variable(tf.zeros(shape=[128]), name='D_b1')
D_W2 = tf.Variable(xavier_init([128, 1]), name='D_W2')
```

```
D_b2 = tf.Variable(tf.zeros(shape=[1]), name='D_b2')
theta_D = [D_W1, D_W2, D_b1, D_b2]


# Discriminator Network
def discriminator(x):
    D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
    D_logit = tf.matmul(D_h1, D_W2) + D_b2
    D_prob = tf.nn.sigmoid(D_logit)

return D_prob, D_logit
```

Whereas the `discriminator(x)` takes `MNIST` image(s) as input and returns a scalar that represents a probability of real image. Now, let's discuss an algorithm for training GAN. Here's the pseudo code for a training algorithm from the paper *arXiv: 1406.2661, 2014*:

for number of training iterations **do**
  for k steps **do**
    - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

Figure 1e: GAN training algorithm pseudo-code

```
G_sample = generator(Z)

D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(G_sample)

# Loss functions according the GAN original paper
D_loss = -tf.reduce_mean(tf.log(D_real) + tf.log(1. -
D_fake))
G_loss = -tf.reduce_mean(tf.log(D_fake))
```

The TensorFlow optimizer can only do minimization, so in order to maximize the `loss` function, we are using a negative sign for the loss as

seen previously. Also, as per the paper's pseudo algorithm, it's better to maximize `tf.reduce_mean(tf.log(D_fake))` instead of minimizing `tf.reduce_mean(1 - tf.log(D_fake)`. Then we train the networks one by one with those preceding `loss` functions:

```
# Only update D(X)'s parameters, so var_list = theta_D
D_solver = tf.train.AdamOptimizer().minimize(D_loss,
var_list=theta_D)
# Only update G(X)'s parameters, so var_list = theta_G
G_solver = tf.train.AdamOptimizer().minimize(G_loss,
var_list=theta_G)

def sample_Z(m, n):
    '''Uniform prior for G(Z)'''
    return np.random.uniform(-1., 1., size=[m, n])

for it in range(1000000):
    X_mb, _ = mnist.train.next_batch(mb_size)

    _, D_loss_curr = sess.run([D_solver, D_loss],
feed_dict={X: X_mb, Z: sample_Z(mb_size, Z_dim)})
    _, G_loss_curr = sess.run([G_solver, G_loss],
feed_dict={Z: sample_Z(mb_size, Z_dim)})
```

After that we start with random noise and as the training continues, `G(Z)` starts moving towards p_{data}. This is proved by the more similar samples generated by `G(Z)` compared to original MNIST images.
Some of the output generated after 60,000 iterations is shown as follows:

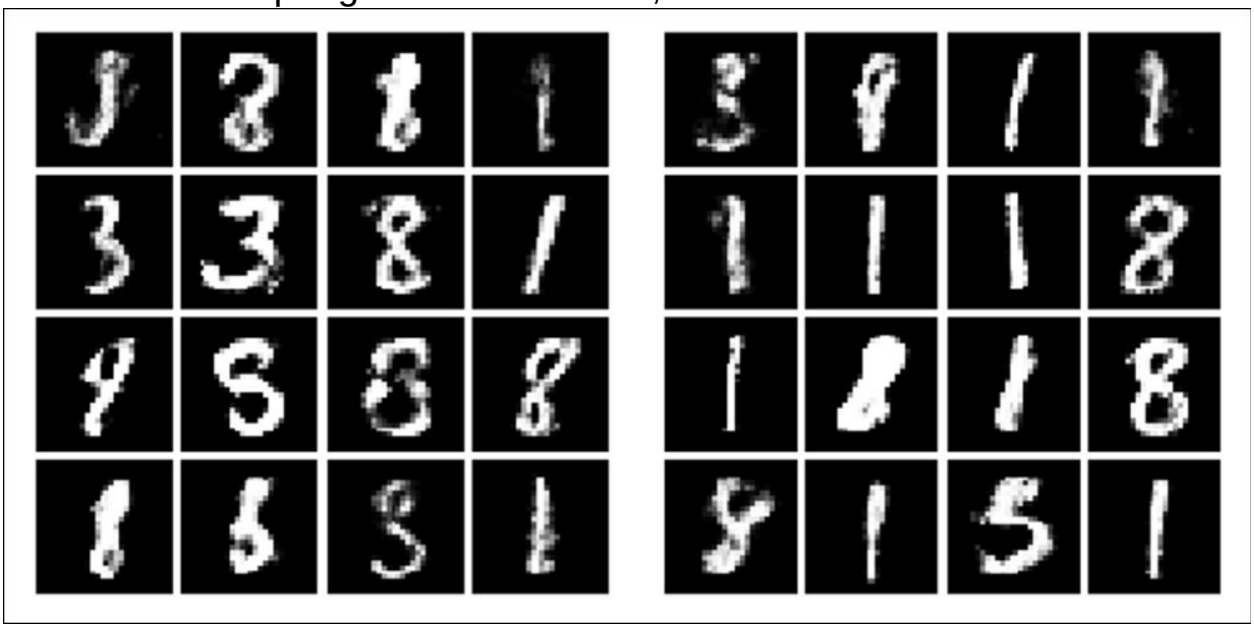


Figure 1f: GAN implementation of a generated output image

Applications of GAN

GAN is generating lots of excitement in a wide variety of fields. Some of the exciting applications of GAN in recent years are listed as follows:

- Translating one image to another (such as horse to zebra) with CycleGAN and performing image editing through Conditional GAN. Details will be covered in Chapter 3, *Transfer Image Style Across Various Domains*.
- Automatic synthesis of realistic images from a textual sentence using StackGAN. And transferring style from one domain to another domain using **Discovery GAN** (**DiscoGAN**). Details will be covered in Chapter 4, *Building Realistic Images from Your Text*.
- Enhancing image quality and generating high resolution images with pre-trained models using SRGAN. Details will be covered in Chapter 5, *Using Various Generative Models to Generate Images*.
- **Generating realistic a image from attributes**: Let's say a burglar comes to your apartment but you don't have a picture of him/her. Now the system at the police station could generate a realistic image of the thief based on the description provided by you and search a database. For more information refer to *arXiv: 1605.05396, 2016*.
- Predicting the next frame in a video or dynamic video generation: (http://carlvondrick.com/tinyvideo/).

Image generation with DCGAN using Keras

The **Deep Convolutional Generative Adversarial Networks** (**DCGAN**) are introduced in the paper: *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, by *A. Radford, L. Metz, and S. Chintala, arXiv:1511.06434, 2015*.

The generator uses a 100-dimensional, uniform distribution space, *Z*, which is then projected into a smaller space by a series of convolution operations. An example is shown in the following figure:

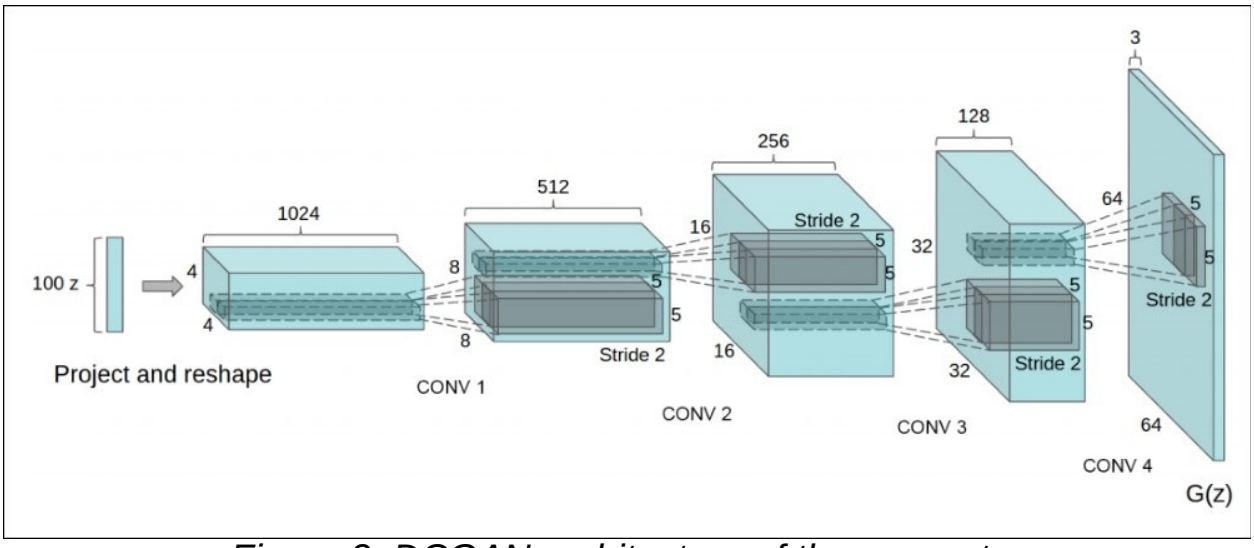


Figure 2: DCGAN architecture of the generator
Source: arXiv, 1511.06434,2015

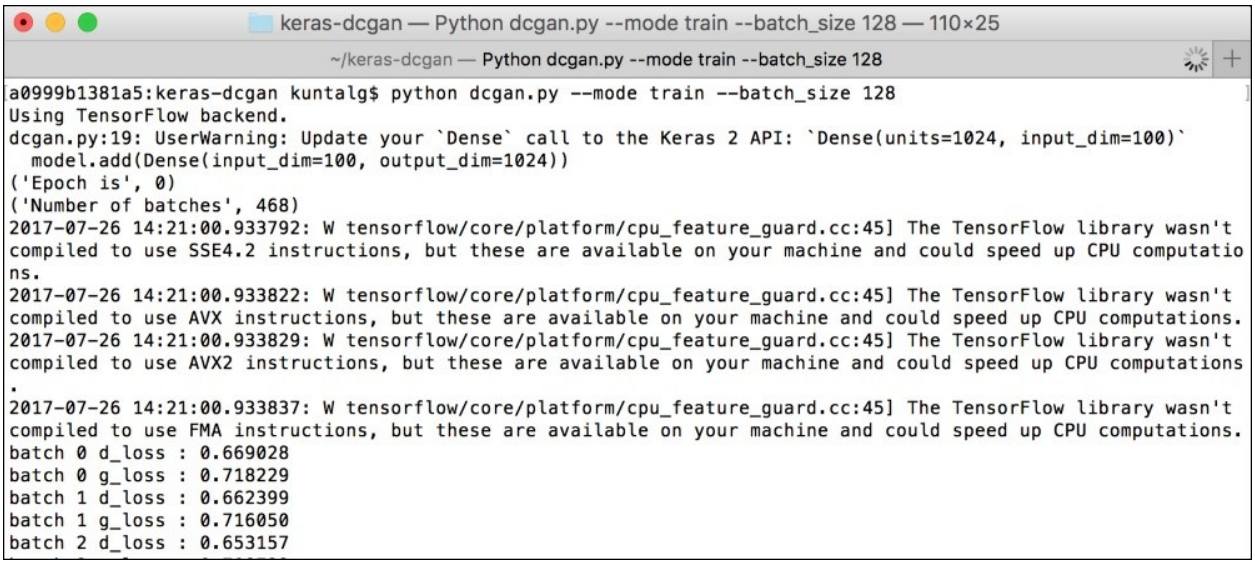
DCGAN stabilizes the networks with the following architectural constraints:

- Replace any pooling layers with strided convolutions in the discriminator and fractional-strided convolutions in the generator
- Use batchnorm in both the generator and the discriminator
- Remove fully connected hidden layers for deeper architectures and simply use average pooling at the end
- Use ReLU activation in the generator for all layers except for the output, which uses `tanh`
- Use leaky ReLU activation in the discriminator for all layers

A DCGAN generator can be described by the following code implemented in Keras, available at: https://github.com/jacobgil/keras-dcgan.

Start the training/generation process with the following command:

```
python dcgan.py --mode train --batch_size <batch_size>
python dcgan.py --mode generate --batch_size
<batch_size> --nice
```



Note that the number of batches printed previously is calculated based on input image shape/batch size (provided).

Now let's jump into the code. The generator can be described with the following:

```
def generator_model():
    model = Sequential()
    model.add(Dense(input_dim=100, output_dim=1024))
    model.add(Activation('tanh'))
    model.add(Dense(128*7*7))
    model.add(BatchNormalization())
    model.add(Activation('tanh'))
    model.add(Reshape((7, 7, 128), input_shape=
(128*7*7,)))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Conv2D(64, (5, 5), padding='same'))
    model.add(Activation('tanh'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Conv2D(1, (5, 5), padding='same'))
    model.add(Activation('tanh'))
    return model
```

The first dense layer of the generator takes as input a vector of 100 dimensions and it produces output of 1,024 dimensions with the activation function `tanh`.

The next dense layer in the network produces data of 128 x 7 x 7 in the output using batch normalization (refer to *Batch Normalization Accelerating Deep Network Training by Reducing Internal Covariate Shift*, by *S. Ioffe* and *C.Szegedy*, *arXiv: 1502.03167*, 2014), a technique that often helps to stabilize learning by normalizing the input with zero mean and unit variance. Batch normalization has been empirically proven to

speed up training in many situations, reduce the problems of poor initialization, and in general produce more accurate results. There is also a `Reshape()` module that produces data of 128 x 7 x 7 (128 channels, 7 width, and 7 height), `dim_ordering` to `tf`, and a `UpSampling()` module that produces a repetition of each one into a 2 x 2 square. After that, we have a convolutional layer that produces 64 filters on 5 x 5 convolutional kernels/patches with `tanh` activation having same padding followed by a new `UpSampling()` and a final convolution with one filter, and on 5 x 5 convolutional kernels with the activation as `tanh`. Note that there are no pooling operations in the ConvNet.
The discriminator can be described with the following code:

```
def discriminator_model():
    model = Sequential()
    model.add(Conv2D(64, (5, 5),
padding='same',input_shape=(28, 28, 1)))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(128, (5, 5)))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(1024))
    model.add(Activation('tanh'))
    model.add(Dense(1))
    model.add(Activation('sigmoid'))
    return model
```

The discriminator takes a standard MNIST image with the shape (`1, 28, 28`) and applies a convolution with 64 filters of size 5 x 5 with `tanh` as the activation function. It is then followed by a max-pooling operation of size 2 x 2 and by a further convolution max-pooling operation.
The last two stages are dense, with the final one being the prediction for forgery, which consists of only a single neuron with a `sigmoid` activation function. For a given number of epochs, the generator and discriminator are trained by using `binary_crossentropy` as a `loss` function. At each epoch, the generator makes a prediction of a number (for example, it creates forged MNIST images) and the discriminator tries to learn after mixing the prediction with real MNIST images. After a few epochs, the generator automatically learns to forge this set of handwritten numbers:

Figure-3: Deep convolutional GAN generated handwritten digit output

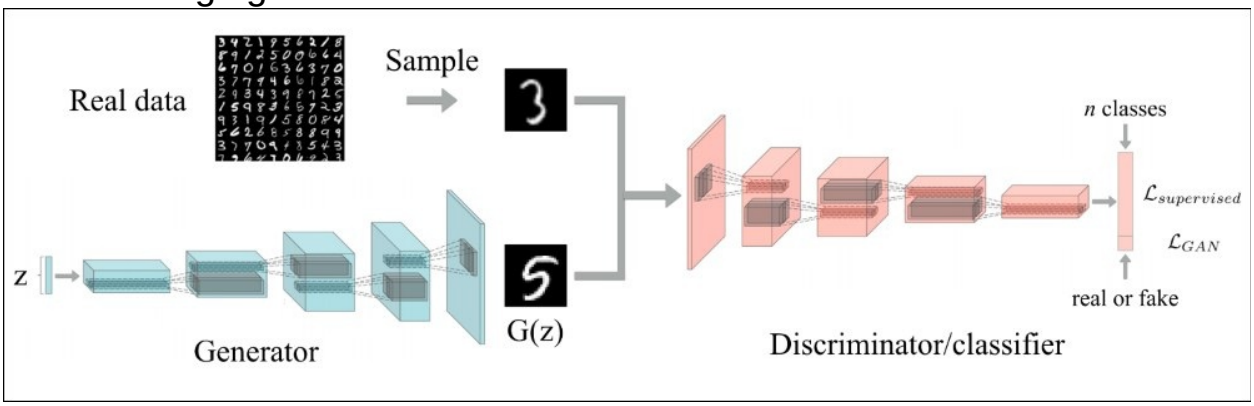
Note that training GANs could be very difficult because it is necessary to find the equilibrium between two players and hence some of the valuable techniques and tips used by practitioners are given in the final section of this chapter.

Implementing SSGAN using TensorFlow

The basic intuition of **Semi-Supervised Learning Generative Adversarial Network** (**SSGAN**) is to exploit the samples generated by generators to enhance the performance of image classification tasks of the discriminator by improving generalization. The key idea is to train one of the networks as both image classifier and discriminator (to identify generated images from real images).

For a dataset having *n* classes, the dual trained (discriminator/classifier) network will take an image as input and classify the real images into the first *n* classes and generated images into the *n+1-th* class, as shown in

the following figure:



Source: https://github.com/gitlimlab/SSGAN-Tensorflow/blob/master/figure/ssgan.png

This multi-tasking learning framework consists of two losses, first the supervised loss:

$$\mathcal{L}_{supervised} = -\mathbb{E}_{x,y\sim p_{data}(x,y)} \log p_{model}\left(y \mid x, y < n+1\right)$$

and second the GAN loss of a discriminator:

$$\mathcal{L}_{GAN} = -\left\{\mathbb{E}_{x\sim p_{data}(x)} \log\left[1 - p_{model}\left(y = n+1 \mid x\right)\right] + \mathbb{E}_{x\sim Generator} \log p_{model}\left(y = n+1 \mid x\right)\right\}$$

During the training phase, both these losses are jointly minimized.

Setting up the environment

Perform the following steps to execute SSGAN on Cifar-10 datasets:

1. Clone the `git` repo: https://github.com/gitlimlab/SSGAN-Tensorflow:

```
a0999b1381a5:~ kuntalg$ git clone https://github.com/gitlimlab/SSGAN-Tensorflow.git
Cloning into 'SSGAN-Tensorflow'...
remote: Counting objects: 494, done.
remote: Total 494 (delta 0), reused 0 (delta 0), pack-reused 494
Receiving objects: 100% (494/494), 50.86 MiB | 1.26 MiB/s, done.
Resolving deltas: 100% (295/295), done.
```

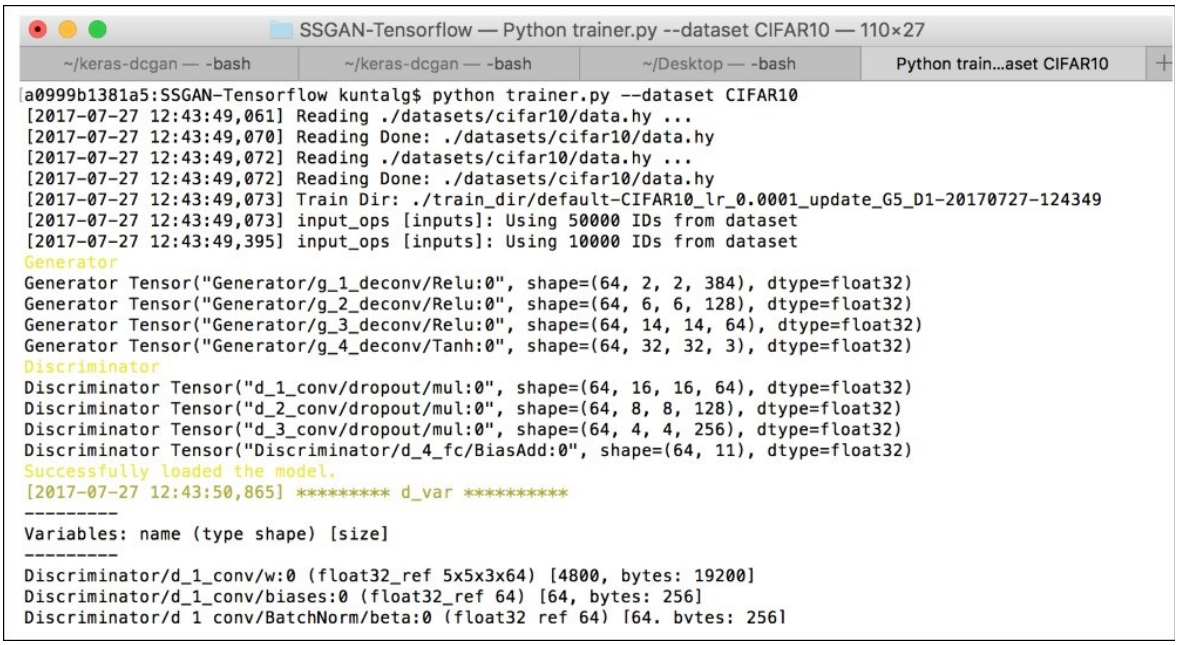
2. Change the directory:

```
cd SSGAN-Tensorflow/
```

3. Download the `CIFAR-10` dataset:

```
a0999b1381a5:SSGAN-Tensorflow kuntalg$ python download.py --dataset CIFAR10
./datasets/cifar10/cifar-10-python.tar.gz
Downloading CIFAR10
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  162M  100  162M    0     0  1248k      0  0:02:13  0:02:13 --:--:-- 1757k
Preprocessing data...
[==================================================================================================] 100%
```

4. Train the model:



5. Test or evaluate the model:

```
python evaler.py --dataset CIFAR10 --checkpoint
ckpt_dir
```

Now let's dive into the code. The generator takes random noise from the uniform distribution:

```
z = tf.random_uniform([self.batch_size, n_z],
minval=-1, maxval=1, dtype=tf.float32)
```

Then the generator model flattens the input noise to 1 dimensional vector using the `reshape` method. It then applies three layers of deconvolution on the input noise having `ReLU` activation and then applies one more deconvolution with `tanh` activation to generate the output image of dimension [*h*=height, *w*=width, *c*], where *c* is the number of channels (grayscale images: 1, color images: 3):

```
# Generator model function
        def G(z, scope='Generator'):
            with tf.variable_scope(scope) as scope:
                print
('\033[93m'+scope.name+'\033[0m')
                z = tf.reshape(z, [self.batch_size, 1,
1, -1])
                g_1 = deconv2d(z, deconv_info[0],
is_train, name='g_1_deconv')
                print (scope.name, g_1)
                g_2 = deconv2d(g_1, deconv_info[1],
is_train, name='g_2_deconv')
                print (scope.name, g_2)
```

```
                g_3 = deconv2d(g_2, deconv_info[2],
is_train, name='g_3_deconv')
                print (scope.name, g_3)
                g_4 = deconv2d(g_3, deconv_info[3],
is_train, name='g_4_deconv', activation_fn='tanh')
                print (scope.name, g_4)
                output = g_4
                assert output.get_shape().as_list() ==
self.image.get_shape().as_list(),
output.get_shape().as_list()
            return output

# Deconvolution method
def deconv2d(input, deconv_info, is_train,
name="deconv2d", stddev=0.02,activation_fn='relu'):
    with tf.variable_scope(name):
        output_shape = deconv_info[0]
        k = deconv_info[1]
        s = deconv_info[2]
        deconv = layers.conv2d_transpose(input,
            num_outputs=output_shape,

weights_initializer=tf.truncated_normal_initializer(st
ddev=stddev),
            biases_initializer=tf.zeros_initializer(),
            kernel_size=[k, k], stride=[s, s],
padding='VALID')
        if activation_fn == 'relu':
            deconv = tf.nn.relu(deconv)
            bn = tf.contrib.layers.batch_norm(deconv,
center=True, scale=True,
                decay=0.9, is_training=is_train,
updates_collections=None)
        elif activation_fn == 'tanh':
            deconv = tf.nn.tanh(deconv)
        else:
            raise ValueError('Invalid activation
function.')
        return deconv
```

The discriminator takes images as input and tries to output into a `n+1` class label. It applies some layers of convolution having leaky ReLU with batch normalization, followed by dropout on the input images, and finally outputs the class `label` using the `softmax` function:

```
# Discriminator model function
        def D(img, scope='Discriminator', reuse=True):
            with tf.variable_scope(scope, reuse=reuse)
as scope:
                if not reuse: print
```

```
('\033[93m'+scope.name+'\033[0m')
                d_1 = conv2d(img, conv_info[0],
is_train, name='d_1_conv')
                d_1 = slim.dropout(d_1, keep_prob=0.5,
is_training=is_train, scope='d_1_conv/')
                if not reuse: print (scope.name, d_1)
                d_2 = conv2d(d_1, conv_info[1],
is_train, name='d_2_conv')
                d_2 = slim.dropout(d_2, keep_prob=0.5,
is_training=is_train, scope='d_2_conv/')
                if not reuse: print (scope.name, d_2)
                d_3 = conv2d(d_2, conv_info[2],
is_train, name='d_3_conv')
                d_3 = slim.dropout(d_3, keep_prob=0.5,
is_training=is_train, scope='d_3_conv/')
                if not reuse: print (scope.name, d_3)
                d_4 = slim.fully_connected(
                    tf.reshape(d_3, [self.batch_size,
-1]), n+1, scope='d_4_fc', activation_fn=None)
                if not reuse: print (scope.name, d_4)
                output = d_4
                assert output.get_shape().as_list() ==
[self.batch_size, n+1]
                return tf.nn.softmax(output), output


# Convolution method with dropout
def conv2d(input, output_shape, is_train, k_h=5,
k_w=5, stddev=0.02, name="conv2d"):
    with tf.variable_scope(name):
        w = tf.get_variable('w', [k_h, k_w,
input.get_shape()[-1], output_shape],

initializer=tf.truncated_normal_initializer(stddev=std
dev))
        conv = tf.nn.conv2d(input, w, strides=[1, 2,
2, 1], padding='SAME')

        biases = tf.get_variable('biases',
[output_shape],
initializer=tf.constant_initializer(0.0))
        conv = lrelu(tf.reshape(tf.nn.bias_add(conv,
biases), conv.get_shape()))
        bn = tf.contrib.layers.batch_norm(conv,
center=True, scale=True,
            decay=0.9, is_training=is_train,
updates_collections=None)
    return bn

# Leaky Relu method
```

```
def lrelu(x, leak=0.2, name="lrelu"):
   with tf.variable_scope(name):
   f1 = 0.5 * (1 + leak)
   f2 = 0.5 * (1 - leak)
return f1 * x + f2 * abs(x)
```

The discriminator network has two `loss` functions, one (`s_loss`) for the supervise classification of real data from CIFAR-10 images using huber loss (Huber loss is robust to outliers compared to squared error loss) and the other (`d_loss`) loss to classify the generated images by the generator as real/fake in scalar form using `softmax` function with cross entropy:

```
# Discriminator/classifier loss
s_loss = tf.reduce_mean(huber_loss(label, d_real[:,
:-1]))
```

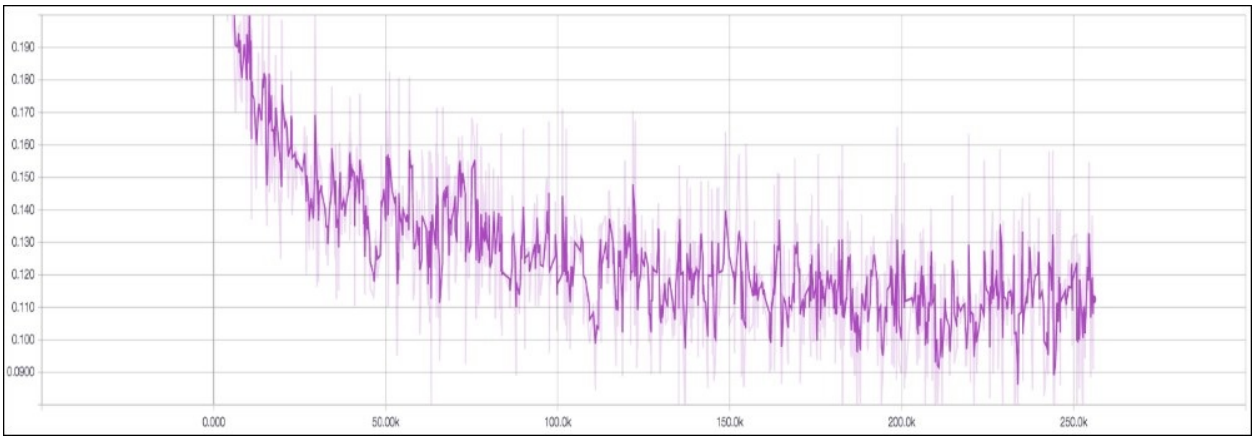


Figure: 4a: Supervise loss of discriminator

```
d_loss_real =
tf.nn.softmax_cross_entropy_with_logits(logits=d_real_
logits, labels=real_label)
 d_loss_fake =
tf.nn.softmax_cross_entropy_with_logits(logits=d_fake_
logits, labels=fake_label)
d_loss = tf.reduce_mean(d_loss_real + d_loss_fake)
```

Figure: 4b: Total discriminator loss (real + fake loss)

```
# Huber loss
def huber_loss(labels, predictions, delta=1.0):
```

```
    residual = tf.abs(predictions - labels)
    condition = tf.less(residual, delta)
    small_res = 0.5 * tf.square(residual)
    large_res = delta * residual - 0.5 *
tf.square(delta)
    return tf.where(condition, small_res, large_res)


# Generator loss
g_loss = tf.reduce_mean(tf.log(d_fake[:, -1]))

g_loss += tf.reduce_mean(huber_loss(real_image,
fake_image)) * self.recon_weight
```

Note

Note: Weight annealing is done as an auxiliary loss to help the generator get rid of the initial local minimum.

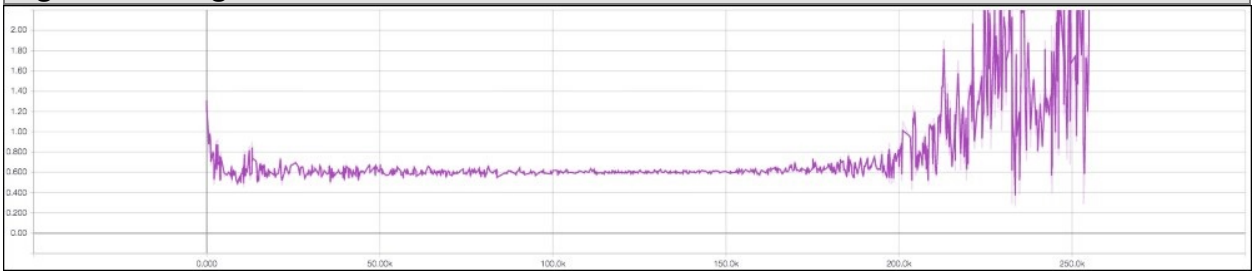


Figure: 4c: Generator loss

The `loss` function of both the generator and discriminator network is optimized with `AdamOptimizer` and gradient clipping (`clip_gradients`) is applied with it to stabilize the training:

```
# Optimizer for discriminator
self.d_optimizer = tf.contrib.layers.optimize_loss(
loss=self.model.d_loss,
global_step=self.global_step,
learning_rate=self.learning_rate*0.5,
optimizer=tf.train.AdamOptimizer(beta1=0.5),
clip_gradients=20.0,
name='d_optimize_loss',
variables=d_var
)


# Optimizer for generator
self.g_optimizer = tf.contrib.layers.optimize_loss(
loss=self.model.g_loss,
global_step=self.global_step,
learning_rate=self.learning_rate,
optimizer=tf.train.AdamOptimizer(beta1=0.5),
```

```
clip_gradients=20.0,
name='g_optimize_loss',
variables=g_var
)
```

Finally, both the supervised loss (`s_loss`) and generative adversarial loss (which is the combination of discriminator loss (`d_loss`) and generator loss (`g_loss`)) are trained jointly to minimize the total loss:

```
for s in xrange(max_steps):
                step, accuracy, summary, d_loss, g_loss,
s_loss, step_time, prediction_train, gt_train, g_img =
\
                  self.run_single_step(self.batch_train,
step=s, is_train=True)
```

The output of generated samples after 150 epochs is as follows:

Challenges of GAN models

Training a GAN is basically about two networks, generator *G(z)* and discriminator *D(z)*, trying to race against each other and trying to reach an optimum, more specifically a nash equilibrium. The definition of nash equilibrium as per Wikipedia (in economics and game theory) is a stable state of a system involving the interaction of different participants, in which no participant can gain by a unilateral change of strategy if the strategies of the others remain unchanged.

Setting up failure and bad initialization

If you think about it, this is exactly what GAN is trying to do; the generator and discriminator reach a state where they cannot improve any further given the other is kept unchanged. Now the setup of gradient descent is to take a step in a direction that reduces the loss measure defined on the problem—but we are by no means enforcing the networks to reach Nash equilibrium in GAN, which have non-convex objective with continuous high dimensional parameters. The networks try to take successive steps to minimize a non-convex objective and end up in an oscillating process rather than decreasing the underlying true objective.

In most cases, when your discriminator attains a loss very close to zero, then right away you can figure out something is wrong with your model. But the biggest difficulty is figuring out what is wrong.

Another practical thing done during the training of GAN is to purposefully make one of the networks stall or learn slower, so that the other network can catch up. And in most scenarios, it's the generator that lags behind so we usually let the discriminator wait. This might be fine to some extent, but remember that for the generator to get better, it requires a good discriminator and vice versa. Ideally the system would want both the networks to learn at a rate where both get better over time. The ideal minimum loss for the discriminator is close to 0.5— this is where the generated images are indistinguishable from the real images from the perspective of the discriminator.

Mode collapse

One of the main failure modes with training a generative adversarial network is called mode collapse or sometimes the helvetica scenario. The basic idea is that the generator can accidentally start to produce several copies of exactly the same image, so the reason is related to the

game theory setup. We can think of the way that we train generative adversarial networks as first maximizing with respect to the discriminator and then minimizing with respect to the generator. If we fully maximize with respect to the discriminator before we start to minimize with respect to the generator, everything works out just fine. But if we go the other way around and we minimize with respect to the generator and then maximize with respect to the discriminator, everything will actually break and the reason is that if we hold the discriminator constant, it will describe a single region in space as being the point that is most likely to be real rather than fake and then the generator will choose to map all noise input values to that same most likely to be real point.

Problems with counting

GANs can sometimes be far-sighted and fail to differentiate the number of particular objects that should occur at a location. As we can see, it gives more numbers of eyes in the head than originally present:

Source: NIPS 2016- arXiv: 1701.00160, 2017

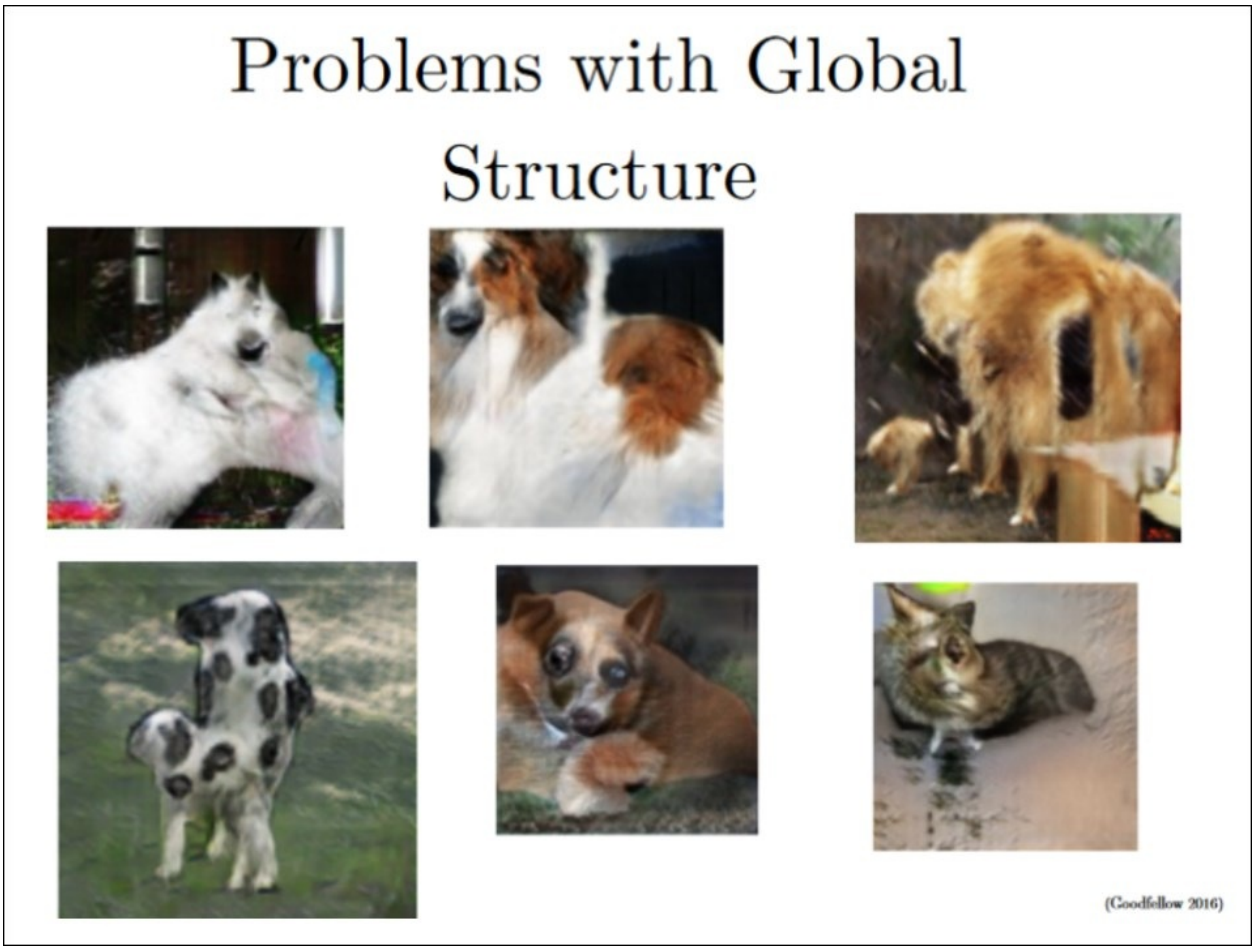
Problems with perspective

GANs sometimes are not capable of differentiating between front and back view and hence fail to adapt well with 3D objects while generating 2D representations from it, as follows:

Source: NIPS 2016- arXiv: 1701.00160, 2017

Problems with global structures

GANs do not understand holistic structures, similar to problems with perspective. For example, in the bottom left image, it generates an image of a quadruple cow, that is, a cow standing on its hind legs and simultaneously on all four legs. That is definitely unrealistic and not possible in real life!



Source: NIPS 2016- arXiv: 1701 .00160, 2017

Improved training approaches and tips for GAN

In order to overcome the difficulties of GAN models, deep learning practitioners carry out various hacks depending on the nature of the problem. Some of the improvisation techniques are mentioned in the following sections.

Feature matching

The instability of GANs is addressed by specifying a new objective for the generator that prevents it from overtraining on the current discriminator. The idea is to use the features at the intermediate layers in the discriminator to match for real and fake images and make this as a supervisory signal to train the generator.

Specifically, we train the generator to generate data that matches the statistics of the real data, and match the expected value of the features on an intermediate layer of the discriminator. By training the discriminator, we ask it to find those features that are most discriminative of real data versus data generated by the current model.

Mini batch

The problem of mode collapse can be addressed by adding some extra features to the discriminator where the discriminator actually looks at an entire "mini batch of samples" at a time rather than looking at a single sample. If those features measure things such as distance to other samples, then the discriminator can detect if the generator is starting to collapse in this way instead of encouraging every sample from the generator to move towards the single most likely point. The mini batch as a whole has to look realistic and have the correct amount of spacing between different samples.

Historical averaging

The idea of historical averaging is to add a penalty term that punishes weights that are rather far away from their historical average values. For example, the cost is:

