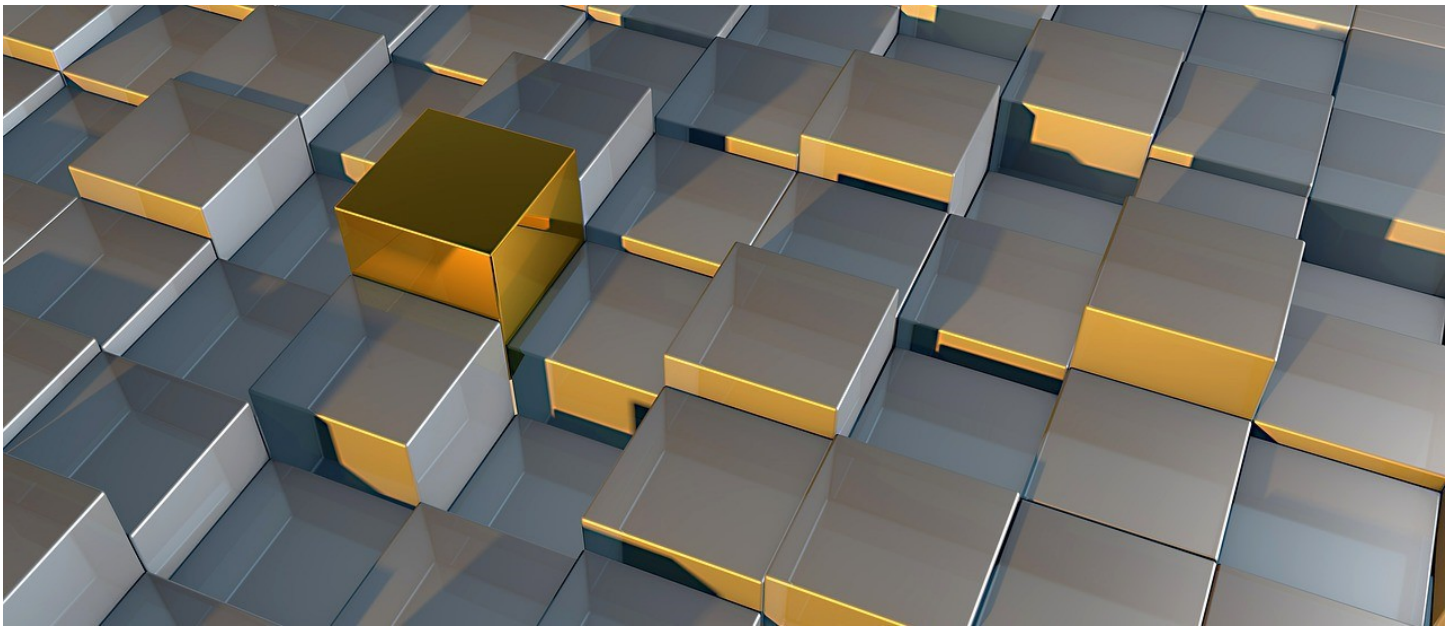
Dipanjan (DJ) Sarkar
Data Scientist @Intel, Author, Mentor @Springboard, Writer & Editor @TDataScience. Feel free to connect with me at https://www.linkedin.com/in/dipanzan
Nov 15 · 45 min read

A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning

Deep Learning on Steroids with the Power of Knowledge Transfer!



Source: Pixabay

Introduction

Humans have an inherent ability to transfer knowledge across tasks. What we acquire as knowledge while learning about one task, we utilize in the same way to solve related tasks. The more related the tasks, the easier it is for us to transfer, or cross-utilize our knowledge. Some simple examples would be,

- Know how to ride a motorbike ☐ Learn how to ride a car
- Know how to play classic piano ☐ Learn how to play jazz piano

- Know math and statistics ☐ Learn machine learning

In each of the above scenarios, we don't learn everything from scratch when we attempt to learn new aspects or topics. We transfer and leverage our knowledge from what we have learnt in the past!

Conventional machine learning and deep learning algorithms, so far, have been traditionally designed to work in isolation. These algorithms are trained to solve specific tasks. The models have to be rebuilt from scratch once the feature-space distribution changes. Transfer learning is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones. In this article, we will do a comprehensive coverage of the concepts, scope and real-world applications of transfer learning and even showcase some hands-on examples. To be more specific, we will be covering the following.

- **Motivation for Transfer Learning**
- **Understanding Transfer Learning**
- **Transfer Learning Strategies**
- **Transfer Learning for Deep Learning**
- **Deep Transfer Learning Strategies**
- **Types of Deep Transfer Learning**
- **Applications of Transfer Learning**

- **Case Study 1: Image Classification with a Data Availability Constraint**
- **Case Study 2: Multi-Class Fine-grained Image Classification with Large Number of Classes and Less Data Availability**
- **Transfer Learning Advantages**
- **Transfer Learning Challenges**
- **Conclusion & Future Scope**

We will look at transfer learning as a general high-level concept which started right from the days of machine learning and statistical modeling, however, we will be more focused around deep learning in this article.

> ***Note:*** *All the case studies will cover step by step details with code and outputs. The case studies depicted here and their results are purely based on actual experiments which we conducted when we implemented and tested these models while working on our book:* ***Hands on Transfer Learning with Python*** *(details at the end of this article).*

This article aims to be an attempt to cover theoretical concepts as well as demonstrate practical hands-on examples of deep learning applications in one place given the information overload which is out there on the web. All examples will be covered in Python using keras with a tensorflow backend, a perfect match for people who are veterans or just getting started with deep learning! *Interested in PyTorch? Feel free to convert these examples and contact me and I'll feature your work here and on GitHub!*

Motivation for Transfer Learning

We have already briefly discussed that humans don't learn everything from the ground up and leverage and transfer their knowledge from previously learnt domains to newer domains and tasks. Given the craze for driving towards ***True Artificial General Intelligence***, transfer learning is something which data scientists and researchers believe can further our progress towards *AGI*. In fact, *Andrew Ng*, renowned professor and data scientist who has been associated with Google Brain, Baidu, Stanford and Coursera, recently gave an amazing tutorial in NIPS 2016 called ***'Nuts and bolts of building AI applications using Deep Learning'*** where he mentioned,

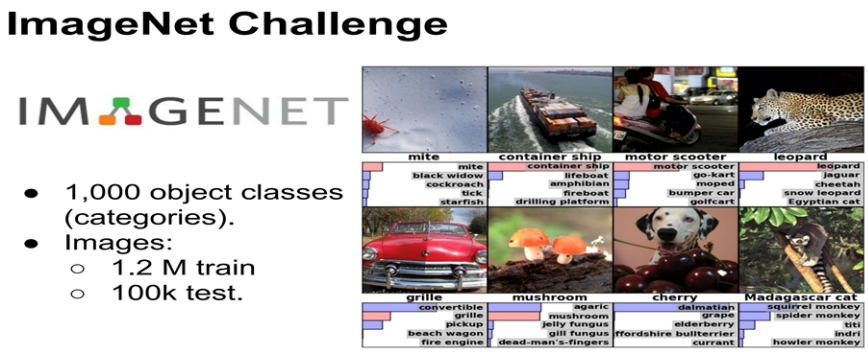
> *After supervised learning—Transfer Learning will be the next driver of ML success*

I recommend interested folks to check out ***his interesting tutorial*** from NIPS 2016.

In fact, transfer learning is not a concept which just cropped up in the 2010s. The Neural Information Processing Systems (NIPS) 1995 workshop *Learning to Learn: Knowledge Consolidation and Transfer in Inductive Systems* is believed to have provided the initial motivations for research in this field. Since then, terms such as *Learning to Learn*, *Knowledge Consolidation*, and *Inductive Transfer* have been used interchangeably with transfer learning. Invariably, different researchers and academic texts provide definitions from different contexts. In their famous book, *Deep Learning*, Goodfellow et al. refer to transfer learning in the context of generalization. Their definition is as follows:

> *Situation where what has been learned in one setting is exploited to improve generalization in another setting.*

Thus the key motivation, especially considering the context of deep learning is the fact that most models which solve complex problems need a whole lot of data, and getting vast amounts of labeled data for supervised models can be really difficult, considering the time and effort it takes to label data points. A simple example would be the ***ImageNet dataset*** which has millions of images pertaining to different categories thanks to years of hard work starting at Stanford!

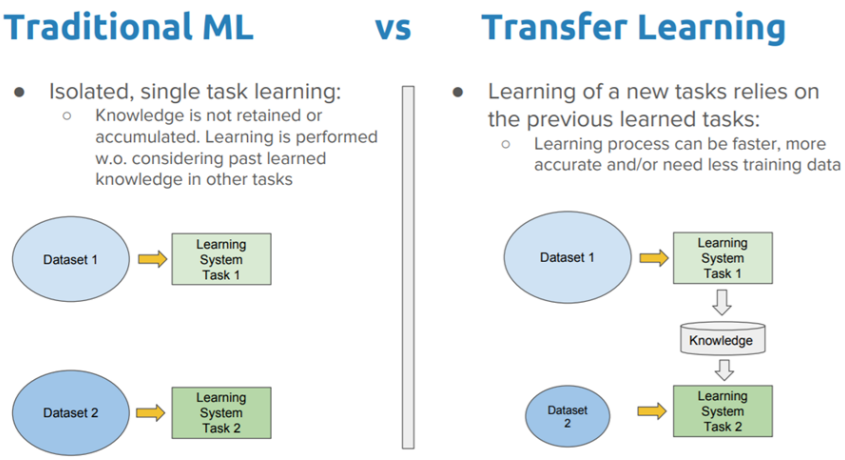


The popular ImageNet Challenge based on the ImageNet Database

However, getting such a dataset for every domain is tough. Besides this, most deep learning models are very specialized to a particular domain or even a specific task. While these might be state-of-the-art models, with really high accuracy and beating all benchmarks, it would be only on very specific datasets and end up suffering a significant loss in performance when used in a new task which might still be similar to the one it was trained on. This forms the motivation for transfer learning which goes beyond specific tasks and domains and tries to see how to leverage knowledge from pre-trained models and use it to solve new problems!

Understanding Transfer Learning

The first thing to remember here is that, transfer learning, is not a new concept which is very specific to deep learning. There is a stark difference between the traditional approach of building and training machine learning models and using a methodology following transfer learning principles.



Traditional Learning vs Transfer Learning

Traditional learning is isolated and occurs purely based on specific tasks, datasets and training separate isolated models on them. No knowledge is retained which can be transferred from one model to another. In transfer learning, you can leverage knowledge (features, weights etc.) from previously trained models for training newer models and even tackle problems like having less data for the newer task!

Let's understand the preceding explanation with the help of an example. Let's assume our task is to identify objects in images within a restricted domain of a restaurant. Let's mark this task in its defined scope as ***T1***. Given the dataset for this task, we train a model and tune it to perform well (generalize) on unseen data points from the same domain (restaurant). Traditional supervised ML algorithms break down when we do not have sufficient training examples for the required tasks in given domains. Suppose we now must detect objects from images in a park or a café (say, task ***T2***). Ideally, we should be able to apply the model trained for ***T1***, but in reality we face performance degradation and models that do not generalize well. This happens for a variety of reasons, which we can liberally and collectively term as the model's bias toward training data and domain.

Transfer learning should enable us to utilize knowledge from previously learned tasks and apply them to newer, related ones. If we have significantly more data for task ***T1***, we may utilize its learning and generalize this knowledge (features, weights) for task ***T2*** (which has significantly less data). In the case of problems in the computer vision domain, certain low-level features, such as edges, shapes, corners and intensity, can be shared across tasks and thus enable knowledge transfer among tasks! Also as we have depicted in the earlier figure, knowledge from an existing task acts as an additional input when learning a new target task.

Formal Definition

Let's now take a look at a formal definition for transfer learning and then utilize it to understand different strategies. In their paper, *A Survey on Transfer Learning*, Pan and Yang use domain, task, and marginal probabilities to present a framework for understanding transfer learning. The framework is defined as follows:

A domain, ***D***, is defined as a two-element tuple consisting of feature space, □, and marginal probability, ***P(X)***, where ***X*** is a sample data point. Thus, we can represent the domain mathematically as ***D = {□, P(X)}***

A **Domain** consists of two components: $D = \{\mathcal{X}, P(X)\}$

- Feature space: $\mathcal{X}$
- Marginal distribution: $P(X)$, $X = \{x_1, \cdots, x_n\}, x_i \in \mathcal{X}$

Here $\boldsymbol{x_i}$ represents a specific vector as represented in the above depiction. A task, $\boldsymbol{T}$, on the other hand, can be defined as a two-element tuple of the label space, $\boldsymbol{\gamma}$, and objective function, $\boldsymbol{\eta}$. The objective function can also be denoted as $\boldsymbol{P(\gamma | X)}$ from a probabilistic view point.

For a given domain **D**, a **Task** is defined by two components:

$$T = \{\mathcal{Y}, P(Y|X)\} = \{\mathcal{Y}, \eta\} \qquad Y = \{y_1, \cdots, y_n\}, y_i \in \mathcal{Y}$$

- A label space: $\mathcal{Y}$
- A predictive function η, learned from *feature vector/label* pairs, (x_i, y_i), $x_i \in \mathcal{X}, y_i \in \mathcal{Y}$
- For each feature vector in the domain, η predicts its corresponding label: $\eta(x_i) = y_i$

Thus, armed with these definitions and representations, we can define transfer learning as follows, thanks to *an excellent article* from Sebastian Ruder.

> Given a source domain $\mathcal{D}_S$, a corresponding source task $\mathcal{T}_S$, as well as a target domain $\mathcal{D}_T$ and a target task $\mathcal{T}_T$, the objective of transfer learning now is to enable us to learn the target conditional probability distribution $P(Y_T|X_T)$ in $\mathcal{D}_T$ with the information gained from $\mathcal{D}_S$ and $\mathcal{T}_S$ where $\mathcal{D}_S \neq \mathcal{D}_T$ or $\mathcal{T}_S \neq \mathcal{T}_T$. In most cases, a limited number of labeled target examples, which is exponentially smaller than the number of labeled source examples are assumed to be available.

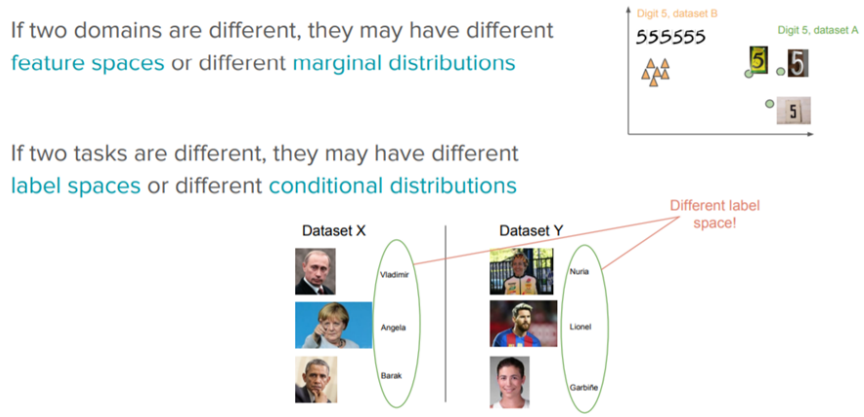
Scenarios

Let's now take a look at the typical scenarios involving transfer learning based on our previous definition.