```
distance (current parameters, average of parameters
over the last t batches)
```

One-sided label smoothing

Usually one would use the labels 0 (image is real) and 1 (image is fake). Instead using some smoother labels (0.1 and 0.9) seems to make networks more resistant to adversarial examples.

Normalizing the inputs

Most of the time it is good to normalize the images between -1 and 1 and use `tanh` as the last layer of the generator output.

Batch norm

The idea is to construct different mini batches for real and fake, that is, each mini batch needs to contain only all real images or all generated images. But when batch norm is not an option, you can use instance normalization (for each sample, subtract mean and divide by standard deviation).

Avoiding sparse gradients with ReLU, MaxPool

The stability of the GAN game suffers if you have sparse gradients. Leaky ReLU is a good fit for both generator and discriminator.
In case of down-sampling, use a combination of average pooling, `Conv2d + stride`, whereas for up-sampling, use the combination of `PixelShuffle`, `ConvTranspose2d + stride`:

```
PixelShuffle- arXiv: 1609.05158, 2016
```

Optimizer and noise

Use the ADAM optimizer for generators and SGD for discriminators. And provide noise in the form of dropout to several layers of generator.

Don't balance loss through statistics only

Instead have a principled approach to it, rather than intuition:

```
while lossD > A:
  train D
while lossG > B:
  train G
```

Note

Note: Despite all these tips and training enhancement steps, the Generative Adversarial model is still relatively new in the field of AI and deep learning and so like any other fast growing field, it too requires a lot of improvement.

Summary

So far you have learned how deep learning made its progress into the unsupervised arena through the concepts of GAN. You have already generated some realistic images such as handwritten digits, airplanes, cars, birds, and so on using `MNIST`, `CIFAR` datasets. Also, you have understood various challenges related to Generative Adversarial Network and how to overcome it with practical tuning tips.

In the next few chapters, we will continue our journey with a different variety of GAN-based architecture to perform some magnificent tasks with real datasets.

Chapter 3. Transfer Image Style Across Various Domains

Generative Adversarial Network is the most rapidly emerging branch of deep learning that is suitable for a wide range of creative applications (such as image editing or painting, style transfer, object transfiguration, photo enhancement, and many more).

In this chapter, you will first learn the technique of generating or editing images based on certain conditions or characteristics. Then, you will stabilize GAN training to overcome the mode-collapse problem and apply a convergence measure metric with the **Boundary Equilibrium** approach. Finally, you will perform image to image translation across various domains (such as changing apple to orange or horse to zebra) using **Cycle Consistent Generative Network**.

We will cover the following topics in this chapter:

- What is CGAN? Its concept and architecture
- Generating fashion wardrobe from `Fashion-MNIST` data using CGAN
- Stabilizing GAN training using Boundary Equilibrium GAN with Wasserstein distance
- Image style transfer across different domains using CycleGAN
- Generating oranges from apples using Tensorflow
- Changing horse images into zebras automatically

Bridging the gap between supervised and unsupervised learning

Humans learn by observing and experiencing the physical world and our brains are very good at prediction without doing explicit computations to arrive at the correct answer. Supervised learning is all about predicting a label associated with the data and the goal is to generalize to new unseen data. In unsupervised learning, the data comes in with no labels, and the goal is often not to generalize any kind of prediction to new data. In the real world, labeled data is often scarce and expensive. The Generative Adversarial Network takes up a supervised learning approach to do unsupervised learning, by generating fake/synthetic looking data,

and tries to determine if the generated sample is fake or real. This part (a discriminator doing classification) is a supervised component. But the actual goal of GAN is to understand what the data looks like (that is, its distribution or density estimation) and be able to generate new examples of what it has learned.

Introduction to Conditional GAN

A **Generative Adversarial Network** (**GAN**) simultaneously trains two networks—a generator that learns to generate fake samples from an unknown distribution or noise and a discriminator that learns to distinguish fake from real samples.

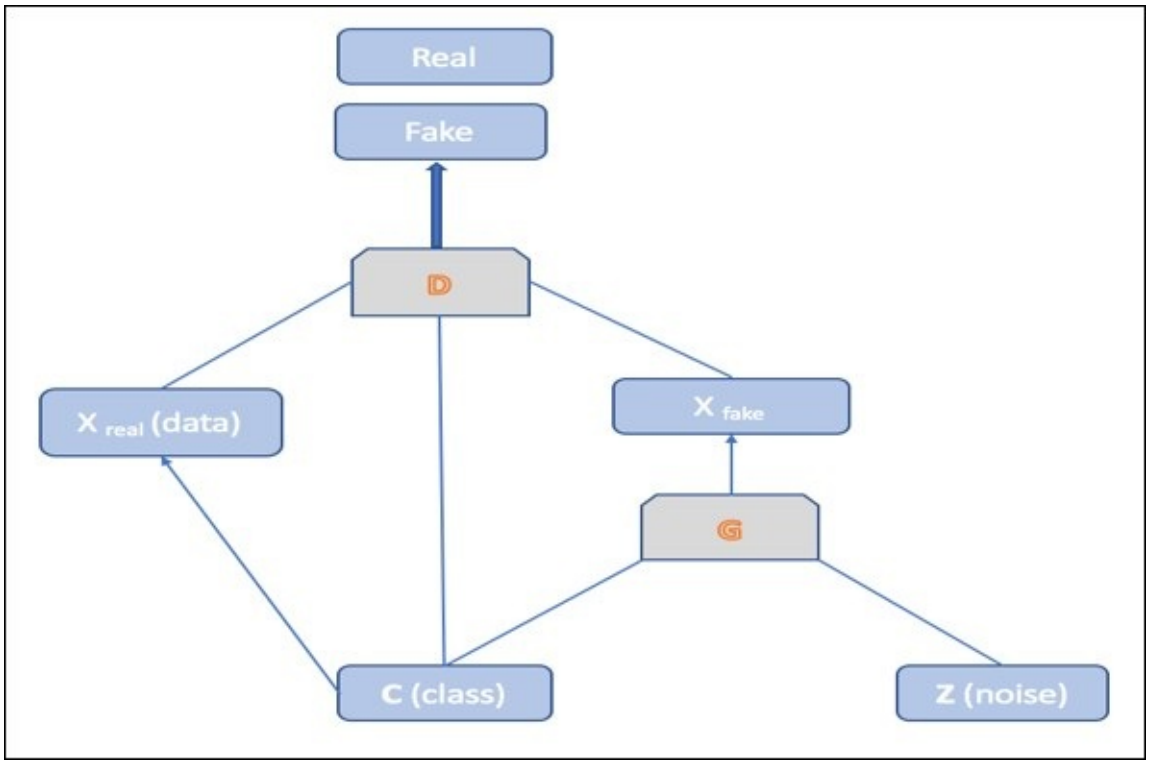
In the **Conditional GAN** (**CGAN**), the generator learns to generate a fake sample with a specific condition or characteristics (such as a label associated with an image or more detailed tag) rather than a generic sample from unknown noise distribution. Now, to add such a condition to both generator and discriminator, we will simply feed some vector *y*, into both networks. Hence, both the discriminator *D(X,y)* and generator *G(z,y)* are jointly conditioned to two variables, *z* or *X* and *y*.

Now, the objective function of CGAN is:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1-D\left(G(z)\right)\right)\right] \Rightarrow \text{Gan}$$

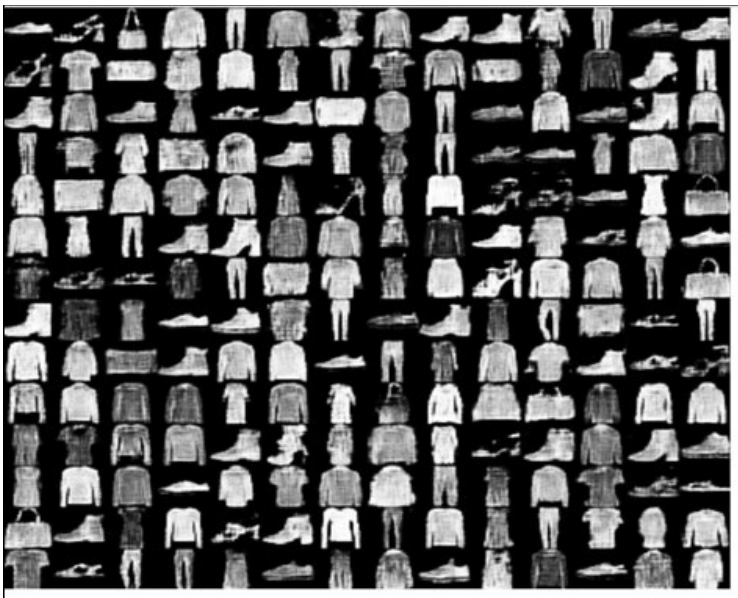
$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1-D\left(G(z,y),y\right)\right)\right] \Rightarrow \text{CGAN}$$

The difference between GAN loss and CGAN loss lies in the additional parameter *y* in both a discriminator and generator function. The architecture of CGAN shown in the following figure now has an additional input layer (in the form of condition vector **C**) that gets fed into both the discriminator network and generator network.



Generating a fashion wardrobe with CGAN

In this example, we will implement conditional GAN to generate a fashion wardrobe using a `Fashion-MNIST` dataset (https://github.com/zalandoresearch/fashion-mnist). The `Fashion-MNIST` dataset is similar to the original `MNIST` dataset with a new set of gray-scale images and labels.



Fashion Wardrobe- images

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Fashion Wardrobe- labels

Let's jump into the code to understand the working of CGAN with simple neural network architecture for both generator and discriminator.
First, we will define a new input variable to hold our condition:

```
Y = tf.placeholder(tf.float32, shape=(None,
num_labels))
```

Next, we incorporate the new variable `y` into the discriminator `D(X)` and generator `G(z)`. Now, the discriminator`(x,y)` and generator`(z,y)` are different than the original GAN:

```
Dhidden = 256  # hidden units of Discriminator's
network
Ghidden = 512  # hidden units of Generator's network
K = 8          # maxout units of Discriminator

# Discriminator Network

def discriminator(x, y):
    u = tf.reshape(tf.matmul(x, DW1x) + tf.matmul(y,
DW1y) + Db1, [-1, K, Dhidden])
    Dh1 = tf.nn.dropout(tf.reduce_max(u,
reduction_indices=[1]), keep_prob)
    return tf.nn.sigmoid(tf.matmul(Dh1, DW2) + Db2)

# Generator Network

def generator(z,y):
    Gh1 = tf.nn.relu(tf.matmul(Z, GW1z) + tf.matmul(Y,
GW1y) + Gb1)
```

```
    G = tf.nn.sigmoid(tf.matmul(Gh1, GW2) + Gb2)
    return G
```

Next, we will use our new networks and define a `loss` function:

```
G_sample = generator(Z, Y)
DG = discriminator(G_sample, Y)

Dloss = -tf.reduce_mean(tf.log(discriminator(X, Y)) +
tf.log(1 - DG))
Gloss = tf.reduce_mean(tf.log(1 - DG) - tf.log(DG +
1e-9))
```

During training, we feed the value of `y` into both a generator network and discriminator network:

```
X_mb, y_mb = mnist.train.next_batch(mini_batch_size)

Z_sample = sample_Z(mini_batch_size, noise_dim)

_, D_loss_curr = sess.run([Doptimizer, Dloss],
feed_dict={X: X_mb, Z: Z_sample, Y:y_mb,
keep_prob:0.5})

_, G_loss_curr = sess.run([Goptimizer, Gloss],
feed_dict={Z: Z_sample, Y:y_mb, keep_prob:1.0})
```

Finally, we generate new data samples based on certain conditions. For this example, we use the image label as our condition and set the label to be `7`, that is, generating the image of `Sneaker`. The conditional variable `y_sample` is a collection of one-hot encoded vectors with value `1` in the seventh index:

```
nsamples=6

      Z_sample = sample_Z(nsamples, noise_dim)
        y_sample = np.zeros(shape=[nsamples,
num_labels])
        y_sample[:, 7] = 1 # generating image based on
label

        samples = sess.run(G_sample, feed_dict={Z:
Z_sample, Y:y_sample})
```

Now let us execute the following steps to generate wardrobe images based on class label condition. First download the `Fashion-MNIST` dataset and save it under the `data`/`fashion` directory by running the `download.py` script:

```
python download.py
```

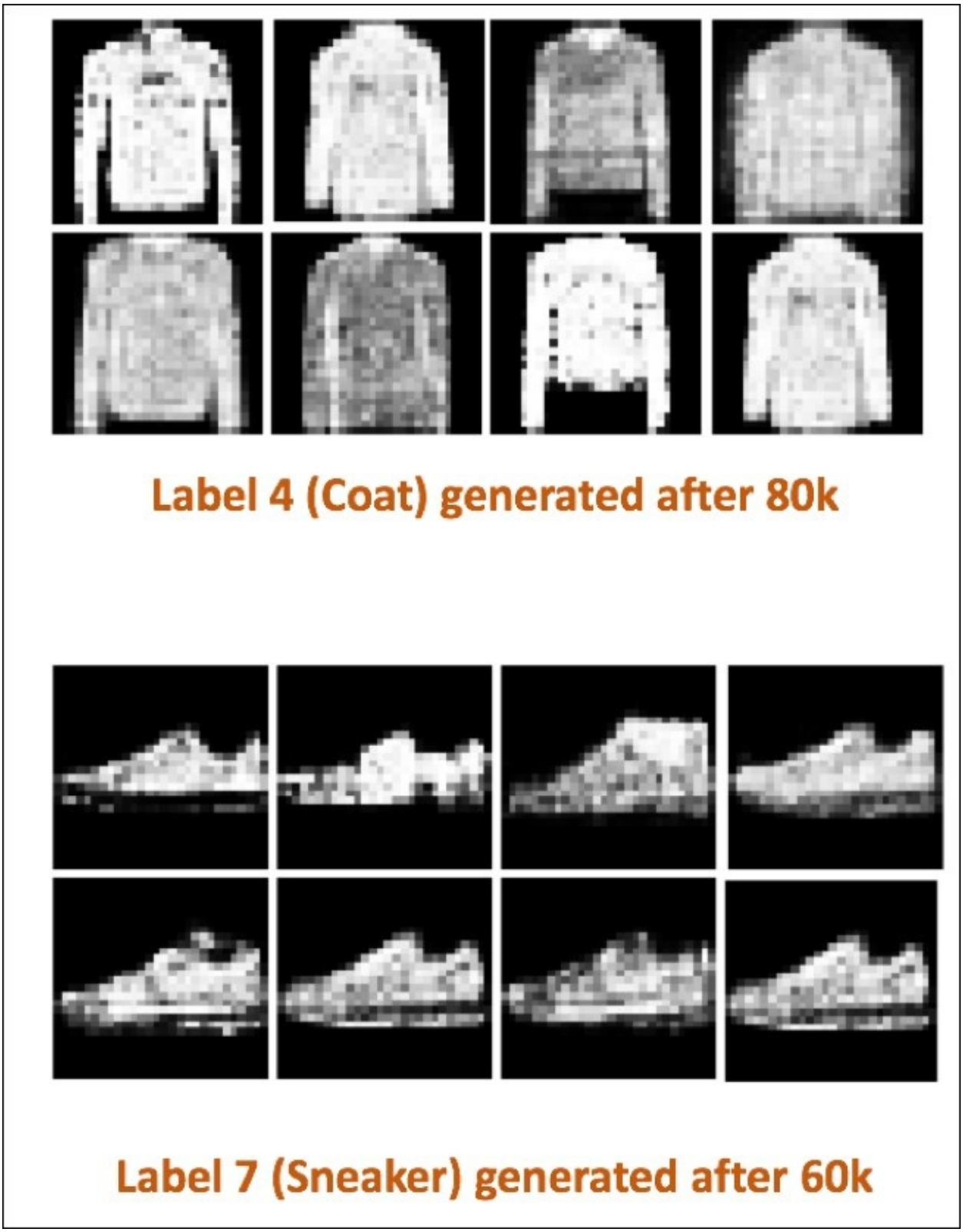
```
a0999b1381a5:CGAN-code kuntalg$ python download.py
http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-uby
te.gz
Downloading  train-images-idx3-ubyte.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
 82 25.2M   82 20.7M    0     0  1640k      0  0:00:15  0:00:12  0:00:03 1725k
```

Next train the CGAN model using the following command, which will generate sample images after every 1000 iterations under the `output` directory:

```
python simple-cgan.py
```

```
a0999b1381a5:CGAN-code kuntalg$ python simple-cgan.py
Extracting ./data/fashion/train-images-idx3-ubyte.gz
Extracting ./data/fashion/train-labels-idx1-ubyte.gz
Extracting ./data/fashion/t10k-images-idx3-ubyte.gz
Extracting ./data/fashion/t10k-labels-idx1-ubyte.gz
2017-08-30 11:50:51.450753: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:51.450788: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use AVX instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:51.450799: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use AVX2 instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:51.450808: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to
 use FMA instructions, but these are available on your machine and could speed up CPU computations.
2017-08-30 11:50:52.412 Python[82549:2076967] ApplePersistenceIgnoreState: Existing state will not be touched. New state
will be written to /var/folders/4h/m2y74lpj7qs_ysw3gpfbcy70k2hbrw/T/org.python.python.savedState
Iter: 0
D loss: 1.382
G_loss: 0.1634
()
```

The following is the output of running CGAN using a condition label set to **4 (Coat)** after **80k** iteration and **7 (Sneaker)** after **60k** iteration:



Stabilizing training with Boundary

Equilibrium GAN

The popularity of GAN is rising rapidly among machine learning researchers. GAN researches can be categorized into two types: one that applies GAN into challenging problems and one that attempts to stabilize the training. Stabilizing GAN training is very crucial as the original GAN architecture suffers and has several shortcomings:

- **Mode collapse**: Where generators collapse into very narrow distribution and the samples generated are not diverse. This problem of course violates the spirit of GAN.
- **Evaluation of convergence metric**: There is no well-defined metric that tells us about the convergence between discriminator loss and generator loss.

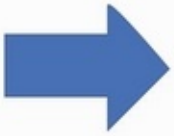
The improved **Wasserstein GAN** (*arXiv: 1704.00028,2017*) is a newly proposed GAN algorithm that promises to solve the preceding problems by minimizing the Wasserstein distance (or Earth-Mover distance) by providing simple gradients to the networks (+1 if the output is considered real and -1 if the output is considered fake).

The main idea behind the **BEGAN** (*arXiv: 1703.10717,2017*) is to have a new `loss` function by using **auto-encoder** as a discriminator, where the real loss is derived from the Wasserstein distance (to cater the problem of mode collapse) between the reconstruction losses of real and generated images:

$$\mathcal{L}(v) = |v - D(v)|^{\eta} \text{ where } \begin{cases} D : \mathbb{R}^{N_x} \mapsto \mathbb{R}^{N_x} & \text{is the autoencoder function.} \\ \eta \in \{1, 2\} & \text{is the target norm.} \\ v \in \mathbb{R}^{N_x} & \text{is a sample of dimension } N_x. \end{cases}$$

A hyper-parameter gamma is added through the use of a weighting parameter *k* to give users the power to control the desired diversity:

$$\begin{cases} \mathcal{L}_D = \mathcal{L}(x) - k_t \mathcal{L}(G(z_D)) & \text{for } \theta_D \\ \mathcal{L}_G = \mathcal{L}(G(z_G)) & \text{for } \theta_G \\ k_{t+1} = k_t + \lambda_x \left(\gamma \mathcal{L}(x) - \mathcal{L}(G(z_G))\right) & \text{for each training step } t \end{cases}$$

Where Gamma  $\gamma = \dfrac{\mathbb{E}\left[\mathcal{L}(G(z))\right]}{\mathbb{E}\left[\mathcal{L}(x)\right]}$

Unlike most GANs where discriminator and the generator are trained alternatively, BEGAN allows simultaneous training of both the networks in an adversarial way at each time step:

$$\arg\min_{\theta_D} \mathcal{L}_D + \arg\min_{\theta_G} \mathcal{L}_G$$

Finally, it allows an approximate measure of convergence *M* to understand the performance of the whole network:

$$\mathcal{M}_{global} = \mathcal{L}(x) + \left|\gamma \mathcal{L}(x) - \mathcal{L}(G(z_G))\right|$$

The training procedure of BEGAN

Steps involved in training BEGAN are described as follows:

1. The discriminator (the autoencoder) updates its weights to minimize the reconstruction loss of real images and in that way, starts to reconstruct real images better.
2. Simultaneously, the discriminator starts to maximize the reconstruction loss of generated images.
3. The generator works in an adversarial way to minimize the reconstruction loss of generated images.

Architecture of BEGAN

As shown in the following figure, the discriminator is a convolutional network with both deep encoder and decoder. The decoder has multiple layers of 3x3 convolution followed by an **Exponential Linear Unit** (**ELU**). Downsampling is done with stride 2 convolutions. The embedding state of the autoencoder is mapped to fully connected layers. Both the generator and the decoder are deep deconvolution with identical architectures, but with different weights, and the upsampling is done using nearest-neighbors:

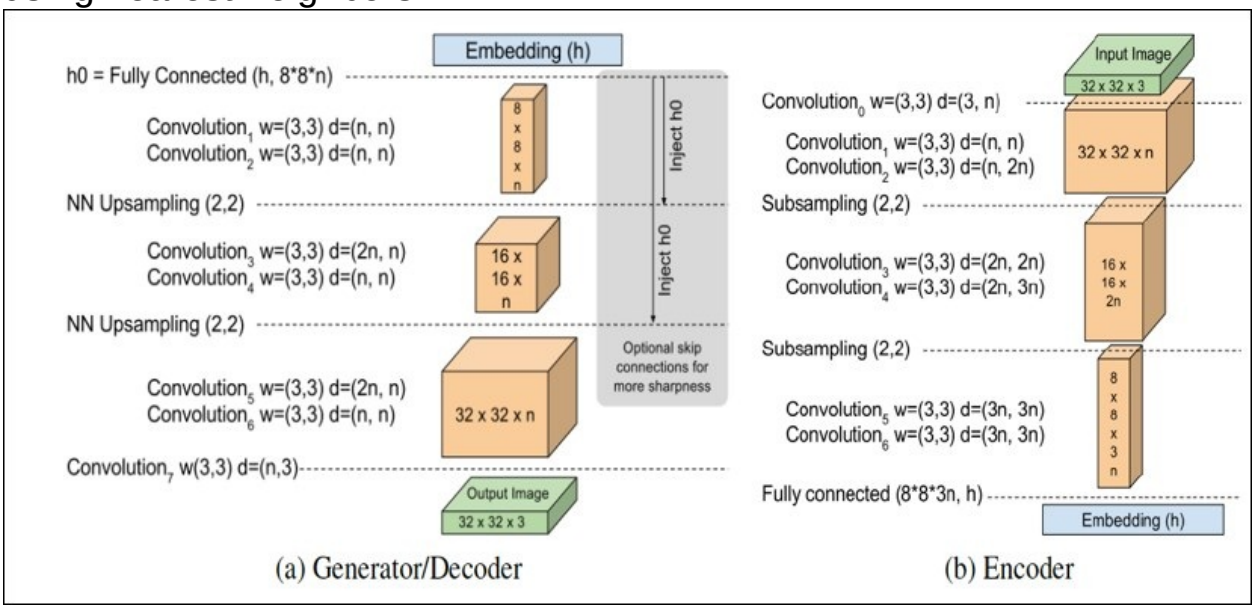


Figure-1: The architecture of the BEGAN.
Source: arXiv: 1703.10717,2017,2017

In the preceding figures, both the generator and the decoder of the discriminator is shown on the left-hand side. The encoder network of the discriminator is shown on the right-hand side.

Implementation of BEGAN using Tensorflow

Let us now dive deep into the code and implement the preceding concept along with the architecture to generate realistic attractive images.
The generator network has multiple layers of 3x3 convolution with an `elu activation` function, followed by nearest neighbor upscaling, except at the final layer. The number of convolution layers is calculated from the height of the image:

```
self.repeat_num = int(np.log2(height)) - 2.

def GeneratorCNN(z, hidden_num, output_num,
repeat_num, data_format, reuse):
    with tf.variable_scope("G", reuse=reuse) as vs:
        num_output = int(np.prod([8, 8, hidden_num]))
        x = slim.fully_connected(z, num_output,
activation_fn=None)
        x = reshape(x, 8, 8, hidden_num, data_format)

        for idx in range(repeat_num):
            x = slim.conv2d(x, hidden_num, 3, 1,
activation_fn=tf.nn.elu, data_format=data_format)
            x = slim.conv2d(x, hidden_num, 3, 1,
activation_fn=tf.nn.elu, data_format=data_format)
            if idx < repeat_num - 1:
                x = upscale(x, 2, data_format)

        out = slim.conv2d(x, 3, 3, 1,
activation_fn=None, data_format=data_format)

    variables = tf.contrib.framework.get_variables(vs)
    return out, variables
```

The encoder of the discriminator network has multiple layers of convolution with the `elu activation` function, followed by down-sampling using maxpooling except at the final convolution layer:

```
def DiscriminatorCNN(x, input_channel, z_num,
repeat_num, hidden_num, data_format):
    with tf.variable_scope("D") as vs:
        # Encoder
        x = slim.conv2d(x, hidden_num, 3, 1,
activation_fn=tf.nn.elu, data_format=data_format)

        prev_channel_num = hidden_num
        for idx in range(repeat_num):
```

```
            channel_num = hidden_num * (idx + 1)
            x = slim.conv2d(x, channel_num, 3, 1,
activation_fn=tf.nn.elu, data_format=data_format)
            x = slim.conv2d(x, channel_num, 3, 1,
activation_fn=tf.nn.elu, data_format=data_format)
            if idx < repeat_num - 1:
                x = slim.conv2d(x, channel_num, 3, 2,
activation_fn=tf.nn.elu, data_format=data_format)
                #x = tf.contrib.layers.max_pool2d(x,
[2, 2], [2, 2], padding='VALID')

        x = tf.reshape(x, [-1, np.prod([8, 8,
channel_num])])
        z = x = slim.fully_connected(x, z_num,
activation_fn=None)
```

The decoder of the discriminator network is similar to the generator network, having multiple layers of convolution with an `elu activation` function followed by upsampling using nearest neighbor except at the final convolution layer:

```
        num_output = int(np.prod([8, 8, hidden_num]))
        x = slim.fully_connected(x, num_output,
activation_fn=None)
        x = reshape(x, 8, 8, hidden_num, data_format)

        for idx in range(repeat_num):
            x = slim.conv2d(x, hidden_num, 3, 1,
activation_fn=tf.nn.elu, data_format=data_format)
            x = slim.conv2d(x, hidden_num, 3, 1,
activation_fn=tf.nn.elu, data_format=data_format)
            if idx < repeat_num - 1:
                x = upscale(x, 2, data_format)

        out = slim.conv2d(x, input_channel, 3, 1,
activation_fn=None, data_format=data_format)

    variables = tf.contrib.framework.get_variables(vs)
```

Now the generator loss and discriminator loss for both real, fake images discussed previously are optimized using **Adam Optimizer** by executing the following code block:

```
d_out, self.D_z, self.D_var = DiscriminatorCNN(
                tf.concat([G, x], 0), self.channel,
self.z_num, self.repeat_num,
                self.conv_hidden_num,
self.data_format)
        AE_G, AE_x = tf.split(d_out, 2)
```

```
        self.G = denorm_img(G, self.data_format)
        self.AE_G, self.AE_x = denorm_img(AE_G,
self.data_format), denorm_img(AE_x, self.data_format)

if self.optimizer == 'adam':
            optimizer = tf.train.AdamOptimizer
        else:
            raise Exception("[!] Caution! Paper didn't
use {} opimizer other than
Adam".format(config.optimizer))

        g_optimizer, d_optimizer =
optimizer(self.g_lr), optimizer(self.d_lr)

        self.d_loss_real = tf.reduce_mean(tf.abs(AE_x
- x))
        self.d_loss_fake = tf.reduce_mean(tf.abs(AE_G
- G))

        self.d_loss = self.d_loss_real - self.k_t *
self.d_loss_fake
        self.g_loss = tf.reduce_mean(tf.abs(AE_G - G))
```

Now it's time to execute the code for generating impressive celebrity images:

1. First clone the following repository and then change the directory:

```
git clone https://github.com/carpedm20/BEGAN-
tensorflow.git
cd BEGAN-tensorflow
```

2. Next, run the following scripts to download the `CelebA` dataset under the `data` directory, and split it into training, validation, and test set:

```
python download.py
```

3. Make sure p7zip is installed on your machine.
4. Now start the training process as follows, which will save the generated samples under the `logs` directory:

```
python main.py --dataset=CelebA --use_gpu=True
```

Note

If you face the error **Conv2DCustomBackpropInputOp only supports NHWC**, then refer to the following issue: https://github.com/carpedm20/BEGAN-tensorflow/ issues/29

After executing the preceding command, while the training is going on you will notice information such as `Model` directory, logging directory, and various losses as follows:

```
[ubuntu@ip-172-31-6-47:~/software/kuntalg/BEGAN-tensorflow$ python main.py --dataset=CelebA --use_g]
pu=True
2017-08-26 12:32:52.771534: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
rary wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could
 speed up CPU computations.
2017-08-26 12:32:52.771590: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
rary wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could
 speed up CPU computations.
2017-08-26 12:32:52.771600: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
rary wasn't compiled to use AVX instructions, but these are available on your machine and could sp
eed up CPU computations.
[*] MODEL dir: logs/CelebA_0826_123249
[*] PARAM path: logs/CelebA_0826_123249/params.json
  0%|                                                                    | 0/500000 [00:00<?, ?it/s]
[0/500000] Loss_D: 0.460274 Loss_G: 0.052054 measure: 0.6384, k_t: 0.0002
[*] Samples saved: logs/CelebA_0826_123249/0_G.png
[*] Samples saved: logs/CelebA_0826_123249/0_D_real.png
[*] Samples saved: logs/CelebA_0826_123249/0_D_fake.png
  0%|                                                  | 50/500000 [06:34<1073:17:56,  7.73s/it]
[50/500000] Loss_D: 0.254775 Loss_G: 0.043506 measure: 0.3390, k_t: 0.0049
  0%|                                                  | 99/500000 [12:57<1084:16:39,  7.81s/it]
```

The output faces generated by BEGAN are visually realistic and attractive as shown in the following screenshot:

Figure-2: Generator output images (64x64) with gamma=0.5 after 350k steps

The following sample output images (128 x 128) are generated after 250k steps:

Figure-3: Generator output images (128x128) with gamma=0.5 after 250k steps

Image to image style transfer with CycleGAN

The **Cycle Consistent Generative Network** (**CycleGAN**), originally proposed in the paper *Unpaired image-to-image translation using CycleGAN—arXiv: 1703.10593, 2017*, aims at finding mapping between the source domain and a target domain for a given image without any pairing information (such as greyscale to color, image to semantic labels, edge-map to photograph, horse to zebra, and so on).
The key idea behind CycleGAN is to have two translator's F and G, where F will translate an image from domain *A* to domain *B*, and *G* will translate an image from domain *B* to domain *A*. So, for an image *x* in domain *A*, we should expect the function *G(F(x))* to be equivalent to *x* and similarly for an image *y* in domain *B*, we should expect the function *F(G(y))* to be equivalent to *y*.

Model formulation of CycleGAN

The main goal of the CycleGAN model is to learn mapping between the two domains *X* and *Y* using the training samples *{xi}Ni=1* ∈ *X* and *{yj}Mj=1* ∈ *Y*. It also has two adversarial discriminators D_X and D_Y: where D_X tries to distinguish between original images *{x}* and translated images *{F(y)}*, and similarly, D_Y tries to distinguish between *{y}* and *{G(x)}*.
CycleGAN model has two `loss` functions:

- **Adversarial loss**: It matches the generated image's distribution to the target domain distribution:

$$\mathcal{L}_{GAN}\left(G, D_Y, X, Y\right) = \mathbb{E}_{y \sim p_{data}(y)}\left[\log D_Y\left(y\right)\right] + \mathbb{E}_{x \sim p_{data}(x)}\left[\log\left(1 - D_Y\left(G\left(x\right)\right)\right)\right]$$

- **Cycle consistency loss**: It prevents the learned mappings *G* and *F* from contradicting each other:

$$\mathcal{L}_{cyc}\left(G, F\right) = \mathbb{E}_{x \sim p_{data}(x)}\left[\left\|F\left(G\left(x\right)\right) - x\right\|1\right] + \mathbb{E}_{y \sim p_{data}(y)}\left[\left\|G\left(F\left(y\right)\right) - y\right\|1\right]$$

The full CycleGAN objective function is given by:

$$G^*, F^* = \arg\min_{F,G} \max_{D_X, D_Y} \mathcal{L}(G, F, D_X, D_Y)$$

where

$$\begin{aligned}\mathcal{L}(G, F, D_X, D_Y) = {} & \mathcal{L}_{GAN}(G, D_Y, X, Y) \\ & + \mathcal{L}_{GAN}(F, D_X, Y, X) \\ & + \lambda \mathcal{L}_{cyc}(G, F)\end{aligned}$$

Transforming apples into oranges using Tensorflow

In this example, we will transfer the style from an image in domain *A* to an image in another domain *B*: more specifically, we will apply CycleGAN to transform apples into oranges or vice-versa by executing the following steps:

1. First clone the following `git` repository and change the directory to CycleGAN-tensorflow:

```
git clone https://github.com/xhujoy/CycleGAN-tensorflow
cd CycleGAN-tensorflow
```

2. Now download the `apple2orange` dataset ZIP file using the `download_dataset.sh` script, extract and save it under the `datasets` directory:

```
bash ./download_dataset.sh apple2orange
```

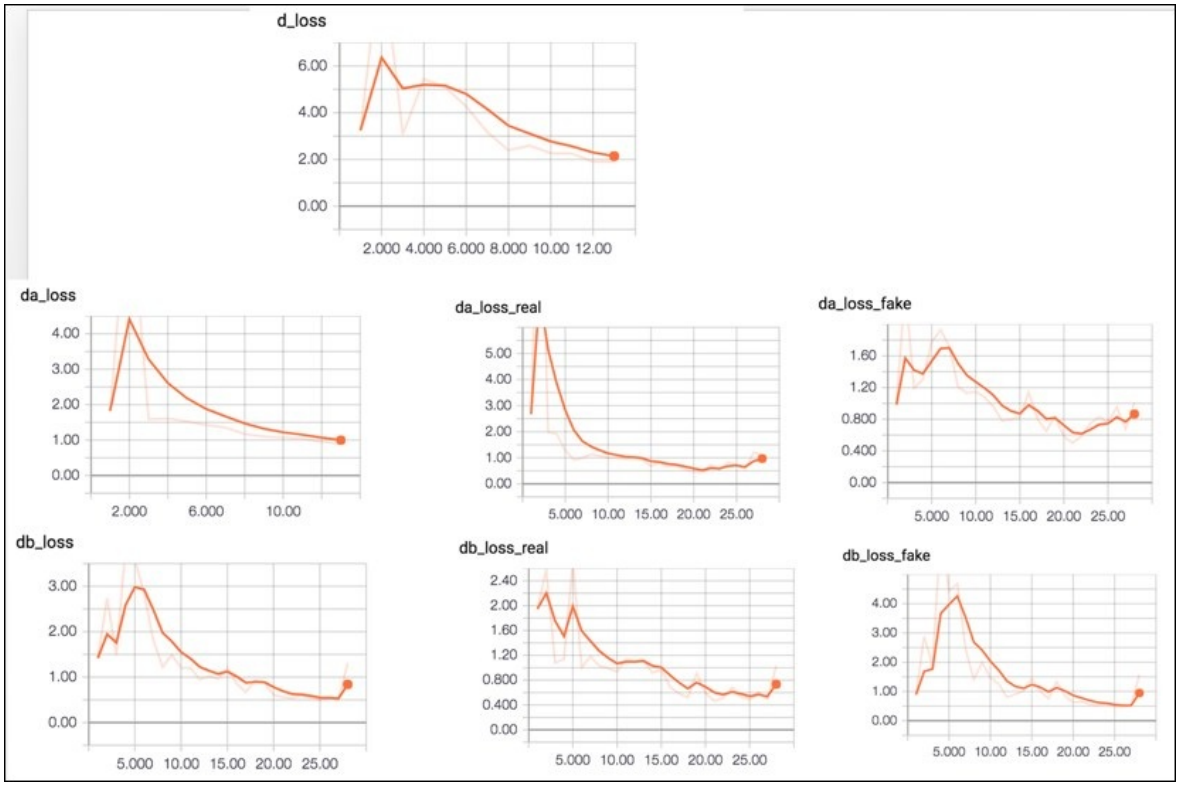
3. Next train the CycleGAN model using the downloaded `apple2orange` dataset. During the training phase, the model will be saved in the `checkpoint` directory and logging is enabled in the `logs` directory for visualization with TensorBoard:

```
python main.py --dataset_dir=apple2orange
```

```
[ubuntu@ip-172-31-6-47:~/software/kuntalg/CycleGAN-tensorflow$ python main.py --dataset_dir=apple2orange
--phase=test --which_direction=AtoB
2017-08-27 10:26:41.727036: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 10:26:41.727083: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 10:26:41.727101: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your machine and could speed up CPU
 computations.
generatorA2B/g_e1_c/Conv/weights:0
generatorA2B/g_e1_bn/scale:0
generatorA2B/g_e1_bn/offset:0
generatorA2B/g_e2_c/Conv/weights:0
generatorA2B/g_e2_bn/scale:0
```

4. Run the following command to visualize various losses (discriminator loss and generator loss) during the training phase in your browser, by navigating to `http://localhost:6006/`:

```
tensorboard --logdir=./logs
```



5. Finally, we will load the trained model from the `checkpoint` directory to transfer a style across images, hence generating oranges from apple or vice-versa (based on the value passed (`AtoB` or `BtoA`) to the `which_direction` parameter that indicates a style transfer from domain 1 to domain 2):

```
python main.py --dataset_dir=apple2orange --
phase=test --which_direction=AtoB
```

6. The following are the sample output images generated in the `test`

phase:

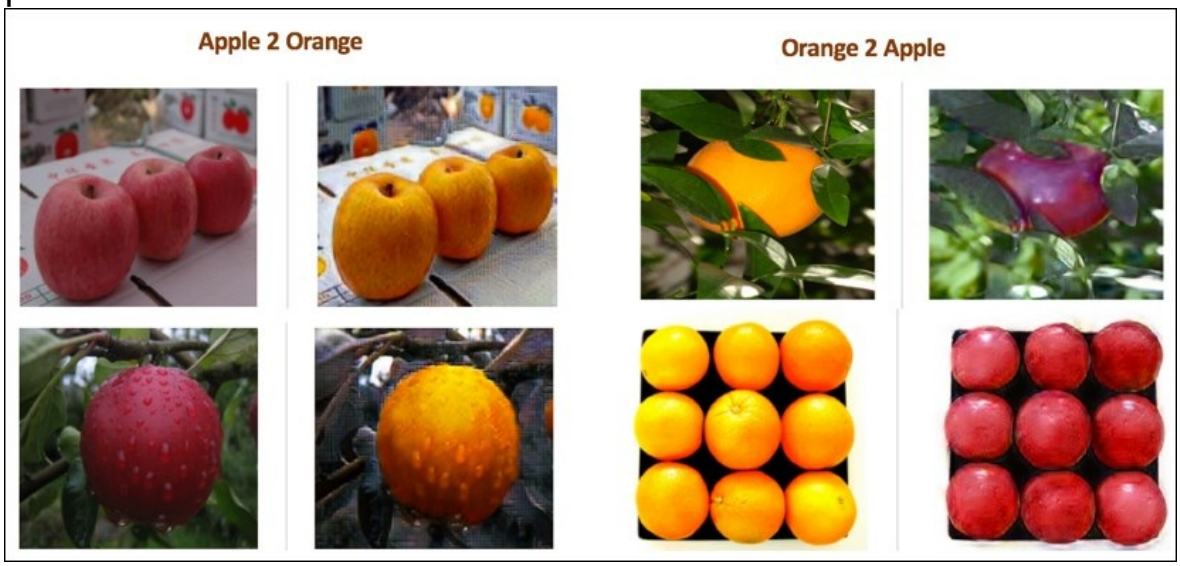


Figure- 4: The left-hand side shows transforming apples to oranges by passing AtoB in the direction parameter, whereas the right-hand side shows the output generated by passing BtoA in the direction parameter.

Transfiguration of a horse into a zebra with CycleGAN

Just like the previous example, in this section we will use CycleGAN to transform a horse into a zebra or vice-versa by executing the following steps:

1. First clone the following `git` repository and change the directory to `CycleGAN-tensorflow` (you can skip this step if you have already executed the previous example):

```
git clone https://github.com/xhujoy/CycleGAN-tensorflow
cd CycleGAN-tensorflow
```

2. Now download the `horse2zebra` ZIP file from Berkley, extract it, and save it under the `datasets` directory using the `download_dataset.sh` script:

```
bash ./download_dataset.sh horse2zebra
```

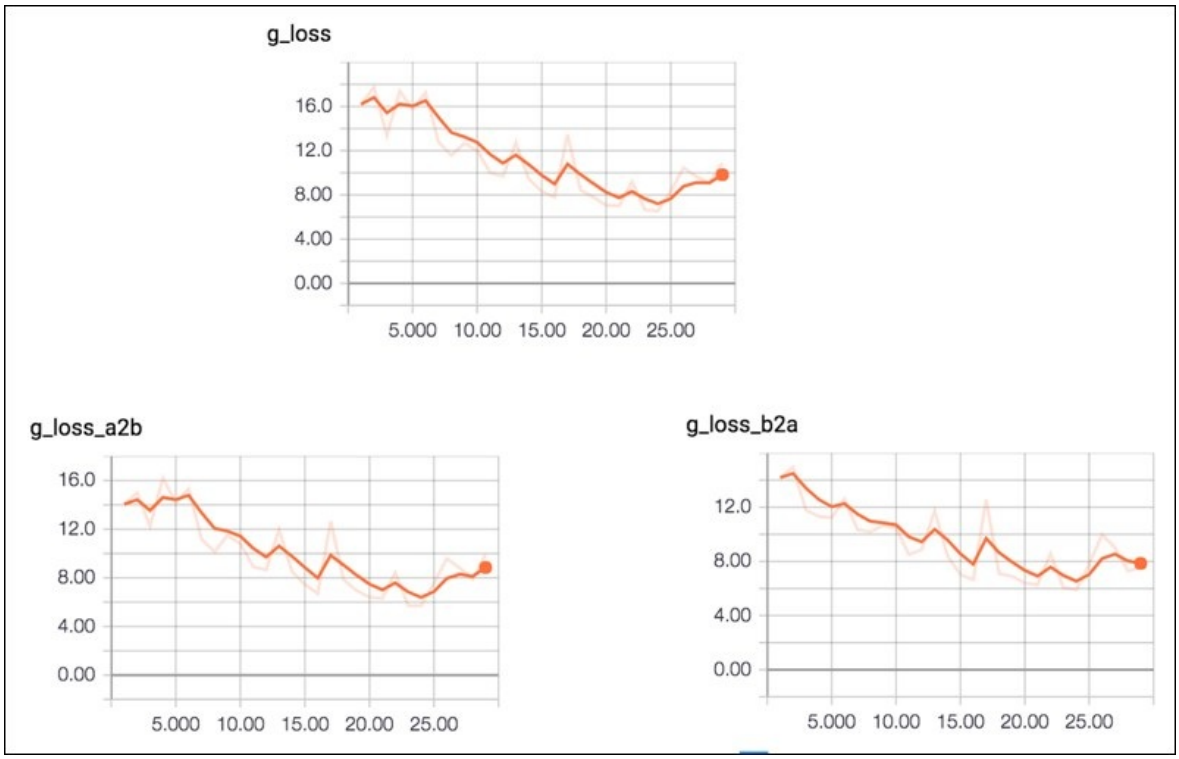
3. Next, we will train our CycleGAN model using the `horse2zebra` dataset and use TensorBoard for visualizing the losses while training is going on:

```
python main.py --dataset_dir=horse2zebra
```

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/CycleGAN-tensorflow$ python main.py --dataset_dir=horse2zebra
2017-08-27 06:12:07.756238: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 06:12:07.756280: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up
CPU computations.
2017-08-27 06:12:07.756299: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your machine and could speed up CPU
 computations.
generatorA2B/g_e1_c/Conv/weights:0
generatorA2B/g_e1_bn/scale:0
generatorA2B/g_e1_bn/offset:0
generatorA2B/g_e2_c/Conv/weights:0
generatorA2B/g_e2_bn/scale:0
generatorA2B/g_e2_bn/offset:0
generatorA2B/g_e3_c/Conv/weights:0
```

4. Run the following command and navigate to `http://localhost:6006/` for the visualizing of various generator or discriminator losses:

```
tensorboard --logdir=./logs
```



5. Finally, we will use the trained model from the `checkpoint` directory to transform a horse into a zebra or vice-versa, depending on whether the value `AtoB` or `BtoA` is passed to the `which_direction` parameter:

```
python main.py --dataset_dir=horse2zebra --
phase=test --which_direction=AtoB
```

The following sample output images are generated in the `test` phase:

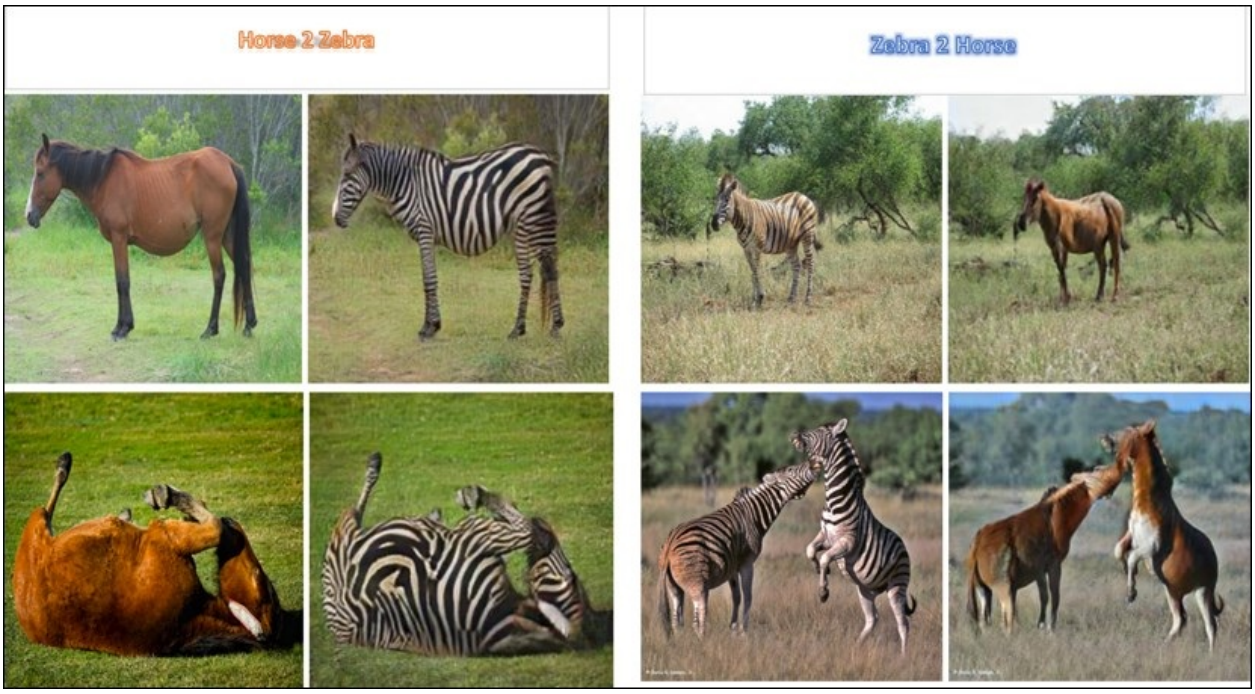


Figure-5: The left-hand side shows transforming horse to zebra, whereas the right-hand side shows translating zebra into horse.

Summary

So far you have learned the approach of creating images based on certain characteristics or conditions, by passing that condition vector into both generator and discriminator. Also, you have understood how to overcome model collapse problems by stabilizing your network training using BEGAN. Finally, you have implemented image to image style transfer by generating an orange from an apple and a zebra from a horse, or vice-versa, using CycleGAN. In the next chapter, we will solve complex real-life problems such as text to image synthesis and cross domain discovery by stacking or coupling two or more GAN models together.

Chapter 4. Building Realistic Images from Your Text

For many real-life complex problems, a single Generative Adversarial Network may not be sufficient to solve it. Instead it's better to decompose the complex problem into multiple simpler sub-problems and use multiple GANs to work on each sub-problem separately. Finally, you can stack or couple the GANs together to find a solution.

In this chapter, we will first learn the technique of stacking multiple generative networks to generate realistic images from textual information. Next, you will couple two generative networks, to automatically discover relationships among various domains (relationships between shoes and handbags or actors and actresses).

We will cover the following topics in this chapter:

- What is StackGAN? Its concept and architecture
- Synthesizing realistic images from text description using TensorFlow
- Discovering cross-domain relationships with DiscoGAN
- Generating handbag images from edges using PyTorch
- Transforming gender (actor-to-actress or vice-versa) with facescrub data

Introduction to StackGAN

The idea of StackGAN was originally proposed by *Han Zhang*, *Tao Xu*, *Hongsheng Li*, *Shaoting Zhang*, *Xiaolei Huang*, *Xiaogang Wang*, and *Dimitris Metaxas* [*arXiv: 1612.03242,2017*] in the paper *Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks*, where GAN has been used to synthesize forged images starting from the text description.

Synthesizing photo realistic images from text is a challenging problem in Computer Vision and has tremendous practical application. The problem of generating images from text can be decomposed into two manageable sub-problems using StackGAN. In this approach, we stack two stages of the generative network based on certain conditions (such as textual description and the output of the previous stage) to achieve this challenging task of realistic image generation from text input.

Let us define some concepts and notation before diving into the model architecture and implementation details:

- *Io*: This is the original image
- *t*: Text description
- *t*: Text embedding
- *μ(t)*: Mean of text embedding
- *∑(t)*: Diagonal covariance matrix of text embedding
- *pdata*: Real data distribution
- *pz*: Gaussian distribution of noise
- *z*: Randomly sampled noise from Gaussian distribution

Conditional augmentation

As we already know from Chapter 2, *Unsupervised Learning with GAN*, in Conditional GAN both the generator and discriminator network receive additional conditioning variables *c* to yield *G(z;c)* and *D(x;c)*. This formulation helps the generator to generate images conditioned on variable *c*. The conditioning augmentation yields more training pairs given a small number of image-text pairs and is useful for modeling text to image translation as the same sentence usually maps to objects with various appearances. The textual description is first converted to text embedding *t* by encoding through an encoder and then transformed nonlinearly using a char-CNN-RNN model to create conditioning latent variables as the input of a stage-I generator network.
Since the latent space for text embedding is usually high dimensional, to mitigate the problem of discontinuity in latent data manifold with a limited amount of data, a conditioning augmentation technique is applied to produce additional conditioning variable *c^* sampled from a Gaussian distribution *N(μ(t), ∑(t))*.

Stage-I

In this stage, the GAN network learns the following:

- Generating rough shapes and basic colors for creating objects conditioned on textual description
- Generating background regions from random noise sampled from prior distribution

The low resolution coarse images generated in this stage might not look real because they have some defects such as object shape distortion, missing object parts, and so on.
The stage-I GAN trains the discriminator *D0* (maximize the loss) and the generator *G0* (minimize the loss), alternatively as shown in the following equation:

Discriminator Loss

$$\mathcal{L}_{D_0} = \mathbb{E}_{(I_0,t)\sim p_{data}}\left[\log D_0\left(I_0,\varphi_t\right)\right] + \mathbb{E}_{z\sim p_z,t\sim p_{data}}\left[\log\left(1-D_0\left(G_0\left(z,\hat{c}_0\right),\varphi_t\right)\right)\right]$$

Generator Loss

$$\mathcal{L}_{G_0} = \mathbb{E}_{z\sim p_z,t\sim p_{data}}\left[\log\left(1-D_0\left(G_0\left(z,\hat{c}_0\right),\varphi_t\right)\right)\right] + \lambda D_{KL}\left(N\left(\mu_0\left(\varphi_t\right),\Sigma_0\left(\varphi_t\right)\right)\middle\|N\left(0,1\right)\right)$$

Kullback-Leibler divergence between standard and conditioned Gaussian distribution as a regularized term.

Stage-II

In this stage, the GAN network only focuses on drawing details and rectifying defects in low resolution images generated from stage-I (such as a lack of vivid object parts, shape distortion, and some omitted details from the text) to generate high resolution realistic images conditioned on textual description.

The stage-II GAN alternatively trains the discriminator *D* (maximize the loss) and generator *G* (minimize the loss), conditioned on the result of low resolution $G_0(z; c\hat{}0)$ and the Gaussian latent variable *c*^:

Discriminator Loss

$$\mathcal{L}_D = \mathbb{E}_{(I,t)\sim p_{data}}\left[\log D\left(I,\varphi_t\right)\right] + \mathbb{E}_{s_0\sim p_{G_0},t\sim p_{data}}\left[\log\left(1-D\left(G\left(s_0,\hat{c}\right),\varphi_t\right)\right)\right]$$

Generator Loss

$$\mathcal{L}_G = \mathbb{E}_{s_0\sim p_{G_0},t\sim p_{data}}\left[\log\left(1-D\left(G\left(s_0,\hat{c}\right),\varphi_t\right)\right)\right] + \lambda D_{KL}\left(N\left(\mu\left(\varphi_t\right),\Sigma\left(\varphi_t\right)\right)\middle\|N\left(0,1\right)\right)$$

Note

Random noise *z* is replaced with Gaussian conditioning variables *c^* in stage-II. Also, the conditioning augmentation in stage-II has different fully connected layers to generate different means and standard deviation of the text embedding.

Architecture details of StackGAN

As illustrated in the following figure, for the generator network G_0 of stage-I, the text embedding *t* is first fed into a fully connected layer to generate *μ0* and *σ0* (*σ0* is the diagonal values of *∑0*) for the Gaussian distribution *N(μ0(t); ∑0(t))* and then the text conditioning variable *c^0* is then sampled from the Gaussian distribution.

For the discriminator network *D0* of stage-I, the text embedding *t* is first compressed to *Nd* dimensions with a fully connected layer and then spatially replicated to *Md* x *Md* x *Nd* tensor. The image is passed through a series of down-sampling blocks to squeeze into *Md* x *Md* spatial dimension and then concatenated using a filter map along the channel dimension with the text tensor. The resulting tensor goes through a 1x1 convolutional layer to jointly learn features across the image and the text and finally output the decision score using one node fully connected layer.

The generator of stage-II is designed as an encoder-decoder network with residual blocks and the text embedding *t* to generate the *Ng* dimensional text conditioning vector *c^*, which is spatially replicated to *Md* x *Md* x *Nd* tensor. The stage-I result *s0* generated is then fed into several down-sampling blocks (that is, encoder) until it is squeezed to spatial size *Mg* x *Mg*. The image features concatenated with text features along the channel dimension are passed through several residual blocks, to learn multi-modal representations across image and text features. Finally, the resulting tensors goes through a series of up-sampling layers (that is, decoder) to generate a *W* x *H* high resolution image.

The discriminator of stage-II is similar to stage-1 with only extra down-sampling blocks to cater for the large image size in this stage. During training of the discriminator, the positive sample pairs is built from the real images and their corresponding text descriptions, whereas the negative sample consists of two groups: one having real images with mismatched text embedding and the other having synthetic images with their corresponding text embedding:

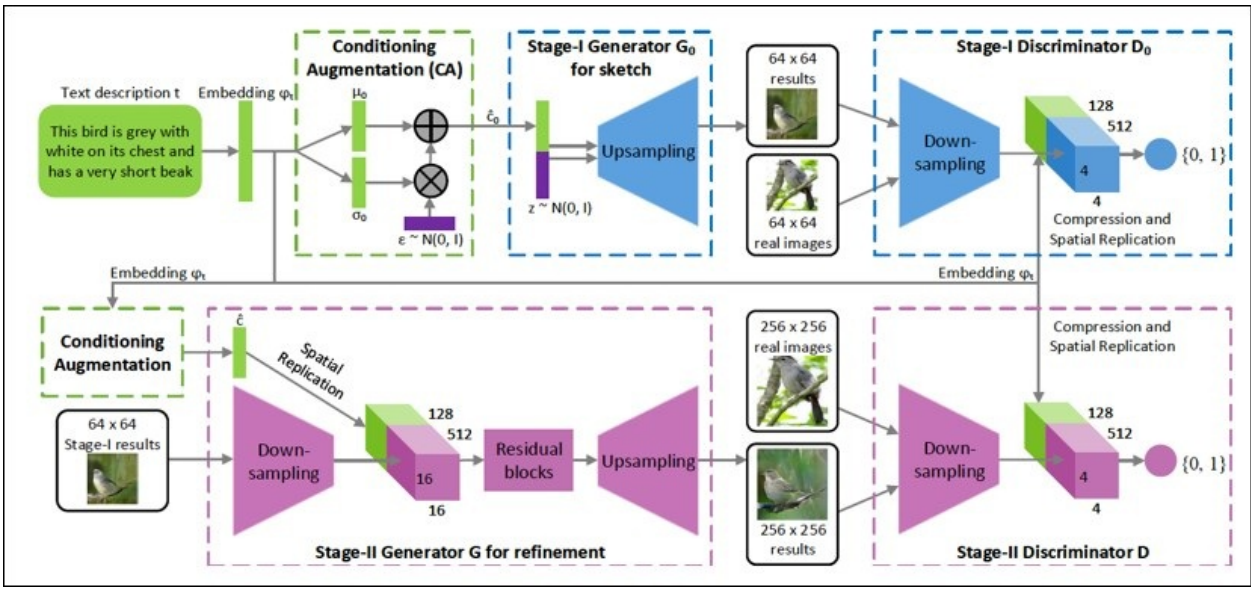


Figure 1. The architecture of the StackGAN.
Source: arXiv: 1612.03242,2017

The stage-I generator first draws a low resolution image by sketching a rough shape and basic colors of the object from the given text and painting the background from a random noise vector. The stage-II generator corrects defects and adds compelling details into stage-I results, yielding a more realistic high resolution image conditioned on stage-I results.

The up-sampling blocks consist of the nearest-neighbor up-sampling followed by the 33 convolutions, each of stride of 1. Batch normalization and `ReLU` activation functions are applied after every convolution except the last one. The residual blocks again consist of 33 convolutions, each of stride 1, followed by batch normalization and `ReLU` activation function. The down-sampling blocks consist of 44 convolutions each of stride 2, followed by batch normalization and Leaky-ReLU, except batch normalization is not present in the first convolution layer.

Synthesizing images from text with TensorFlow

Let us implement the code to synthesize realistic images from text and produce mind blowing result:

1. First clone the `git` repository: https://github.com/Kuntal-G/StackGAN.git and change the directory to `StackGAN`:

```
git clone https://github.com/Kuntal-G/StackGAN.git
cd StackGAN
```

> **Note**
> Currently the code is compatible with an older version of TensorFlow (0.11), so you need to have TensorFlow version below 1.0 to successfully run this code. You can modify your TensorFlow version using: `sudo pip install tensorflow==0.12.0`. Also make sure torch is installed in your system. More information can be found here: http://torch.ch/docs/getting-started.html.

2. Then install the following packages using the `pip` command:

```
sudo pip install prettytensor progressbar python-
dateutil easydict pandas torchfile requests
```

3. Next download the pre-processed char-CNN-RNN text embedding birds model from: https://drive.google.com/file/d/0B3y_msrWZaXLT1BZdVdycDY5TEE/ using the following command:

```
python google-drive-download.py
0B3y_msrWZaXLT1BZdVdycDY5TEE Data/ birds.zip
```

4. Now extract the downloaded file using the `unzip` command:

```
unzip Data/birds.zip
```

5. Next download and extract the birds image data from Caltech-UCSD:

```
wget http://www.vision.caltech.edu/visipedia-
data/CUB-200-2011/
CUB_200_2011.tgz -O Data/birds/CUB_200_2011.tgz
tar -xzf CUB_200_2011.tgz
```

6. Now we will do preprocessing on the images to split into training and test sets and save the images in pickle format:

```
python misc/preprocess_birds.py
```

```
ubuntu@ip-172-31-1-246:~/software/kuntalg/StackGAN$ python misc/preprocess_birds.py
Total filenames:  11788 001.Black_footed_Albatross/Black_Footed_Albatross_0046_18.jpg
Load filenames from: Data/birds/train/filenames.pickle (8855)
Load 100......
Load 200......
Load 300......
Load 400......
Load 500......
Load 600......
Load 700......
Load 800......
Load 900......
Load 1000......

Load 8300......
Load 8400......
Load 8500......
Load 8600......
Load 8700......
Load 8800......
images 8855 (304, 304, 3) (76, 76, 3)
save to:  Data/birds/train/304images.pickle
save to:  Data/birds/train/76images.pickle
Load filenames from: Data/birds/test/filenames.pickle (2933)
Load 100......
Load 200......
Load 300......
Load 400......
Load 500......
Load 600......

Load 2800......
Load 2900......
images 2933 (304, 304, 3) (76, 76, 3)
save to:  Data/birds/test/304images.pickle
save to:  Data/birds/test/76images.pickle
```

7. Now we will download the pre-trained char-CNN-RNN text embedding model from: https://drive.google.com/file/d/0B3y_msrWZaXLNUNKa3BaRjAyTzQ/ and save it to the `models/` directory using:

```
python google-drive-download.py
0B3y_msrWZaXLNUNKa3BaRjAyTzQ models/
birds_model_164000.ckpt
```

8. Also download the char-CNN-RNN text encoder for birds from https://drive.google.com/file/d/0B0ywwgffWnLLU0F3UHA3NzFTNEE/ and save it under the `models/text_encoder` directory:

```
python google-drive-download.py
0B0ywwgffWnLLU0F3UHA3NzFTNEE models/text_encoder/
lm_sje_nc4_cub_hybrid_gru18_a1_c512_0.00070_1_10_t
rainvalids.txt_iter30000.t7
```

9. Next, we will add some sentences to the `example_captions.txt` file to generate some exciting images of birds:

```
A white bird with a black crown and red beak
this bird has red breast and yellow belly
```

10. Finally, we will execute the `birds_demo.sh` file under the `demo` directory to generate realistic bird images from the text description given in the `example_captions.txt` file:

```
sh demo/birds_demo.sh
```

```
[ubuntu@ip-172-31-6-47:~/software/kuntalg/StackGAN$ sh demo/birds_demo.sh
{
  doc_length : 201
  filenames : "Data/birds/example_captions.t7"
  queries : "Data/birds/example_captions.txt"
  net_txt : "models/text_encoder/lm_sje_nc4_cub_hybrid_gru18_a1_c512_0.00070_1_10_trainvalids.txt_iter30000.t7"
}
```

11. Now the generated images will be saved under the `Data/birds/example_captions/` directory as shown in the following screenshot:

Voila, you have now generated impressive bird images from the textual description. Play with your own sentences to describe birds and visually verify the results with the description.

Discovering cross-domain relationships with DiscoGAN

Cross-domain relationships are often natural to humans and they can easily identify the relationship between data from various domains without supervision (for example, recognizing relationships between an English sentence and its translated sentence in Spanish or choosing a shoe to fit the style of a dress), but learning this relation automatically is very challenging and requires a lot of ground truth pairing information that illustrates the relations.

Discovery Generative Adversarial Networks (**DiscoGAN**) *arXiv: 1703.05192 ,2017* discovers the relationship between two visual domains and successfully transfers styles from one domain to another by generating new images of one domain given an image from the other domain without any pairing information. DiscoGAN seeks to have two GANs coupled together that can map each domain to its counterpart domain. The key idea behind DiscoGAN is to make sure that all images in domain 1 are representable by images in domain 2, and use the reconstruction loss to measure how well the original image is reconstructed after the two translations—that is, from domain 1 to domain 2 and back to domain 1.

The architecture and model formulation of DiscoGAN

Before diving into the model formulation and various `loss` functions associated with DiscoGAN, let us first define some related terminology and concepts:

- G_{AB}: The `generator` function that translates input image x_A from domain A into image x_{AB} in domain B
- G_{BA}: The `generator` function that translates input image x_B from domain B into image x_{BA} in domain A
- $G_{AB}(x_A)$: This is the complete set of all possible resulting values for all x_As in domain A that should be contained in domain B
- $G_{BA}(x_B)$: This is the complete set of all possible resulting values for all x_Bs in domain B, that should be contained in domain A
- D_A: The `discriminator` function in domain A

- D_B: The `discriminator` function in domain B

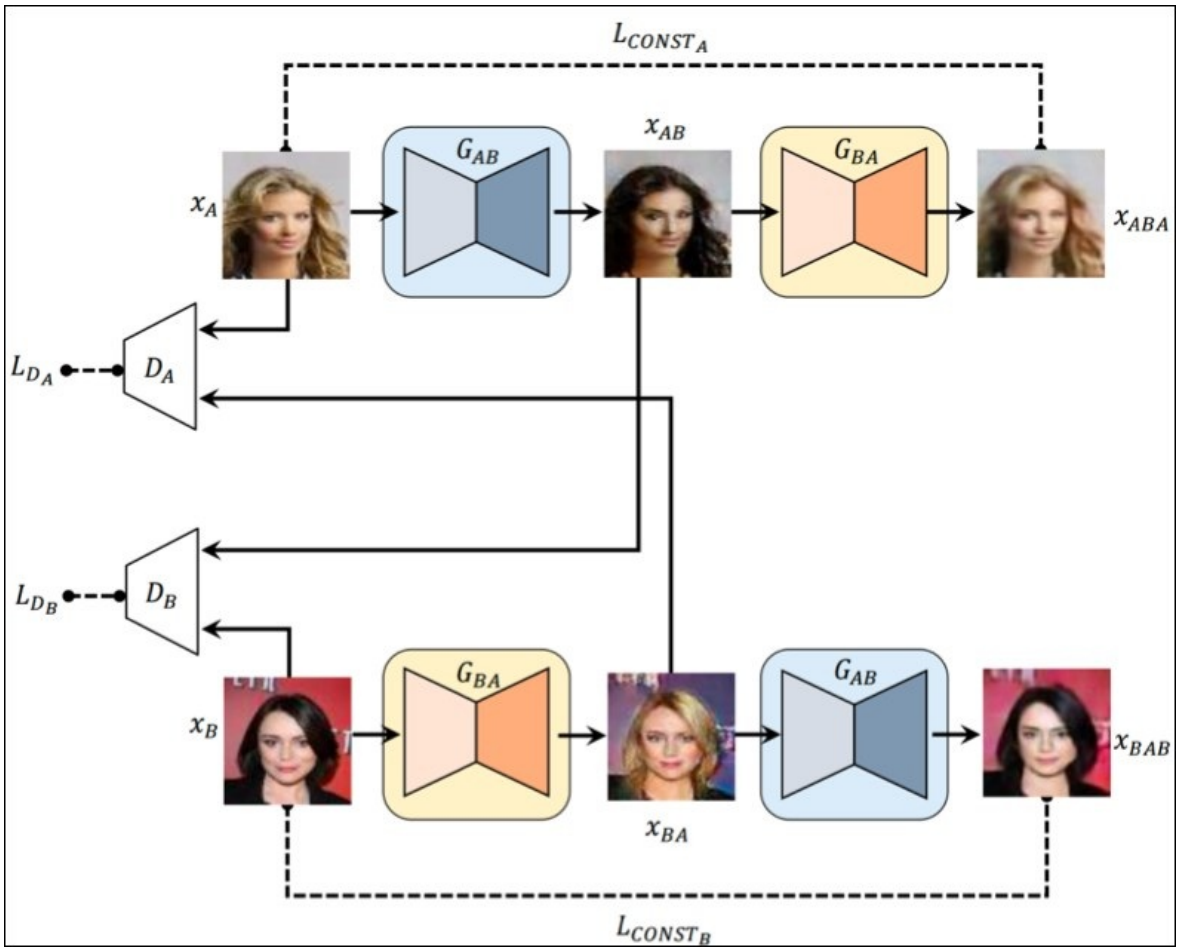


Figure-2: DiscoGAN architecture with two coupled GAN models
Source: arXiv- 1703.05192, 2017

The generator modules of DiscoGAN consist of an encoder-decoder pair to perform back to back image translation. A generator G_{AB} first translates input image x_A from domain A into the image x_{AB} in domain B. Then the generated image is translated back to domain A image x_{ABA} to match the original input image using reconstruction loss (equation-3) with some form of distance metrics, such as MSE, Cosine distance, and hinge-loss. Finally, the translated output image x_{AB} of the generator is fed into the discriminator and gets scored by comparing it to the real image of domain B:

$$x_{AB} = \mathbf{G}_{AB}\left(x_A\right) \quad (1)$$

$$x_{ABA} = \mathbf{G}_{BA}\left(x_{AB}\right) = \mathrm{G}_{BA} \circ \mathrm{G}_{AB}\left(x_A\right) \quad (2)$$

$$L_{CONST_A = d\left(\mathbf{G}_{BA} \circ \mathbf{G}_{AB}\left(x_A\right), x_A\right)} \quad (3)$$

$$L_{GAN_B} = -\mathbb{E}_{x_A \sim P_A}\left[\log \mathbf{D}_B\left(\mathbf{G}_{AB}\left(x_A\right)\right)\right] \quad (4)$$

The generator *GAB* receives two types of losses as shown (equation-5):

- L_{CONSTA}: A reconstruction loss that measures how well the original image is reconstructed after the two translations domain A-> domain B-> domain A
- L_{GANB}: Standard GAN loss that measures how realistic the generated image is in domain B

Whereas the discriminator D_B receives the standard GAN discriminator loss as shown (equation-6):

$$L_{G_{AB}} = L_{GAN_B} + L_{CONST_A} \quad (5)$$

$$L_{D_B} = -\mathbb{E}_{x_B \sim P_B}\left[\log \mathbf{D}_B\left(x_B\right)\right] - \mathbb{E}_{x_A \sim P_A}\left[\log\left(1 - \mathbf{D}_B\left(\mathbf{G}_{AB}\left(x_A\right)\right)\right)\right] \quad (6)$$

The two coupled GANs are trained simultaneously and both the GANs learn mapping from one domain to another along with reverse mapping for reconstruction of the input images from both domains using two reconstruction losses: L_{CONSTA} and L_{CONSTB}.