Given source and target domains $\mathcal{D}_S$ and $\mathcal{D}_T$ where $\mathcal{D} = \{\mathcal{X}, P(X)\}$ and source and target tasks $\mathcal{T}_S$ and $\mathcal{T}_T$ where $\mathcal{T} = \{\mathcal{Y}, P(Y|X)\}$ source and target conditions can vary in four ways, which we will illustrate in the following again using our document classification example:

1. $\mathcal{X}_S \neq \mathcal{X}_T$. The feature spaces of the source and target domain are different, e.g. the documents are written in two different languages. In the context of natural language processing, this is generally referred to as cross-lingual adaptation.
2. $P(X_S) \neq P(X_T)$. The marginal probability distributions of source and target domain are different, e.g. the documents discuss different topics. This scenario is generally known as domain adaptation.
3. $\mathcal{Y}_S \neq \mathcal{Y}_T$. The label spaces between the two tasks are different, e.g. documents need to be assigned different labels in the target task. In practice, this scenario usually occurs with scenario 4, as it is extremely rare for two different tasks to have different label spaces, but exactly the same conditional probability distributions.
4. $P(Y_S|X_S) \neq P(Y_T|X_T)$. The conditional probability distributions of the source and target tasks are different, e.g. source and target documents are unbalanced with regard to their classes. This scenario is quite common in practice and approaches such as over-sampling, under-sampling, or SMOTE are widely used

To give some more clarity on the difference between the terms ***domain*** and ***task***, the following figure tries to explain them with some examples.



Key Takeaways

Transfer learning, as we have seen so far, is having the ability to utilize existing knowledge from the source learner in the target task. During the process of transfer learning, the following three important questions must be answered:

- **What to transfer:** This is the first and the most important step in the whole process. We try to seek answers about which part of the knowledge can be transferred from the source to the target in order to improve the performance of the target task. When trying

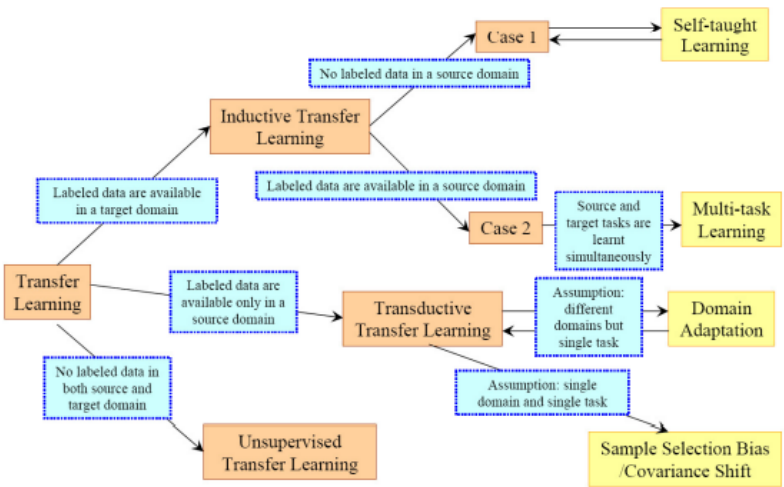
to answer this question, we try to identify which portion of knowledge is source-specific and what is common between the source and the target.

- **When to transfer:** There can be scenarios where transferring knowledge for the sake of it may make matters worse than improving anything (also known as negative transfer). We should aim at utilizing transfer learning to improve target task performance/results and not degrade them. We need to be careful about when to transfer and when not to.
- **How to transfer:** Once the *what* and *when* have been answered, we can proceed toward identifying ways of actually transferring the knowledge across domains/tasks. This involves changes to existing algorithms and different techniques, which we will cover in later sections of this article. Also, specific case studies are lined up in the end for a better understanding of how to transfer.

This should help us define the various scenarios where transfer learning can be applied and possible techniques, which we will discuss in the next section.

Transfer Learning Strategies

There are different transfer learning strategies and techniques which can be applied based on the domain, task at hand and the availability of data. I really like the following figure from the paper on transfer learning we mentioned earlier, *A Survey on Transfer Learning*.



Transfer Learning Strategies

Thus based on the previous figure, transfer learning methods can be categorized based on the type of traditional ML algorithms involved, such as:

- **Inductive Transfer learning:** In this scenario, the source and target domains are the same, yet the source and target tasks are different from each other. The algorithms try to utilize the inductive biases of the source domain to help improve the target task. Depending upon whether the source domain contains labeled data or not, this can be further divided into two subcategories, similar to multitask learning and self-taught learning, respectively.
- **Unsupervised Transfer Learning:** This setting is similar to inductive transfer itself, with a focus on unsupervised tasks in the target domain. The source and target domains are similar, but the tasks are different. In this scenario, labeled data is unavailable in either of the domains.
- **Transductive Transfer Learning:** In this scenario, there are similarities between the source and target tasks but the corresponding domains are different. In this setting, the source domain has a lot of labeled data while the target domain has none. This can be further classified into subcategories, referring to settings where either the feature spaces are different or the marginal probabilities.

We can summarize the different settings and scenarios for each of the above techniques in the following table.

Learning Strategy	Related Areas	Source & Target Domains	Source Domain Labels	Target Domain Labels	Source & Target Tasks	Tasks
Inductive Transfer Learning	Multi-task Learning	The Same	Available	Available	Different but Related	Regression Classification
	Self-taught Learning	The Same	Unavailable	Available	Different but Related	Regression Classification
Unsupervised Transfer Learning		Different but Related	Unavailable	Unavailable	Different but Related	Clustering Dimensionality Reduction
Transductive Transfer Learning	Domain Adaptation, Sample Selection Bias & Co-variate Shift	Different but Related	Available	Unavailable	The Same	Regression Classification

Types of Transfer Learning Strategies and their Settings

The three transfer categories discussed in the previous section outline different settings where transfer learning can be applied and studied in detail. To answer the question of what to transfer across these categories, some of the following approaches can be applied:

- **Instance transfer:** Reusing knowledge from the source domain to the target task is usually an ideal scenario. In most cases, the source domain data cannot be reused directly. Rather, there are certain instances from the source domain that can be reused along with target data to improve results. In case of inductive transfer, modifications such as AdaBoost by Dai and their co-authors help utilize training instances from the source domain for improvements in the target task.

- **Feature-representation transfer:** This approach aims to minimize domain divergence and reduce error rates by identifying good feature representations that can be utilized from the source to target domains. Depending upon the availability of labeled data, supervised or unsupervised methods may be applied for feature-representation-based transfers.

- **Parameter transfer:** This approach works on the assumption that the models for related tasks share some parameters or prior distribution of hyperparameters. Unlike multitask learning, where both the source and target tasks are learned simultaneously, for transfer learning, we may apply additional weightage to the loss of the target domain to improve overall performance.

- **Relational-knowledge transfer:** Unlike the preceding three approaches, the relational-knowledge transfer attempts to handle non-IID data, such as data that is not independent and identically distributed. In other words, data where each data point has a relationship with other data points; for instance, social network data utilizes relational-knowledge-transfer techniques.

The following table clearly summarizes the relationship between different transfer learning strategies and what to transfer.

	Inductive Transfer Learning	Transductive Transfer Learning	Unsupervised Transfer Learning
Instance-transfer	√	√	
Feature-representation-transfer	√	√	√
Parameter-transfer	√		
Relational-knowledge-transfer	√		

Transfer Learning Strategies and Types of Transferable Components

Let's now utilize this understanding and learn how transfer learning is applied in the context of deep learning.

Transfer Learning for Deep Learning

The strategies we discussed in the previous section are general approaches which can be applied towards machine learning techniques, which brings us to the question, can transfer learning really be applied in the context of deep learning?

Transfer learning in DL

Myth: you can't do deep learning unless you have a million labeled examples for your problem.

Reality

- You can learn useful representations from **unlabeled data**
- You can train on a nearby **surrogate objective** for which it is easy to generate labels
- You can **transfer** learned representations from a related task

Deep learning models are representative of what is also known as ***inductive learning***. The objective for inductive-learning algorithms is to infer a mapping from a set of training examples. For instance, in cases of classification, the model learns mapping between input features and class labels. In order for such a learner to generalize well on unseen data, its algorithm works with a set of assumptions related to the distribution of the training data. These sets of assumptions are known as ***inductive bias***. The inductive bias or assumptions can be characterized by multiple factors, such as the hypothesis space it restricts to and the search process through the hypothesis space. Thus, these biases impact how and what is learned by the model on the given task and domain.

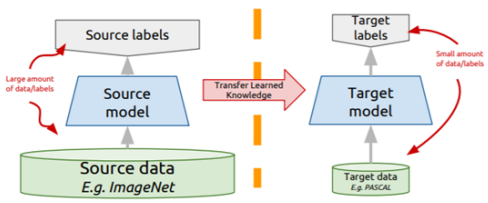
Transfer learning: idea

Instead of training a deep network from scratch for your task:

- Take a network trained on a different domain for a different **source task**
- Adapt it for your domain and your **target task**

Variations:

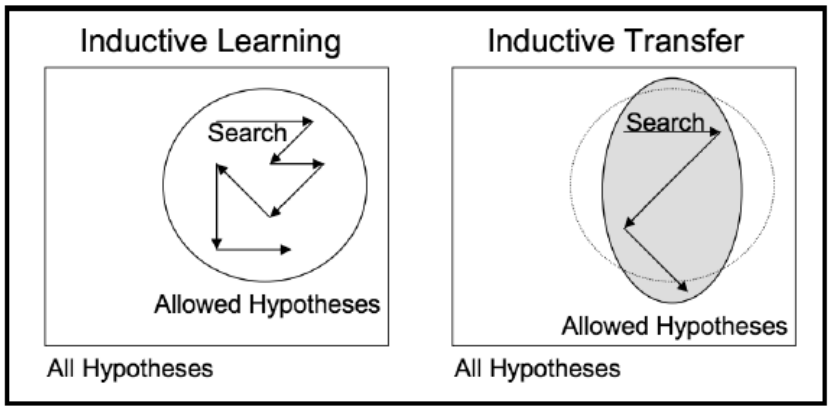
- Same domain, different task
- Different domain, same task



Ideas for deep transfer learning

Inductive transfer techniques utilize the inductive biases of the source task to assist the target task. This can be done in different ways, such as by adjusting the inductive bias of the target task by limiting the model

space, narrowing down the hypothesis space, or making adjustments to the search process itself with the help of knowledge from the source task. This process is depicted visually in the following figure.



Inductive transfer (Source: Transfer learning, Lisa Torrey and Jude Shavlik)

Apart from inductive transfer, inductive-learning algorithms also utilize Bayesian and Hierarchical transfer techniques to assist with improvements in the learning and performance of the target task.

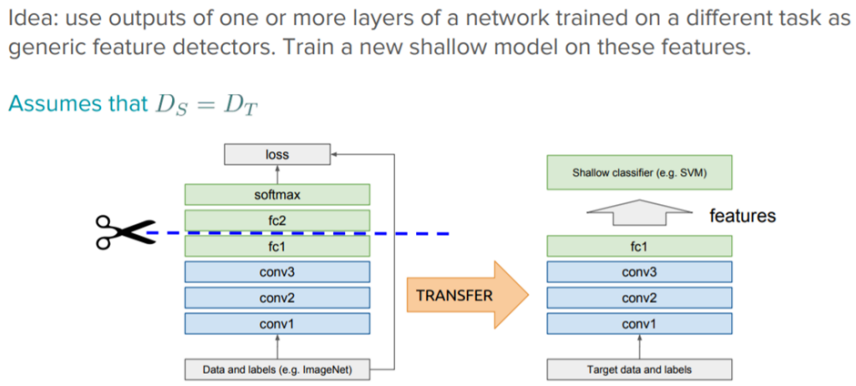
Deep Transfer Learning Strategies

Deep learning has made considerable progress in recent years and the results yielded are amazing based on the type of complex problems we can tackle. However, the training time and the amount of data required for such deep learning systems is in orders of magnitudes which are much larger than traditional ML systems. There are various deep learning networks with state-of-the-art performance (sometimes as good or even better than human performance) that have been developed and tested across domains such as computer vision and natural language processing (NLP). In most cases, teams/people share the details of these networks for others to use. These pretrained networks/models form the basis of transfer learning in the context of deep learning or what I like to call ***'deep transfer learning'***. Let's look at the two most popular strategies for deep transfer learning.

Off-the-shelf Pre-trained Models as Feature Extractors

Deep learning systems and models are layered architectures that learn different features at different layers (hierarchical representations of layered features). These layers are then finally connected to a last layer (usually a fully connected layer, in the case of classification) to get the final output. This layered architecture allows us to utilize a pre-trained

network (such as Inception V3 or VGG) without its final layer as a fixed feature extractor for other tasks.

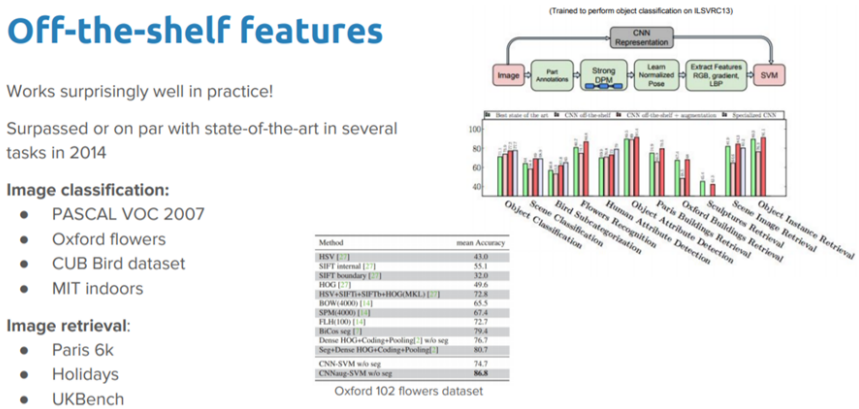


Transfer Learning with Pre-trained Deep Learning Models as Feature Extractors

> *The key idea here is to just leverage the pre-trained model's weighted layers to extract features but not to update the weights of the model's layers during training with new data for the new task.*

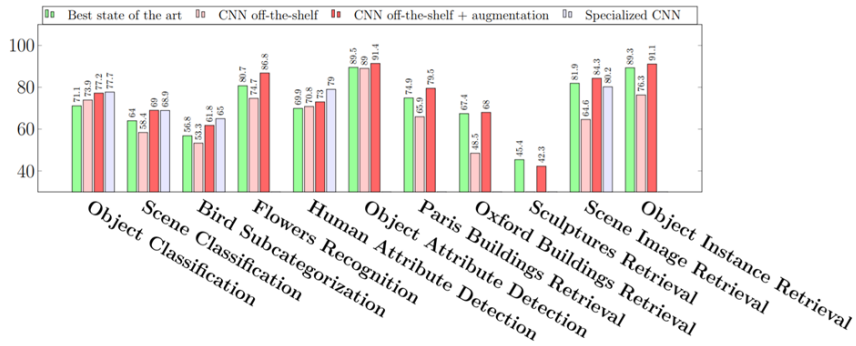
For instance, if we utilize AlexNet without its final classification layer, it will help us transform images from a new domain task into a 4096-dimensional vector based on its hidden states, thus enabling us to extract features from a new domain task, utilizing the knowledge from a source-domain task. This is one of the most widely utilized methods of performing transfer learning using deep neural networks.