The parameters are shared between generators G_{AB} and G_{BA} of two GANs and the generated images x_{BA} and x_{AB} are then fed into the separate discriminators L_{DA} and L_{DB} respectively:

$$L_G = L_{G_{AB}} + L_{G_{BA}} = L_{GAN_B} + L_{CONST_A} + L_{GAN_A} + L_{CONST_B} \quad (7)$$

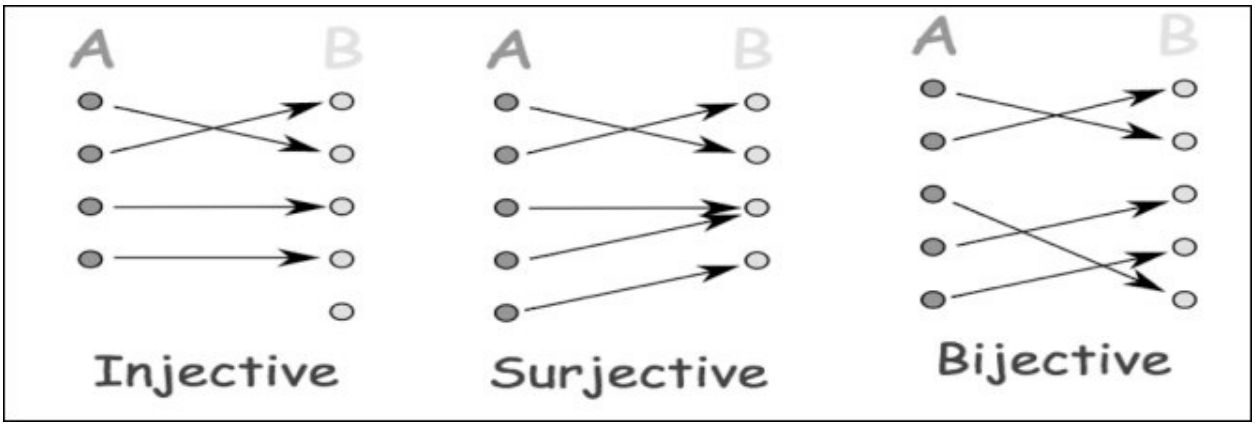
$$L_D = L_{D_A} + L_{D_B} \quad (8)$$

The total generator loss, L_G, is the sum of two GAN losses of the coupled model and reconstruction loss of each partial model as shown (equation-7). And the total discriminator loss, L_D, is the sum of the two

discriminators losses L_{DA} and L_{DB}, which discriminate real and fake images in domain A and domain B, respectively (equation- 8). In order to achieve bijective mapping having one-to-one correspondence, the DiscoGAN model is constrained by two L_{GAN} losses and two L_{CONST} reconstruction losses.

Injective mapping means that every member of **A** has its own unique matching member in **B** and surjective mapping means that every **B** has at least one matching **A**.

Bijective mapping means both injective and surjective are together and there is a perfect one-to-one correspondence between the members of the sets:



Implementation of DiscoGAN

Let's now dig into the code to understand the concept (loss and measuring criteria) along with the architecture of DiscoGAN.

The generator takes an input image of size 64x64x3 and feeds it through an encoder-decoder pair. The encoder part of the generator consists of five convolution layers with 4x4 filters, each followed by batch normalization and Leaky ReLU. The decoder part consists of five deconvolution layers with 4x4 filters, followed by a batch normalization and `ReLU` activation function, and outputs a target domain image of size 64x64x3. The following is the generator code snippet:

```
class Generator(nn.Module):

            self.main = nn.Sequential(
            # Encoder
                nn.Conv2d(3, 64, 4, 2, 1, bias=False),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Conv2d(64, 64 * 2, 4, 2, 1,
bias=False),
                nn.BatchNorm2d(64 * 2),
```

```
                nn.LeakyReLU(0.2, inplace=True),
                nn.Conv2d(64 * 2, 64 * 4, 4, 2, 1,
bias=False),
                nn.BatchNorm2d(64 * 4),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Conv2d(64 * 4, 64 * 8, 4, 2, 1,
bias=False),
                nn.BatchNorm2d(64 * 8),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Conv2d(64 * 8, 100, 4, 1, 0,
bias=False),
                nn.BatchNorm2d(100),
                nn.LeakyReLU(0.2, inplace=True),

             # Decoder
                nn.ConvTranspose2d(100, 64 * 8, 4, 1,
0, bias=False),
                nn.BatchNorm2d(64 * 8),
                nn.ReLU(True),
                nn.ConvTranspose2d(64 * 8, 64 * 4, 4,
2, 1, bias=False),
                nn.BatchNorm2d(64 * 4),
                nn.ReLU(True),
                nn.ConvTranspose2d(64 * 4, 64 * 2, 4,
2, 1, bias=False),
                nn.BatchNorm2d(64 * 2),
                nn.ReLU(True),
                nn.ConvTranspose2d(64 * 2, 64, 4, 2,
1, bias=False),
                nn.BatchNorm2d(64),
                nn.ReLU(True),
                nn.ConvTranspose2d(64,3, 4, 2, 1,
bias=False),
                nn.Sigmoid()

                . . .

             )
```

The discriminator is similar to the encoder part of the generator and consists of five convolution layers with 4x4 filters, each followed by a batch normalization and `LeakyReLU` activation function. Finally, we apply the `sigmoid` function on the final convolution layer (`conv-5`) to generate a scalar probability score between [0,1] to judge real/fake data. The following is the discriminator code snippet:

```
class Discriminator(nn.Module):

        self.conv1 = nn.Conv2d(3, 64, 4, 2, 1,
```

```
bias=False)
        self.relu1 = nn.LeakyReLU(0.2, inplace=True)

        self.conv2 = nn.Conv2d(64, 64 * 2, 4, 2, 1,
bias=False)
        self.bn2 = nn.BatchNorm2d(64 * 2)
        self.relu2 = nn.LeakyReLU(0.2, inplace=True)

        self.conv3 = nn.Conv2d(64 * 2, 64 * 4, 4, 2,
1, bias=False)
        self.bn3 = nn.BatchNorm2d(64 * 4)
        self.relu3 = nn.LeakyReLU(0.2, inplace=True)

        self.conv4 = nn.Conv2d(64 * 4, 64 * 8, 4, 2,
1, bias=False)
        self.bn4 = nn.BatchNorm2d(64 * 8)
        self.relu4 = nn.LeakyReLU(0.2, inplace=True)

        self.conv5 = nn.Conv2d(64 * 8, 1, 4, 1, 0,
bias=False)

        . . . .

   return torch.sigmoid( conv5 ), [relu2, relu3,
relu4]
```

Then we define the loss criteria for the generator and reconstruction using mean square error and binary cross entropy measures:

```
recon_criterion = nn.MSELoss()
gan_criterion = nn.BCELoss()


optim_gen = optim.Adam( gen_params,
lr=args.learning_rate, betas=(0.5,0.999),
weight_decay=0.00001)
optim_dis = optim.Adam( dis_params,
lr=args.learning_rate, betas=(0.5,0.999),
weight_decay=0.00001)
```

Now we start generating images from one domain to other and calculate the reconstruction loss to understand how well the original image is reconstructed after two translations (`ABA` or `BAB`):

```
AB = generator_B(A)
BA = generator_A(B)

ABA = generator_A(AB)
BAB = generator_B(BA)
```

```
# Reconstruction Loss
recon_loss_A = recon_criterion( ABA, A )
recon_loss_B = recon_criterion( BAB, B )
```

Next, we calculate the generator loss and discriminator loss across each domain:

```
# Real/Fake GAN Loss (A)
A_dis_real, A_feats_real = discriminator_A( A )
A_dis_fake, A_feats_fake = discriminator_A( BA )

dis_loss_A, gen_loss_A = get_gan_loss( A_dis_real,
A_dis_fake, gan_criterion, cuda )
fm_loss_A = get_fm_loss(A_feats_real, A_feats_fake,
feat_criterion)

# Real/Fake GAN Loss (B)
B_dis_real, B_feats_real = discriminator_B( B )
B_dis_fake, B_feats_fake = discriminator_B( AB )

dis_loss_B, gen_loss_B = get_gan_loss( B_dis_real,
B_dis_fake, gan_criterion, cuda )
fm_loss_B = get_fm_loss( B_feats_real, B_feats_fake,
feat_criterion )

gen_loss_A_total = (gen_loss_B*0.1 + fm_loss_B*0.9) *
(1.-rate) + recon_loss_A * rate
gen_loss_B_total = (gen_loss_A*0.1 + fm_loss_A*0.9) *
(1.-rate) + recon_loss_B * rate
```

Finally, we calculate the total loss of the `discogan` model by summing up the losses from two cross domains (`A` and `B`):

```
if args.model_arch == 'discogan':
    gen_loss = gen_loss_A_total + gen_loss_B_total
    dis_loss = dis_loss_A + dis_loss_B
```

Generating handbags from edges with PyTorch

In this example, we will generate realistic handbag images from corresponding edges using the `pix2pix` dataset from Berkley. Make sure you have PyTorch (http://pytorch.org/) and OpenCV (http://docs.opencv.org/trunk/d7/d9f/tutorial_linux_install.html) installed on your machine before going through the following steps:

1. First clone the `git` repository and change the directory to `DiscoGAN`:

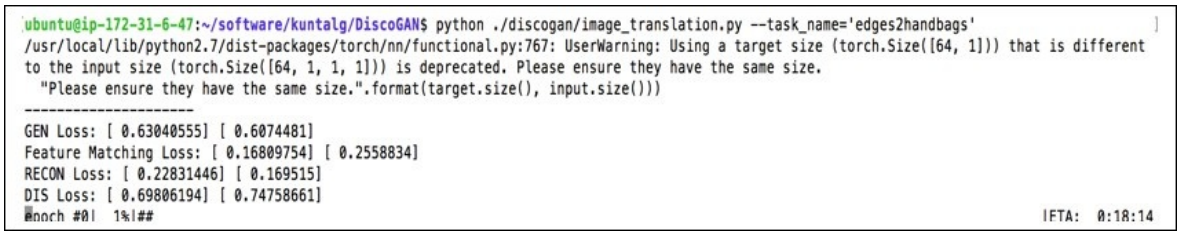
```
git clone https://github.com/SKTBrain/DiscoGAN.git
cd DiscoGAN
```

2. Next download the `edges2handbags` dataset using the following command:

```
python ./datasets/download.py edges2handbags
```

3. And then apply image translation between two domains: edges and handbags with the downloaded dataset:

```
python ./discogan/image_translation.py --
task_name='edges2handbags'
```



4. Now, the images will be saved after every 1,000 iterations (as per the `image_save_interval` argument) per epoch, under the `results` directory with the respective task name used previously during the image translation step:

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DiscoGAN/results$ ls
edges2handbags  facescrub
```

The following is a sample output of the cross-domain images generated from domain A to domain B:

Figure-3: On the left-hand side are cross-domain (A -> B -> A) generated images (edges -> handbags -> edges), while on the right-hand side are cross-domain (B -> A -> B) generated images of (handbags -> edges -> handbags)

Gender transformation using PyTorch

In this example, we will transform the gender of actor-to-actress or vice versa using facial images of celebrities from the `facescrub` dataset. Just like the previous example, please make sure you have PyTorch and OpenCV installed on your machine before executing the following steps:

1. First clone the `git` repository and change directory to `DiscoGAN` (you can skip this step if you executed the previous example of generating handbags from edges):

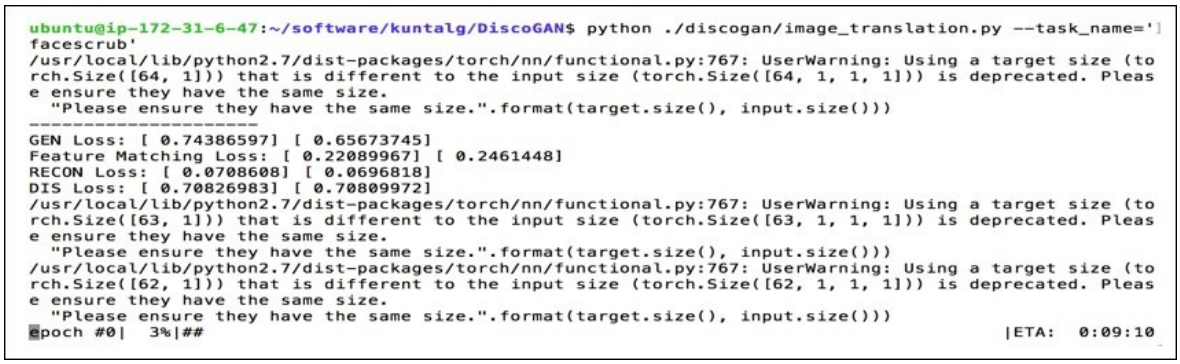
```
git clone https://github.com/SKTBrain/DiscoGAN.git
cd DiscoGAN
```

2. Next download the `facescrub` dataset using the following command:

```
python ./datasets/download.py facescrub
```

3. And then apply image translation between two domains, male and female, with the downloaded dataset:

```
python ./discogan/image_translation.py --
task_name= facescrub
```



4. Now, the images will be saved after every 1,000 iterations (as per the `image_save_interval` argument) per epoch, under the `results` directory with the respective task name (`facescrub`) and epoch interval:

The following is the sample output of the cross-domain images generated from domain A (male) to domain B (female):

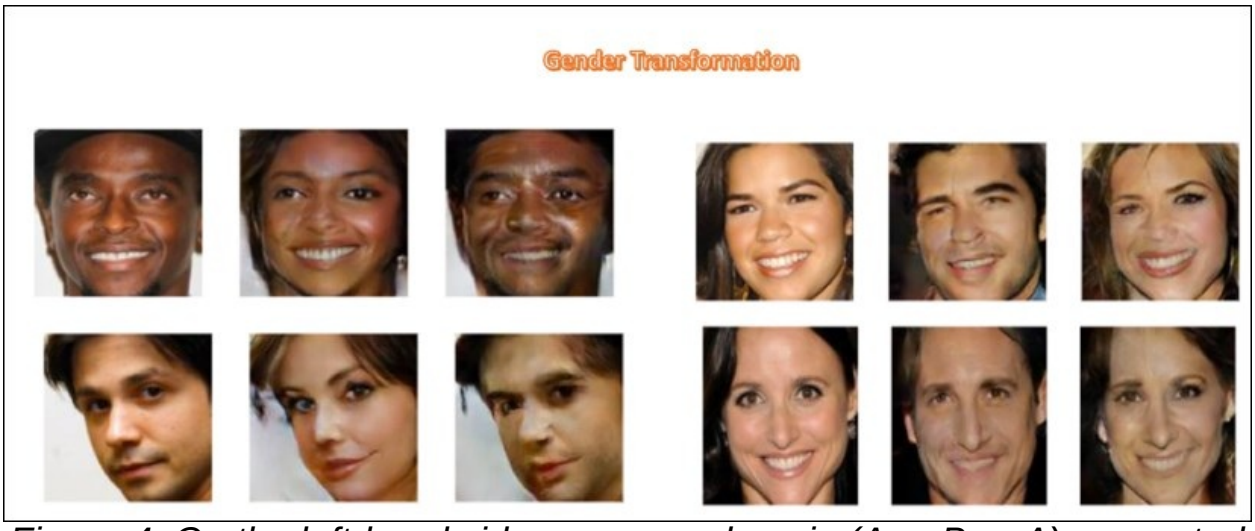


Figure-4: On the left-hand side are cross-domain (A -> B -> A) generated images (Male -> Female -> Male), while on the right-hand side are cross-domain (B -> A -> B) generated images of (Female -> Male -> Female)

DiscoGAN versus CycleGAN

The main objective of both DiscoGAN (discussed previously) and CycleGAN (discussed in Chapter 2, *Unsupervised Learning with GAN*) introduces a new approach to the problem of image-to-image translation by finding a mapping between a source domain X and a target domain Y for a given image, without pairing information.

From an architecture standpoint both the models consist of two GANs that map one domain to its counterpart domain and compose their losses as functions of the traditional generator loss (normally seen in GANs) and the reconstruction loss/cycle consistency loss.

There isn't a huge dissimilarity between the two models, except from the fact that DiscoGAN uses two reconstruction losses (a measure of how well the original image is reconstructed after the two translations X->Y->X), whereas CycleGAN uses a single cycle consistency loss with two translators F and G (F translates the image from domain X to domain Y and G performs the reverse) to make sure the two equilibriums (*F(G(b)) = b and G(F(a)) = a*, given *a*, *b* are images in domain X , Y respectively) are maintained.

Summary

So far you have learned the approach of solving complex real-life problems (such as synthesizing images from text and discovering cross-domain relationships) by combining multiple GAN models together using StackGAN and DiscoGAN. In the next chapter, you will learn an important technique for dealing with small datasets in deep learning using pre-trained models and feature transfer and how to run your deep models at a large scale on a distributed system.

Chapter 5. Using Various Generative Models to Generate Images

Deep learning shines with big data and deeper models. It has millions of parameters that can take even weeks to train. Some real-life scenarios may not have sufficient data, hardware, or resources to train bigger networks in order to achieve the desired accuracy. Is there any alternative approach or do we need to reinvent the training wheel from scratch all the time?

In this chapter, we will first look at the powerful and widely used training approach in modern deep learning based applications named **Transfer Learning** through hands-on examples with real datasets (`MNIST`, `cars vs cats vs dogs vs flower`, `LFW`). Also, you will build deep learning-based network over large distributed clusters using Apache Spark and BigDL. Then you will combine both Transfer Learning and GAN to generate high resolution realistic images with facial datasets. Finally, you will also understand how to create artistic hallucination on images beyond GAN.

We will cover the following topics in this chapter:

- What is Transfer Learning?—its benefits and applications
- Classifying `cars vs dog vs flower` with pre-trained VGG model using Keras
- Training and deploying a deep network over large distributed clusters with Apache Spark—deep learning pipeline
- Identifying handwritten digits through feature extraction and fine tuning using BigDL
- High resolution image generation using pre-trained model and SRGAN
- Generating artistic hallucinated images with DeepDream and image generation with VAE

Building a deep learning model from scratch requires sophisticated resources and also it is very time consuming. And hence you don't always want to build such deep models from scratch to solve your problem at hand. Instead of reinventing the same wheel, you will reuse an already existing model built for similar problems to satisfy your use case.

Let's say you want to build a self-driving car. You can either to spend years building a decent image recognition algorithm from scratch or you can simply take the pre-trained inception model built by Google from a huge dataset of ImageNet. A pre-trained model may not reach the desired accuracy level for your application, but it saves huge effort required to reinvent the wheel. And with some fine tuning and tricks your accuracy level will definitely improve.

Introduction to Transfer Learning

Pre-trained models are not optimized for tackling user specific datasets, but they are extremely useful for the task at hand that has similarity with the trained model task.
For example, a popular model, InceptionV3, is optimized for classifying images on a broad set of 1000 categories, but our domain might be to classify some dog breeds. A well-known technique used in deep learning that adapts an existing trained model for a similar task to the task at hand is known as Transfer Learning.
And this is why Transfer Learning has gained a lot of popularity among deep learning practitioners and in recent years has become the go-to technique in many real-life use cases. It is all about transferring knowledge (or features) among related domains.

The purpose of Transfer Learning

Let's say you have trained a deep neural network to differentiate between fresh mango and rotten mango. During training the network will have required thousands of rotten and fresh mango images and hours of training to learn knowledge such as if any fruit is rotten, liquid will come out of it and it will produce a bad smell. Now with this training experience the network can be used for different tasks/use-cases to differentiate between rotten apples and fresh apples using the knowledge of the rotten features learned during training of mango images.
The general approach of Transfer Learning is to train a base network and then copy its first n layers to the first n layers of a target network. The remaining layers of the target network are initialized randomly and trained toward the targeted use case.
The main scenarios for using Transfer Learning in your deep learning workflow are as follows:

- **Smaller datasets**: When you have a smaller dataset, building a deep learning model from scratch won't work well. Transfer Learning

provides the way to apply a pre-trained model to new classes of data. Let's say a pre-trained model built from one million images of ImageNet data will converge to a decent solution (after training on just a fraction of the available smaller training data, for example, CIFAR-10) compared to a deep learning model built with a smaller dataset from scratch.

- **Less resources**: Deep learning processes (such as convolution) require a significant amount of resource and time. Deep learning processes are well suited to run on high grade GPU-based machines. But with pre-trained models, you can easily train across a full training set (let's say 50,000 images) in less than a minute using your laptop/notebook without GPU, since the majority of time a model is modified in the final layer with a simple update of just a classifier or regressor.

Various approaches of using pre-trained models

We will discuss how pre-trained model could be used in different ways:

- **Using pre-trained architecture**: Instead of transferring weights of the trained model, we can only use the architecture and initialize our own random weights to our new dataset.
- **Feature extractor**: A pre-trained model can be used as a feature extraction mechanism just by simply removing the output layer of the network (that gives the probabilities for being in each of the n classes) and then freezing all the previous layers of the network as a fixed feature extractor for the new dataset.
- **Partially freezing the network**: Instead of replacing only the final layer and extracting features from all previous layers, sometimes we might train our new model partially (that is, to keep the weights of initial layers of the network frozen while retraining only the higher layers). The choice of the number of frozen layers can be considered as one more hyper-parameter.

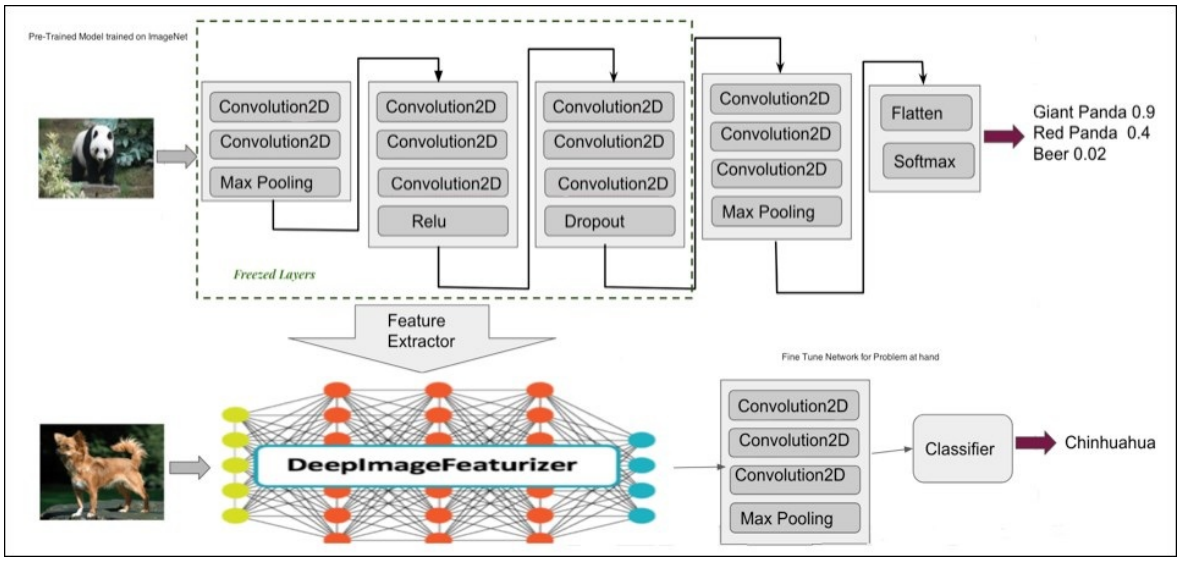


Figure-1: Transfer Learning with a pre-trained model

Depending mainly on data size and dataset similarity, you might have to decide on how to proceed with Transfer Learning. The following table discusses these scenarios:

	High data similarity	**Low data similarity**
Data size small	In the scenario of small data size but high data similarity, we will modify only the output layers of the pre-trained model and use it as a feature extractor.	When both data size as well as data similarity is low, we can freeze initial *k* layers of the pre-trained network and train only the *(n-k)* remaining layers again. This will help the top layers to customize to the new dataset and the small data size will also get compensated by frozen initial *k* layers of the network.
Data size large	In this scenario, we can use the architecture and initial weights of the pre-trained model.	Although we have a large dataset, the data is very different compared to the one used for training the pre-trained model, so using it in this scenario would not be effective. Instead it is better to train the deep network from scratch.

In case of image recognition Transfer Learning utilize the pre-trained convolutional layers to extract features about the new input images, that means only a small part of the original model (mainly the dense layers) are retrained. The rest of the network remains frozen. In this way, it saves a lot of time and resource by passing the raw images through the frozen part of the network only once, and then never goes through that part of

the network again.

Classifying car vs cat vs dog vs flower using Keras

Let us implement the concept of Transfer Learning and fine-tuning to identify customizable object categories using a customized dataset consisting of 150 training images and 50 validation images for each category of car, cat, dog, and flower.

Note that the dataset is prepared by taking images from the *Kaggle Dogs vs.Cats* (https://www.kaggle.com/c/dogs-vs-cats), Stanford cars (http://ai.stanford.edu/~jkrause/cars/car_dataset.html), and `Oxford flower` dataset (http://www.robots.ox.ac.uk/~vgg/data/flowers).

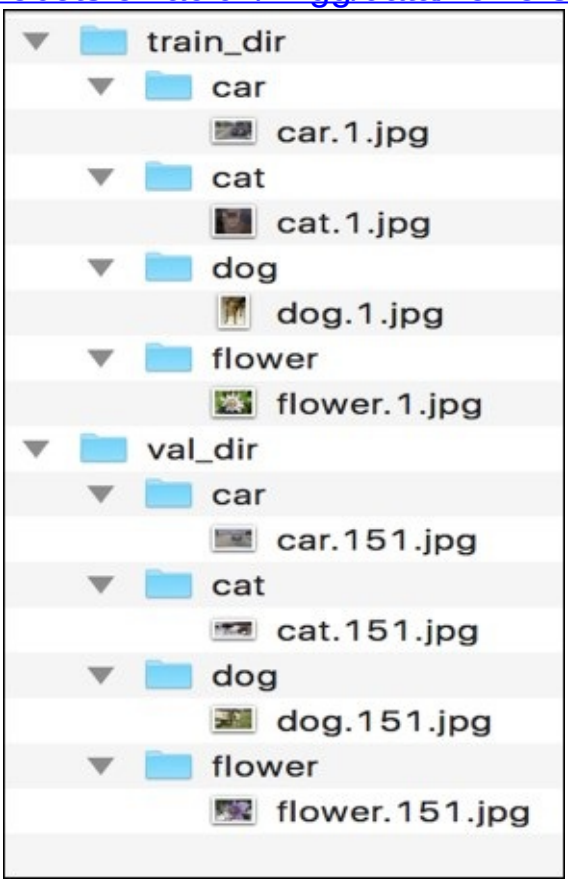


Figure-2: Car vs Cat vs Dog vs Flower dataset structure

We need to perform some preprocessing using the `preprocessing` function and apply various data augmentation transformation through `rotation`, `shift`, `shear`, `zoom`, and `flip` parameters:

```
train_datagen = ImageDataGenerator(
        preprocessing_function=preprocess_input,
        rotation_range=30,
```

```
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
)
test_datagen = ImageDataGenerator(
        preprocessing_function=preprocess_input,
        rotation_range=30,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
)
train_generator = train_datagen.flow_from_directory(
        args.train_dir,
        target_size=(IM_WIDTH, IM_HEIGHT),
        batch_size=batch_size,
)
validation_generator =
test_datagen.flow_from_directory(
        args.val_dir,
        target_size=(IM_WIDTH, IM_HEIGHT),
        batch_size=batch_size,
)
```

Next, we need to load the InceptionV3 model from the `keras.applications` module. The flag `include_top=False` is used to leave out the weights of the last fully connected layer:

```
base_model = InceptionV3(weights='imagenet',
include_top=False)
```

Initialize a new last layer by adding fully-connected `Dense` layer of size 1024, followed by a `softmax` function on the output to squeeze the values between `[0,1]`:

```
def addNewLastLayer(base_model, nb_classes):
        x = base_model.output
        x = GlobalAveragePooling2D()(x)
        x = Dense(FC_SIZE, activation='relu')(x)
        predictions = Dense(nb_classes,
activation='softmax')(x)
        model = Model(input=base_model.input,
output=predictions)
        return model
```

Once the last layer of the network is stabilized (Transfer Learning), we

can move onto retraining more layers (fine-tuning).
Use a utility method to freeze all layers and compile the model:

```
def setupTransferLearn(model, base_model):
        for layer in base_model.layers:
          layer.trainable = False
model.compile(optimizer='rmsprop',

loss='categorical_crossentropy',
              metrics=['accuracy'])
```

This is another utility method to freeze the bottom of the top two inception blocks in the InceptionV3 architecture and retrain the remaining top:

```
def setupFineTune(model):
   for layer in
model.layers[:NB_IV3_LAYERS_TO_FREEZE]:
      layer.trainable = False
   for layer in
model.layers[NB_IV3_LAYERS_TO_FREEZE:]:
          layer.trainable = True
   model.compile(optimizer=SGD(lr=0.0001,
momentum=0.9),
loss='categorical_crossentropy')
```

Now we're all ready for training using the `fit_generator` method and finally save our model:

```
history = model.fit_generator(
        train_generator,
        samples_per_epoch=nb_train_samples,
        nb_epoch=nb_epoch,
        validation_data=validation_generator,
        nb_val_samples=nb_val_samples,
        class_weight='auto')
model.save(args.output_model_file)
```

Run the following command for training and fine tuning:

```
python training-fine-tune.py --train_dir <path to
training images> --val_dir <path to validation images>
```

```
ubuntu@ip-172-31-1-246:~/software/dataset/cat-dog-flower-car$ python training-fine-tune.py --train_dir /home/ubuntu/software/dataset/cat-
dog-car-flower/train_dir --val_dir /home/ubuntu/software/dataset/cat-dog-car-flower/val_dir

Using TensorFlow backend.
Found 600 images belonging to 4 classes.
Found 200 images belonging to 4 classes.
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.2/inception_v3_weights_tf_dim_ordering_tf_ker
```

Even with such a small dataset size, we are able to achieve an accuracy

of 98.5 percent on the validation set by utilizing the power of the pre-trained model and Transfer Learning:

```
Epoch 1/3
128/600 [=====>........................] - ETA: 15s - loss: 0.0424 - acc: 0.98442017-08-13 07:06:31.582692: I tensorflow/core/common_runtime/gpu/pool_allocator.cc:247] PoolAllocator: After 11762 get requests, put_count=11717 evicted_count=1000 eviction_rate=0.0853461 and unsatisfied allocation rate=0.0973474
2017-08-13 07:06:31.582730: I tensorflow/core/common_runtime/gpu/pool_allocator.cc:259] Raising pool_size_limit_ from 100 to 110
600/600 [==============================] - 15s - loss: 0.0522 - acc: 0.9833 - val_loss: 0.0708 - val_acc: 0.9800
Epoch 2/3
600/600 [==============================] - 13s - loss: 0.0424 - acc: 0.9883 - val_loss: 0.0657 - val_acc: 0.9850
Epoch 3/3
600/600 [==============================] - 13s - loss: 0.0240 - acc: 0.9933 - val_loss: 0.1065 - val_acc: 0.9850
```

Voila, we can now use the saved model to predict images (either from the local filesystem or from the URL) with test data:

```
python predict.py --image_url https://goo.gl/DCbuq8 --
model inceptionv3-ft.model
```

```
ubuntu@ip-172-31-1-246:~/software/dataset/cat-dog-flower-car$ python predict.py --image_url https://goo.gl/DCbuq8 --model
inceptionv3-ft.model
```

```
Car Probability
---------------
1.0
Cat Probability
---------------
8.61596e-10
Dog Probability
---------------
1.48996e-10
Flower Probability
---------------
2.36165e-10
```

Large scale deep learning with Apache Spark

Deep learning is a resource hungry and computationally intensive process and you get better result with more data and bigger network, but its speed gets impacted by the size of the datasets as well. And in practice, deep learning requires experimenting with different values for training parameters known as hyper-parameter tuning, where you have to run your deep networks on a large dataset iteratively or many times and speed does matter. Some common ways to tackle this problem is to use faster hardware (usually GPUs), optimized code (with a proper production-ready framework), and scaling out over distributed clusters to achieve some form of parallelism.

Data parallelism is a concept of sharding large datasets into multiple chunks of data and then processing chunks over neural networks running on separate nodes of distributed clusters.

Apache Spark is a fast, general-purpose, fault-tolerant framework for interactive and iterative computations on large, distributed datasets by doing in-memory processing of RDDs or DataFrames instead of saving data to hard disks. It supports a wide variety of data sources as well as storage layers. It provides unified data access to combine different data formats, streaming data, and defining complex operations using high-level, composable operators.

Today Spark is the superpower of big data processing and makes big data accessible to everyone. But Spark or its core modules alone are not capable of training or running deep networks over the clusters. In the next few sections, we will develop deep learning applications over Apache Spark cluster with optimized libraries.

> **Note**
> For coding purposes, we will not cover the distributed Spark cluster setup, instead use Apache Spark standalone mode. More information about Spark cluster mode can be found at: https://spark.apache.org/docs/latest/cluster-overview.html.

Running pre-trained models using Spark deep learning

Deep learning pipelines is an open-source library that leverages the

power of Apache Spark cluster to easily integrate scalable deep learning into machine learning workflows. It is built on top of Apache Spark's ML Pipelines for training, and uses Spark DataFrames and SQL for deploying models. It provides high-level APIs for running Transfer Learning in a distributed manner by integrating pre-trained model as transformer in Spark ML Pipeline.
Deep learning pipelines make Transfer Learning easier with the concept of a featurizer. The featurizer (or `DeepImageFeaturizer` in case of image operation) automatically removes the last layer of a pre-trained neural network model and uses all the previous layers output as features for the classification algorithm (for example, logistic regression) specific to the new problem domain.
Let us implement deep learning pipelines for predicting sample images of the `flower` dataset (http://download.tensorflow.org/example_images/flower_photos.tgz) with a pre-trained model over the Spark cluster:

1. First start the PySpark with the deep learning pipeline package:

```
pyspark --master local[*] --packages
databricks:spark-deep-learning:0.1.0-spark2.1-
s_2.11
```

> **Tip**
> Note: If you get the error **No module named sparkdl** while starting the PySpark with deep learning, please check the GitHub page for workaround: https://github.com/databricks/spark-deep-learning/issues/18

2. First read the images and randomly split it into `train`, `test` set.

```
img_dir= "path to base image directory"
roses_df = readImages(img_dir +
"/roses").withColumn("label", lit(1))
daisy_df = readImages(img_dir +
"/daisy").withColumn("label", lit(0))
roses_train, roses_test =
roses_df.randomSplit([0.6, 0.4])
daisy_train, daisy_test =
daisy_df.randomSplit([0.6, 0.4])
train_df = roses_train.unionAll(daisy_train)
```

3. Then create a pipeline with `DeepImageFeaturizer` using the `InceptionV3` model:

```
featurizer = DeepImageFeaturizer(inputCol="image",
outputCol="features", modelName="InceptionV3")
```

```
lr = LogisticRegression(maxIter=20, regParam=0.05,
elasticNetParam=0.3, labelCol="label")
p = Pipeline(stages=[featurizer, lr])
```

4. Now fit the images with an existing pre-trained model, where `train_images_df` is a dataset of images and labels:

```
p_model = p.fit(train_df)
```

5. Finally, we will evaluate the accuracy:

```
tested_df = p_model.transform(test_df)
evaluator =
MulticlassClassificationEvaluator(metricName="accu
racy")
print("Test set accuracy = " +
str(evaluator.evaluate(tested_df.select("predictio
n", "label"))))
```

```
>>> evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
>>> print("Test set accuracy = " + str(evaluator.evaluate(tested_df.select("prediction", "label"))))
Test set accuracy = 0.962264150943
```

In addition to `DeepImageFeaturizer`, we can also utilize the pre-existing model just to do prediction, without any retraining or fine tuning using `DeepImagePredictor`:

```
sample_img_dir=<path to your image>

image_df = readImages(sample_img_dir)

predictor = DeepImagePredictor(inputCol="image",
outputCol="predicted_labels", modelName="InceptionV3",
decodePredictions=True, topK=10)
predictions_df = predictor.transform(image_df)

predictions_df.select("filePath",
"predicted_labels").show(10,False)
```

The input image and its top five predictions are shown as follows:

```
>>> predictions_df.select("filePath", "predicted_labels").show(10,False)
+----------------------------------------------------+--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------+
|filePath                                            |predicted_labels
                                                                                      |
+----------------------------------------------------+--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------+
|file:/Users/kuntalg/Downloads/flower_photos/elephant.jpg|[[n01871265,tusker,0.59073424], [n02504458,African_elephant,0.32260254], [n02504013,Indian_elepha
nt,0.015373577], [n03075370,combination_lock,3.9325625E-4], [n03874599,padlock,3.417398E-4]]|
+----------------------------------------------------+--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------+
```

In addition to using the built-in pre-trained models, deep learning pipeline allows users to plug in Keras models or TensorFlow graphs in a Spark prediction pipeline. This really turns any single-node deep models running on a single-node machine into one that can be trained and deployed in a distributed fashion, on a large amount of data.

First, we will load the Keras built-in InceptionV3 model and save it in the file:

```
model = InceptionV3(weights="imagenet")
model.save('model-full.h5')
```

During the prediction phase, we will simply load the model and pass images through it to get the desired prediction:

```
def loadAndPreprocessKerasInceptionV3(uri):
    # this is a typical way to load and prep images in
keras
    image = img_to_array(load_img(uri, target_size=
(299, 299)))
    image = np.expand_dims(image, axis=0)
    return preprocess_input(image)
```

```
transformer =
KerasImageFileTransformer(inputCol="uri",
  outputCol="predictions",

modelFile="model-full.h5",

imageLoader=loadAndPreprocessKerasInceptionV3,

outputMode="vector")
dirpath=<path to mix-img>

files = [os.path.abspath(os.path.join(dirpath, f)) for
f in os.listdir(dirpath) if f.endswith('.jpg')]
uri_df = sqlContext.createDataFrame(files,
StringType()).toDF("uri")

final_df = transformer.transform(uri_df)
final_df.select("uri", "predictions").show()
```

Deep learning pipeline is a really fast way of doing Transfer Learning over distributed Spark clusters. But you must have noticed that featurizer only allow us to change the final layer of the pre-trained model. But in some scenarios, you might have to modify more than one layer of the pre-trained network to get the desired result and deep learning pipeline doesn't provide this full capability.

Handwritten digit recognition at a large scale using BigDL

BigDL is an open-source distributed high performance deep learning library that can run directly on top of Apache Spark clusters. Its high performance is achieved by combing **Intel® Math Kernel Library** (**MKL**) along with multithreaded programming in each Spark task. BigDL provides Keras style (both sequential and function) high-level APIs to build deep learning application and scale out to perform analytics at a large scale. The main purposes of using BigDL are:

- Running deep learning model at a large scale and analyzing massive amount of data residing in a Spark or Hadoop cluster (in say Hive, HDFS, or HBase)
- Adding deep learning functionality (both training and prediction) to your big data workflow

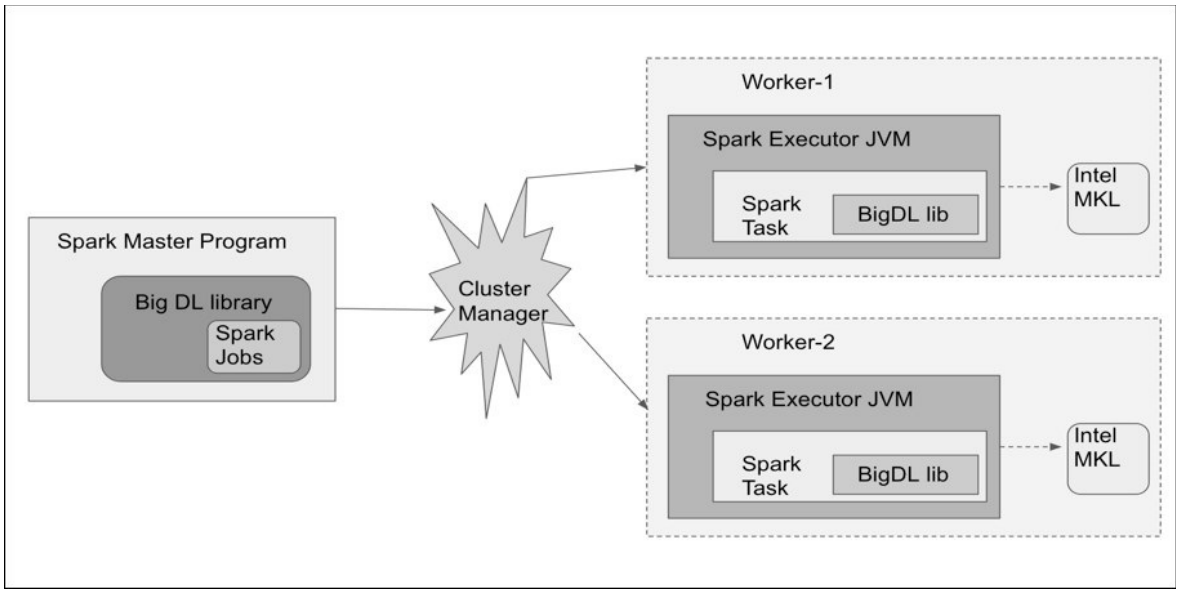


Figure-3: BigDL execution over Spark cluster

As you can see from the figure, the BigDL driver program is first launched in the Spark master node of the cluster. Then with the help of **cluster manager** and the driver program, Spark tasks are distributed across the Spark executors on the worker nodes. And BigDL interacts with Intel MKL to enable faster execution of those tasks.

Let us implement a deep neural network at a large scale for identifying hand-written digits with the `mnist` dataset. First we will prepare training and validation samples:

```
mnist_path = "datasets/mnist"
(train_data, test_data) = get_mnist(sc, mnist_path)
print train_data.count()
print test_data.count()
```

Then we will create the LeNet architecture consisting of two sets of convolutional, activation, and pooling layers, followed by a fully-connected layer, activation, another fully-connected, and finally a `SoftMax` classifier. LeNet is small, yet powerful enough to provide interesting results:

```
def build_model(class_num):
    model = Sequential()
    model.add(Reshape([1, 28, 28]))
    model.add(SpatialConvolution(1, 6, 5,
5).set_name('conv1'))
    model.add(Tanh())
    model.add(SpatialMaxPooling(2, 2, 2,
2).set_name('pool1'))
    model.add(Tanh())
```

```
    model.add(SpatialConvolution(6, 12, 5,
5).set_name('conv2'))
    model.add(SpatialMaxPooling(2, 2, 2,
2).set_name('pool2'))
    model.add(Reshape([12 * 4 * 4]))
    model.add(Linear(12 * 4 * 4, 100).set_name('fc1'))
    model.add(Tanh())
    model.add(Linear(100,
class_num).set_name('score'))
    model.add(LogSoftMax())
    return model
lenet_model = build_model(10)
```

Now we will configure an `Optimizer` and set the validation logic:

```
optimizer = Optimizer(
    model=lenet_model,
    training_rdd=train_data,
    criterion=ClassNLLCriterion(),
    optim_method=SGD(learningrate=0.4,
learningrate_decay=0.0002),
    end_trigger=MaxEpoch(20),
    batch_size=2048)

optimizer.set_validation(
    batch_size=2048,
    val_rdd=test_data,
    trigger=EveryEpoch(),
    val_method=[Top1Accuracy()]
)

trained_model = optimizer.optimize()
```

Then we will take a few test samples, and make prediction by checking both the predicted labels and the ground truth labels:

```
predictions = trained_model.predict(test_data)
```

Finally, we will train the LeNet model in the Spark cluster using the `spark-submit` command. Download the BigDL distribution (https://bigdl-project.github.io/master/#release-download/) based on your Apache Spark version and then execute the file (`run.sh`) provided with the code to submit the job in the Spark cluster:

```
SPARK_HOME= <path to Spark>
BigDL_HOME= <path to BigDL>
PYTHON_API_ZIP_PATH=${BigDL_HOME}/bigdl-python-
<version>.zip
BigDL_JAR_PATH=${BigDL_HOME}/bigdl-SPARK-<version>.jar
export
PYTHONPATH=${PYTHON_API_ZIP_PATH}:${BigDL_HOME}/conf/s
```

```
park-bigdl.conf:$PYTHONPATH

${SPARK_HOME}/bin/spark-submit \
      --master <local or spark master url>\
      --driver-cores 5  \
      --driver-memory 5g  \
      --total-executor-cores 16  \
      --executor-cores 8  \
      --executor-memory 10g \
      --py-files
${PYTHON_API_ZIP_PATH},${BigDL_HOME}/BigDL-MNIST.py\
      --properties-file ${BigDL_HOME}/conf/spark-
bigdl.conf \
      --jars ${BigDL_JAR_PATH} \
      --conf
spark.driver.extraClassPath=${BigDL_JAR_PATH} \
      --conf spark.executor.extraClassPath=bigdl-
SPARK<version>.jar \
      ${BigDL_HOME}/BigDL-MNIST.py
```

More information regarding `spark-submit` can be found at: https://spark.apache.org/docs/latest/submitting-applications.html. Once you have submitted the job, you can track the progress on the **Spark Master** application page as follows:

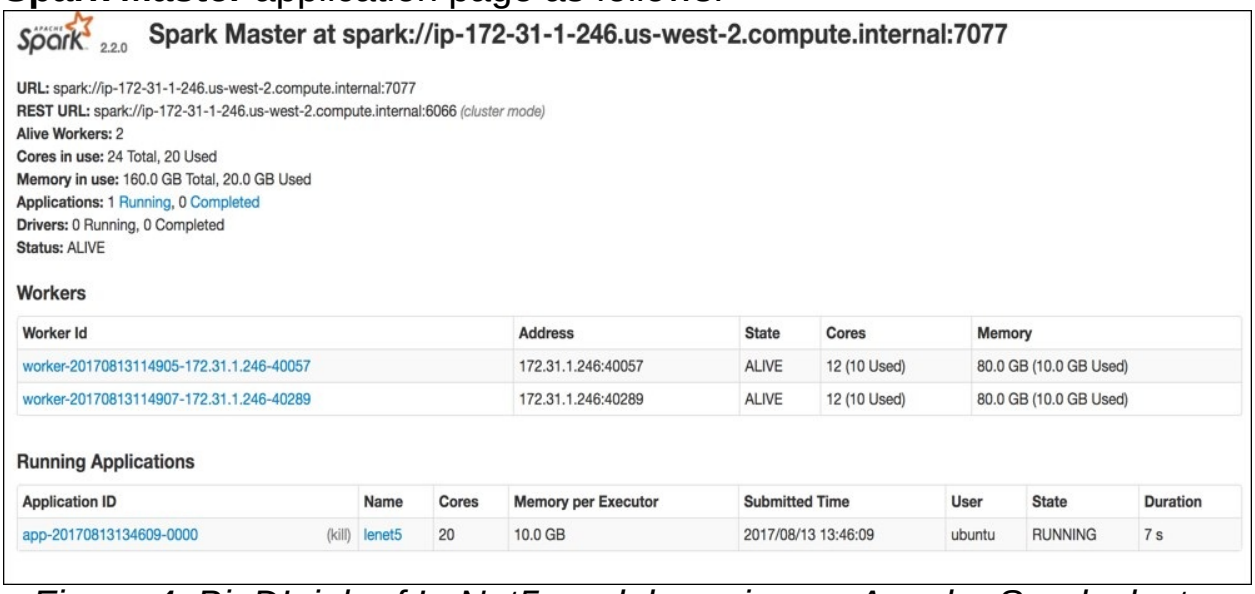


Figure-4: BigDL job of LeNet5 model running on Apache Spark cluster

After the job has successfully finished, you can search the logs of the Spark workers to verify the accuracy of your model similar to the one shown as follows:

```
INFO  DistriOptimizer$:536 - Top1Accuracy is
Accuracy(correct: 9568, count: 10000, accuracy:
0.9568)
```

High resolution image generation using SRGAN

Super Resolution Generative Network (**SRGAN)** excels in generating high resolution images from its low-resolution counterpart. During the training phase, a high resolution image is transformed to a low resolution image by applying the Gaussian filter to a high resolution image followed by the down-sampling operation.

Let us define some notation before diving into the network architecture:

- I^{LR}: Low resolution image having the size width(*W*) x height(*H*) x color channels(*C*)
- I^{HR}: High resolution image having the size $rW \times rH \times C$
- I^{SR}: Super resolution image having the size $rW \times rH \times C$
- *r*: down sampling factor
- $G_{\theta G}$: Generator network
- $D_{\theta D}$: Discriminator network

To achieve the goal of estimating a high resolution input image from its corresponding low resolution counterpart, the generator network is trained as a feed-forward convolution neural network $G_{\theta G}$ parametrized by *θG* where *θG* is represented by weights (*W1:L*) and biases (*b1:L*) of the L-layer of the deep network and is obtained by optimizing super resolution specific `loss` function. For training images having high resolution I_n^{HR}, *n=1*; N along with its corresponding low resolution I_n^{LR}, *n=1*, N, we can solve for *θG*, as follows:

$$\hat{\theta}G = \arg\min_{\theta G} \frac{1}{N} \sum_{n=1}^{N} l^{SR}\left(G_{\theta_G}\left(I_n^{LR}\right), I_n^{HR}\right)$$

Note

The formulation of the perceptual loss function *ISR* is critical for the performance of the generator network. In general, perceptual loss is commonly modeled based on **Mean Square Error** (**MSE**), but to avoid unsatisfying solutions with overly smooth textures, a new content loss based on the ReLU activation layers of the pre-trained 19 layer VGG network is formulated.

The perceptual loss is the weighted combination of several `loss` functions that map important characteristics of the super resolution image as

follows:

$$l^{SR} = \underbrace{\underbrace{l^{SR}_{VGG/i.j}}_{\text{content loss}} + \underbrace{10^{-3} l^{SR}_{Gen}}_{\text{adversarial loss}}}_{\text{perceptual loss (for VGG based content losses)}}$$

- **Content loss**: The VGG-based content loss is defined as the Euclidean distance between the feature representations of a reconstructed image $G_{\theta G}$ (I^{LR}) and the corresponding high resolution image *IHR*. Here $\varphi_{i,j}$ indicate the feature map obtained by the jth convolution (after activation) before the ith max-pooling layer within the VGG19 network. And *Wi,j*, *Hi,j* describe the dimensions of the respective feature maps within the *VGG* network:

$$l^{SR}_{VGG/i.j} = \frac{1}{W_{i,j} H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} \left(\varnothing_{i,j}\left(I^{HR}\right)x, y - \varnothing_{i,j}\left(G_{\theta_G}\left(I^{LR}\right)\right)x, y \right)^2$$

- **Adversarial loss**: The generative loss is based on the probabilities of the discriminator $D_{\theta D}(G_{\theta G}(I_{LR}))$ over all training images and encourages the network to favor solutions residing on the manifold of natural images, in order to fool the discriminator network:

$$l^{SR}_{Gen} = \sum_{n=1}^{N} -\log D_{\theta_D}\left(G_{\theta_G}\left(I^{LR}\right)\right)$$

Similar to the concept of adversarial network, the general idea behind the SRGAN approach is to train a generator *G* with the goal of fooling a discriminator *D* that is trained to distinguish super-resolution images from real images:

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{I^{HR} \sim p_{train}\left(I^{HR}\right)} \left[\log D_{\theta_D}\left(I^{HR}\right) \right] + \mathbb{E}_{I^{LR} \sim p_G\left(I^{LR}\right)} \left[\log\left(1 - D_{\theta_D}\right)\left(G_{\theta_G}\left(I^{LR}\right)\right) \right]$$

Based on this approach the generator learns to create solutions that are highly similar to real images and thus hard to classify by discriminator *D*. And this encourages perceptually superior solutions residing in the subspace, the manifold, of natural images.

Architecture of the SRGAN

As illustrated in the following figure, the generator network **G** consists of **B residual blocks** with identical layout. Each block has two convolutional layers with small 3×3 kernels and 64 feature maps followed by batch-

normalization layers [32] and ParametricReLU [28] as the `activation` function. The resolution of the input image is increased by two trained sub-pixel convolution layers.
The discriminator network uses Leaky ReLU activation (with $\alpha = 0.2$) and consists of eight convolutional layers with an increasing number of 3×3 filter kernels, increasing by a factor of 2 from 64 to 512 kernels. Each time the number of features is doubled, strided convolutions are used to reduce the image resolution.
The resulting 512 feature maps go through two dense layers followed by a final sigmoid activation layer to obtain a classification probability for a generated image sample:

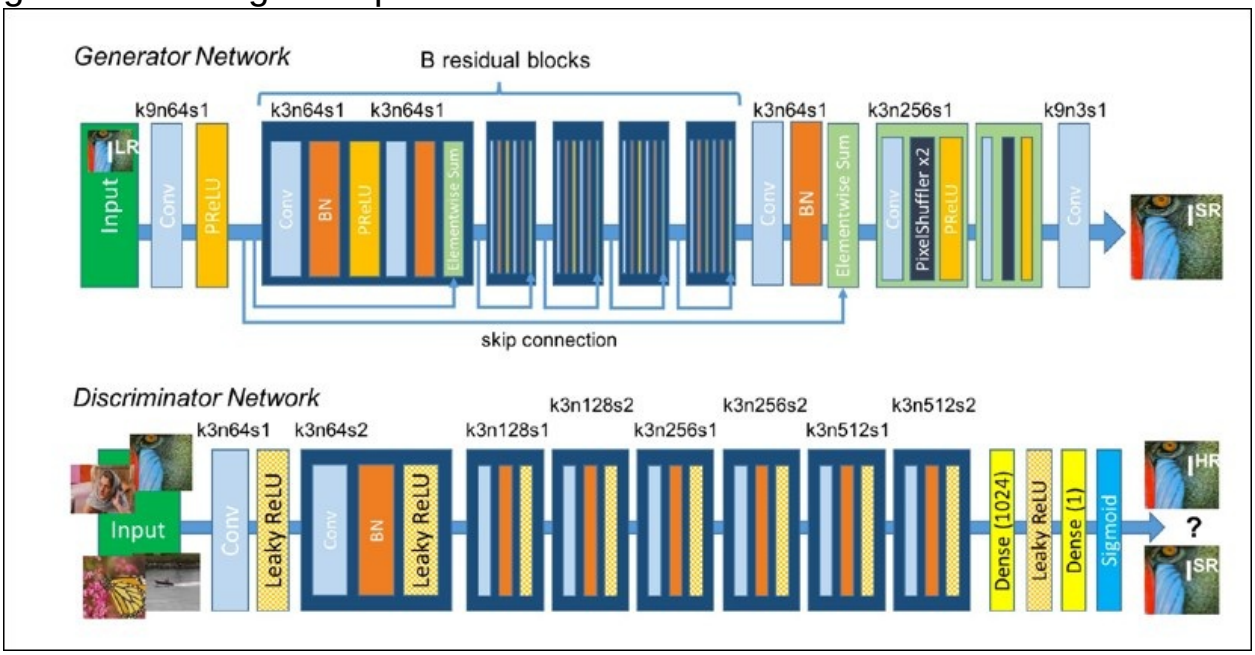


Figure-5: Architecture of generator and discriminator network with corresponding kernel size (k), number of feature maps (n) and stride (s) indicated for each convolutional layer.
Source: arXiv, 1609.04802, 2017

Now it's time to deep dive into the code with TensorFlow and generate high resolution images using an LFW facial dataset.
The generator network is first built as a single deconvolution layer with 3×3 kernels and 64 feature maps followed by ReLU as an `activation` function. Then there are five residual blocks with identical layout of each block having two convolutional layers followed by batch-normalization and ReLU. Finally, the resolution of the input image is increased by two trained pixel-shuffle layers:

```
def generator(self, x, is_training, reuse):
```

```
    with tf.variable_scope('generator', reuse=reuse):
      with tf.variable_scope('deconv1'):
          x = deconv_layer(
               x, [3, 3, 64, 3], [self.batch_size, 24,
24, 64], 1)
          x = tf.nn.relu(x)
          shortcut = x
# 5 Residual block with identical layout of
deconvolution layers having batch norm and relu as
activation function.
    for i in range(5):
       mid = x
       with tf.variable_scope('block{}a'.format(i+1)):
         x = deconv_layer(x, [3, 3, 64, 64],
[self.batch_size, 24,
24, 64], 1)
         x = batch_normalize(x, is_training)
         x = tf.nn.relu(x)

# 2 deconvolution layers having pixel-suffle and relu
as activation function.
     with tf.variable_scope('deconv3'):
         x = deconv_layer(x, [3, 3, 256, 64],
[self.batch_size, 24,
24, 256], 1)
         x = pixel_shuffle_layer(x, 2, 64) # n_split =
256 / 2 ** 2
         x = tf.nn.relu(x)
     with tf.variable_scope('deconv4'):
         x = deconv_layer(x, [3, 3, 64, 64],
[self.batch_size, 48,
48, 64], 1)
         x = pixel_shuffle_layer(x, 2, 16)
         x = tf.nn.relu(x)

     . . . . . [code omitted for clarity]

return x
```

The `deconvolution layer` function is defined with a TensorFlow `conv2d_transpose` method with Xavier initialization as follows:

```
def deconv_layer(x, filter_shape, output_shape,
stride, trainable=True):
    filter_ = tf.get_variable(
        name='weight',
        shape=filter_shape,
        dtype=tf.float32,

initializer=tf.contrib.layers.xavier_initializer(),
```

```
            trainable=trainable)
        return tf.nn.conv2d_transpose(
            value=x,
            filter=filter_,
            output_shape=output_shape,
            strides=[1, stride, stride, 1])
```

The discriminator network consists of eight convolutional layers having 3×3 filter kernels that get increased by a factor of 2 from 64 to 512 kernels. The resulting 512 feature maps are flattened and go through two dense fully connected layers followed by a final softmax layer to obtain a classification probability for a generated image sample:

```
def discriminator(self, x, is_training, reuse):
        with tf.variable_scope('discriminator',
reuse=reuse):
            with tf.variable_scope('conv1'):
                x = conv_layer(x, [3, 3, 3, 64], 1)
                x = lrelu(x)
            with tf.variable_scope('conv2'):
                x = conv_layer(x, [3, 3, 64, 64], 2)
                x = lrelu(x)
                x = batch_normalize(x, is_training)

         .  .   .   .   .   . [code omitted for
clarity]

            x = flatten_layer(x)
            with tf.variable_scope('fc'):
                x = full_connection_layer(x, 1024)
                x = lrelu(x)
            with tf.variable_scope('softmax'):
                x = full_connection_layer(x, 1)

return x
```

The network uses LeakyReLU (with $\alpha = 0.2$) as an `activation` function with the convolutional layer:

```
def lrelu(x, trainbable=None):
    alpha = 0.2
    return tf.maximum(alpha * x, x)
```

The `convolution layer` function is defined with a TensorFlow `conv2d` method with Xavier initialization as follows:

```
def conv_layer(x, filter_shape, stride,
trainable=True):
    filter_ = tf.get_variable(
        name='weight',
```

```
            shape=filter_shape,
            dtype=tf.float32,

    initializer=tf.contrib.layers.xavier_initializer(),
            trainable=trainable)
        return tf.nn.conv2d(
            input=x,
            filter=filter_,
            strides=[1, stride, stride, 1],
            padding='SAME')
```

Please note that the code implementation uses the least squares `loss` function (to avoid the vanishing gradient problem) for the discriminator, instead of the sigmoid cross entropy `loss` function as proposed in the original paper of SRGAN (*arXiv, 1609.04802, 2017*):

```
def inference_adversarial_loss(real_output,
fake_output):
alpha = 1e-5
g_loss = tf.reduce_mean(
tf.nn.l2_loss(fake_output -
tf.ones_like(fake_output)))
d_loss_real = tf.reduce_mean(
tf.nn.l2_loss(real_output -
tf.ones_like(true_output)))
d_loss_fake = tf.reduce_mean(
tf.nn.l2_loss(fake_output +
tf.zeros_like(fake_output)))
d_loss = d_loss_real + d_loss_fake
return (g_loss * alpha, d_loss * alpha)

generator_loss, discriminator_loss = (
inference_adversarial_loss(true_output, fake_output))
```

More information about **Least Square GAN** can be found at:
https://arxiv.org/abs/1611.04076
The `code` directory structure for running the SRGAN is shown as follows:

```
ubuntu@ip-172-31-1-246:~/software/dataset/srgan/code$ ls -lth
total 52K
drwxrwxr-x 3 ubuntu ubuntu 4.0K Aug 13 14:10 backup
drwxr-xr-x 2 ubuntu ubuntu 4.0K Aug 13 14:08 result
drwxr-xr-x 5 ubuntu ubuntu 4.0K Aug 13 14:07 data
-rwxr-xr-x 1 ubuntu ubuntu 3.3K Aug 13 13:03 trainSrgan.py
-rwxr-xr-x 1 ubuntu ubuntu  150 Aug 13 12:36 loadLfw.py
-rwxr-xr-x 1 ubuntu ubuntu 4.2K Aug 13 12:14 vgg19.py
-rwxr-xr-x 1 ubuntu ubuntu 4.2K Aug 13 12:12 utilLayer.py
-rwxr-xr-x 1 ubuntu ubuntu 1.2K Aug 13 12:12 utilAugment.py
-rwxr-xr-x 1 ubuntu ubuntu 6.9K Aug 13 12:12 srgan.py
-rwxr-xr-x 1 ubuntu ubuntu 2.7K Aug 13 12:12 download-preprocess-lfw.py
```

First let us download an `LFW facial` dataset and do some preprocessing (frontal face detection and splitting the dataset as train and test) and store the dataset under the `data/` directory:

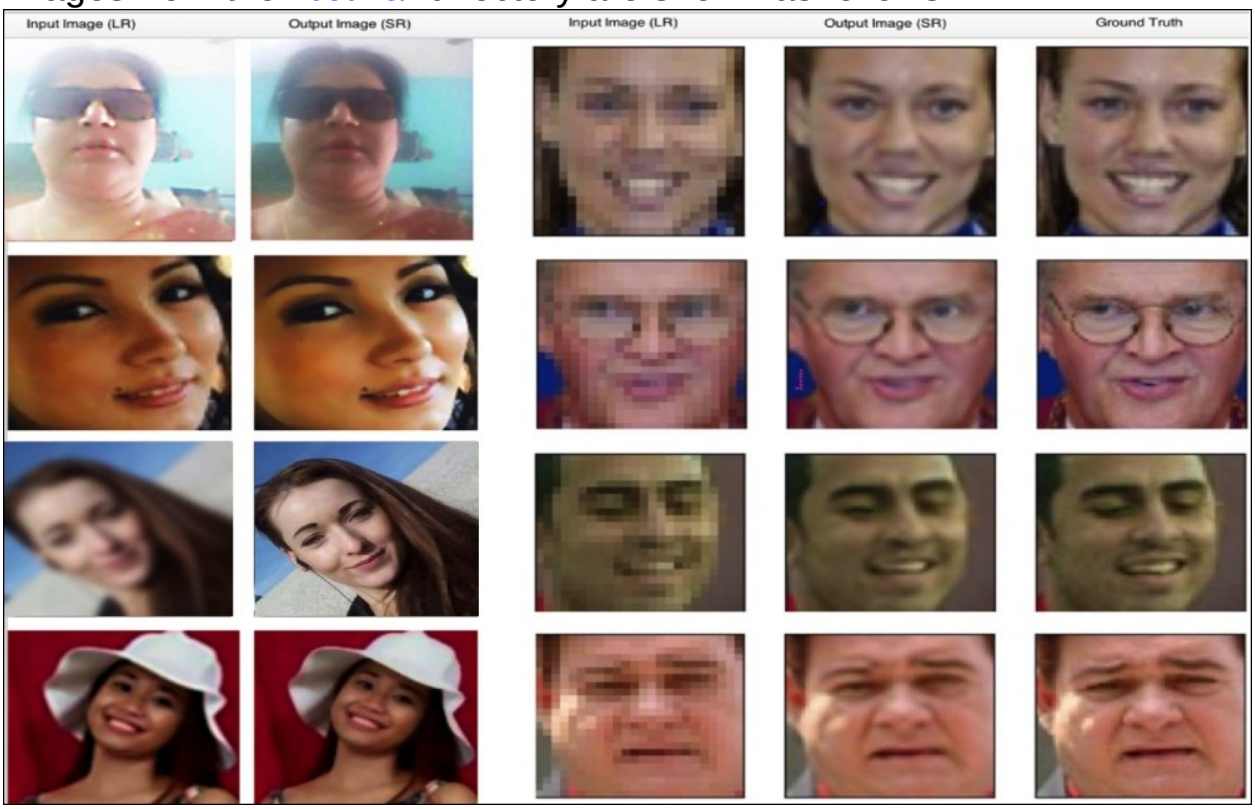
```
python download-preprocess-lfw.py
```

```
ubuntu@ip-172-31-1-246:~/software/dataset/srgan/code$ python download-preprocess-lfw.py
... loading data
100%|████████████████████████████████████████████████████████████| 12577/12577 [11:45<00:00, 17.82it/s]
100%|████████████████████████████████████████████████████████████| 662/662 [00:36<00:00, 17.92it/s]
```

Next download the pre-trained VGG19 model from the following link, extract it, and save it under the `backup/` directory:
https://drive.google.com/open?id=0B-s6ok7B0V9vcXNfSzdjZ0lCc0k
Next execute the `trainSrgan.py` file to start the SRGAN operation using a VGG19 model:

```
python trainSrgan.py
```

Once the training is going on the generator network will start generating super resolution images in the `result/` directory. Some of the sample images from the `result/` directory are shown as follows:



Generating artistic hallucinated images using DeepDream

The DeepDream algorithm is a modified neural network that has capability of producing impressive surrealistic, dream-like hallucinogenic appearances by changing the image in the direction of training data. It uses backpropagation to change the image instead of changing weights through the network.

Broadly, the algorithm can be summarized in the following steps:

1. Select a layer of the network and a filter that you feel is interesting.
2. Then compute the activations of the image up to that layer.
3. Back-propagate the activations of the filter back to the input image.
4. Multiply the gradients with learning rate and add them to the input image.
5. Repeat steps 2 to 4 until you are satisfied with the result.

Applying the algorithm iteratively on the output and applying some zooming after each iteration helps the network to generate an endless stream of new impressions by exploring the set of things that it knows about.

Let's deep dive into the code to generate a hallucinogenic dreamy image. We will apply the following settings to the various layers of the pre-trained VGG16 model available in Keras. Note that instead of using pre-trained we can apply this setting to a fresh neural network architecture of your choice as well:

```
settings_preset = {
    'dreamy': {
        'features': {
            'block5_conv1': 0.05,
            'block5_conv2': 0.02
        },
        'continuity': 0.1,
        'dream_l2': 0.02,
        'jitter': 0
    }
}

settings = settings_preset['dreamy']
```

This utility function basically loads, resizes, and formats images to an appropriate tensors format:

```
def preprocess_image(image_path):
    img = load_img(image_path, target_size=
(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg16.preprocess_input(img)
    return img
```

Then we calculate the continuity loss to give the image a local coherence and avoid messy blurs that look like a variant of the total variation loss discussed in the paper (http://www.robots.ox.ac.uk/~vedaldi/assets/pubs/mahendran15understan

```
def continuity_loss(x):
    assert K.ndim(x) == 4
    a = K.square(x[:, :img_height-1, :img_width-1, :]
-
                 x[:, 1:, :img_width-1, :])
    b = K.square(x[:, :img_height-1, :img_width-1, :]
-
                 x[:, :img_height-1, 1:, :])

# (a+b) is the squared spatial gradient, 1.25 is a
hyperparameter # that should be >1.0 as discussed in
the aforementioned paper
    return K.sum(K.pow(a+b, 1.25))
```

Next, we will load the VGG16 model with pretrained weights:

```
model = vgg16.VGG16(input_tensor=dream,
weights='imagenet', include_top=False)
```

After that we will add the `l2` norm of the features of a layer to the `loss` and then add continuity loss to give the image local coherence followed by adding again `l2` norm to loss in order to prevent pixels from taking very high values and then compute the gradients of the dream with respect to the loss:

```
loss += settings['continuity'] *
continuity_loss(dream) / np.prod(img_size)
loss += settings['dream_l2'] * K.sum(K.square(dream))
/ np.prod(img_size)
grads = K.gradients(loss, dream)
```

Finally, we will add a `random_jitter` to the input image and run the `L-BFGS` optimizer over the pixels of the generated image to minimize the loss:

```
random_jitter = (settings['jitter']*2) *
(np.random.random(img_size)-0.5)
x += random_jitter
```

```
# run L-BFGS
x, min_val, info = fmin_l_bfgs_b(evaluator.loss,
x.flatten(),
                                 fprime=evaluator.grads,
maxfun=7)
```

At the end, we will decode the dream and save it in an output image file:

```
x = x.reshape(img_size)
x -= random_jitter
img = deprocess_image(np.copy(x))
fn = result_prefix + '_at_iteration_%d.png' % i
imsave(fn, img)
```

The dreamy output generated just after five iterations is shown as follows:

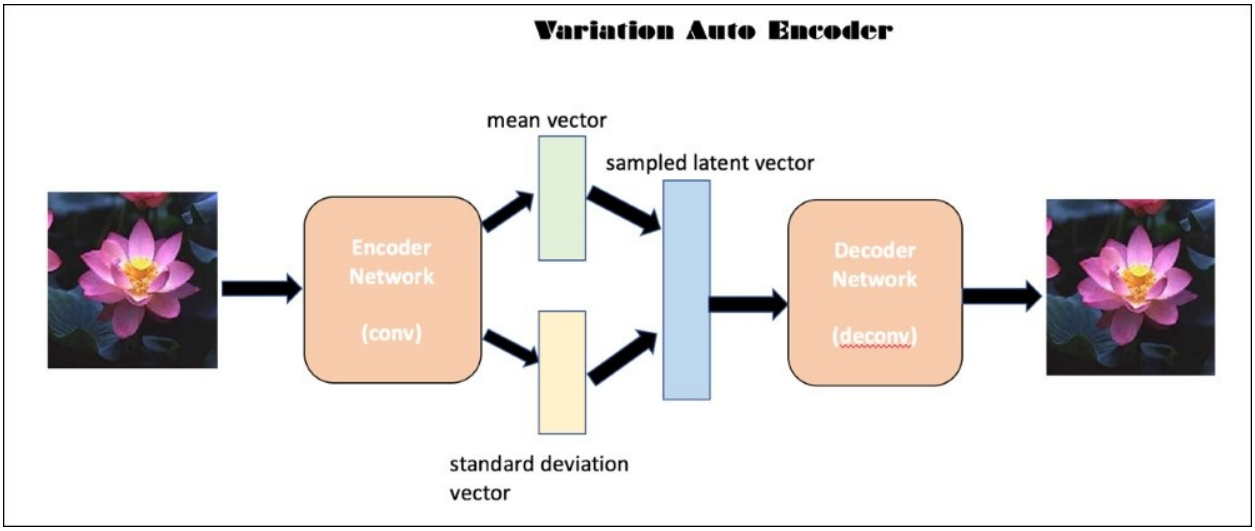
Figure-6: Left showing the original input image and right showing the dreamy artistic image created by deep dream

Generating handwritten digits with VAE using TensorFlow

The **Variational Autoencoder** (**VAE**) nicely synthesizes unsupervised learning with variational Bayesian methods into a sleek package. It applies a probabilistic turn on the basic autoencoder approach by treating inputs, hidden representations, and reconstructed outputs as probabilistic random variables within a directed graphical model.

From Bayesian perspective, the encoder becomes a variational inference network, that maps the observed inputs to posterior distributions over latent space, and the decoder becomes a generative network that maps the arbitrary latent coordinates back to distributions over the original data space.

VAE is all about adding a constraint on the encoding network that generates latent vectors that roughly follow a unit Gaussian distribution (this constraint that separates a VAE from a standard autoencoder) and then reconstructs the image back by passing the latent vector through the decoder network:



A real world analogy of VAE

Let's say we want to generate data (an animal) and a good way of doing it is to first decide what kind of data we want to generate, before actually generating the data. So, we must imagine some criteria about representing the animal, like it should have four legs and be able to swim. Once we have those criteria, we can then generate the animal by sampling from the animal kingdom. Our imagination criteria are analogous to latent variables. Deciding the latent variable in the first place helps to describe the data well, otherwise it is like generating data blindly.

The basic idea of VAE is to infer *p(z)* using *p(z|x)*. Let's now expand with some mathematical notation:

- *x*: Represents the data (that is, animal) we want to generate
- *z*: Represents the latent variable (that is, our imagination)
- *p(x)*: Represents the distribution of data (that is, animal kingdom)
- *p(z)*: Represents the normal probability distribution of the latent variable (that is, the source of imagination—our brain)
- *p(x|z)*: Probability distribution of generating data given the latent variable (that is, turning imagination into a realistic animal)

As per the analogous example, we want to restrict our imagination only on the animal kingdom domain, so that we shouldn't imagine about things such as root, leaf, money, glass, GPU, refrigerator, carpet as it's very unlikely that those things have anything in common with the animal kingdom.

The `loss` function of the VAE is basically the negative log-likelihood with a regularizer as follows:

$$l_i(\theta,\varnothing) = -E_{Z\sim q_\theta(Z/x_i)}\left[\log p_\varnothing(x_i/z)\right] + KL\left(q_\theta(z/x_i)\middle\|p(z)\right)$$

The first term is the expected negative log-likelihood or reconstruction loss of the ith data point where the expectation is calculated with respect to the encoder's distribution over the representations. This term helps the decoder to reconstruct the data well and incur large costs if it fails to do so. The second term represents the Kullback-Leibler divergence between encoder distribution *q(z|x)* and *p(z)* and acts as a regularizer that adds a penalty to the loss when an encoder's output representation *z* differs from normal distribution.

Now let's dive deep into the code for generating handwritten digits from

the MNIST dataset with VAE using TensorFlow. First let us create the encoder network *Q(z|X)* with a single hidden layer that will take *X* as input and output *μ(X)* and *Σ(X)* as part of Gaussian distribution:

```
X = tf.placeholder(tf.float32, shape=[None, X_dim])
z = tf.placeholder(tf.float32, shape=[None, z_dim])

Q_W1 = tf.Variable(xavier_init([X_dim, h_dim]))
Q_b1 = tf.Variable(tf.zeros(shape=[h_dim]))

Q_W2_mu = tf.Variable(xavier_init([h_dim, z_dim]))
Q_b2_mu = tf.Variable(tf.zeros(shape=[z_dim]))
Q_W2_sigma = tf.Variable(xavier_init([h_dim, z_dim]))
Q_b2_sigma = tf.Variable(tf.zeros(shape=[z_dim]))


def Q(X):
    h = tf.nn.relu(tf.matmul(X, Q_W1) + Q_b1)
    z_mu = tf.matmul(h, Q_W2_mu) + Q_b2_mu
    z_logvar = tf.matmul(h, Q_W2_sigma) + Q_b2_sigma
    return z_mu, z_logvar
```

Now we create the decoder network *P(X|z)*:

```
P_W1 = tf.Variable(xavier_init([z_dim, h_dim]))
P_b1 = tf.Variable(tf.zeros(shape=[h_dim]))

P_W2 = tf.Variable(xavier_init([h_dim, X_dim]))
P_b2 = tf.Variable(tf.zeros(shape=[X_dim]))


def P(z):
    h = tf.nn.relu(tf.matmul(z, P_W1) + P_b1)
    logits = tf.matmul(h, P_W2) + P_b2
    prob = tf.nn.sigmoid(logits)
    return prob, logits
```

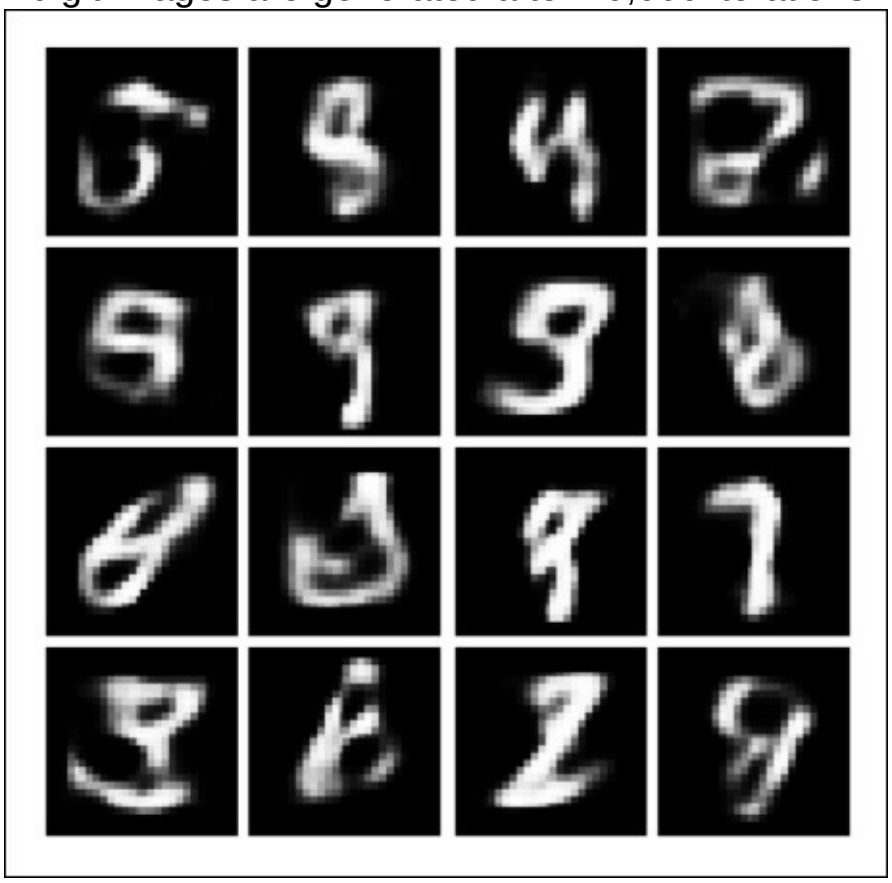
Then we calculate the reconstruction loss and Kullback-Leibler divergence loss and sum them up to get the total loss of the VAE network:

```
recon_loss =
tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(
logits=logits, labels=X), 1)
kl_loss = 0.5 * tf.reduce_sum(tf.exp(z_logvar) +
z_mu**2 - 1. - z_logvar, 1)
# VAE loss
vae_loss = tf.reduce_mean(recon_loss + kl_loss)
```

And then use an `AdamOptimizer` to minimize the loss:

```
solver = tf.train.AdamOptimizer().minimize(vae_loss)
```

Run the file (`VAE.py` or `VAE.ipynb`) to start VAE operations on an `MNIST` dataset and the images will be generated in the output folder. The sample handwritten digit images are generated after 10,000 iterations:



A comparison of two generative models—GAN and VAE

Although both are very exciting approaches of the generative model and have helped researchers to make inroads into the unsupervised domain along with generating capability, these two models differ in how they are trained. GAN is rooted in game theory, with an objective to find the nash equilibrium between the discriminator network and generator network. Whereas VAE is basically a probabilistic graphical model rooted in Bayesian inference, whose goal is latent modeling, that is, it tries to model the probability distribution of underlying data, in order to sample new data from that distribution.

VAE has a clear known way of evaluating the quality of the model (such as log-likelihood, either estimated by importance sampling or lower-bounded) compared to GAN, but the problem with VAE is that it uses direct mean squared error in the calculation of latent loss, instead of an adversarial network, so they over simplify the objective task as they are bound to work in a latent space and as a result they often generate blured images compared to GAN.

Summary

Transfer Learning addresses the problem of dealing with small data effectively without re-inventing the training wheel from scratch. You have learned to extract and transfer features from a pre-trained model and apply it to your own problem domain. Also, you have mastered training and running deeper models over a large scale distributed system using Spark and its related components. Then, you have generated a realistic high resolution image leveraging the power of Transfer Learning within SRGAN. Also, you have mastered the concepts of other generative model approaches such as VAE and DeepDream for artistic image generation. In the last chapter, we will shift our focus from training deep models or generative models and learn various approaches of deploying your deep learning-based applications in production.

Chapter 6. Taking Machine Learning to Production

A lot of machine learning and deep learning tutorials, text books, and videos focus on the training and evaluation of models only. But how do you take your trained model to production and use it for real-time scenarios or make it available to your customers?

In this chapter, you will develop a facial image correction system using the `LFW` dataset to automatically correct corrupted images using your trained GAN model. Then, you will learn several techniques to deploy machine learning or deep learning models in production, both on data centers and clouds with microservice-based containerized environments. Finally, you will learn a way to run deep models in a serverless environment and with managed cloud services.

We will cover the following topics in this chapter:

- Building an image correction system using DCGAN
- The challenges of deploying machine learning models
- Microservice architecture with containers
- Various approaches to deploying deep learning models
- Serving Keras-based deep models on Docker
- Deploying deep models on the cloud with GKE
- Serverless image recognition with audio using AWS Lambda and Polly
- Running face detection with a cloud managed service

Building an image correction system using DCGAN

Image correction and inpainting are related technologies used for filling in or completing missing or corrupted parts of images. Building a system that can |fill in the missing pieces broadly requires two pieces of information:

- **Contextual information**: Helps to infer missing pixels based on information provided by the surrounding pixels
- **Perceptual information**: Helps to interpret the filled/completed portions as being normal, as seen in real life or other pictures

In this example, we will develop an image correction or completion

system with the **Labeled Face in the Wild** (`LFW`) dataset using DCGAN. Refer to Chapter 2, *Unsupervised Learning with GAN*, for DCGAN and its architecture.

Let's define some notation and `loss` function before diving into the steps for building an image correction system:

- *x*: Corrupted image.
- *M*: Represents a binary mask that has a value of either 1 (meaning the part of the image we want to keep) or 0 (meaning the part of the image we want to complete/correct). The element-wise multiplication between the two matrices *x* and *M* represented by $\mathrm{M} \odot \mathrm{x}$ returns the original part of the image.
- *pdata*: The unknown distribution of sampled data.

Once we have trained the discriminator *D(x)* and generator *G(z)* of DCGAN, we can leverage it to complete missing pixels in an image, *x*, by maximizing *D(x)* over those missing pixels.

Contextual loss penalizes *G(z)* for not creating a similar image for the known pixel location in the input image by element-wise subtracting the pixels in *x* from *G(z)* and finding the difference between them:

$$\mathrm{contextual}(z) = \left\| \mathrm{M} \odot \mathrm{G}(z) - \mathrm{M} \odot \mathrm{x} \right\|$$

Perceptual loss has the same criterion used in training DCGAN to make sure that the recovered image looks real:

$$\mathrm{perceptual}(z) = \log\left(1 - \mathrm{D}\left(\mathrm{G}(z)\right)\right)$$

Next, we need to find an image from the generator, *G(z)*, that provides a reasonable reconstruction of the missing pixels. Then, the completed pixels $(1-\mathrm{M}) \odot \mathrm{G}(z)$ can be added to the original pixels to generate the reconstructed image:

$$\mathrm{xreconstructed} = \mathrm{M} \odot \mathrm{x} + (1-\mathrm{M}) \odot \mathrm{G}(z)$$

> **Tip**
> Training a deep convolutional network over a CPU may be prohibitively slow, so it is recommended to use a CUDA-enabled GPU for deep learning activities involving images with convolution or transposed convolution.

Steps for building an image correction

system

Make sure you have downloaded the code for this chapter:

1. The `DCGAN-ImageCorrection` project will have the following directory structure:

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DCGAN-ImageCorrection$ ls -lrth
total 544K
-rwxr-xr-x    1 ubuntu ubuntu 1.2K Aug  9 19:57 utils.py
-rwxr-xr-x    1 ubuntu ubuntu 7.8K Aug  9 19:57 dcgan.py
drwxr-xr-x    2 ubuntu ubuntu 4.0K Aug 27 15:30 complete
drwxr-xr-x    2 ubuntu ubuntu 4.0K Aug 27 15:36 complete_src
drwxrwxr-x 5751 ubuntu ubuntu 200K Aug 31 13:56 lfw
-rwxr-xr-x    1 ubuntu ubuntu 1.2K Aug 31 13:57 create_tfrecords.py
drwxr-xr-x    2 ubuntu ubuntu 288K Aug 31 13:58 data
-rwxrwxr-x    1 ubuntu ubuntu   15 Aug 31 16:02 test.sh
drwxr-xr-x    2 ubuntu ubuntu 4.0K Sep 10 13:53 checkpoints
-rwxrwxr-x    1 ubuntu ubuntu 6.4K Sep 10 13:54 train_generate.py
-rwxrwxr-x    1 ubuntu ubuntu 5.8K Sep 10 13:55 image_correction.py
```

2. Now download the `LFW` dataset (aligned with deep funnelling) from http://vis-www.cs.umass.edu/lfw and extract its content under the `lfw` directory:

```
wget http://vis-www.cs.umass.edu/lfw/lfw-
funneled.tgz
tar -xvzf lfw-funneled.tgz
```

3. Next, execute `create_tfrecords.py` to generate the TensorFlow standard format from the `LFW` images. Modify the path of your `LFW` image location in the Python file:

```
base_path = <Path to lfw directory>
python create_tfrecords.py
```

   This will generate the `tfrecords` file in the `data` directory as follows:

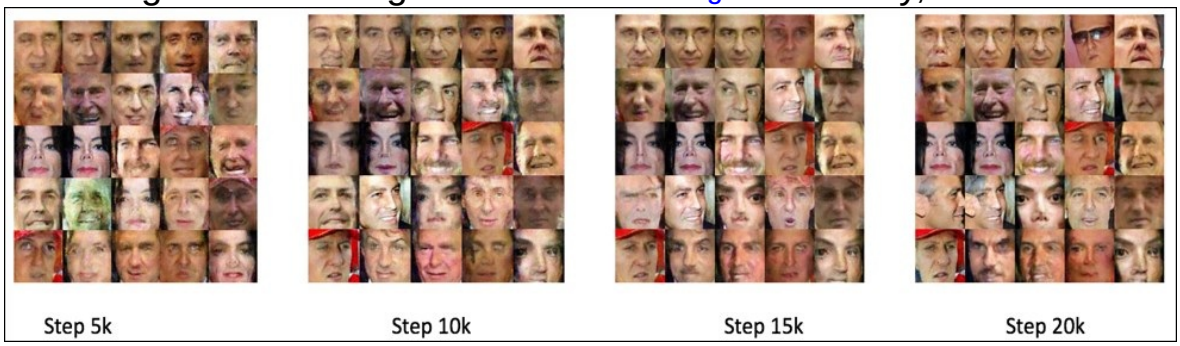
```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DCGAN-ImageCorrection$ ls data
Aaron_Eckhart.tfrecords
Aaron_Guiel.tfrecords
Aaron_Patterson.tfrecords
Aaron_Peirsol.tfrecords
Aaron_Pena.tfrecords
Aaron_Sorkin.tfrecords
Aaron_Tippin.tfrecords
Abba_Eban.tfrecords
Abbas_Kiarostami.tfrecords
Abdel_Aziz_Al-Hakim.tfrecords
Abdel_Madi_Shabneh.tfrecords
```

4. Now train the DCGAN model by executing the following command:

```
python train_generate.py
```

   You can modify the `max_itr` attribute in the Python files to determine

the maximum number of iterations the training should continue for. Once the training is going on, after every 5,000 iterations, you will find the generated images under the `lfw-gen` directory, as follows:



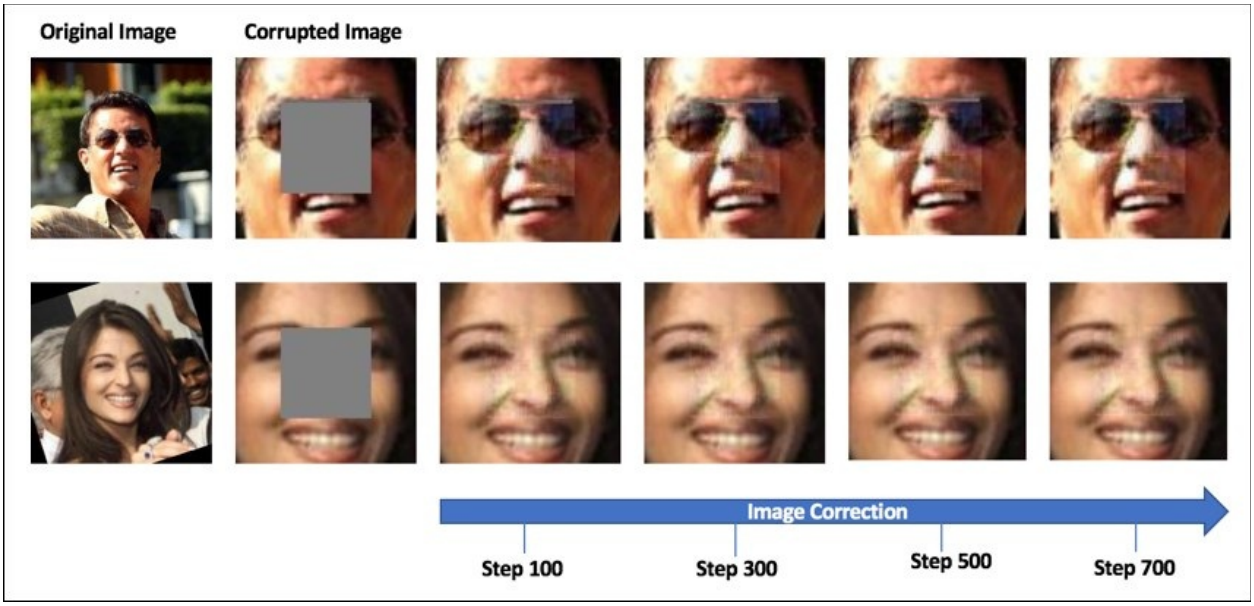
5. Finally, you can use the trained DCGAN model to correct corrupted images. You need to put your corrupted images under the `complete_src` directory and execute the following command:

```
python image_correction.py --is_complete True --
latest_ckpt  <checkpoint number>
```

You can also alter the type of masking by specifying `center` or `random` with the `masktype` attribute in the preceding command.

```
ubuntu@ip-172-31-6-47:~/software/kuntalg/DCGAN-ImageCorrection$ python image_correction.py --is_complet
e True --latest_ckpt 20000
2017-09-16 14:13:32.984245: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up
CPU computations.
2017-09-16 14:13:32.984376: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up
CPU computations.
2017-09-16 14:13:32.984400: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your machine and could speed up CPU
 computations.
Restoring variables:
 g/reshape/dense/kernel:0
 g/reshape/dense/bias:0
 g/reshape/batch_normalization/beta:0
 g/reshape/batch_normalization/gamma:0
 g/deconv1/conv2d_transpose/kernel:0
```

The preceding command will generate corrected or completed images under the complete directory, as follows:



Challenges of deploying models to production

Most researchers and machine learning practitioners focus on the training and evaluation side of machine learning or deep learning models. A real-world analogy of building models during research is similar to cooking at home, whereas building or deploying that model in production is like cooking for a wide variety of customers (whose taste changes over time) in a restaurant.

Some of the common challenges that often arise during the production deployment of models are as follows:

- **Scalability**: A real-world production environment is quite different from a training or research environment. You often need to cater for a high volume of requests without impacting the performance. Your model should automatically scale up/out based on the traffic and then scale down/in when the traffic is low.
- **Automated model training or updates**: Real-world data has temporal dynamics and, as your model enters the real-world production environment, the data starts looking different from that on which the model was originally trained. This means you need to retrain your model (sometimes automatically) and then switch between models seamlessly.
- **Interoperation between development languages**: Often, two different people or groups are responsible for researching (training) the model and productionizing it, and the language for research may

be different from the preferred language for production. This causes a bunch of problems, as machine learning models have different implementations in different programming languages, even though the model is essentially the same.

- **Knowledge of training set metadata**: Real-world production data might have missing values and you will need to apply a missing value imputation technique to deal with this. Although, in production systems, you don't keep information about training data, but in order to correctly impute the missing values arriving in production test samples, you have to store the knowledge of the training set statistics needed for imputation.
- **Real-time evaluation of model performance**: Evaluation of a model's performance in production often requires you to collect ground truth data (or other real-time metrics) and generate dynamic pages as a model processes more data. Also, you might need to carry out **A/B** testing by deploying two or more models serving the same functionality simultaneously to evaluate performance in production.

Microservice architecture using containers

In traditional **monolithic** architecture, an application puts all its functionality into single packages such as EAR or WAR and deploys it on an application server (such as JBoss, Tomcat, or WebLogic). Even though a monolithic application has separate and distinguishable components, all are packaged under one roof.

Drawbacks of monolithic architecture

Some of the common pitfalls of monolithic design are as follows:

- The functional components in monolithic architecture are packed under one application and are not isolated. Hence, changing a single component requires updating an entire application, thereby bringing the entire application to a halt. This is not desirable in a production scenario.
- Scaling a monolithic application is not efficient, because to scale you have to deploy each copy of the application (WAR or EAR) in various servers that will utilize the same amount of resources.
- Often, in the real world, one or two functional components are heavily used compared to other components. But in monolithic design, all components will utilize the same resources, hence it is hard to segregate highly used components to improve the performance of the overall application.

Microservices is a technique that decomposes large software projects into loosely coupled modules/services that communicate with each other through simple APIs. A microservices-based architecture puts each functionality into separate services to overcome the drawbacks of monolithic design.

Benefits of microservice architecture

Some of the advantages of microservice design pattern are as follows:

- **Single responsibility principle**: Microservice architecture makes sure that each functionality is deployed or exposed as a separate service through simple APIs.
- **High scalability**: Highly utilized or demanding services can be deployed in multiple servers to serve a high number of requests/traffic, thereby enhancing performance. This is difficult to

achieve with single, large monolithic services.

- **Improves fault tolerance**: The failure of single modules/services doesn't affect the larger application, and you can quickly recover or bring back the failed module, since the module is running as a separate service. Whereas a monolithic or bulky service having errors in one component/module can impact other modules/functionality.
- **Freedom of the technology stack**: Microservices allows you to choose the technology that is best suited for a particular functionality and helps you to try out a new technology stack on an individual service.

The best way to deploy microservices-based applications is inside containers.

Containers

Containers are shareable, lightweight processes that sit on top of a host operating system and share the kernels (binaries and libraries) of the host OS. Containers solve a bunch of complex problems simultaneously through a layer of abstraction. The popularity of containers can be described by the wonderful triad: *Isolation*! *Portability*! *Repeatability*!.

Docker

Docker is one of the hottest open source projects and is a very popular containerization engine that allows a convenient way to pack your service/application with all dependencies together to be deployed locally or in the cloud.

Kubernetes

Kubernetes is another open source project at Google and it provides orchestration to containers, allowing automated horizontal scaling, service discovery, load balancing, and much more. Simply put, it automates the management of your containerized applications/services in the cloud.

> **Note**
> We will refer to Docker as the container engine for illustration in this section, although other container engines would also provide similar features or functionality.

Benefits of using containers

Some of the pros of using container are discussed as follows:

- **Continuous deployment and testing**: Often, release life cycles involving different environments, such as development and production, have some differences because of different package versions or dependencies. Docker fills that gap by ensuring consistent environments by maintaining all configurations and dependencies internally from development to production. As a result, you can use the same container from development to production without any discrepancies or manual intervention.
- **Multi-cloud platforms**: One of the greatest benefits of Docker is its portability across various environments and platforms. All major cloud providers, such as **Amazon Web Services** (**AWS**) and **Google Compute Platform** (**GCP**), have embraced Docker's availability by adding individual support (AWS ECS or Google GKE). Docker containers can run inside a Virtual Machine VM instance (Amazon EC2 or Google Compute Engine) provided the host OS supports Docker.
- **Version control**: Docker containers work as a version control system just like `Git`/`SVN` repositories, so that you can commit changes to your Docker images and version control then.
- **Isolation and security**: Docker ensures that applications running inside containers are completely isolated and segregated from each other, granting complete control over traffic flow and management. No Docker container has access to the processes running inside another container. From an architectural standpoint, each container has its own set of resources.

You can combine an advanced machine learning or deep learning application with the deployment capabilities of containers to make the system more efficient and shareable.

Various approaches to deploying deep models

Machine learning is exciting and fun! It has its challenges though, both during the modeling part and also during deployment, when you want your model to serve real people and systems.
Deploying machine learning models into production can be done in a wide variety of ways and the different ways of productionalizing machine learning models is really governed by various factors:

- Do you want your model to be part of real-time streaming analytics or batch analytics?
- Do you want to have multiple models serving the same functionality or do you need to perform A/B testing on your models?
- How often do you want your model to be updated?
- How do you scale your model based on traffic?
- How do you integrate with other services or fit the ML service into a pipeline?

Approach 1 - offline modeling and microservice-based containerized deployment

In this approach, you will train and evaluate your model offline and then use your pretrained model to build a RESTful service and deploy it inside a container. Next, you can run the container within your data center or cloud depending on cost, security, scaling, and infrastructure requirement. This approach is well suited to when your machine learning or deep learning service will have continuous traffic flow and need to dynamically scale based on spikes in requests.

Approach 2 - offline modeling and serverless deployment

In this approach, you will train your model offline and deploy your service in a serverless environment such as AWS Lambda (where you will be charged only for invoking the API; you don't have to pay for running containers or VM instances on an hourly/minute basis). This approach is well suited to when your model service will not be used continuously but

will instead be invoked after a certain time. But even if there is continuous traffic flow (depending on the number of requests you hit), this approach might still be cost-effective compared to approach 1.

Approach 3 - online learning

Sometimes, you need to perform real-time streaming analytics by integrating your machine learning service with the pipeline (such as putting it at the consumer end of a message queue having IOT sensor data). Data might change very frequently in real-time streaming situations. In that scenario, offline model training is not the right choice. Instead, you need your model to adapt automatically to the data as it is sees it—that is it will update weights/parameters based on data using something like SGD or its mini batch variant.

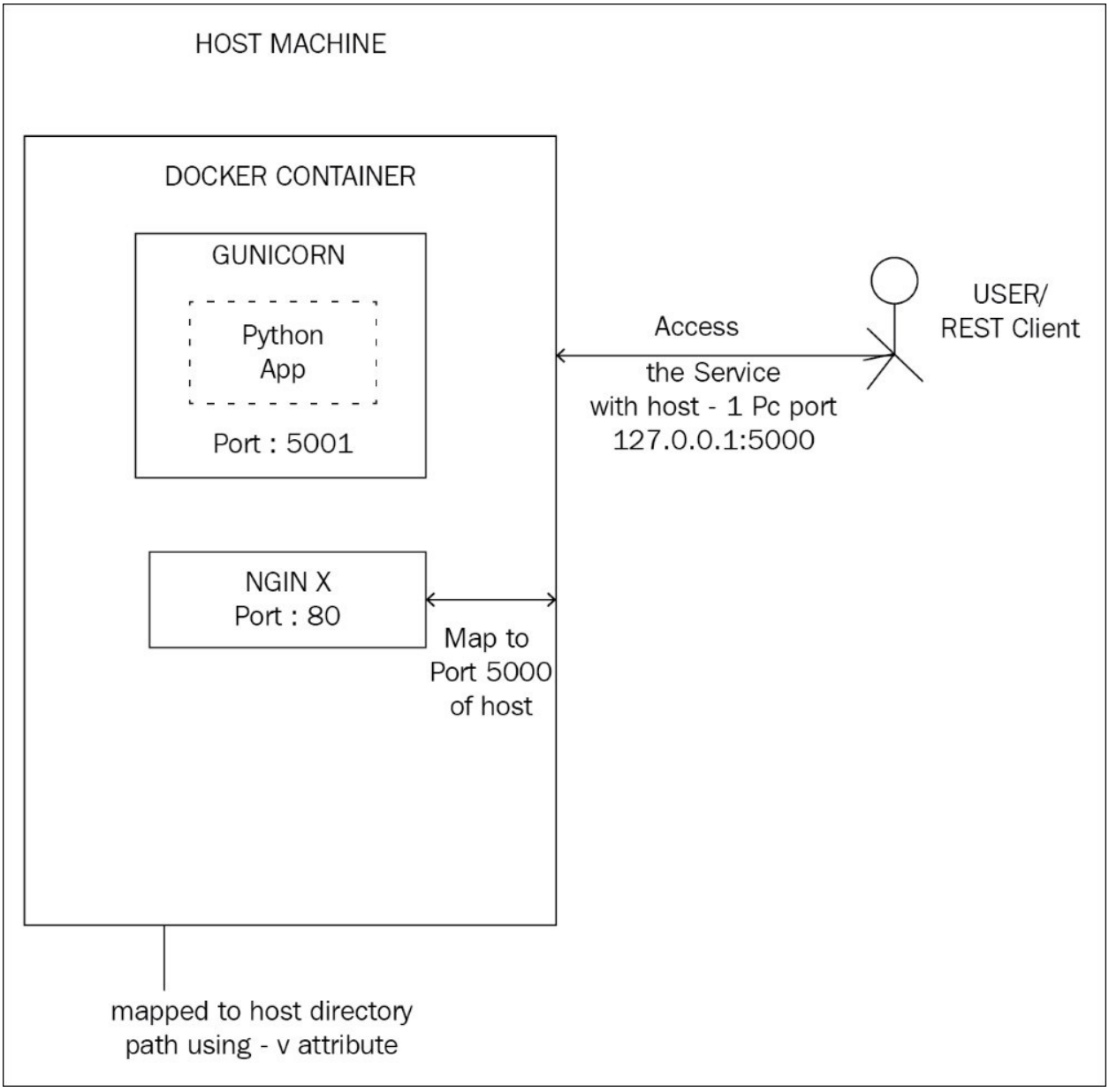
Approach 4 - using a managed machine learning service

This approach is well suited to when you don't have the resources or team members to build machine learning models in-house. Instead, you utilize the available cloud-based managed machine learning or deep learning services, such as Google Cloud ML, Azure ML, AWS Rekognition, AWS Polly, Google Cloud Vision, and so on to fulfill your requirements by invoking simple API calls.

Next, we will illustrate the deployment approaches mentioned previously through hands-on examples.

Serving Keras-based deep models on Docker

In this example, we will build an image identification system with a pretrained Keras InceptionV3 model and deploy it on a container in the local machine. Refer to Chapter 4, *Building Realistic Images from Your Text*, for more information about pretrained models. Our pretrained Keras model will run inside a Docker container exposed as a REST API using Flask.



Make sure you have the `keras-microservice` project available and then

perform the following steps to run a Keras-based deep model inside a `docker` container:

1. First, check that the Dockerfile is in your current working directory and then build a Docker image:

```
docker build -t keras-recognition-service .
```

2. Once the Docker image is built successfully, use the image to run a container with the `docker run` command:

```
docker run -it --rm -d -p <host port>:<container
port> -v <host path>:<container path> keras-
recognition-service
```

   For example:

```
docker run -it --rm -d -p 5000:80 -v
/Users/kuntalg/knowledge:/deep/model keras-
recognition-service
```

> **Note**
> Inside the `docker` container, the Keras model is running on port `5001` inside a WSGI HTTP Python server named **Gunicorn**, which is load balanced by an **Nginx** proxy server on port `80`. We used the `-p` attribute previously to map the host port with the container port. Also, we used the `-v` volume attribute to map the host path with the container path, so that we can load the pretrained model from this path.