Now a question might arise, how well do these pre-trained off-the-shelf features really work in practice with different tasks?



It definitely seems to work really well in real-world tasks and if the chart in the above table is not very clear, the following figure should

make things more clear with regard to their performance in different computer vision based tasks!

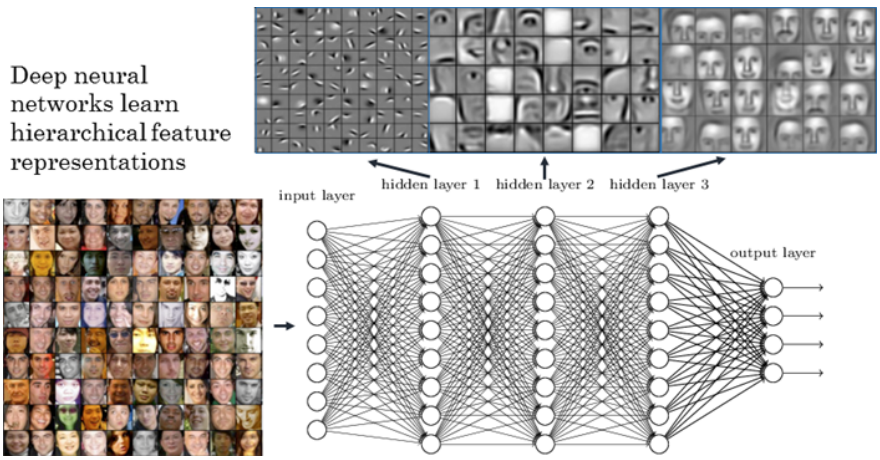


Performance of off-the-shelf pre-trained models vs. specialized task-focused deep learning models

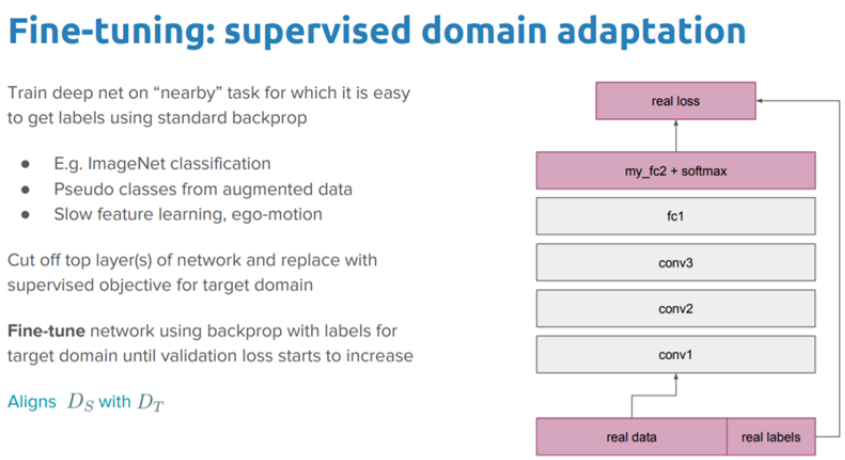
Based on the red and pink bars in the above figure, you can clearly see that the features from the pre-trained models consistently out-perform very specialized task-focused deep learning models.

Fine Tuning Off-the-shelf Pre-trained Models

This is a more involved technique where we do not just replace the final layer (for classification/regression), but we also selectively retrain some of the previous layers. Deep neural networks are highly configurable architectures with various hyperparameters. As discussed earlier, the initial layers have been seen to capture generic features, while the later ones focus more on the specific task at hand. An example is depicted in the following figure on a face-recognition problem, where initial lower layers of the network learn very generic features and the higher layers learn very task-specific features.

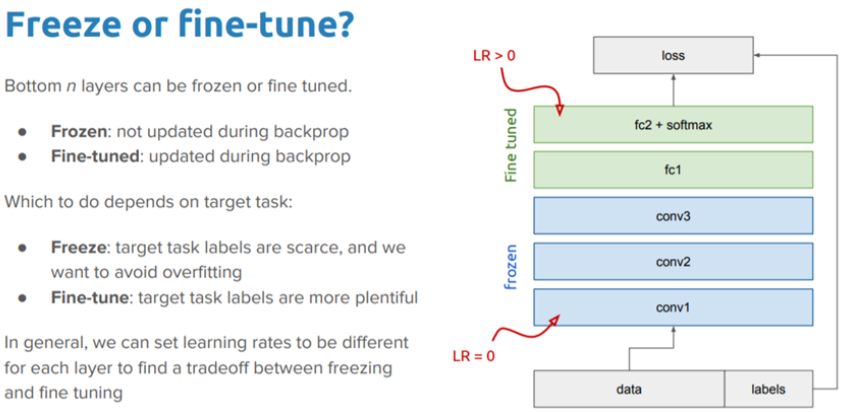


Using this insight, we may freeze (fix weights) certain layers while retraining, or fine-tune the rest of them to suit our needs. In this case, we utilize the knowledge in terms of the overall architecture of the network and use its states as the starting point for our retraining step. This, in turn, helps us achieve better performance with less training time.



Freezing or Fine-tuning?

This brings us to the question, should we freeze layers in the network to use them as feature extractors or should we also fine-tune layers in the process?



This should give us a good perspective on what each of these strategies are and when should they be used!

Pre-trained Models

One of the fundamental requirements for transfer learning is the presence of models that perform well on source tasks. Luckily, the deep learning world believes in sharing. Many of the state-of-the art deep

learning architectures have been openly shared by their respective teams. These span across different domains, such as computer vision and NLP, the two most popular domains for deep learning applications. Pre-trained models are usually shared in the form of the millions of parameters/weights the model achieved while being trained to a stable state. Pre-trained models are available for everyone to use through different means. The famous deep learning Python library, keras, provides an interface to download some popular models. You can also access pre-trained models from the web since most of them have been open-sourced.

For computer vision, you can leverage some popular models including,

- VGG-16
- VGG-19
- Inception V3
- XCeption
- ResNet-50

For natural language processing tasks, things become more difficult due to the varied nature of NLP tasks. You can leverage word embedding models including,

- Word2Vec
- GloVe
- FastText

But wait, that's not all! Recently, there have been some excellent advancements towards transfer learning for NLP. Most notably,

- Universal Sentence Encoder by Google
- Bidirectional Encoder Representations from Transformers (BERT) by Google

They definitely hold a lot of promise and I'm sure they will be widely adopted pretty soon for real-world applications.

Types of Deep Transfer Learning

The literature on transfer learning has gone through a lot of iterations, and as mentioned at the start of this chapter, the terms associated with it have been used loosely and often interchangeably. Hence, it is sometimes confusing to differentiate between *transfer learning*, *domain adaptation*, and *multi-task learning*. Rest assured, these are all related and try to solve similar problems. In general you should always think of transfer learning as a general concept or principle, where we will try to solve a target task using source task-domain knowledge.

Domain Adaptation

Domain adaption is usually referred to in scenarios where the marginal probabilities between the source and target domains are different, such as $P(X_S) \neq P(X_T)$. There is an inherent shift or drift in the data distribution of the source and target domains that requires tweaks to transfer the learning. For instance, a corpus of movie reviews labeled as positive or negative would be different from a corpus of product-review sentiments. A classifier trained on movie-review sentiment would see a different distribution if utilized to classify product reviews. Thus, domain adaptation techniques are utilized in transfer learning in these scenarios.

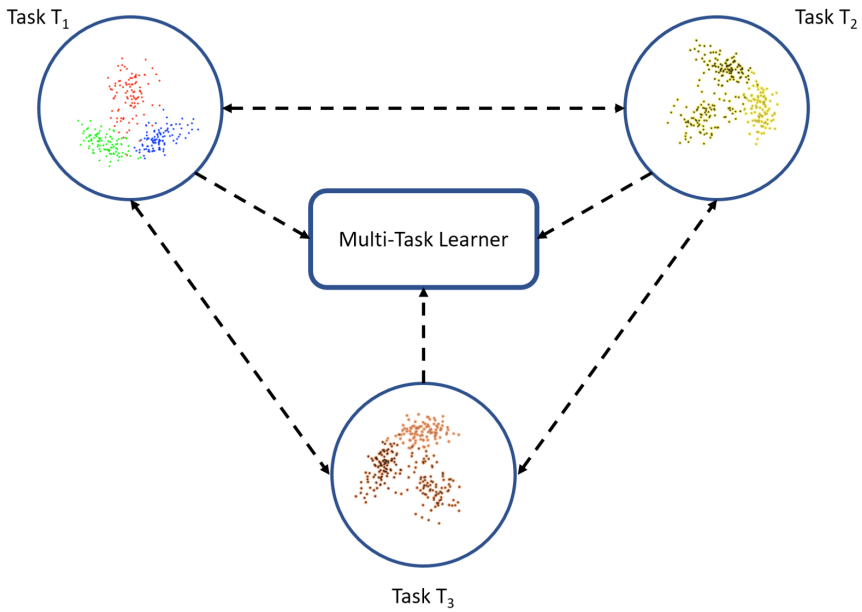
Domain Confusion

We learned different transfer learning strategies and even discussed the three questions of what, when, and how to transfer knowledge from the source to the target. In particular, we discussed how feature-representation transfer can be useful. It is worth re-iterating that different layers in a deep learning network capture different sets of features. We can utilize this fact to learn domain-invariant features and improve their transferability across domains. Instead of allowing the model to learn any representation, we nudge the representations of both domains to be as similar as possible. This can be achieved by applying certain pre-processing steps directly to the representations themselves. Some of these have been discussed by Baochen Sun, Jiashi Feng, and Kate Saenko in their paper *'Return of Frustratingly Easy Domain Adaptation'*. This nudge toward the similarity of representation has also been presented by Ganin et. al. in their paper, *'Domain-Adversarial Training of Neural Networks'*. The basic idea behind this technique is to add another objective to the source model to encourage similarity by confusing the domain itself, hence domain confusion.

Multitask Learning

Multitask learning is a slightly different flavor of the transfer learning world. In the case of multitask learning, several tasks are learned

simultaneously without distinction between the source and targets. In this case, the learner receives information about multiple tasks at once, as compared to transfer learning, where the learner initially has no idea about the target task. This is depicted in the following figure.



Multitask learning: Learner receives information from all tasks simultaneously

One-shot Learning

Deep learning systems are data hungry by nature, such that they need many training examples to learn the weights. This is one of the limiting aspects of deep neural networks, though such is not the case with human learning. For instance, once a child is shown what an apple looks like, they can easily identify a different variety of apple (with one or a few training examples); this is not the case with ML and deep learning algorithms. One-shot learning is a variant of transfer learning where we try to infer the required output based on just one or a few training examples. This is essentially helpful in real-world scenarios where it is not possible to have labeled data for every possible class (if it is a classification task) and in scenarios where new classes can be added often. The landmark paper by Fei-Fei and their co-authors, *'One Shot Learning of Object Categories'*, is supposedly what coined the term one-shot learning and the research in this sub-field. This paper presented a variation on a Bayesian framework for representation learning for object categorization. This approach has since been improved upon, and applied using deep learning systems.

Zero-shot Learning

Zero-shot learning is another extreme variant of transfer learning, which relies on no labeled examples to learn a task. This might sound unbelievable, especially when learning using examples is what most supervised learning algorithms are about. Zero-data learning or zero-short learning methods, make clever adjustments during the training stage itself to exploit additional information to understand unseen data. In their book on *Deep Learning*, Goodfellow and their co-authors present zero-shot learning as a scenario where three variables are learned, such as the traditional input variable, $\boldsymbol{x}$, the traditional output variable, $\boldsymbol{y}$, and the additional random variable that describes the task, $\boldsymbol{T}$. The model is thus trained to learn the conditional probability distribution of $\boldsymbol{P(y \mid x, T)}$. Zero-shot learning comes in handy in scenarios such as machine translation, where we may not even have labels in the target language.

Applications of Transfer Learning

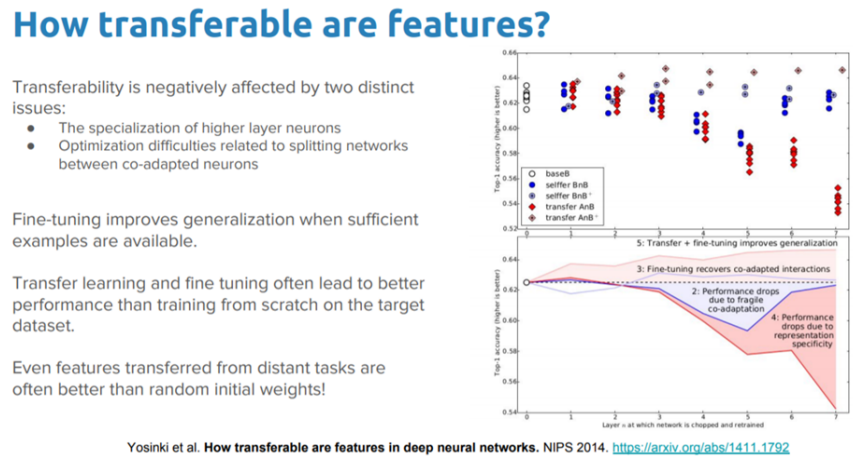
Deep learning is definitely one of the specific categories of algorithms that has been utilized to reap the benefits of transfer learning very successfully. The following are a few examples:

- **Transfer learning for NLP:** Textual data presents all sorts of challenges when it comes to ML and deep learning. These are usually transformed or vectorized using different techniques. Embeddings, such as Word2vec and FastText, have been prepared using different training datasets. These are utilized in different tasks, such as sentiment analysis and document classification, by transferring the knowledge from the source tasks. Besides this, newer models like the Universal Sentence Encoder and BERT definitely present a myriad of possibilities for the future.

- **Transfer learning for Audio/Speech:** Similar to domains like NLP and Computer Vision, deep learning has been successfully used for tasks based on audio data. For instance, Automatic Speech Recognition (ASR) models developed for English have been successfully used to improve speech recognition performance for other languages, such as German. Also, automated-speaker identification is another example where transfer learning has greatly helped.

- **Transfer learning for Computer Vision:** Deep learning has been quite successfully utilized for various computer vision tasks, such as object recognition and identification, using different CNN architectures. In their paper, How transferable are features in deep neural networks, Yosinski and their co-authors

(https://arxiv.org/abs/1411.1792) present their findings on how the lower layers act as conventional computer-vision feature extractors, such as edge detectors, while the final layers work toward task-specific features.



Yosinki et al. **How transferable are features in deep neural networks.** NIPS 2014. https://arxiv.org/abs/1411.1792

Thus, these findings have helped in utilizing existing state-of-the-art models, such as VGG, AlexNet, and Inceptions, for target tasks, such as style transfer and face detection, that were different from what these models were trained for initially. Let's explore some real-world case studies now and build some deep transfer learning models!

Case Study 1: Image Classification with a Data Availability Constraint

In this simple case study, will be working on an image categorization problem with the constraint of having a very small number of training samples per category. The dataset for our problem is available on Kaggle and is one of the most popular computer vision based datasets out there.

Main Objective

The dataset we will be using comes from the very popular ***Dogs vs. Cats Challenge***, where our primary objective is to build a deep learning model that can successfully recognize and categorize images into either a ***cat*** or a ***dog***.