   Now it's time to test our image identification service by executing the `test.sh` script. The script contains a `curl` command to call and test the REST API of our exposed image identification service:

```
#!/bin/bash
echo "Prediction for 1st Image:"
echo "-------------------------------"
(echo -n '{"data": "'; base64 test-1.jpg; echo
'"}') | curl -X POST -H "Content-Type:
application/json" -d @- http://127.0.0.1:5000

echo "Prediction for 2nd Image:"
echo "-------------------------------"
(echo -n '{"data": "'; base64 test-1.jpg; echo
'"}') | curl -X POST -H "Content-Type:
application/json" -d @- http://127.0.0.1:5000
```

3. Finally, execute the script to generate a prediction from our Keras service:

```
./test_requests.sh
```

```
a0999b1381a5:keras-microservice kuntalg$ ./test_requests.sh
Prediction for 1st Image:
---------------------------------
{
  "predictions": [
    {
      "description": "cheetah",
      "label": "n02130308",
      "probability": 61.97998523712158
    },
    {
      "description": "leopard",
      "label": "n02128385",
      "probability": 23.502658307552338
    },
    {
      "description": "jaguar",
      "label": "n02128925",
      "probability": 1.483460795134306
    }
  ]
}
Prediction for 2nd Image:
---------------------------------
{
  "predictions": [
    {
      "description": "macaw",
      "label": "n01818515",
      "probability": 69.18083429336548
    },
    {
      "description": "bee_eater",
      "label": "n01828970",
      "probability": 4.089190810918808
    },
    {
      "description": "lorikeet",
      "label": "n01820546",
      "probability": 1.0934238322079182
    }
  ]
}
```

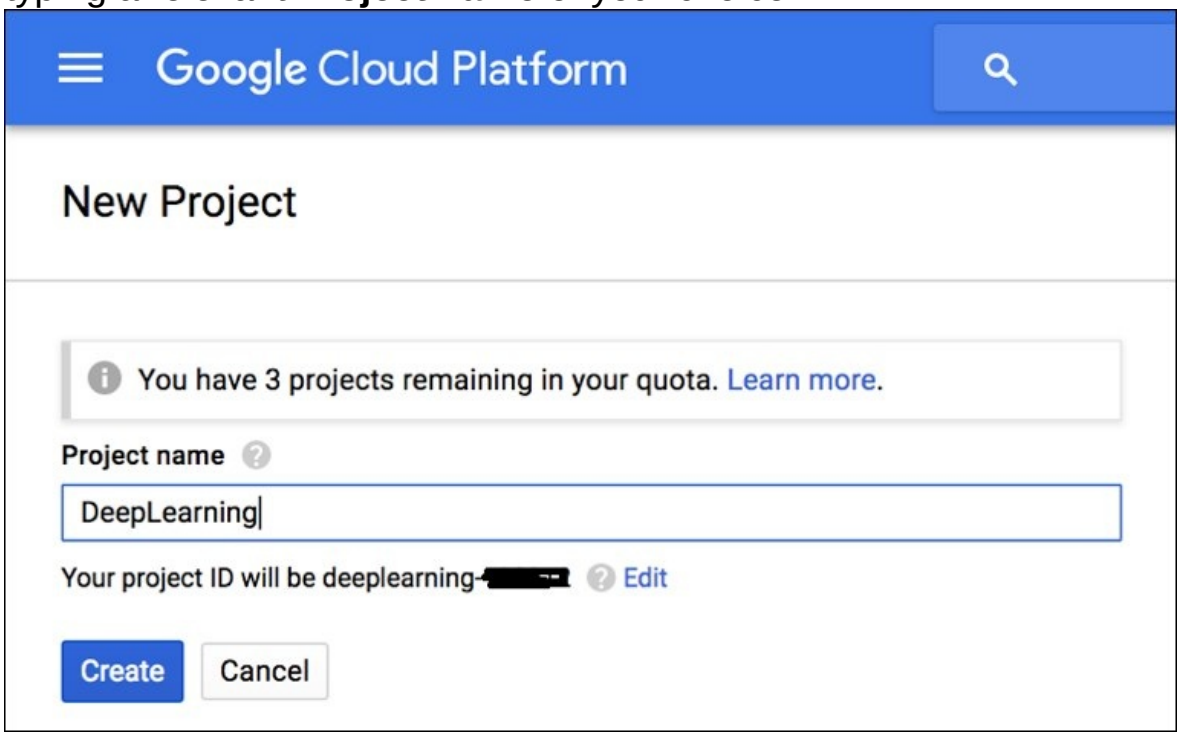
Voila! We have successfully deployed our first Keras-based deep learning model inside a container.

Deploying a deep model on the cloud with GKE

Once the deep learning model is created, deployed inside the container, and generating predictions locally, it's time to take the model to the cloud (for example, Google Cloud in this example) using Docker and Kubernetes.

Perform the following steps to take a locally created containerized model to the cloud:

1. Sign up for a Google Cloud trial account (https://cloud.google.com/free) and then create a **New Project** by typing a relevant **Project name** of your choice:



   Please note down the **project ID** containing your **Project name** along with some numeric digits of the format `<project name>-xxxxxx`. We will need the **project ID** later for deploying our local model to the c
2. d SDK on your machine (https://cloud.google.com/sdk). And then install kubectl to manage the Kubernetes cluster:

```
gcloud components install kubectl
```

   The `gcloud` command is included in the Google Cloud SDK.
3. Set some environment variables with the `gcloud` command-line tool

`config` command:

```
gcloud config set project <project ID>
gcloud config set compute/zone <zone name such as
us-central1-b>
export PROJECT_ID="$(gcloud config get-value
project -q)
```

4. Now build the docker image with a tag or version (`v1` in this example):

```
docker build -t gcr.io/<project ID>/keras-
recognition-service:v1 .
```

   For example:

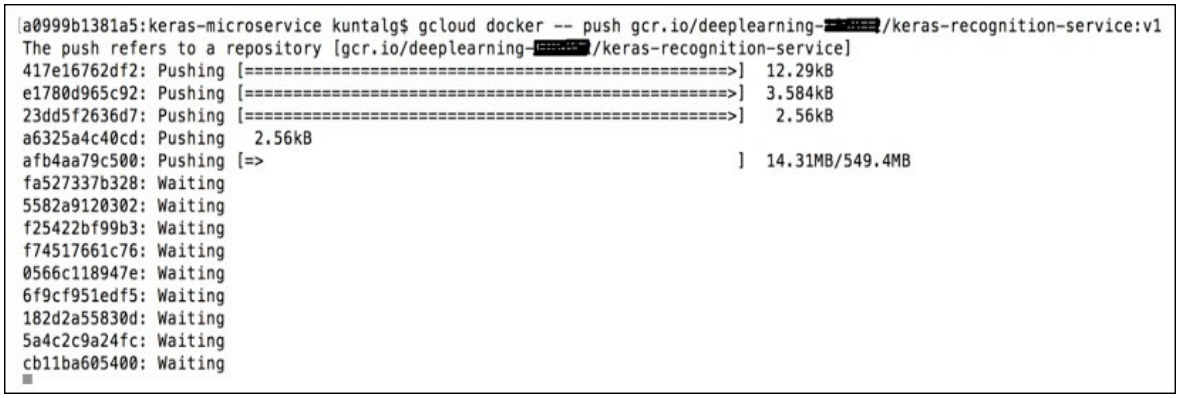
```
docker build -t gcr.io/deeplearning-123456/keras-
recognition-service:v1 .
```

5. Next, upload the image built previously with the `docker push` command to the Google Container Registry:

```
gcloud docker -- push gcr.io/<project ID>/keras-
recognition-service:v1
```

   For example:

```
gcloud docker -- push gcr.io/deeplearning-
123456/keras-recognition-service:v1
```



6. Once the container image is stored in a registry, we need to create a container cluster by specifying the number of Compute Engine VM instances. This cluster will be orchestrated and managed by Kubernetes. Execute the following command to create a two-node cluster named `dl-cluster`:

```
gcloud container clusters create dl-cluster --num-
nodes=2
```

7. We will use the Kubernetes `kubectl` command-line tool to deploy and run an application on a Container Engine cluster, listening on port `80`:

```
gcloud container clusters get-credentials dl-
cluster

kubectl run keras-recognition-service --
image=gcr.io/deeplearning-123456/keras-
recognition-service:v1 --port 80
```

```
a0999b1381a5:keras-microservice kuntalg$ kubectl run keras-recognition-service --image=gcr.io/deeplearning-[redacted]/keras-recogni
tion-service:v1 --port 80
```

8. Now attach the application running inside the container cluster to the load balancer, so that we can expose our image identification service to a real- world user:

```
kubectl expose deployment keras-recognition-
service --type=LoadBalancer --port 80 --target-
port 80
```

9. Next, run the following `kubectl` command to get the external IP of our service:

```
kubectl get service
```

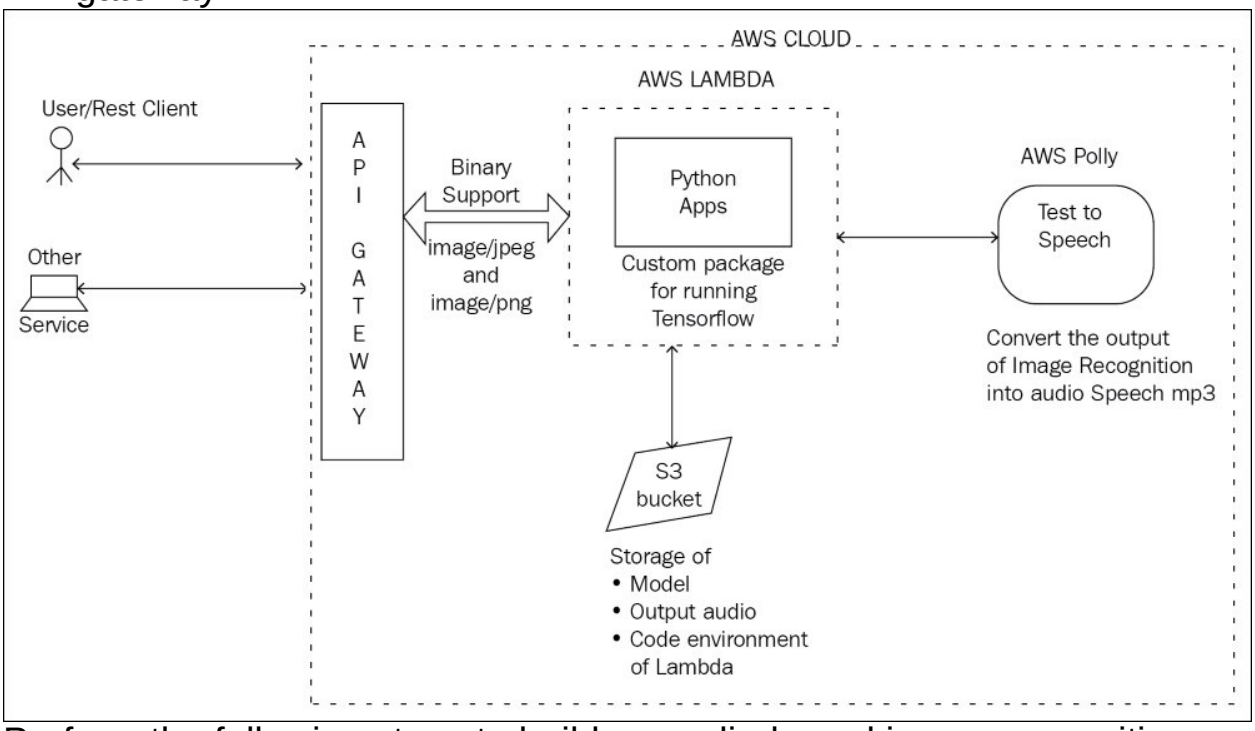
10. Finally, execute the following command to get a prediction from our image recognition service hosted inside a container cluster hosted on the cloud:

```
(echo -n '{"data": "'; base64 test-1.jpeg; echo
'"}') | curl -X POST -H "Content-Type:
application/json" -d @- http://<External IP>:80
```

```
a0999b1381a5:keras-microservice kuntalg$ (echo -n '{"data": "'; base64 test-1.jpg; echo '"}') | curl -X POST -H "Content-Type:
application/json" -d @- http://[redacted]:80
{
  "predictions": [
    {
      "description": "cheetah",
      "label": "n02130308",
      "probability": 61.979931592941284
    },
    {
      "description": "leopard",
      "label": "n02128385",
      "probability": 23.502694070339203
    },
    {
      "description": "jaguar",
      "label": "n02128925",
      "probability": 1.4834615401923656
    }
  ]
```

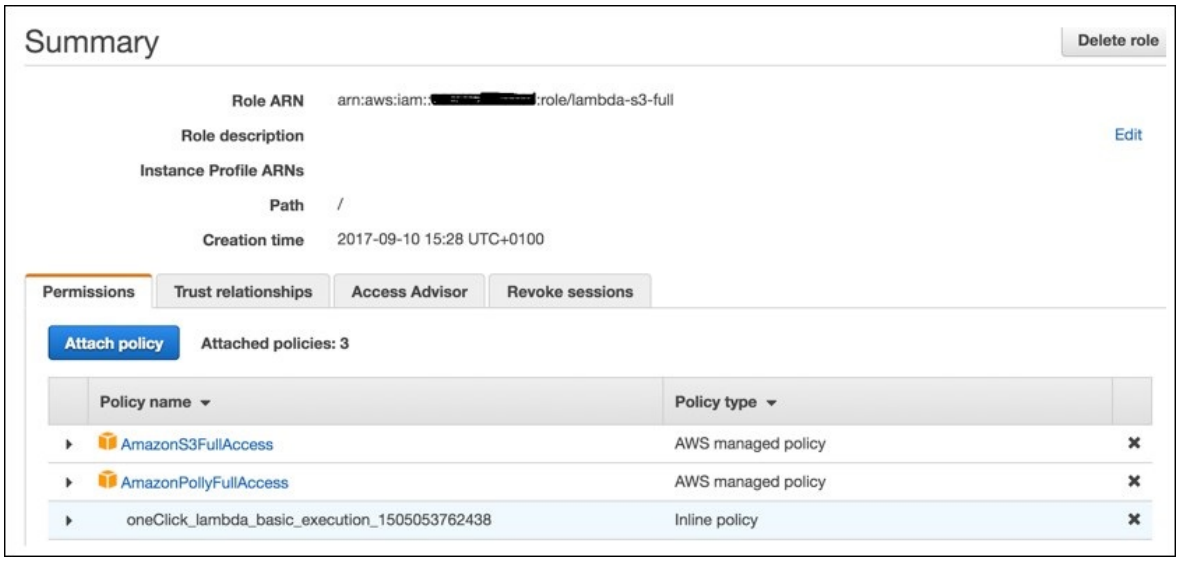
Serverless image recognition with audio using AWS Lambda and Polly

In this example, we will build an audio-based image prediction system using TensorFlow pretrained InceptionV3 model and deploy it on a serverless environment of AWS Lambda. We will run our image prediction code on AWS Lambda and load our pretrained model from S3, and then expose the service to our real-world customer through the AWS API gateway.

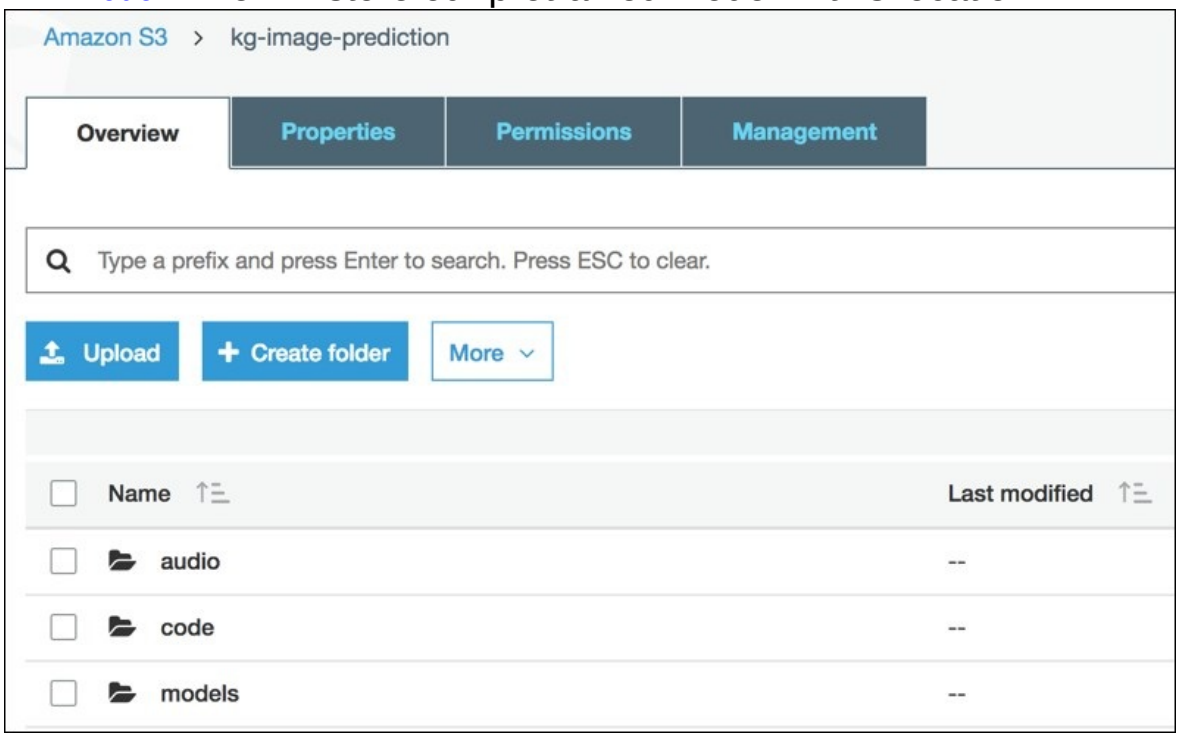


Perform the following steps to build an audio-based image recognition system on a serverless platform:

1. Sign up for an AWS trial account (https://aws.amazon.com/free/) and navigate to the **IAM** service to create a new role for AWS Lambda. Attach two new managed policies: **S3FullAccess** and **PollyFullAccess** alongside the inline policy of **lambda_basic_execution**.



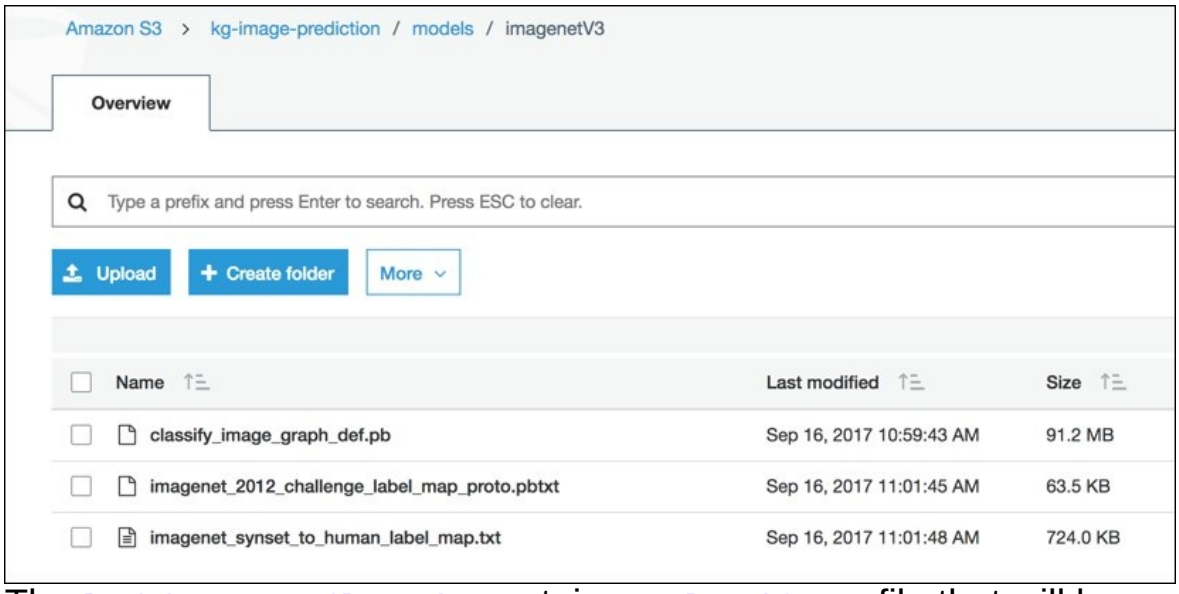
2. Next, create an S3 bucket, where we will store our lambda code (consisting of custom Python packages such as `numpy`, `scipy`, `tensorflow`, and so on). Also create three folders within your S3 bucket:
    - `code`: We will store our code for the lambda environment here
    - `audio`: Our prediction audio will be saved in this location
    - `model`: We will store our pretrained model in this location



3. Download the pretrained TensorFlow model (http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz), extract it, and then upload the following files to the

`S3` bucket under the `model` directory:

```
classify_image_graph_def.pb
imagenet_synset_to_human_label_map.txt
imagenet_synset_to_human_label_map.txt
```

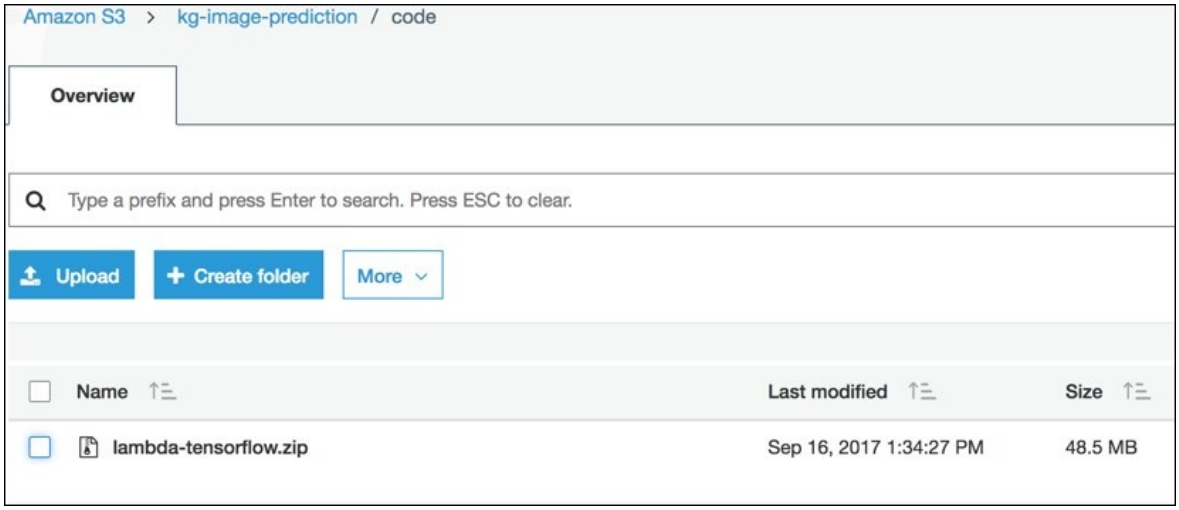


4. The `lambda_tensorflow.zip` contains a `classify.py` file that will be invoked during the `lambda` function execution. Change the bucket name, and inside the `classify.py`, zip it again and upload it to the S3 bucket under the `code` directory:

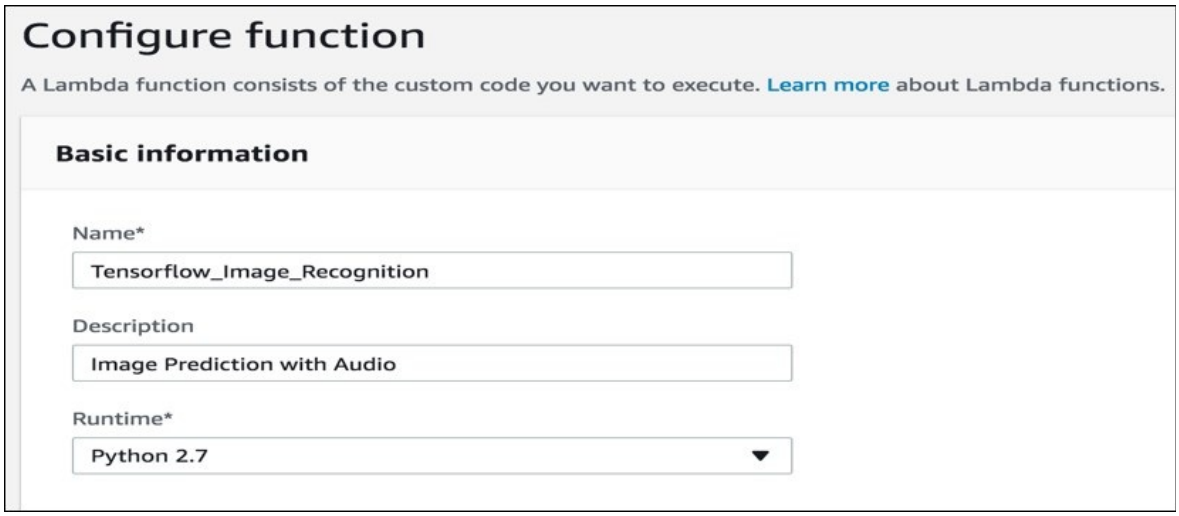
```
def lambda_handler(event, context):

    if not os.path.exists('/tmp/imagenetV3/'):
        os.makedirs('/tmp/imagenetV3/')

    # imagenet_synset_to_human_label_map.txt:
    #   Map from synset ID to a human readable
string.
    strBucket = 'kg-image-prediction'
    strKey =
'models/imagenetV3/imagenet_synset_to_human_label_
map.txt'
    strFile =
'/tmp/imagenetV3/imagenet_synset_to_human_label_ma
p.txt'
    downloadFromS3(strBucket,strKey,strFile)
    print(strFile)

    #
imagenet_2012_challenge_label_map_proto.pbtxt:
    #   Text representation of a protocol buffer
mapping a label to synset ID.
```

```
    strBucket = 'kg-image-prediction'
    strKey =
'models/imagenetV3/imagenet_2012_challenge_label_m
ap_proto.pbtxt'
    strFile =
'/tmp/imagenetV3/imagenet_2012_challenge_label_map
_proto.pbtxt'
    downloadFromS3(strBucket,strKey,strFile)
    print(strFile)

    # classify_image_graph_def.pb:
    #   Binary representation of the GraphDef
protocol buffer.
    strBucket = 'kg-image-prediction'
    strKey =
'models/imagenetV3/classify_image_graph_def.pb'
    strFile =
'/tmp/imagenetV3/classify_image_graph_def.pb'
    downloadFromS3(strBucket,strKey,strFile)
    print(strFile)
    data = base64.b64decode(event['base64Image'])
    imageName= event['imageName']

    image=io.BytesIO(data)
    strBucket = 'kg-image-prediction'

    strKey = 'raw-
image/tensorflow/'+imageName+'.png'
    uploadToS3(image, strBucket, strKey)
    print("Image file uploaded to S3")


    audioKey=imageName+'.mp3'
    print(audioKey)
    print("Ready to Run inference")

    strBucket = 'kg-image-prediction'
    strKey = 'raw-
image/tensorflow/'+imageName+'.png'
    imageFile = '/tmp/'+imageName+'.png'
    downloadFromS3(strBucket,strKey,imageFile)
    print("Image downloaded from S3")

    strResult = run_inference_on_image(imageFile)

    # Invoke AWS Polly to generate Speech from
text
    polly_client=boto3.client('polly')
    response = polly_client.synthesize_speech(Text
```

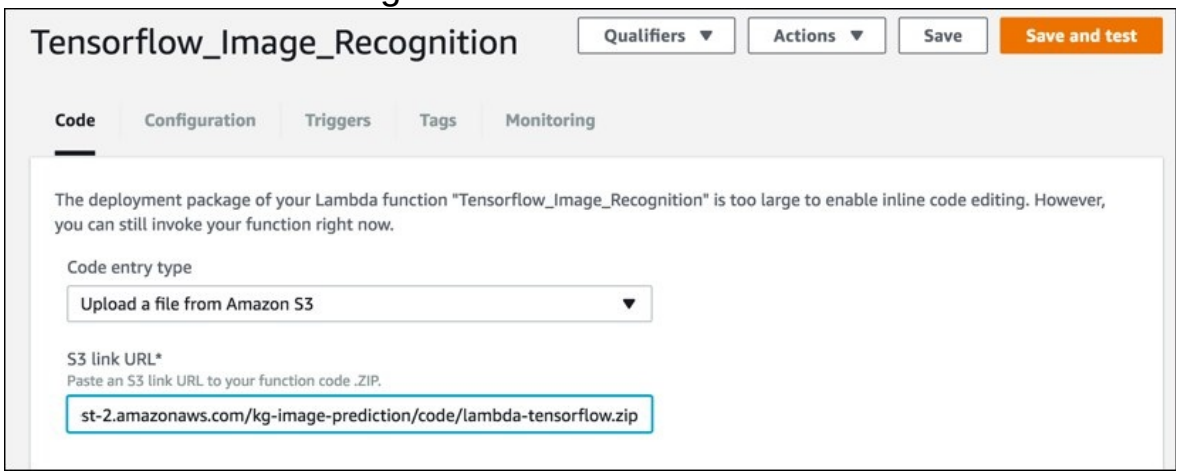
```
=strResult,OutputFormat = "mp3",VoiceId =
"Joanna")
    if "AudioStream" in response:
        output = os.path.join("/tmp", audioKey)
        with open(output, "wb") as file:

file.write(response["AudioStream"].read())

    #Upload speech to S3
    print("Ready upload to S3 audio")
    strBucket = 'kg-image-prediction'
    strKey = 'audio/'+audioKey
    strFile = '/tmp/'+audioKey

    with open(strFile, 'rb') as data:
        uploadToS3(data,strBucket,strKey)
    # Clean up directory
    os.remove(imageFile)
    os.remove(strFile)


    return strResult
```



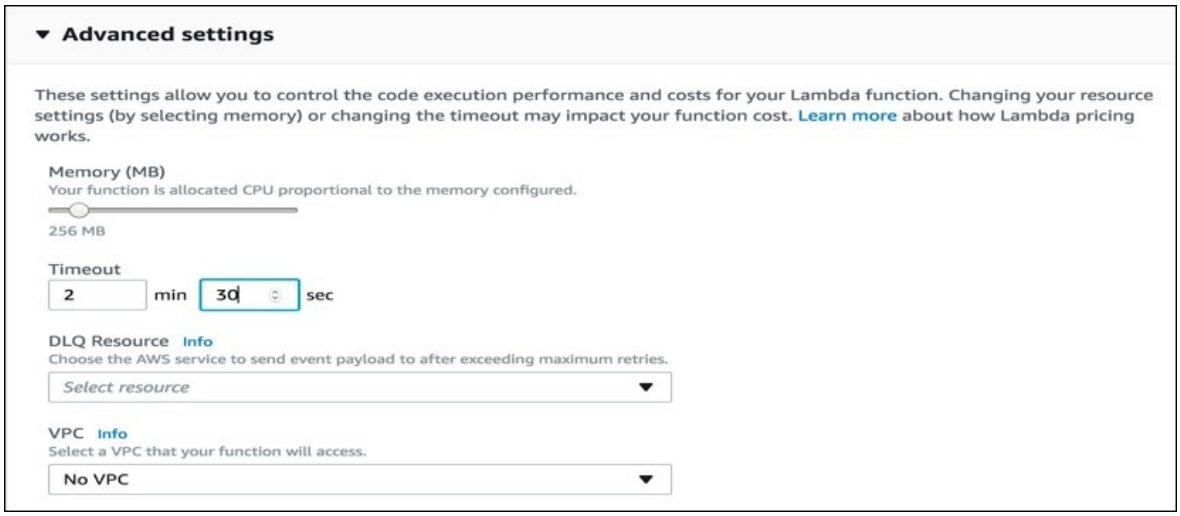
5. Now navigate to the **Lambda** service from the web console to create a new Lambda function from scratch. Provide **Name*** and **Description** for the function; choose **Runtime** as **Python 2.7** and attach the `IAM` role created previously to this Lambda function.



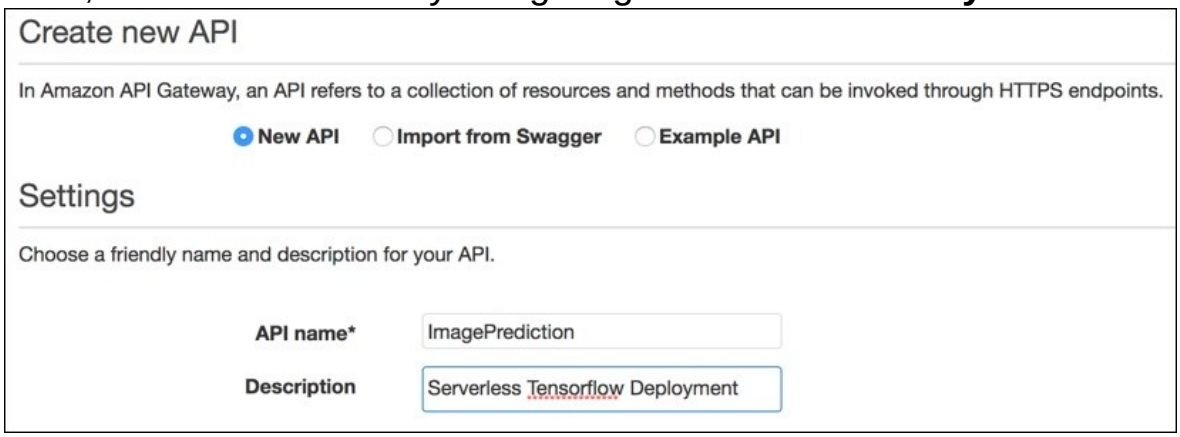
6. And then specify the code (`lambda_tensorflow.zip`) location in your Lambda function configuration:



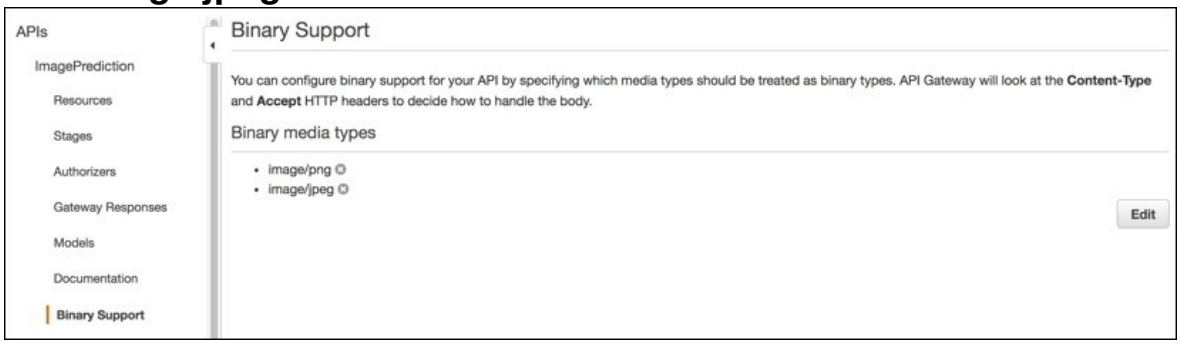
7. Also increase the **Memory(MB)** and **Timeout** of your Lambda function under the **Advance Settings** tab. The first time, the lambda execution will take some time due to the loading of the pretrained model from S3.