Source: becominghuman.ai

In terms of ML, this is a binary classification problem based on images. Before getting started, I would like to thank *Francois Chollet* for not only creating the amazing deep learning framework, `keras`, but also for talking about the real-world problem where transfer learning is effective in his book, *'Deep Learning with Python'*. I've have taken that as an inspiration to portray the true power of transfer learning in this chapter and all results are based on building and running each model in my own GPU-based cloud setup (AWS p2.x)

Building Datasets

To start, download the `train.zip` file from the dataset page and store it in your local system. Once downloaded, unzip it into a folder. This folder will contain 25,000 images of dogs and cats; that is, 12,500 images per category. While we can use all 25,000 images and build some nice models on them, if you remember, our problem objective includes the added constraint of having a small number of images per category. Let's build our own dataset for this purpose.

```
import glob
import numpy as np
import os
import shutil
```

```
np.random.seed(42)
```

Let's now load up all the images in our original training data folder as follows:

```
files = glob.glob('train/*')

cat_files = [fn for fn in files if 'cat' in fn]
dog_files = [fn for fn in files if 'dog' in fn]
```

```
(12500, 12500)
```

We can verify with the preceding output that we have 12,500 images for each category. Let's now build our smaller dataset so that we have 3,000 images for training, 1,000 images for validation, and 1,000 images for our test dataset (with equal representation for the two animal categories).

```
cat_train = np.random.choice(cat_files, size=1500, re
dog_train = np.random.choice(dog_files, size=1500, re
cat_files = list(set(cat_files) - set(cat_train))
dog_files = list(set(dog_files) - set(dog_train))

cat_val = np.random.choice(cat_files, size=500, repla
dog_val = np.random.choice(dog_files, size=500, repla
cat_files = list(set(cat_files) - set(cat_val))
dog_files = list(set(dog_files) - set(dog_val))

cat_test = np.random.choice(cat_files, size=500, repl
```

```
Cat datasets: (1500,) (500,) (500,)
Dog datasets: (1500,) (500,) (500,)
```

Now that our datasets have been created, let's write them out to our disk in separate folders, so that we can come back to them anytime in the future without worrying if they are present in our main memory.

```
train_dir = 'training_data'
val_dir = 'validation_data'
test_dir = 'test_data'

train_files = np.concatenate([cat_train, dog_train])
validate_files = np.concatenate([cat_val, dog_val])
test_files = np.concatenate([cat_test, dog_test])

os.mkdir(train_dir) if not os.path.isdir(train_dir) e
os.mkdir(val_dir) if not os.path.isdir(val_dir) else
os.mkdir(test_dir) if not os.path.isdir(test_dir) els

for fn in train_files:
    shutil.copy(fn, train_dir)
```

Since this is an image categorization problem, we will be leveraging CNN models or ConvNets to try and tackle this problem. We will start by building simple CNN models from scratch, then try to improve using techniques such as regularization and image augmentation. Then, we will try and leverage pre-trained models to unleash the true power of transfer learning!

Preparing Datasets

Before we jump into modeling, let's load and prepare our datasets. To start with, we load up some basic dependencies.

```
import glob
import numpy as np
import matplotlib.pyplot as plt
from keras.preprocessing.image import
ImageDataGenerator, load_img, img_to_array,
array_to_img
```

```
%matplotlib inline
```

Let's now load our datasets, using the following code snippet.

```
IMG_DIM = (150, 150)

train_files = glob.glob('training_data/*')
train_imgs = [img_to_array(load_img(img, target_size=
train_imgs = np.array(train_imgs)
train_labels = [fn.split('\\')[1].split('.')[0].strip

validation_files = glob.glob('validation_data/*')
validation_imgs = [img_to_array(load_img(img, target_
validation_imgs = np.array(validation_imgs)
```

```
Train dataset shape: (3000, 150, 150, 3)
Validation dataset shape: (1000, 150, 150, 3)
```

We can clearly see that we have 3000 training images and 1000 validation images. Each image is of size `150 x 150` and has three channels for red, green, and blue (RGB), hence giving each image the `(150, 150, 3)` dimensions. We will now scale each image with pixel values between `(0, 255)` to values between `(0, 1)` because deep learning models work really well with small input values.

```
train_imgs_scaled = train_imgs.astype('float32')
validation_imgs_scaled  = validation_imgs.astype('floa
train_imgs_scaled /= 255
validation_imgs_scaled /= 255

print(train_imgs[0].shape)
```

```
(150, 150, 3)
```

The preceding output shows one of the sample images from our training dataset. Let's now set up some basic configuration parameters

and also encode our text class labels into numeric values (otherwise, Keras will throw an error).

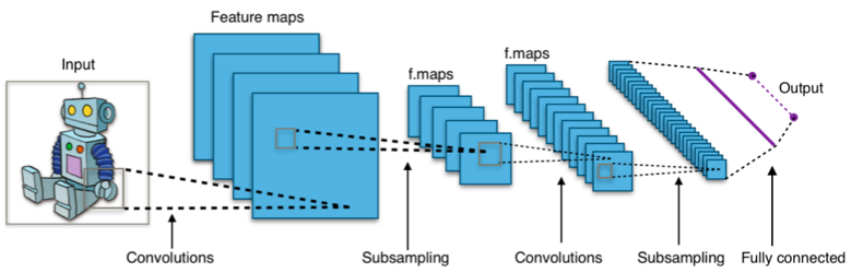
```
batch_size = 30
num_classes = 2
epochs = 30
input_shape = (150, 150, 3)

# encode text category labels
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
le.fit(train_labels)
```

```
['cat', 'cat', 'cat', 'cat', 'cat', 'dog', 'dog',
'dog', 'dog', 'dog'] [0 0 0 0 0 1 1 1 1 1]
```

We can see that our encoding scheme assigns the number `0` to the `cat` labels and `1` to the `dog` labels. We are now ready to build our first CNN-based deep learning model.

Simple CNN Model from Scratch

We will start by building a basic CNN model with three convolutional layers, coupled with max pooling for auto-extraction of features from our images and also downsampling the output convolution feature maps.



A Typical CNN (Source: Wikipedia)

We assume you have enough knowledge about CNNs and hence won't cover theoretical details. Feel free to refer to my book or any other resources on the web which explain convolutional neural networks! Let's leverage Keras and build our CNN model architecture now.

```
from keras.layers import Conv2D, MaxPooling2D, Flatte
from keras.models import Sequential
from keras import optimizers

model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation='
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, kernel_size=(3, 3), activation='
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation=
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(1, activation='sigmoid'))


model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(),
              metrics=['accuracy'])

model.summary()


'''
Layer (type)                 Output Shape
=================================================
conv2d_1 (Conv2D)            (None, 148, 148, 16)
_________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 16)

```

The preceding output shows us our basic CNN model summary. Just like we mentioned before, we are using three convolutional layers for feature extraction. The flatten layer is used to flatten out `128` of the `17 x 17` feature maps that we get as output from the third convolution layer. This is fed to our dense layers to get the final prediction of whether the image should be a ***dog (1)*** or a ***cat (0)***. All of this is part of the model training process, so let's train our model using the following snippet which leverages the `fit(…)` function.

The following terminology is very important with regard to training our model:

- The `batch_size` indicates the total number of images passed to the model per iteration.
- The weights of the units in layers are updated after each iteration.
- The total number of iterations is always equal to the total number of training samples divided by the `batch_size`
- An epoch is when the complete dataset has passed through the network once, that is, all the iterations are completed based on data batches.

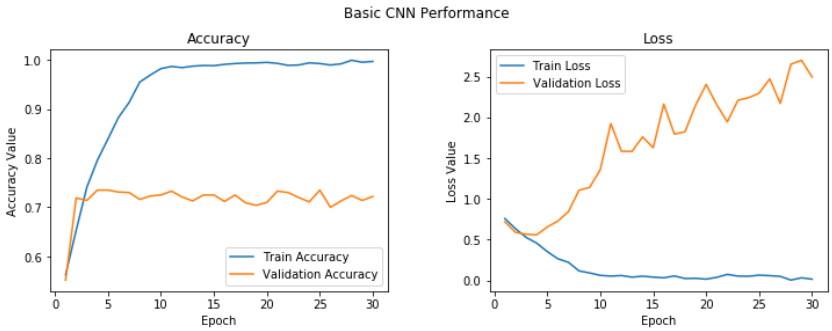
We use a `batch_size` of 30 and our training data has a total of 3,000 samples, which indicates that there will be a total of 100 iterations per epoch. We train the model for a total of 30 epochs and validate it consequently on our validation set of 1,000 images.

```
history = model.fit(x=train_imgs_scaled, y=train_label
                    validation_data=(validation_imgs_s
                    batch_size=batch_size,
                    epochs=epochs,
```

```
Train on 3000 samples, validate on 1000 samples
Epoch 1/30
3000/3000 - 10s - loss: 0.7583 - acc: 0.5627 -
val_loss: 0.7182 - val_acc: 0.5520
Epoch 2/30
3000/3000 - 8s - loss: 0.6343 - acc: 0.6533 -
val_loss: 0.5891 - val_acc: 0.7190
...
...
Epoch 29/30
3000/3000 - 8s - loss: 0.0314 - acc: 0.9950 -
val_loss: 2.7014 - val_acc: 0.7140
Epoch 30/30
3000/3000 - 8s - loss: 0.0147 - acc: 0.9967 -
val_loss: 2.4963 - val_acc: 0.7220
```

Looks like our model is kind of overfitting, based on the training and validation accuracy values. We can plot our model accuracy and errors using the following snippet to get a better perspective.

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
t = f.suptitle('Basic CNN Performance', fontsize=12)
f.subplots_adjust(top=0.85, wspace=0.3)

epoch_list = list(range(1,31))
ax1.plot(epoch_list, history.history['acc'], label='T
ax1.plot(epoch_list, history.history['val_acc'], labe
ax1.set_xticks(np.arange(0, 31, 5))
ax1.set_ylabel('Accuracy Value')
ax1.set_xlabel('Epoch')
ax1.set_title('Accuracy')
l1 = ax1.legend(loc="best")

ax2.plot(epoch_list, history.history['loss'], label='
```



Vanilla CNN Model Performance

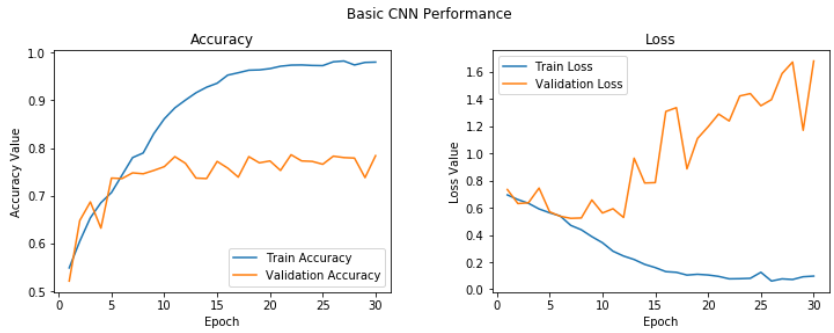
You can clearly see that after 2–3 epochs the model starts overfitting on the training data. The average accuracy we get in our validation set is around **72%**, which is not a bad start! Can we improve upon this model?

CNN Model with Regularization

Let's improve upon our base CNN model by adding in one more convolution layer, another dense hidden layer. Besides this, we will add dropout of `0.3` after each hidden dense layer to enable regularization. Basically, dropout is a powerful method of regularizing in deep neural nets. It can be applied separately to both input layers and the hidden layers. Dropout randomly masks the outputs of a fraction of units from a layer by setting their output to zero (in our case, it is 30% of the units in our dense layers).

```
model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation='
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, kernel_size=(3, 3), activation='
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation=
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation=
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

```

```
Train on 3000 samples, validate on 1000 samples
Epoch 1/30
3000/3000 - 7s - loss: 0.6945 - acc: 0.5487 -
val_loss: 0.7341 - val_acc: 0.5210
Epoch 2/30
3000/3000 - 7s - loss: 0.6601 - acc: 0.6047 -
val_loss: 0.6308 - val_acc: 0.6480
...
...
Epoch 29/30
3000/3000 - 7s - loss: 0.0927 - acc: 0.9797 -
val_loss: 1.1696 - val_acc: 0.7380
Epoch 30/30
3000/3000 - 7s - loss: 0.0975 - acc: 0.9803 -
val_loss: 1.6790 - val_acc: 0.7840
```



Vanilla CNN Model with Regularization Performance

You can clearly see from the preceding outputs that we still end up overfitting the model, though it takes slightly longer and we also get a slightly better validation accuracy of around **78%**, which is decent but not amazing. The reason for model overfitting is because we have much less training data and the model keeps seeing the same instances over time across each epoch. A way to combat this would be to leverage an image augmentation strategy to augment our existing training data with images that are slight variations of the existing images. We will cover this in detail in the following section. Let's save this model for the time being so we can use it later to evaluate its performance on the test data.

```
model.save('cats_dogs_basic_cnn.h5')
```

CNN Model with Image Augmentation

Let's improve upon our regularized CNN model by adding in more data using a proper image augmentation strategy. Since our previous model was trained on the same small sample of data points each time, it wasn't able to generalize well and ended up overfitting after a few epochs. The idea behind image augmentation is that we follow a set process of taking in existing images from our training dataset and applying some image transformation operations to them, such as rotation, shearing, translation, zooming, and so on, to produce new, altered versions of existing images. Due to these random transformations, we don't get the same images each time, and we will leverage Python generators to feed in these new images to our model during training.

The Keras framework has an excellent utility called `ImageDataGenerator` that can help us in doing all the preceding

operations. Let's initialize two of the data generators for our training and validation datasets.

```
train_datagen = ImageDataGenerator(rescale=1./255, zoc
                                   width_shift_range=0
                                   horizontal_flip=Tru

```

There are a lot of options available in `ImageDataGenerator` and we have just utilized a few of them. Feel free to check out the ***documentation*** to get a more detailed perspective. In our training data generator, we take in the raw images and then perform several transformations on them to generate new images. These include the following.

- Zooming the image randomly by a factor of `0.3` using the `zoom_range` parameter.
- Rotating the image randomly by `50` degrees using the `rotation_range` parameter.
- Translating the image randomly horizontally or vertically by a `0.2` factor of the image's width or height using the `width_shift_range` and the `height_shift_range` parameters.
- Applying shear-based transformations randomly using the `shear_range` parameter.
- Randomly flipping half of the images horizontally using the `horizontal_flip` parameter.
- Leveraging the `fill_mode` parameter to fill in new pixels for images after we apply any of the preceding operations (especially rotation or translation). In this case, we just fill in the new pixels with their nearest surrounding pixel values.

Let's see how some of these generated images might look so that you can understand them better. We will take two sample images from our training dataset to illustrate the same. The first image is an image of a cat.

```
img_id = 2595
cat_generator = train_datagen.flow(train_imgs[img_id:i
                                   batch_size=1)
cat = [next(cat_generator) for i in range(0,5)]
fig, ax = plt.subplots(1,5, figsize=(16, 6))
print('Labels:', [item[1][0] for item in cat])
```

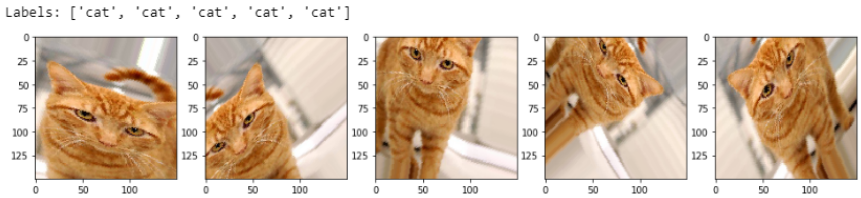


Image Augmentation on a Cat Image

You can clearly see in the previous output that we generate a new version of our training image each time (with translations, rotations, and zoom) and also we assign a label of cat to it so that the model can extract relevant features from these images and also remember that these are cats. Let's look at how image augmentation works on a sample dog image now.

```
img_id = 1991
dog_generator = train_datagen.flow(train_imgs[img_id:i
                                   batch_size=1)
dog = [next(dog_generator) for i in range(0,5)]
fig, ax = plt.subplots(1,5, figsize=(15, 6))
print('Labels:', [item[1][0] for item in dog])
```

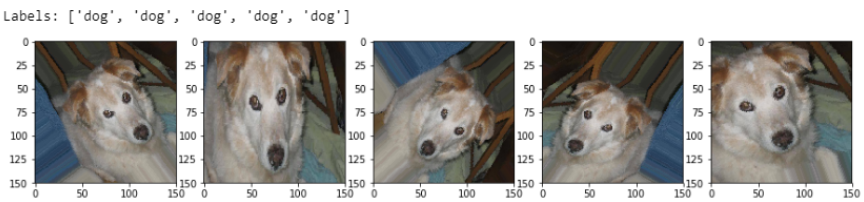


Image Augmentation on a Dog Image

This shows us how image augmentation helps in creating new images, and how training a model on them should help in combating overfitting. Remember for our validation generator, we just need to send the validation images (original ones) to the model for evaluation; hence, we just scale the image pixels (between 0–1) and do not apply any transformations. We just apply image augmentation transformations only on our training images. Let's now train a CNN

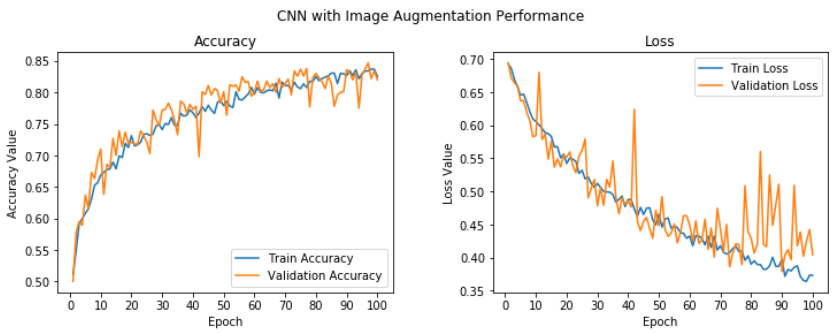
model with regularization using the image augmentation data generators we created. We will use the same model architecture from before.

```
train_generator = train_datagen.flow(train_imgs, trai
val_generator = val_datagen.flow(validation_imgs, val
input_shape = (150, 150, 3)

from keras.layers import Conv2D, MaxPooling2D, Flatte
from keras.models import Sequential
from keras import optimizers

model = Sequential()

model.add(Conv2D(16, kernel_size=(3, 3), activation='
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, kernel_size=(3, 3), activation='
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation=
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, kernel_size=(3, 3), activation=
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
```

We reduce the default learning rate by a factor of 10 here for our optimizer to prevent the model from getting stuck in a local minima or overfit, as we will be sending a lot of images with random transformations. To train the model, we need to slightly modify our approach now, since we are using data generators. We will leverage the `fit_generator(…)` function from Keras to train this model. The `train_generator` generates 30 images each time, so we will use the `steps_per_epoch` parameter and set it to 100 to train the model on 3,000 randomly generated images from the training data for each epoch. Our `val_generator` generates 20 images each time so we will set the `validation_steps` parameter to 50 to validate our model accuracy on all the 1,000 validation images (remember we are not augmenting our validation dataset).

```
Epoch 1/100
100/100 - 12s - loss: 0.6924 - acc: 0.5113 - val_loss:
0.6943 - val_acc: 0.5000
Epoch 2/100
100/100 - 11s - loss: 0.6855 - acc: 0.5490 - val_loss:
0.6711 - val_acc: 0.5780
Epoch 3/100
100/100 - 11s - loss: 0.6691 - acc: 0.5920 - val_loss:
0.6642 - val_acc: 0.5950
...
...
Epoch 99/100
100/100 - 11s - loss: 0.3735 - acc: 0.8367 - val_loss:
0.4425 - val_acc: 0.8340
Epoch 100/100
100/100 - 11s - loss: 0.3733 - acc: 0.8257 - val_loss:
0.4046 - val_acc: 0.8200
```

We get a validation accuracy jump to around **82%**, which is almost **4–5%** better than our previous model. Also, our training accuracy is very similar to our validation accuracy, indicating our model isn't overfitting anymore. The following depict the model accuracy and loss per epoch.



Vanilla CNN Model with Image Augmentation Performance

While there are some spikes in the validation accuracy and loss, overall, we see that it is much closer to the training accuracy, with the loss indicating that we obtained a model that generalizes much better as compared to our previous models. Let's save this model now so we can evaluate it later on our test dataset.

```
model.save('cats_dogs_cnn_img_aug.h5')
```

We will now try and leverage the power of transfer learning to see if we can build a better model!

Leveraging Transfer Learning with Pre-trained CNN Models

Pre-trained models are used in the following two popular ways when building new models or reusing them:

- Using a pre-trained model as a feature extractor
- Fine-tuning the pre-trained model

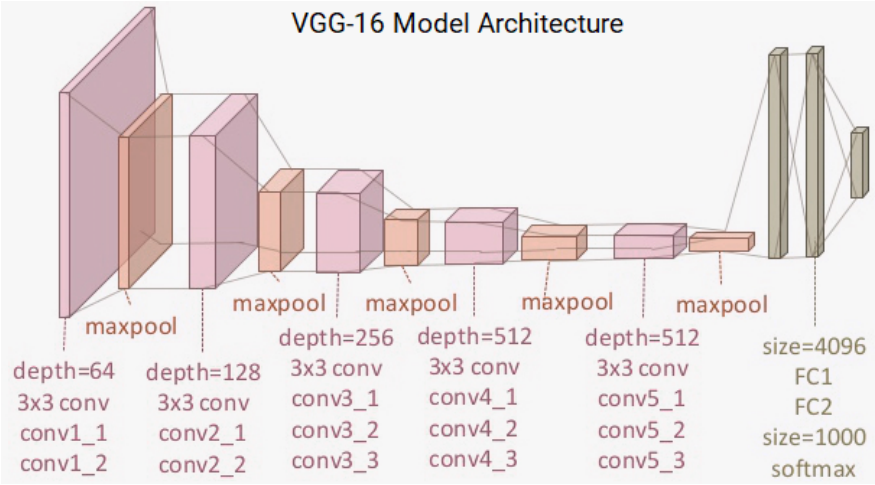
We will cover both of them in detail in this section. The pre-trained model we will be using in this chapter is the popular VGG-16 model, created by the ***Visual Geometry Group*** at the University of Oxford, which specializes in building very deep convolutional networks for large-scale visual recognition.

A pretrained model like the VGG-16 is an already pre-trained model on a huge dataset (ImageNet) with a lot of diverse image categories. Considering this fact, the model should have learned a robust hierarchy of features, which are spatial, rotation, and translation invariant with regard to features learned by CNN models. Hence, the model, having learned a good representation of features for over a million images belonging to 1,000 different categories, can act as a good feature extractor for new images suitable for computer vision problems. These new images might never exist in the ImageNet dataset or might be of totally different categories, but the model should still be able to extract relevant features from these images.

This gives us an advantage of using pre-trained models as effective feature extractors for new images, to solve diverse and complex computer vision tasks, such as solving our cat versus dog classifier with fewer images, or even building a dog breed classifier, a facial expression classifier, and much more! Let's briefly discuss the VGG-16 model architecture before unleashing the power of transfer learning on our problem.

Understanding the VGG-16 model

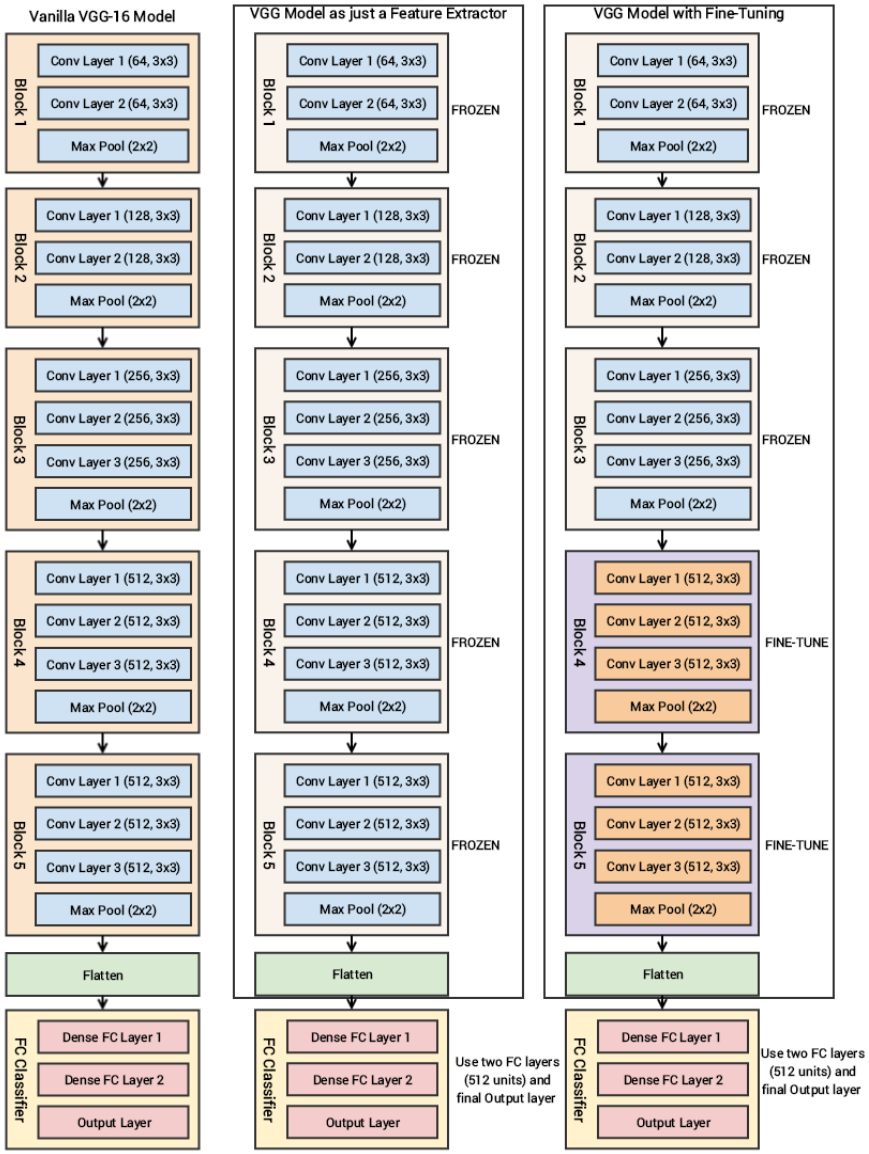
The VGG-16 model is a 16-layer (convolution and fully connected) network built on the ImageNet database, which is built for the purpose of image recognition and classification. This model was built by Karen Simonyan and Andrew Zisserman and is mentioned in their paper titled *'Very Deep Convolutional Networks for Large-Scale Image Recognition'*. I recommend all interested readers to go and read up on the excellent literature in this paper. The architecture of the VGG-16 model is depicted in the following figure.



VGG-16 Model Architecture

You can clearly see that we have a total of `13` convolution layers using `3 x 3` convolution filters along with max pooling layers for downsampling and a total of two fully connected hidden layers of `4096` units in each layer followed by a dense layer of `1000` units, where each unit represents one of the image categories in the ImageNet database. We do not need the last three layers since we will be using our own fully connected dense layers to predict whether images will be a dog or a cat. We are more concerned with the first five blocks, so that we can leverage the VGG model as an effective feature extractor.

For one of the models, we will use it as a simple feature extractor by freezing all the five convolution blocks to make sure their weights don't get updated after each epoch. For the last model, we will apply fine-tuning to the VGG model, where we will unfreeze the last two blocks (Block 4 and Block 5) so that their weights get updated in each epoch (per batch of data) as we train our own model. We represent the preceding architecture, along with the two variants (basic feature extractor and fine-tuning) that we will be using, in the following block diagram, so you can get a better visual perspective.



Block Diagram showing Transfer Learning Strategies on the VGG-16 Model

Thus, we are mostly concerned with leveraging the convolution blocks of the VGG-16 model and then flattening the final output (from the feature maps) so that we can feed it into our own dense layers for our classifier.

Pre-trained CNN model as a Feature Extractor

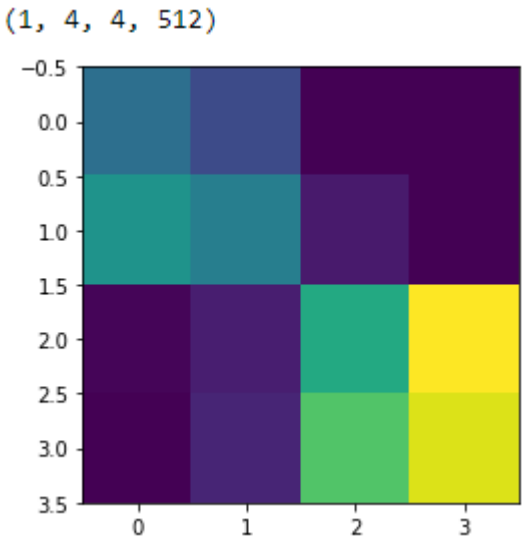
Let's leverage Keras, load up the VGG-16 model, and freeze the convolution blocks so that we can use it as just an image feature extractor.

```
from keras.applications import vgg16
from keras.models import Model
import keras

vgg = vgg16.VGG16(include_top=False, weights='imagene
                                     input_shape=inpu

output = vgg.layers[-1].output
output = keras.layers.Flatten()(output)
vgg_model = Model(vgg.input, output)

vgg_model.trainable = False
for layer in vgg_model.layers:
```

	Layer Type	Layer Name	Layer Trainable
0	<keras.engine.topology.InputLayer object at 0x7f26c86b2518>	input_1	False
1	<keras.layers.convolutional.Conv2D object at 0x7f277c9fc080>	block1_conv1	False
2	<keras.layers.convolutional.Conv2D object at 0x7f26c86b26d8>	block1_conv2	False
3	<keras.layers.pooling.MaxPooling2D object at 0x7f26c86e6c88>	block1_pool	False
4	<keras.layers.convolutional.Conv2D object at 0x7f26c867dc18>	block2_conv1	False
5	<keras.layers.convolutional.Conv2D object at 0x7f26c8690f28>	block2_conv2	False
6	<keras.layers.pooling.MaxPooling2D object at 0x7f26c869e5c0>	block2_pool	False
7	<keras.layers.convolutional.Conv2D object at 0x7f26c863f828>	block3_conv1	False
8	<keras.layers.convolutional.Conv2D object at 0x7f26c863f128>	block3_conv2	False
9	<keras.layers.convolutional.Conv2D object at 0x7f26c86607b8>	block3_conv3	False
10	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83d7d68>	block3_pool	False
11	<keras.layers.convolutional.Conv2D object at 0x7f26c83fd358>	block4_conv1	False
12	<keras.layers.convolutional.Conv2D object at 0x7f26c83fddd8>	block4_conv2	False
13	<keras.layers.convolutional.Conv2D object at 0x7f26c839da20>	block4_conv3	False
14	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83ac1d0>	block4_pool	False
15	<keras.layers.convolutional.Conv2D object at 0x7f26c834e978>	block5_conv1	False
16	<keras.layers.convolutional.Conv2D object at 0x7f271a15eb38>	block5_conv2	False
17	<keras.layers.convolutional.Conv2D object at 0x7f26c8371d68>	block5_conv3	False
18	<keras.layers.pooling.MaxPooling2D object at 0x7f26c8314b00>	block5_pool	False
19	<keras.layers.core.Flatten object at 0x7f26c828bda0>	flatten_1	False

It is quite clear from the preceding output that all the layers of the VGG-16 model are frozen, which is good because we don't want their weights to change during model training. The last activation feature map in the VGG-16 model (output from `block5_pool`) gives us the bottleneck features, which can then be flattened and fed to a fully connected deep neural network classifier. The following snippet shows what the bottleneck features look like for a sample image from our training data.