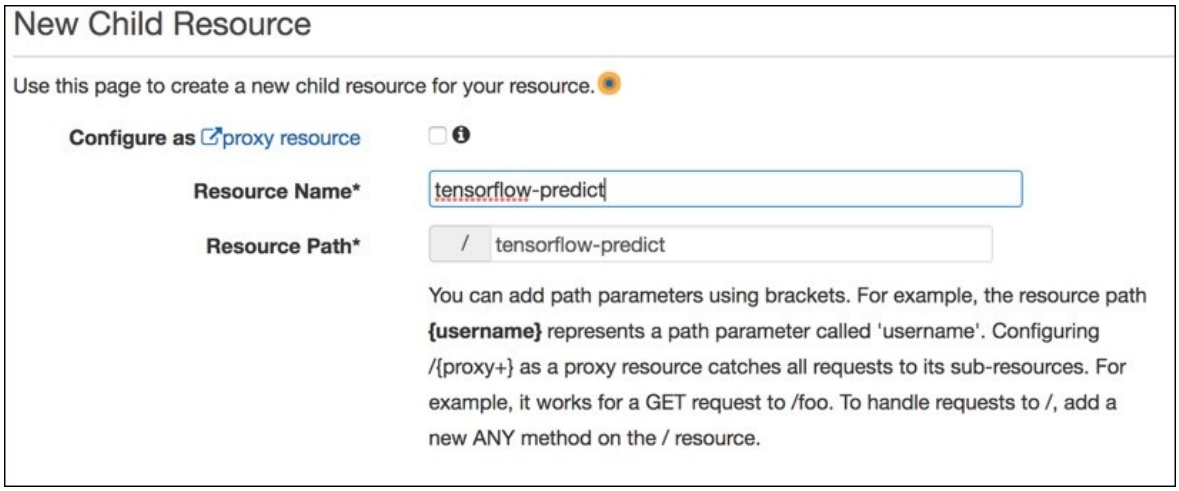
8. Next, create a new API by navigating to the **API Gateway** service:



9. Then, click the **Binary Support** tab on the left panel of your API to add the following content type:
   - **image/png**
   - **image/jpeg**

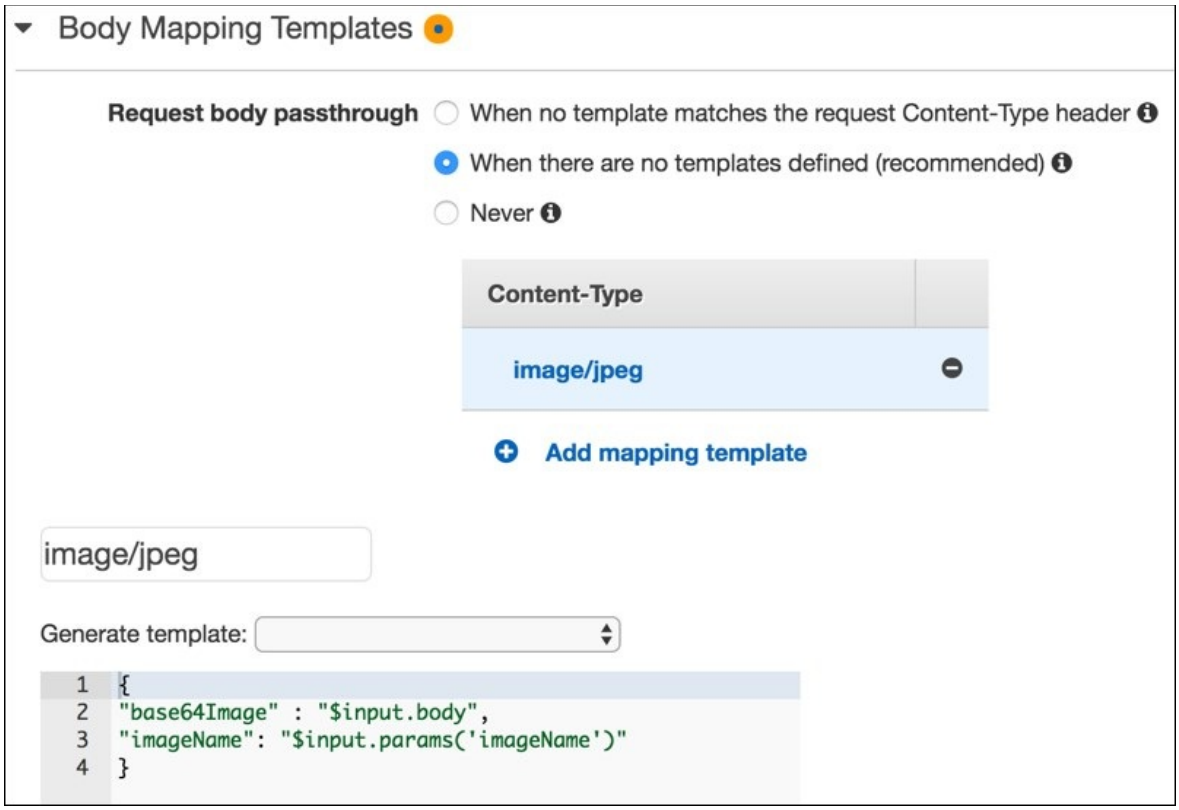


10. Next, create a **New Child Resource** by specifying the **Resource Path** (for example, `tensorflow-predict`):



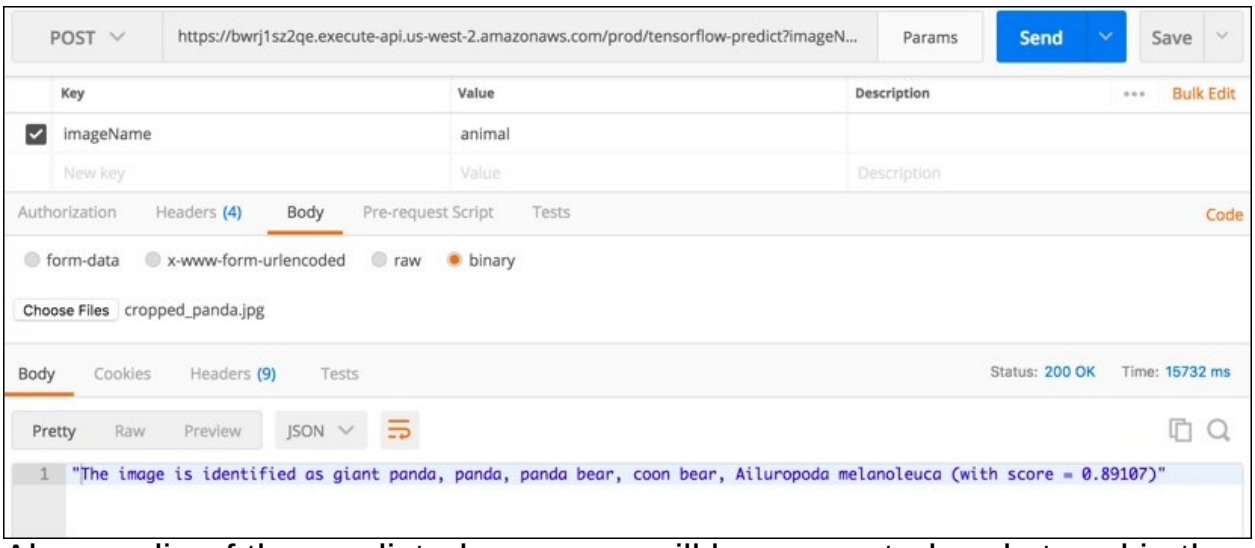
11. Next, add a method (**POST**) to your child resource by clicking **Create Method** from the **Action** menu. Add the Lambda function we created previously to AMP with this API resource. You may need to specify the correct region to find your Lambda function from the dropdown.
12. Once the **POST** method is created, click on **Integration Request** and expand the **Body Mapping Templates** tab. Under **Request body passthrough**, choose **When there are no templates defined (recommended)**. Then, add an image/jpeg under **Content-Type** and add the following under the **Generate template** section:

```
{
"base64Image": "$input.body",
"imageName": "$input.params(imageName)"
}
```

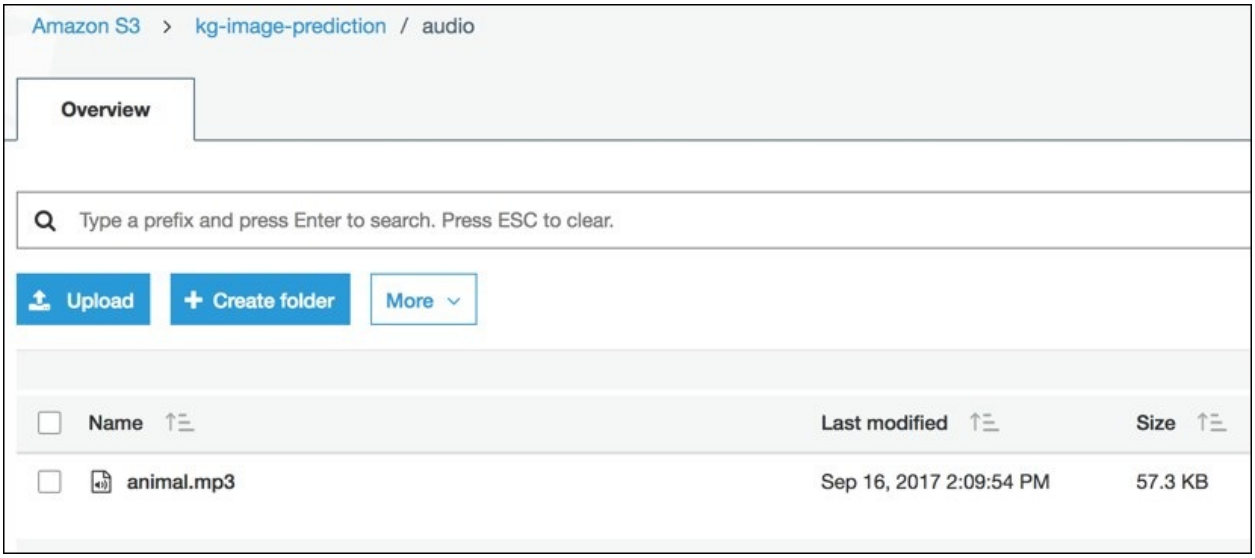


13. Finally, deploy the API from the **Action** menu and define the **Stage name** (for example, `prod` or `dev`). Once your API is deployed, you will find your API URL as follows:
`https://<API ID>.execute-api.<region>.amazonaws.com/`
14. Next, access your API from the REST client, such as **POSTMAN** shown in this example, to invoke your image prediction service. In the **API Request**, set the **Content-Type** as **image/jpeg** and add the parameter name **imageName** with a value (such as `animal`). Add an image in the body as a `binary` file that our service will predict:
`https://<API ID>.execute-api.<region>.amazonaws.com/prod/tensorflow-predict?imageName=animal`

Voila! You will see the following output from the serverless service in your Postman response:
"The image is identified as giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (with score = 0.89107)"



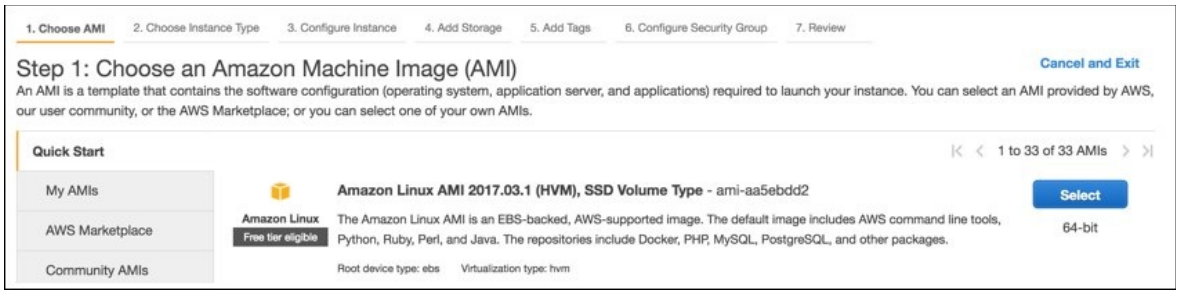
Also, audio of the predicted response will be generated and stored in the S3 bucket under the `audio` folder.



Steps to modify code and packages for lambda environments

If you need to add an additional Python package for your service or update any existing package, perform the following:

1. Launch an EC2 instance with **Amazon Linux AMI 2017.03.1 (HVM), SSD Volume Type**:



2. Log in to the EC2 instance and copy the current lambda code in the instance. Then, create a directory and extract the ZIP file inside that directory:

```
mkdir lambda
cd lambda
unzip lambda_tensorflow.zip
```

```
[ec2-user@ip-172-31-7-67 lambda]$ ls
classify.py              Keras-Tensorflow       pkg_resources                      tensorflow
easy_install.py          __MACOSX               protobuf-3.1.0.post1.dist-info     tensorflow-1.0.0.data
enum-0.4.6.dist-info     mock                   protobuf-3.1.0.post1-py2.7-nspkg.pth tensorflow-1.0.0.dist-info
enum.py                  mock-2.0.0.dist-info   pytz                               wheel
funcsigs                 numpy                  pytz-2016.10.dist-info             wheel-0.30.0a0.dist-info
funcsigs-1.0.2.dist-info numpy-1.11.2.data      PyYAML-3.10.egg-info               yaml
google                   numpy-1.11.2.dist-info six-1.10.0.dist-info
keras                    pbr                    six.py
keras_lambda.py          pbr-1.10.0.dist-info   six.pyc
```

3. To update any existing package, first remove it and then install it with the `pip` command. To add a new package install with `pip` (if that package depends on shared `.so` libraries, then you need to create a `lib` folder and copy those files in it from the `//usr/lib` and `/usr/lib64` directory):

```
rm -rf tensorflow*
pip install tensorflow==1.2.0 -t /home/ec2-
user/lambda
```

4. Then create the ZIP file of the full directory:

```
zip -r lambda_tensorflow.zip *
```

5. And finally, copy the ZIP file to S3 and update your Lambda function by mentioning the new ZIP file location on S3.

Note

You may need to strip down some packages or irrelevant directories from packages to make sure the total size of unzipped files in the `code` directory is less than 250 MB; otherwise, Lambda won't deploy your code.

Go to the following link for more information about custom package deployment on Lambdahttp://docs.aws.amazon.com/lambda/latest/dg/lambda-python-

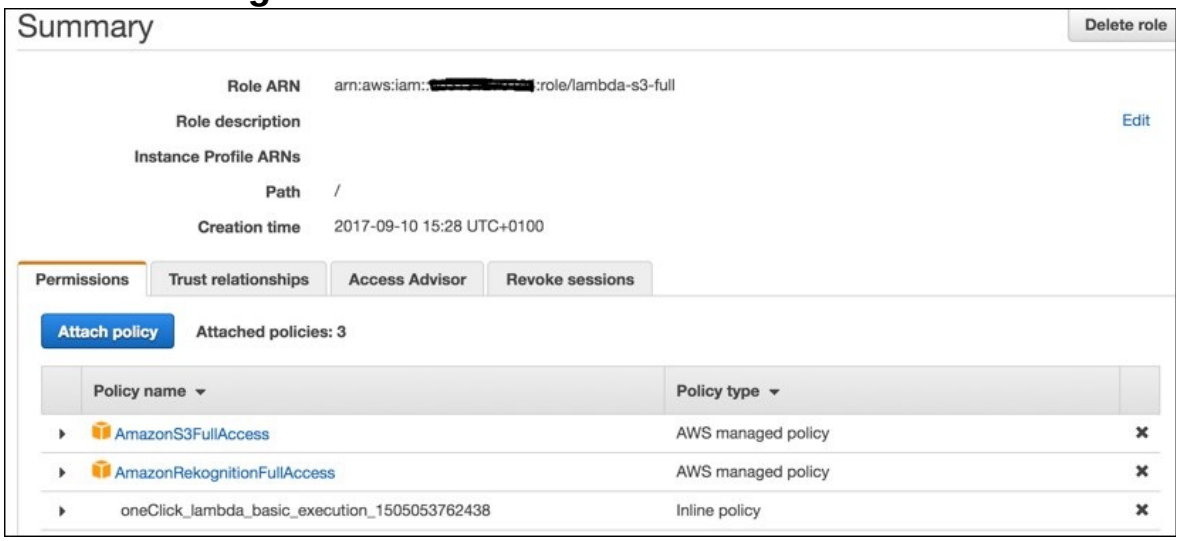
how-to-create-deployment-package.html.

Running face detection with a cloud managed service

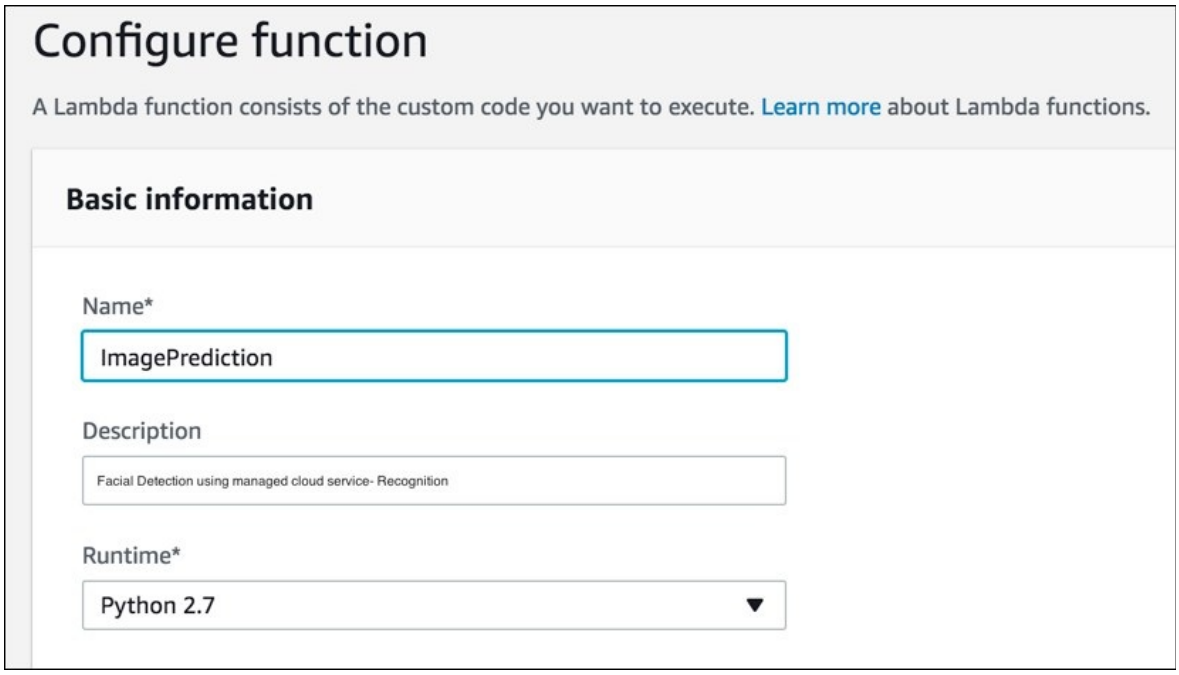
In this example, we will use a deep learning-based managed cloud service for our label identification and face detection system. We will continue to leverage a serverless environment AWS Lambda and utilize a deep learning-based cloud managed service, AWS Rekognition for facial attribute identification.

Perform the following steps to build a facial detection system using managed cloud deep learning services on a serverless platform:

1. First, update the IAM Lambda execution role from the previous example and attach a new managed policy **AmazonRekognitionFullAccess** as follows:



2. Next, create a new Lambda function that will be used for building the facial detection service. Select **Runtime*** as **Python 2.7** and keep all other settings as default. Attach the updated IAM role with this Lambda function:

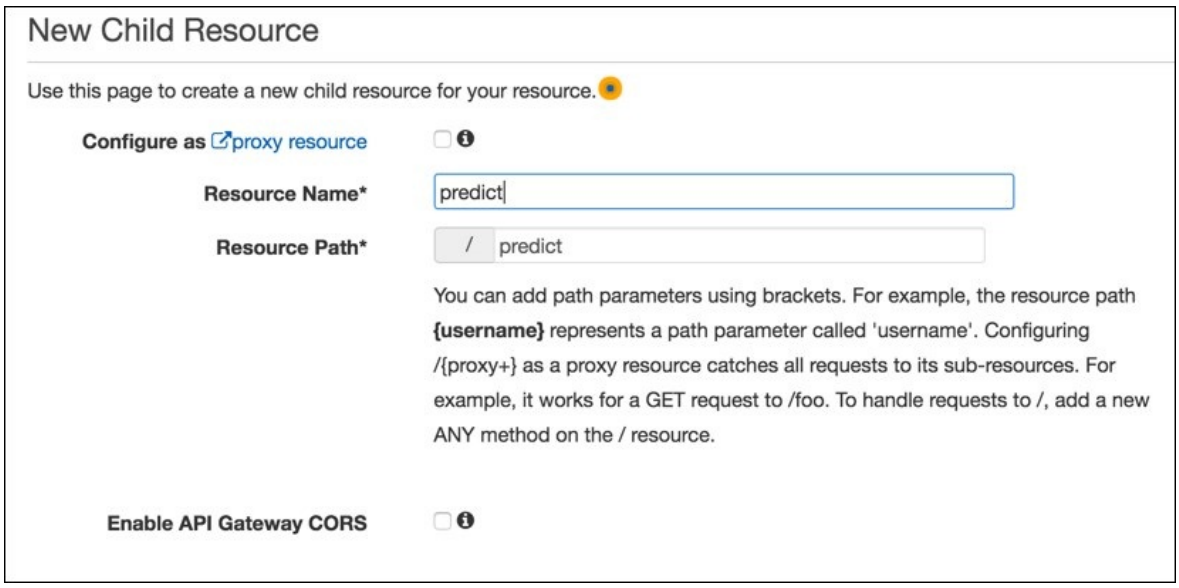


3. Then, paste the following code in the **Lambda function code** section area. Update the S3 **Bucket Name** and AWS **Region** information in the `boto3.client` of the code as follows:

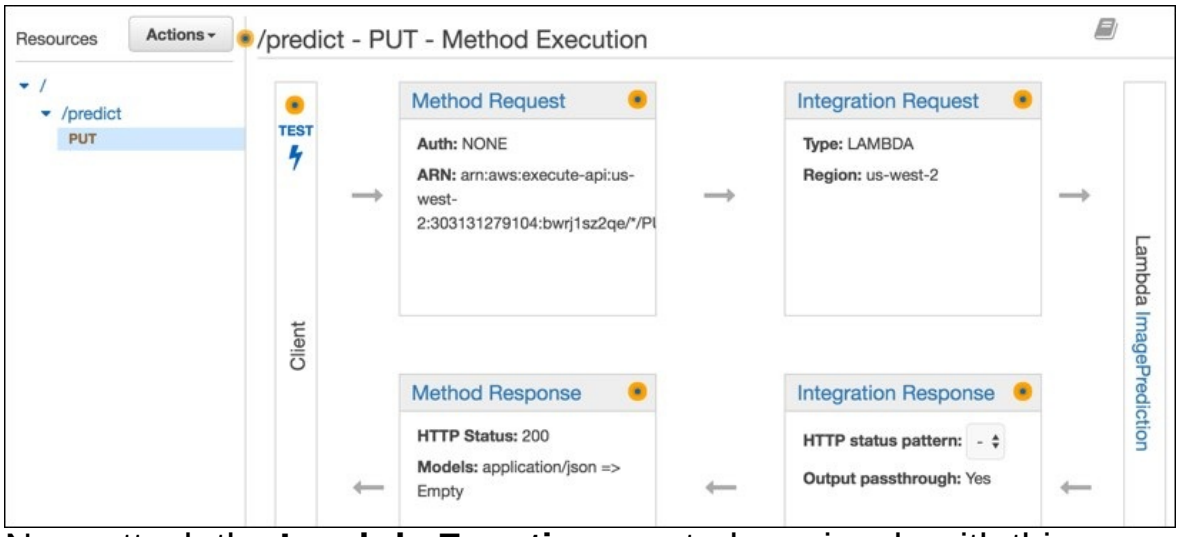
```
from __future__ import print_function

import json
import urllib
import boto3
import base64
import io

print('Loading function')

s3 = boto3.client('s3')
rekognition = boto3.client("rekognition", <aws-
region name like us-west-1>)

bucket=<Put your Bucket Name>
key_path='raw-image/'

def lambda_handler(event, context):

    output={}
    try:
        if event['operation']=='label-detect':
            print('Detecting label')
            fileName= event['fileName']
            bucket_key=key_path + fileName
            data =
```

```
base64.b64decode(event['base64Image'])
            image=io.BytesIO(data)
            s3.upload_fileobj(image, bucket,
bucket_key)
            rekog_response =
rekognition.detect_labels(Image={"S3Object":
{"Bucket": bucket,"Name":
bucket_key,}},MaxLabels=5,MinConfidence=90,)
            for label in rekog_response['Labels']:

output[label['Name']]=label['Confidence']
        else:
            print('Detecting faces')
            FEATURES_BLACKLIST = ("Landmarks",
"Emotions", "Pose", "Quality", "BoundingBox",
"Confidence")
            fileName= event['fileName']
            bucket_key=key_path + fileName
            data =
base64.b64decode(event['base64Image'])
            image=io.BytesIO(data)
            s3.upload_fileobj(image, bucket,
bucket_key)
            face_response =
rekognition.detect_faces(Image={"S3Object":
{"Bucket": bucket,  "Name": bucket_key, }},
Attributes=['ALL'],)
            for face in
face_response['FaceDetails']:
                output['Face']=face['Confidence']
                for emotion in face['Emotions']:

output[emotion['Type']]=emotion['Confidence']
                for feature, data in
face.iteritems():
                    if feature not in
FEATURES_BLACKLIST:
                        output[feature]=data
    except Exception as e:
        print(e)
        raise e

    return output
```

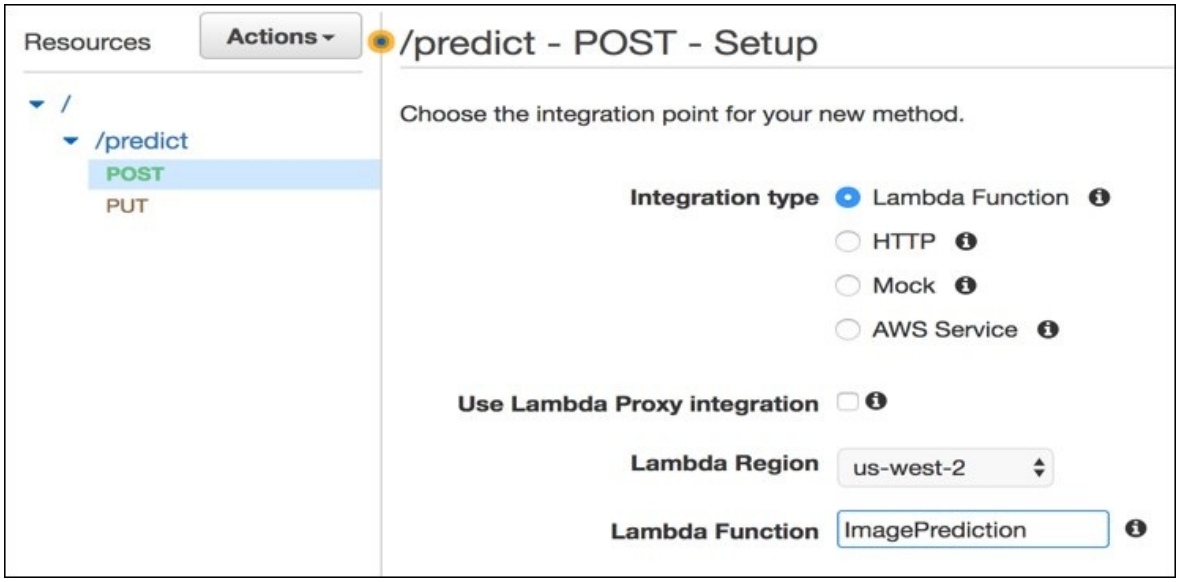
4. Once you have created the Lambda function, we will create an API gateway child resource for this service as follows:



5. Next, we will add a method (**PUT** in this case) to our new child resource (**predict**), then click on **Integration Request** of the **PUT** method.

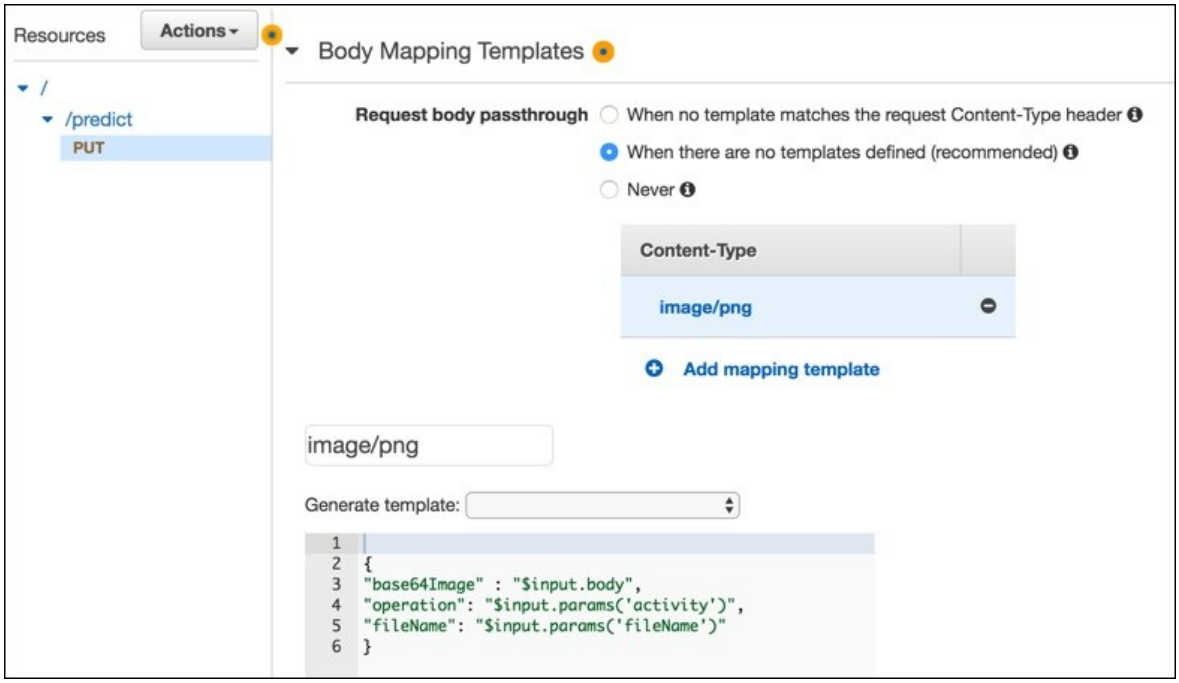


6. Now, attach the **Lambda Function** created previously with this resource method. You need to select the AWS **Lambda Region** where you have created your **Lambda Function** to get the Lambda function name in the drop-down list:



7. Next, expand the **Body Mapping Templates** section and select **When there are no templates defined (recommended)** in the **Request body passthrough** section. Then, add a mapping template **image/png** in the **Content-Type** and paste the following code in the **General template** area:

```
{
"base64Image": "$input.body",
"operation": "$input.params('activity')",
"fileName": "$input.params('fileName')"
}
```

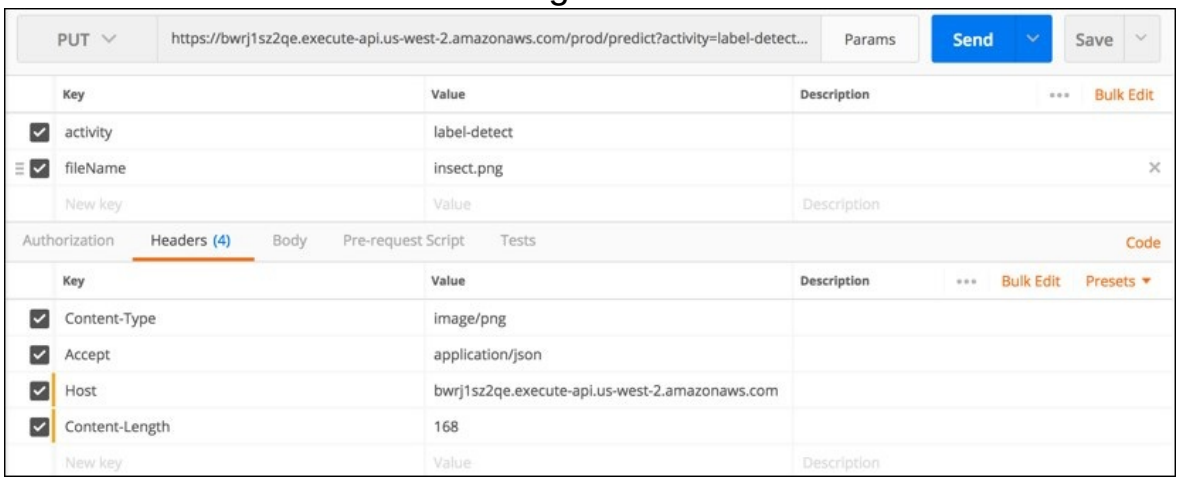


8. Now deploy your API Gateway resource API by clicking **Deploy API**

from the **Action** menu. Once your resource is deployed, you will get an API of the gateway that you will use to invoke the face detection service. We will continue to use the REST client **Postman** (https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehl=en) from our previous example, but you can use any other REST client of your choice as well. The API Gateway URL will look as follows:

```
https://<API ID>.execute-api.<AWS Region>.amazonaws.com/prod/predict
```

9. Add **Content-Type** as **image/png** and add two request parameter activities and filenames in the request. The parameter **activity** takes two values (**label-detect** for image recognition or label detection) and (**face-detect** for face detection). And the **fileName** parameter will be used to save the raw image to S3 with that name.



10. Now, invoke your service to detect a label or face and get the response output in JSON as follows:

Label Detection

"Butterfly": 99.25298309326172,
"Insect": 99.25298309326172,
"Potted Plant": 98.92184448242188,
"Plant": 98.92184448242188,
"Invertebrate": 99.25298309326172

Face Detection


```
{
    "Eyeglasses": {
        "Confidence": 99.96630096435547,
        "Value": false
    },
    "Sunglasses": {
        "Confidence": 99.28678894042969,
        "Value": false
    },
    "Gender": {
        "Confidence": 99.92697143554688,
        "Value": "Male"
    },
    "CONFUSED": 4.443657398223877,
    "Face": 99.99605560302734,
    "CALM": 3.5979151725769043,
    "EyesOpen": {
        "Confidence": 99.7735824584961,
        "Value": true
    },
    "AgeRange": {
        "High": 38,
        "Low": 20
    },
    "Smile": {
        "Confidence": 89.05831909179688,
        "Value": true
    },
    "Beard": {
        "Confidence": 83.59758758544922,
        "Value": false
    },
    "MouthOpen": {
        "Confidence": 99.88780212402344,
        "Value": false
    },
    "Mustache": {
        "Confidence": 99.93497467041016,
        "Value": false
    },
    "HAPPY": 88.41159057617188
}
```

Summary

So far, you have learned and implemented various ways of deploying the trained deep model and making predictions for new data samples. You have also learned how to take your model from your local machine or data center to the cloud using Docker containers smoothly. I hope, throughout the book, with lots of hands-on examples using real-world public datasets, you have understood the concept of GANs, and its variant architecture (SSGAN, BEGAN, DCGAN, CycleGAN, StackGAN, DiscoGAN) well. Once you have played around with the code and examples in this book, I would definitely encourage you to do the following:

Participate in the Kaggle Adversarial Network competition: https://www.kaggle.com/c/nips-2017-defense-against-adversarial-attack.

Keep your knowledge updated about deep learning and GANs by attending or viewing the following conferences:

- **Neural Information Processing Systems** (**NIPS**): https://nips.cc/
- **International Conference on Learning Representations** (ICLR): HTTP://WWW.ICLR.CC/

Index

A

B

C

    - used, for image to image style transfer / Image to image style transfer with CycleGAN
    - model formulation / Model formulation of CycleGAN
    - used, for transfiguration of horse into zebra / Transfiguration of a horse into a zebra with CycleGAN

D

- DCGAN
    - used, for building image correction system / Building an image correction system using DCGAN
- deconvolution / Deconvolution or transpose convolution
- Deep Convolutional Generative Adversarial Networks (DCGAN)
    - used, for image generation using Keras / Image generation with DCGAN using Keras
    - architectural constraints / Image generation with DCGAN using Keras
    - reference / Image generation with DCGAN using Keras
- DeepDream
    - used, for generating artistic hallucinated images / Generating artistic hallucinated images using DeepDream
- deep learning
    - evolution / Evolution of deep learning
    - sigmoid activation / Sigmoid activation
    - Rectified Linear Unit (ReLU) / Rectified Linear Unit (ReLU)
    - Exponential Linear Unit (ELU) / Exponential Linear Unit (ELU)
    - Stochastic Gradient Descent (SGD) / Stochastic Gradient Descent (SGD)
    - learning rate tuning learning / Learning rate tuning
    - regularization / Regularization
    - shared weights / Shared weights and pooling
    - pooling / Shared weights and pooling
    - local receptive field / Local receptive field
    - Convolutional network (ConvNet) / Convolutional network (ConvNet)
- deep models
    - deploying, approaches / Various approaches to deploying deep models
    - deploying, on cloud with GKE / Deploying a deep model on the cloud with GKE

G

H

I

K

L

M

R

S