```
bottleneck_feature_example =
vgg.predict(train_imgs_scaled[0:1])
print(bottleneck_feature_example.shape)
plt.imshow(bottleneck_feature_example[0][:,:,0])
```



Sample Bottleneck Features

We flatten the bottleneck features in the vgg_model object to make them ready to be fed to our fully connected classifier. A way to save time in model training is to use this model and extract out all the features from our training and validation datasets and then feed them as inputs to our classifier. Let's extract out the bottleneck features from our training and validation sets now.

```
def get_bottleneck_features(model, input_imgs):
    features = model.predict(input_imgs, verbose=0)
    return features

train_features_vgg = get_bottleneck_features(vgg_model
validation_features_vgg = get_bottleneck_features(vgg_
```

```
Train Bottleneck Features: (3000, 8192)
Validation Bottleneck Features: (1000, 8192)
```

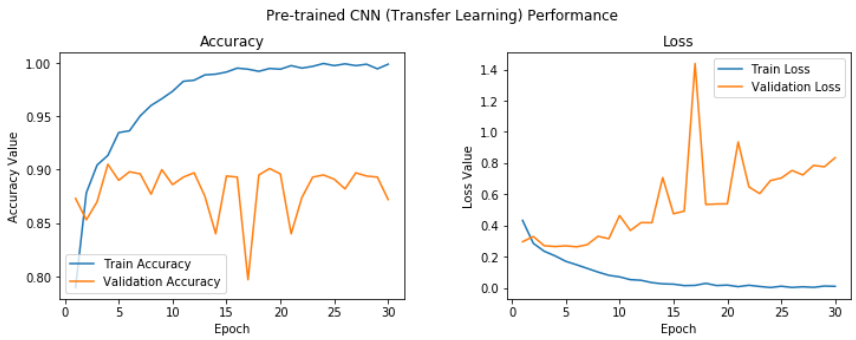
The preceding output tells us that we have successfully extracted the flattened bottleneck features of dimension `1 x 8192` for our 3,000 training images and our 1,000 validation images. Let's build the architecture of our deep neural network classifier now, which will take these features as input.

```
from keras.layers import Conv2D, MaxPooling2D, Flatte
from keras.models import Sequential
from keras import optimizers

input_shape = vgg_model.output_shape[1]

model = Sequential()
model.add(InputLayer(input_shape=(input_shape,)))
model.add(Dense(512, activation='relu', input_dim=inp
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['accuracy'])

model.summary()


'''
_________________________________________________________
Layer (type)                 Output Shape
=========================================================
input_2 (InputLayer)         (None, 8192)
_________________________________________________________
```

Just like we mentioned previously, bottleneck feature vectors of size 8192 serve as input to our classification model. We use the same architecture as our previous models here with regard to the dense layers. Let's train this model now.

```
history = model.fit(x=train_features_vgg, y=train_labe
                    validation_data=(validation_featur
                    batch_size=batch_size,
                    epochs=epochs,
```

```
Train on 3000 samples, validate on 1000 samples
Epoch 1/30
3000/3000 - 1s 373us/step - loss: 0.4325 - acc: 0.7897
- val_loss: 0.2958 - val_acc: 0.8730
Epoch 2/30
3000/3000 - 1s 286us/step - loss: 0.2857 - acc: 0.8783
- val_loss: 0.3294 - val_acc: 0.8530
Epoch 3/30
3000/3000 - 1s 289us/step - loss: 0.2353 - acc: 0.9043
- val_loss: 0.2708 - val_acc: 0.8700
...
...
Epoch 29/30
3000/3000 - 1s 287us/step - loss: 0.0121 - acc: 0.9943
- val_loss: 0.7760 - val_acc: 0.8930
Epoch 30/30
3000/3000 - 1s 287us/step - loss: 0.0102 - acc: 0.9987
- val_loss: 0.8344 - val_acc: 0.8720
```



Pre-trained CNN (feature extractor) Performance

We get a model with a validation accuracy of close to 88%, almost a 5–6% improvement from our basic CNN model with image augmentation, which is excellent. The model does seem to be overfitting though. There is a decent gap between the model train and validation accuracy after the fifth epoch, which kind of makes it clear that the model is overfitting on the training data after that. But overall, this seems to be the best model so far. Let's try using our image augmentation strategy on this model. Before that, we save this model to disk using the following code.

```
model.save('cats_dogs_tlearn_basic_cnn.h5')
```

Pre-trained CNN model as a Feature Extractor with Image Augmentation

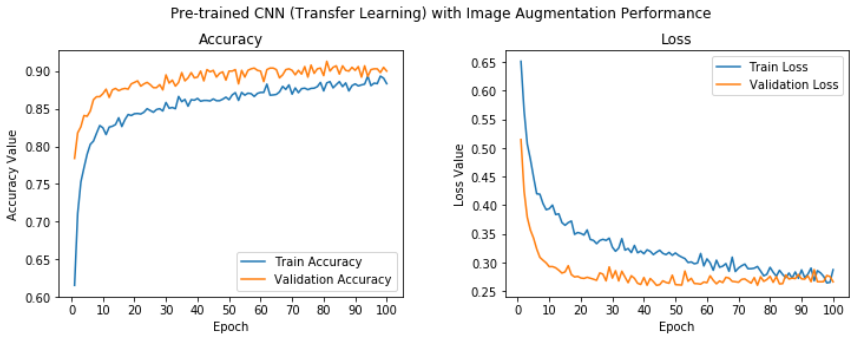
We will leverage the same data generators for our train and validation datasets that we used before. The code for building them is depicted as follows for ease of understanding.

```
train_datagen = ImageDataGenerator(rescale=1./255, zoo
                                   width_shift_range=0
                                   horizontal_flip=Tru

val_datagen = ImageDataGenerator(rescale=1./255)

```

Let's now build our deep learning model and train it. We won't extract the bottleneck features like last time since we will be training on data generators; hence, we will be passing the `vgg_model` object as an input to our own model. We bring the learning rate slightly down since we will be training for 100 epochs and don't want to make any sudden abrupt weight adjustments to our model layers. Do remember that the VGG-16 model's layers are still frozen here and we are still using it as a basic feature extractor only.

```
from keras.layers import Conv2D, MaxPooling2D, Flatte
from keras.models import Sequential
from keras import optimizers

model = Sequential()
model.add(vgg_model)
model.add(Dense(512, activation='relu', input_dim=inp
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
```

```
Epoch 1/100
100/100 – 45s 449ms/step – loss: 0.6511 – acc: 0.6153
– val_loss: 0.5147 – val_acc: 0.7840
Epoch 2/100
100/100 – 41s 414ms/step – loss: 0.5651 – acc: 0.7110
– val_loss: 0.4249 – val_acc: 0.8180
Epoch 3/100
100/100 – 41s 415ms/step – loss: 0.5069 – acc: 0.7527
– val_loss: 0.3790 – val_acc: 0.8260
...
...
Epoch 99/100
100/100 – 42s 417ms/step – loss: 0.2656 – acc: 0.8907
– val_loss: 0.2757 – val_acc: 0.9050
Epoch 100/100
100/100 – 42s 418ms/step – loss: 0.2876 – acc: 0.8833
– val_loss: 0.2665 – val_acc: 0.9000
```



Pre-trained CNN (feature extractor) with Image Augmentation Performance

We can see that our model has an overall validation accuracy of **90%**, which is a slight improvement from our previous model, and also the train and validation accuracy are quite close to each other, indicating that the model is not overfitting. Let's save this model on the disk now for future evaluation on the test data.

```
model.save('cats_dogs_tlearn_img_aug_cnn.h5')
```

We will now fine-tune the VGG-16 model to build our last classifier, where we will unfreeze blocks 4 and 5, as we depicted in our block diagram earlier.

Pre-trained CNN model with Fine-tuning and Image Augmentation

We will now leverage our VGG-16 model object stored in the `vgg_model` variable and unfreeze convolution blocks 4 and 5 while keeping the first three blocks frozen. The following code helps us achieve this.

```
vgg_model.trainable = True

set_trainable = False
for layer in vgg_model.layers:
    if layer.name in ['block5_conv1', 'block4_conv1']
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
```

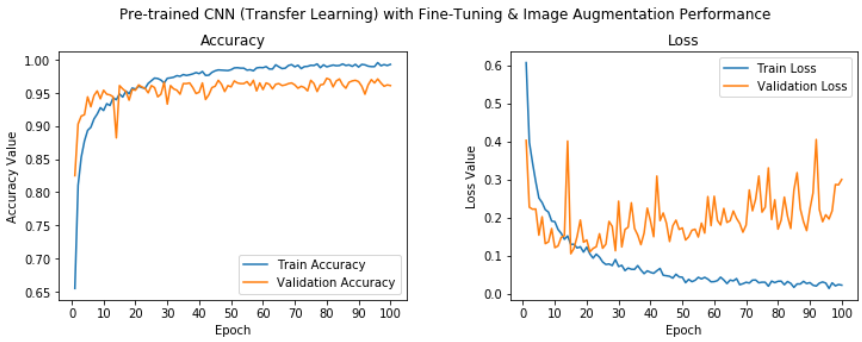
	Layer Type	Layer Name	Layer Trainable
0	<keras.engine.topology.InputLayer object at 0x7f26c86b2518>	input_1	False
1	<keras.layers.convolutional.Conv2D object at 0x7f277c9fc080>	block1_conv1	False
2	<keras.layers.convolutional.Conv2D object at 0x7f26c86b26d8>	block1_conv2	False
3	<keras.layers.pooling.MaxPooling2D object at 0x7f26c86e6c88>	block1_pool	False
4	<keras.layers.convolutional.Conv2D object at 0x7f26c867dc18>	block2_conv1	False
5	<keras.layers.convolutional.Conv2D object at 0x7f26c8690f28>	block2_conv2	False
6	<keras.layers.pooling.MaxPooling2D object at 0x7f26c869e5c0>	block2_pool	False
7	<keras.layers.convolutional.Conv2D object at 0x7f26c863f828>	block3_conv1	False
8	<keras.layers.convolutional.Conv2D object at 0x7f26c863f128>	block3_conv2	False
9	<keras.layers.convolutional.Conv2D object at 0x7f26c86607b8>	block3_conv3	False
10	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83d7d68>	block3_pool	False
11	<keras.layers.convolutional.Conv2D object at 0x7f26c83fd358>	block4_conv1	True
12	<keras.layers.convolutional.Conv2D object at 0x7f26c83fddd8>	block4_conv2	True
13	<keras.layers.convolutional.Conv2D object at 0x7f26c839da20>	block4_conv3	True
14	<keras.layers.pooling.MaxPooling2D object at 0x7f26c83ac1d0>	block4_pool	True
15	<keras.layers.convolutional.Conv2D object at 0x7f26c834e978>	block5_conv1	True
16	<keras.layers.convolutional.Conv2D object at 0x7f271a15eb38>	block5_conv2	True
17	<keras.layers.convolutional.Conv2D object at 0x7f26c8371d68>	block5_conv3	True
18	<keras.layers.pooling.MaxPooling2D object at 0x7f26c8314b00>	block5_pool	True
19	<keras.layers.core.Flatten object at 0x7f26c828bda0>	flatten_1	True

You can clearly see from the preceding output that the convolution and pooling layers pertaining to blocks 4 and 5 are now trainable. This means the weights for these layers will also get updated with backpropagation in each epoch as we pass each batch of data. We will use the same data generators and model architecture as our previous model and train our model. We reduce the learning rate slightly since we don't want to get stuck at any local minimal, and we also do not

want to suddenly update the weights of the trainable VGG-16 model layers by a big factor that might adversely affect the model.

```
train_datagen = ImageDataGenerator(rescale=1./255, zc
                                   width_shift_range=
                                   horizontal_flip=Tr
val_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow(train_imgs, trai
val_generator = val_datagen.flow(validation_imgs, val

from keras.layers import Conv2D, MaxPooling2D, Flatte
from keras.models import Sequential
from keras import optimizers

model = Sequential()
model.add(vgg_model)
model.add(Dense(512, activation='relu', input_dim=inp
model.add(Dropout(0.3))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.3))
```

```
Epoch 1/100
100/100 - 64s 642ms/step - loss: 0.6070 - acc: 0.6547
- val_loss: 0.4029 - val_acc: 0.8250
Epoch 2/100
100/100 - 63s 630ms/step - loss: 0.3976 - acc: 0.8103
- val_loss: 0.2273 - val_acc: 0.9030
Epoch 3/100
100/100 - 63s 631ms/step - loss: 0.3440 - acc: 0.8530
- val_loss: 0.2221 - val_acc: 0.9150
...
...
Epoch 99/100
100/100 - 63s 629ms/step - loss: 0.0243 - acc: 0.9913
- val_loss: 0.2861 - val_acc: 0.9620
Epoch 100/100
100/100 - 63s 629ms/step - loss: 0.0226 - acc: 0.9930
- val_loss: 0.3002 - val_acc: 0.9610
```



Pre-trained CNN (fine-tuning) with Image Augmentation Performance

We can see from the preceding output that our model has obtained a validation accuracy of around **96%**, which is a **6%** improvement from our previous model. Overall, this model has gained a **24%** improvement in validation accuracy from our first basic CNN model. This really shows how useful transfer learning can be. We can see that accuracy values are really excellent here, and although the model looks like it might be slightly overfitting on the training data, we still get great validation accuracy. Let's save this model to disk now using the following code.

```
model.save('cats_dogs_tlearn_finetune_img_aug_cnn.h5')
```

Let's now put all our models to the test by actually evaluating their performance on our test dataset.

Evaluating our Deep Learning Models on Test Data

We will now evaluate the five different models we built so far by first testing them on our test dataset, because just validation is not enough! We have also built a nifty utility module called `model_evaluation_utils`, which we will be using to evaluate the performance of our deep learning models. Let's load up the necessary dependencies and our saved models before getting started.

```
# load dependencies
import glob
import numpy as np
import matplotlib.pyplot as plt
from keras.preprocessing.image import load_img, img_t
from keras.models import load_model
import model_evaluation_utils as meu
%matplotlib inline

# load saved models
basic_cnn = load_model('cats_dogs_basic_cnn.h5')
img_aug_cnn = load_model('cats_dogs_cnn_img_aug.h5')
tl_cnn = load_model('cats_dogs_tlearn_basic_cnn.h5')
tl_img_aug_cnn = load_model('cats_dogs_tlearn_img_aug
tl_img_aug_finetune_cnn = load_model('cats_dogs_tlear

# load other configurations
IMG_DIM = (150, 150)
input_shape = (150, 150, 3)
num2class_label_transformer = lambda l: ['cat' if x =
class2num_label_transformer = lambda l: [0 if x == 'c

# load VGG model for bottleneck features
from keras.applications import vgg16
```

It's time now for the final test, where we literally test the performance of our models by making predictions on our test dataset. Let's load up and prepare our test dataset first before we try making predictions.

```
IMG_DIM = (150, 150)

test_files = glob.glob('test_data/*')
test_imgs = [img_to_array(load_img(img, target_size=I
test_imgs = np.array(test_imgs)
test_labels = [fn.split('/')[1].split('.')[0].strip()

test_imgs_scaled = test_imgs.astype('float32')
test_imgs_scaled /= 255
```

```
Test dataset shape: (1000, 150, 150, 3)
['dog', 'dog', 'dog', 'dog', 'dog'] [1, 1, 1, 1, 1]
```

Now that we have our scaled dataset ready, let's evaluate each model by making predictions for all the test images, and then evaluate the model performance by checking how accurate are the predictions.

Model 1: Basic CNN Performance

```
predictions = basic_cnn.predict_classes(test_imgs_scal
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test
                                      classes=list(set
```

```
Model Performance metrics:
--------------------------
Accuracy: 0.776
Precision: 0.7769
Recall: 0.776
F1 Score: 0.7758

Model Classification report:
----------------------------
             precision    recall  f1-score   support

        cat       0.76      0.80      0.78       500
        dog       0.79      0.75      0.77       500

avg / total       0.78      0.78      0.78      1000

Prediction Confusion Matrix:
----------------------------
            Predicted:
                cat  dog
Actual: cat     402   98
        dog     126  374
```

Model 2: Basic CNN with Image Augmentation Performance

```
predictions = img_aug_cnn.predict_classes(test_imgs_sc
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test
                                      classes=list(set
```

```
Model Performance metrics:
--------------------------
Accuracy: 0.844
Precision: 0.844
Recall: 0.844
F1 Score: 0.844

Model Classification report:
----------------------------
             precision    recall  f1-score   support

        cat       0.84      0.84      0.84       500
        dog       0.84      0.84      0.84       500

avg / total       0.84      0.84      0.84      1000

Prediction Confusion Matrix:
----------------------------
            Predicted:
                cat  dog
Actual: cat     422   78
        dog      78  422
```

Model 3: Transfer Learning—Pre-trained CNN as a Feature Extractor Performance

```
test_bottleneck_features = get_bottleneck_features(vgg

predictions = tl_cnn.predict_classes(test_bottleneck_f
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test
```

```
Model Performance metrics:
-------------------------
Accuracy: 0.888
Precision: 0.8898
Recall: 0.888
F1 Score: 0.8879

Model Classification report:
---------------------------
              precision    recall  f1-score   support

         cat       0.92      0.85      0.88       500
         dog       0.86      0.92      0.89       500

   avg / total       0.89      0.89      0.89      1000

Prediction Confusion Matrix:
---------------------------
             Predicted:
                   cat  dog
Actual: cat        427   73
        dog         39  461
```

Model 4: Transfer Learning—Pre-trained CNN as a Feature Extractor with Image Augmentation Performance

```
predictions = tl_img_aug_cnn.predict_classes(test_imgs
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test
                                      classes=list(set
```

```
Model Performance metrics:
-------------------------
Accuracy: 0.898
Precision: 0.8981
Recall: 0.898
F1 Score: 0.898

Model Classification report:
---------------------------
              precision    recall  f1-score   support

         cat       0.89      0.91      0.90       500
         dog       0.90      0.89      0.90       500

   avg / total       0.90      0.90      0.90      1000

Prediction Confusion Matrix:
---------------------------
             Predicted:
                   cat  dog
Actual: cat        453   47
        dog         55  445
```

Model 5: Transfer Learning—Pre-trained CNN with Fine-tuning and Image Augmentation Performance

```
predictions = tl_img_aug_finetune_cnn.predict_classes(
predictions = num2class_label_transformer(predictions)
meu.display_model_performance_metrics(true_labels=test
                                      classes=list(set
```

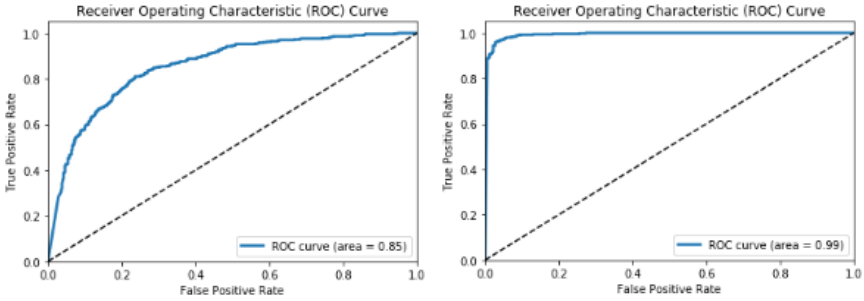
```
Model Performance metrics:
-------------------------
Accuracy: 0.961
Precision: 0.9611
Recall: 0.961
F1 Score: 0.961

Model Classification report:
---------------------------
              precision    recall  f1-score   support

         cat       0.97      0.95      0.96       500
         dog       0.95      0.97      0.96       500

   avg / total       0.96      0.96      0.96      1000

Prediction Confusion Matrix:
---------------------------
             Predicted:
                   cat  dog
Actual: cat        476   24
        dog         15  485
```

We can see that we definitely have some interesting results. Each subsequent model performs better than the previous model, which is expected since we tried more advanced techniques with each new model.

Our worst model is our basic CNN model, with a model accuracy and F1-score of around **78%**, and our best model is our fine-tuned model with transfer learning and image augmentation, which gives us a model accuracy and F1-score of **96%**, which is really amazing

considering we trained our model from our 3,000 image training dataset. Let's plot the ROC curves of our worst and best models now.

```
# worst model - basic CNN
meu.plot_model_roc_curve(basic_cnn, test_imgs_scaled,
                         true_labels=test_labels_enc,
                         class_names=[0, 1])

# best model - transfer learning with fine-tuning & im
meu.plot_model_roc_curve(tl_img_aug_finetune_cnn, test
```



ROC curve of our worst vs. best model

This should give you a good idea of how much of a difference pre-trained models and transfer learning can make, especially in tackling complex problems when we have constraints like less data. We encourage you to try out similar strategies with your own data!

Case Study 2: Multi-Class Fine-grained Image Classification with Large Number of Classes and Less Data Availability

Now in this case study, let us level up the game and make the task of image classification even more exciting. We built a simple binary classification model in the previous case study (albeit we used some complex techniques for solving the small data constraint problem!). In this case-study, we will be concentrating toward the task of fine-grained image classification. Unlike usual image classification tasks, fine-grained image classification refers to the task of recognizing different sub-classes within a higher-level class.

Main Objective

To help understand this task better, we will be focusing our discussion around the **Stanford Dogs** dataset. This dataset, as the name suggests,

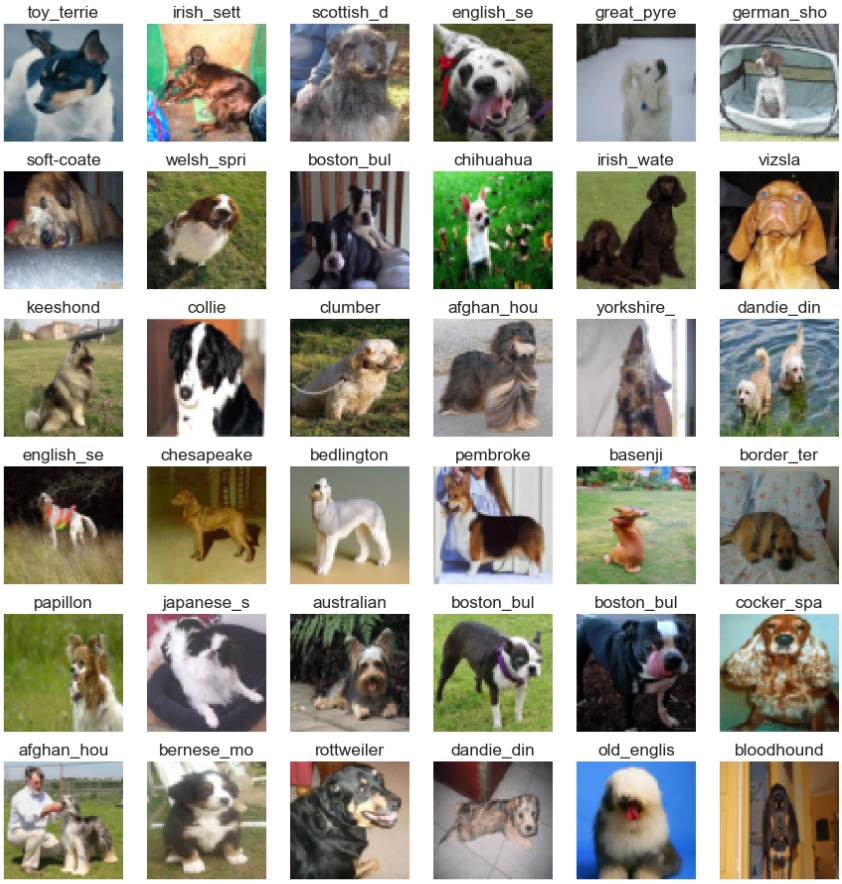
contains images of different dog breeds. In this case, the task is to identify each of those dog breeds. Hence, the high-level concept is the dog itself, while the task is to categorize different subconcepts or subclasses—in this case, breeds—correctly.

We will be leveraging the dataset available through **Kaggle** available here. We will only be using the train dataset since it has labeled data. This dataset contains around 10,000 labeled images of 120 different dog breeds. Thus our task is to build a fine-grained 120-class classification model to categorize 120 different dog breeds. Definitely challenging!

Loading and Exploring the Dataset

Let's take a look at how our dataset looks like by loading the data and viewing a sample batch of images.

```
import scipy as sp
import numpy as np
import pandas as pd
import PIL
import scipy.ndimage as spi
import matplotlib.pyplot as plt
%matplotlib inline
np.random.seed(42)

DATASET_PATH = r'../kaggle_train/'
LABEL_PATH = r'../kaggle_labels/labels.csv'

# This function prepares a random batch from the data
def load_batch(dataset_df, batch_size = 25):
    batch_df = dataset_df.loc[np.random.permutation(r

    return batch_df

# This function plots sample images in specified size
def plot_batch(images_df, grid_width, grid_height, im
    f, ax = plt.subplots(grid_width, grid_height)
    f.set_size_inches(12, 12)

    img_idx = 0
    for i in range(0, grid_width):
        for j in range(0, grid_height):
```



Sample dog breed images and labels

From the preceding grid, we can see that there is a lot of variation, in terms of resolution, lighting, zoom levels, and so on, available along with the fact that images do not just contain just a single dog but other dogs and surrounding items as well. This is going to be a challenge!

Building Datasets

Let's start by looking at how the dataset labels look like to get an idea of what we are dealing with.

```
data_labels = pd.read_csv('labels/labels.csv')
target_labels = data_labels['breed']
```

```
print(len(set(target_labels)))
data_labels.head()
```

```
------------------
120
```

	id	breed
0	000bec180eb18c7604dcecc8fe0dba07	boston_bull
1	001513dfcb2ffafc82cccf4d8bbaba97	dingo
2	001cdf01b096e06d78e9e5112d419397	pekinese
3	00214f311d5d2247d5dfe4fe24b2303d	bluetick
4	0021f9ceb3235effd7fcde7f7538ed62	golden_retriever

What we do next is to add in the exact image path for each image present in the disk using the following code. This will help us in easily locating and loading up the images during model training.

```
train_folder = 'train/'
data_labels['image_path'] = data_labels.apply(lambda r
                                              axis=1)
data_labels.head()
```

	id	breed	image_path
0	000bec180eb18c7604dcecc8fe0dba07	boston_bull	train/000bec180eb18c7604dcecc8fe0dba07.jpg
1	001513dfcb2ffafc82cccf4d8bbaba97	dingo	train/001513dfcb2ffafc82cccf4d8bbaba97.jpg
2	001cdf01b096e06d78e9e5112d419397	pekinese	train/001cdf01b096e06d78e9e5112d419397.jpg
3	00214f311d5d2247d5dfe4fe24b2303d	bluetick	train/00214f311d5d2247d5dfe4fe24b2303d.jpg
4	0021f9ceb3235effd7fcde7f7538ed62	golden_retriever	train/0021f9ceb3235effd7fcde7f7538ed62.jpg

It's now time to prepare our train, test and validation datasets. We will leverage the following code to help us build these datasets!

```
from sklearn.model_selection import train_test_split
from keras.preprocessing.image import img_to_array, l

# load dataset
train_data = np.array([img_to_array(load_img(img, tar
                            for img in data_labels['im
                        ]).astype('float32')

# create train and test datasets
x_train, x_test, y_train, y_test = train_test_split(t
                                                    t
                                                    s
                                                    r

# create train and validation datasets
x_train, x_val, y_train, y_val = train_test_split(x_t
```

```
Initial Dataset Size: (10222, 299, 299, 3)

Initial Train and Test Datasets Size: (7155, 299, 299,
3) (3067, 299, 299, 3)

Train and Validation Datasets Size: (6081, 299, 299,
3) (1074, 299, 299, 3)

Train, Test and Validation Datasets Size: (6081, 299,
299, 3) (3067, 299, 299, 3) (1074, 299, 299, 3)
```

We also need to convert the text class labels to one-hot encoded labels else our model will not run.

```
y_train_ohe = pd.get_dummies(y_train.reset_index(drop=
y_val_ohe = pd.get_dummies(y_val.reset_index(drop=True
y_test_ohe = pd.get_dummies(y_test.reset_index(drop=Tr

```

```
((6081, 120), (3067, 120), (1074, 120))
```

Everything looks to be in order. Now, if you remember from the previous case study, image augmentation is a great way to deal with having less data per class. In this case, we have a total of 10222 samples and 120 classes. This means, an average of only 85 images per class! We do this using the `ImageDataGenerator` utility from `keras.`

```
from keras.preprocessing.image import ImageDataGenera
BATCH_SIZE = 32

# Create train generator.
train_datagen = ImageDataGenerator(rescale=1./255,
                                   rotation_range=30,
                                   width_shift_range=
                                   height_shift_range
                                   horizontal_flip =
train_generator = train_datagen.flow(x_train, y_train
                                     batch_size=BATCH
```

Now that we have our data ready, the next step is to actually build our deep learning model!

Transfer Learning with Google's Inception V3 Model

Now that our datasets are ready, let's get started with the modeling process. We already know how to build a deep convolutional network from scratch. We also understand the amount of fine-tuning required to achieve good performance. For this task, we will be utilizing concepts of transfer learning. A pre-trained model is the basic ingredient required to begin with the task of transfer learning.

In this case study, we will concentrate on utilizing a pre-trained model as a feature extractor. We know, a deep learning model is basically a stacking of interconnected layers of neurons, with the final one acting as a classifier. This architecture enables deep neural networks to capture different features at different levels in the network. Thus, we can utilize this property to use them as feature extractors. This is made possible by removing the final layer or using the output from the penultimate layer. This output from the penultimate layer is then fed into an additional set of layers, followed by a classification layer. We will be using the **Inception V3 Model** from Google as our pre-trained model.

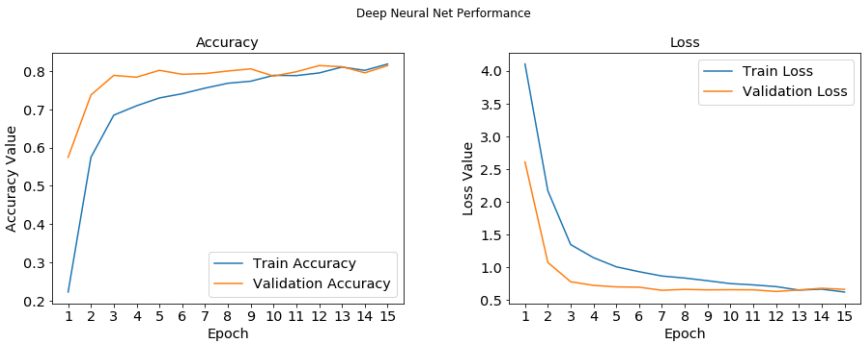
```
from keras.models import Model
from keras.optimizers import Adam
from keras.layers import GlobalAveragePooling2D
from keras.layers import Dense
from keras.applications.inception_v3 import Inceptior
from keras.utils.np_utils import to_categorical

# Get the InceptionV3 model so we can do transfer lea
base_inception = InceptionV3(weights='imagenet', incl
                             input_shape=(299, 299, 3

# Add a global spatial average pooling layer
out = base_inception.output
out = GlobalAveragePooling2D()(out)
out = Dense(512, activation='relu')(out)
out = Dense(512, activation='relu')(out)
total_classes = y_train_ohe.shape[1]
predictions = Dense(total_classes, activation='softma

model = Model(inputs=base_inception.input, outputs=pr

# only if we want to freeze layers
for layer in base_inception.layers:
    layer.trainable = False

# Compile
model.compile(Adam(lr=.0001), loss='categorical_cross
model.summary()


'''
Layer (type)                 Output Shape
=================================================
input_1 (InputLayer)         (None, 299, 299, 3)
_________________________________________________
conv2d_1 (Conv2D)            (None, 149, 149, 32)
_________________________________________________
batch_normalization_1 (BatchNor (None, 149, 149, 32)
_________________________________________________
activation_1 (Activation)    (None, 149, 149, 32)
```

Based on the previous output, you can clearly see that the Inception V3 model is huge with a lot of layers and parameters. Let's start training our model now. We train the model using the `fit_generator(...)`

method to leverage the data augmentation prepared in the previous step. We set the batch size to `32`, and train the model for 15 epochs.

```
# Train the model
batch_size = BATCH_SIZE
train_steps_per_epoch = x_train.shape[0] // batch_siz
val_steps_per_epoch = x_val.shape[0] // batch_size

history = model.fit_generator(train_generator,
                              steps_per_epoch=train_s
```

```
Epoch 1/15
190/190 - 155s 816ms/step - loss: 4.1095 - acc: 0.2216
- val_loss: 2.6067 - val_acc: 0.5748
Epoch 2/15
190/190 - 159s 836ms/step - loss: 2.1797 - acc: 0.5719
- val_loss: 1.0696 - val_acc: 0.7377
Epoch 3/15
190/190 - 155s 815ms/step - loss: 1.3583 - acc: 0.6814
- val_loss: 0.7742 - val_acc: 0.7888
...
...
Epoch 14/15
190/190 - 156s 823ms/step - loss: 0.6686 - acc: 0.8030
- val_loss: 0.6745 - val_acc: 0.7955
Epoch 15/15
190/190 - 161s 850ms/step - loss: 0.6276 - acc: 0.8194
- val_loss: 0.6579 - val_acc: 0.8144
```



Performance of our Inception V3 Model (feature extractor) on the Dog Breed Dataset

The model achieves a commendable performance of more than **80%** accuracy on both train and validation sets within just 15 epochs. The plot on the right-hand side shows how quickly the loss drops and

converges to around `0.5` . This is a clear example of how powerful, yet simple, transfer learning can be.

Evaluating our Deep Learning Model on Test Data

Training and validation performance is pretty good, but how about performance on unseen data? Since we already divided our original dataset into three separate portions. The important thing to remember here is that the test dataset has to undergo similar pre-processing as the training dataset. To account for this, we scale the test dataset as well, before feeding it into the function.

```
# scaling test features
x_test /= 255.

# getting model predictions
test_predictions = model.predict(x_test)
predictions = pd.DataFrame(test_predictions, columns=
predictions = list(predictions.idxmax(axis=1))
test_labels = list(y_test)

```

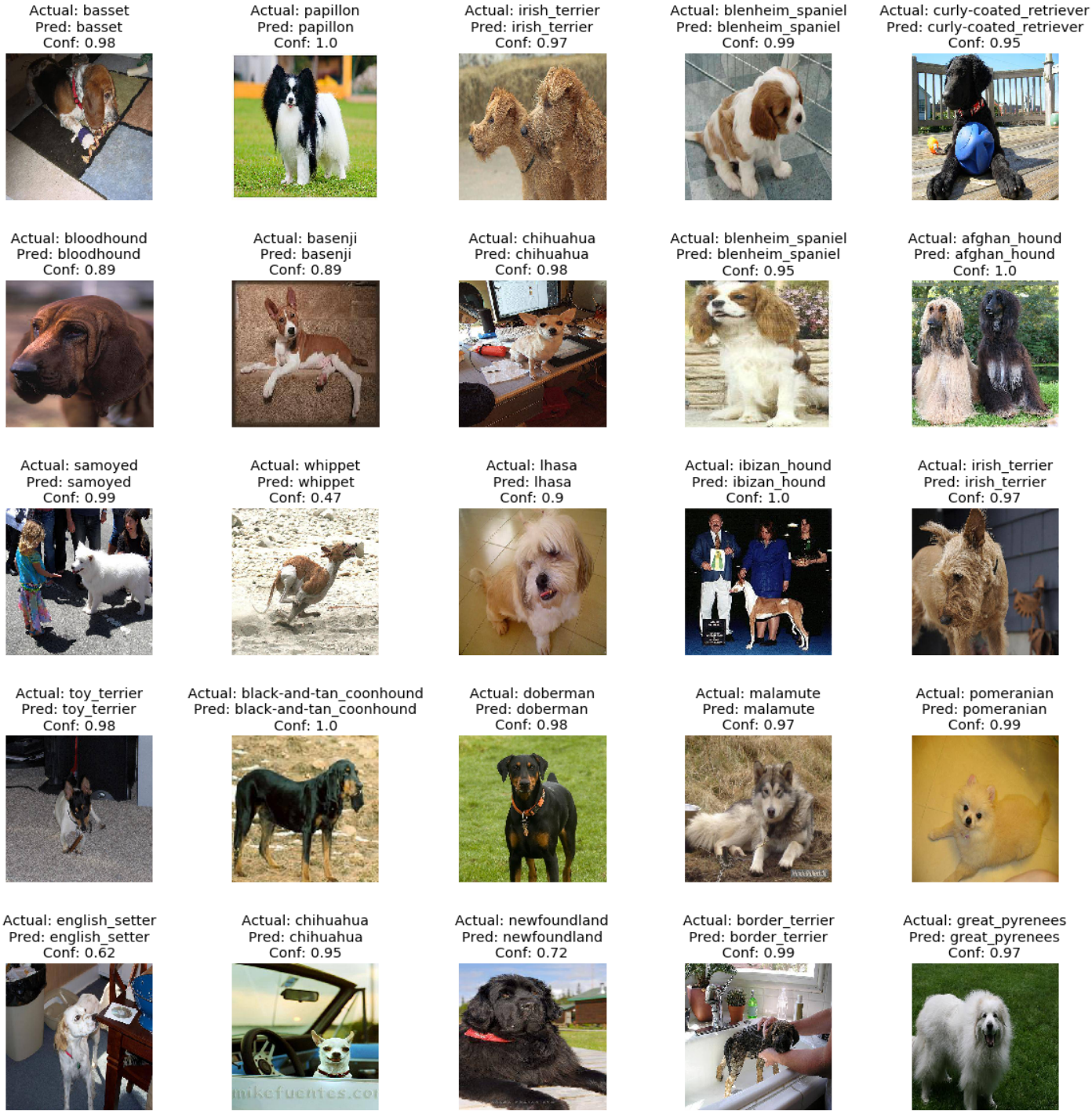
```
Accuracy: 0.864
Precision: 0.8783
Recall: 0.864
F1 Score: 0.8591
```

The model achieves an amazing **86%** accuracy as well as F1-score on the test dataset. Given that we just trained for 15 epochs with minimal inputs from our side, transfer learning helped us achieve a pretty decent classifier. We can also check the per-class classification metrics using the following code.

```
meu.display_classification_report(true_labels=test_la
                                  predicted_labels=pr
                                  classes=list(labels


'''
_________________________________________________________
                                  precision    recall

                 affenpinscher       0.92      0.92
                  afghan_hound       1.00      0.97
             african_hunting_dog       1.00      1.00
                       airedale       0.91      0.94
american_staffordshire_terrier       0.59      0.77
                    appenzeller       0.94      0.70
             australian_terrier       0.90      0.61
                        basenji       1.00      0.79
                         basset       0.83      0.96
                         beagle       0.81      0.97
             bedlington_terrier       1.00      1.00
            bernese_mountain_dog       0.85      1.00
        black-and-tan coonhound       0.83      0.87
```

We can also visualize model predictions in a visually appealing way using the following code.

```
grid_width = 5
grid_height = 5
f, ax = plt.subplots(grid_width, grid_height)
f.set_size_inches(15, 15)
batch_size = 25
dataset = x_test

labels_ohe_names = pd.get_dummies(target_labels, spar
labels_ohe = np.asarray(labels_ohe_names)
label_dict = dict(enumerate(labels_ohe_names.columns.
model_input_shape = (1,)+model.get_input_shape_at(0)[
random_batch_indx = np.random.permutation(np.arange(0

img_idx = 0
for i in range(0, grid_width):
    for j in range(0, grid_height):
        actual_label = np.array(y_test)[random_batch_
        prediction = model.predict(dataset[random_bat
```



Model predictions on test data for dog breeds!

The preceding image presents a visual proof of the model's performance. As we can see, in most of the cases the model is not only predicting the correct dog breed, it also does so with very high confidence.

Transfer Learning Advantages

We have already covered several advantages of transfer learning in some way or the other in the previous sections. Typically transfer learning enables us to build more robust models which can perform a wide variety of tasks.

- Helps solve complex real-world problems with several constraints
- Tackle problems like having little or almost no labeled data availability
- Ease of transfering knowledge from one model to another based on domains and tasks
- Provides a path towards achieving Artificial General Intelligence some day in the future!

Transfer Learning Challenges

Transfer learning has immense potential and is a commonly required enhancement for existing learning algorithms. Yet, there are certain pertinent issues related to transfer learning that need more research and exploration. Apart from the difficulty of answering the questions of what, when, and how to transfer, negative transfer and transfer bounds present major challenges.

- **Negative Transfer:** The cases we have discussed so far talk about improvements in target tasks based on knowledge transfer from the source task. There are cases when transfer learning can lead to a drop in performance. Negative transfer refers to scenarios where the transfer of knowledge from the source to the target does not lead to any improvements, but rather causes a drop in the overall performance of the target task. There can be various reasons for negative transfer, such as cases when the source task is not sufficiently related to the target task or if the transfer method could not leverage the relationship between the source and target tasks very well. Avoiding negative transfer is very important and requires careful investigation. In their work, Rosenstien and their co-authors present empirically how brute-force transfer degrades performance in target tasks when the source and target are too dissimilar. Bayesian approaches by Bakker and their co-authors, along with other techniques exploring clustering-based solutions to identify relatedness, are being researched to avoid negative transfers.
- **Transfer Bounds:** Quantifying the transfer in transfer learning is also very important affecting about the quality of the transfer and

its viability. To gauge the amount for the transfer, Hassan Mahmud and their co-authors used Kolmogorov complexity to prove certain theoretical bounds to analyze transfer learning and measure relatedness between tasks. Eaton and their co-authors presented a novel graph-based approach to measure knowledge transfer. Detailed discussions of these techniques are outside the scope of this article. Readers are encouraged to explore more on these topics using the publications outlined in this section!

Conclusion & Future Scope

This concludes perhaps one of my longest articles with a comprehensive coverage about transfer learning concepts, strategies, focus on deep transfer learning, challenges and advantages. We also covered two hands-on real-world case studies to give you a good idea of how to implement these techniques. If you are reading this section, kudos on reading through this pretty long article!

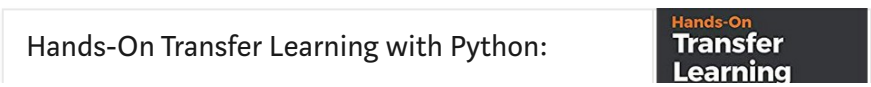
Transfer learning is definitely going to be one of the key drivers for machine learning and deep learning success in mainstream adoption in the industry. I definitely hope to see more pre-trained models and innovative case studies which leverage this concept and methodology. For some of my future articles, you can definitely expect some of the following.

- Transfer Learning for NLP
- Transfer Learning on Audio Data
- Transfer Learning for Generative Deep Learning
- More complex Computer Vision problems like Image Captioning

Let's hope for more success stories around transfer learning and deep learning which enables us to build more intelligent systems to make the world a better place and drive our own personal goals!

. . .

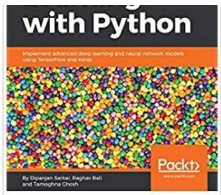
All of the content above has been adopted in some form from my recent book, ***'Hands on Transfer Learning with Python'*** which is available on the **Packt** website as well as on **Amazon**.



Implement advanced deep learning and neur...

Deep learning simplified by taking supervised, unsupervised, and reinforcement learning to the...

www.amazon.com



Don't have the time to read through the book or can't spend right now? Don't worry, you can still access all the wonderful examples and case studies we implemented on our ***GitHub repository***!

dipanjanS/hands-on-transfer-learning-with-python

Deep learning simplified by transferring prior learning using the Python deep learning...

github.com

. . .

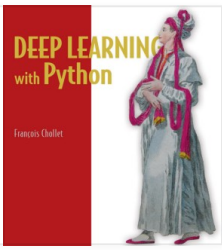
A big shoutout goes to my co-authors ***Raghav*** & ***Tamoghna*** for working with me on the book which paved the way for this content!

Thanks to Francois Chollet and his amazing book ***'Deep Learning with Python'*** for a lot of the motivation and inspiration behind some of the examples used in this article.

Deep Learning with Python

The clearest explanation of deep learning I have come across...it was a joy to read.

www.manning.com



Have feedback for me? Or interested in working with me on research, data science, artificial intelligence or even publishing an article on ***TDS***? You can reach out to me on **LinkedIn.**

Dipanjan Sarkar — Data Scientist — Intel Corporation | LinkedIn

View Dipanjan Sarkar's profile on LinkedIn, the world's largest professional community. Dipanja...