Karlsruhe Institute of Technology

Intelligent Process Automation and Robotics Lab

Pose Estimation for Industrial Robots via Transfer Learning from Sim to Real

Master Thesis

of

Zhuoyi Han

At the Department of Computer Science
Institute for Anthropomatics and Robotics (IAR) -
Intelligent Process Automation and Robotics Lab (IPR)

First reviewer:	Prof. Dr.-Ing. Torsten Kröger
Second reviewer:	Prof. Dr.-Ing. habil. Björn Hein
Advisor:	M.Sc. Yongzhou Zhang

11. November 2018 – 31. April 2019

Institute for Anthropomatics and Robotics (IAR) -
Intelligent Process Automation and Robotics Lab (IPR)
KIT Department of Informatics
Karlsruhe Institute of Technology
Engler-Bunte-Ring 8
76131 Karlsruhe

Zhuoyi Han
Bernhardstr.11
76131 Karlsruhe
uxehr@studen.kit.edu

Todo list

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, May 6, 2019

. . . . . .

(**Zhuoyi Han**)

Acknowledgement

Foremost, I would like to express my sincere thanks to my supervisor Mr. Yongzhou Zhang, for offering me the chance to write my master thesis in the Intelligent Process Automation and Robotics Lab in Institute for Anthropomatics and Robotics. His supportive guidance and advice to my master thesis motivated me and I learned a lot from him.

Further, I would like to thank Prof. Hein and Prof. Furmans for offering me the mentoring privilege in both IPR and IFL, KIT.

At last, I appreciate all the encouragement and support from my friends and family, without them I would not have so much courage and confidence to arrive where I am standing today.

Abstract

Pose Estimation for Industrial Robots via Transfer Learning from Sim to Real

English abstract.

Keywords: *Keywords, of, my, Thesis*

Zusammenfassung

Mein Titel
ist lang

Deutsche Zusammenfassung **Stickwörter:** *Die, Stichwörter, für, meine, Arbeit*

Contents

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, May 6, 2019

. . . . . .

(**Zhuoyi Han**)

Acknowledgement

Foremost, I would like to express my sincere thanks to my supervisor Mr. Yongzhou Zhang, for offering me the chance to write my master thesis in the Intelligent Process Automation and Robotics Lab in Institute for Anthropomatics and Robotics. His supportive guidance and advice to my master thesis motivated me and I learned a lot from him.

Further, I would like to thank Prof. Hein and Prof. Furmans for offering me the mentoring privilege in both IPR and IFL, KIT.

At last, I appreciate all the encouragement and support from my friends and family, without them I would not have so much courage and confidence to arrive where I am standing today.

1. Introduction

Robot learning has attracted an increasing amount of attention in recent years. With the development of technology robots are required to conduct complex tasks and tackle unforeseen circumstances. Giving robots the ability to learn is paramount for ensuring the success of future robotic system. Perceiving and processing information from environment are particularly important for robot learning.

Pose estimation of an object from pixels is a well-studied problem of perceiving task in robotics. Pose estimation can be achieved by a variety of methods, including the approach based on image. Estimation from camera pixels is attractive due to the low cost of cameras and the rich data they provide, but challenging because it involves processing high-dimensional input data. Recent work has shown that supervised learning with deep neural networks is a powerful tool for learning generalizable representations from high-dimensional inputs [24], and state-of- the-art methods employ complex, hand-engineered image processing pipelines (e.g., [9], [56]). This work is a first step toward the goal of using deep learning to improve the accuracy of object detection pipelines. But deep learning relies on a large amount of labeled data. Labeled data is difficult to obtain in the real world for precise robotic manipulation behaviors, but it is easy to generate in a physics simulator. So the concept of transfer learning was proposed. The main idea is to realize the learning process in simulation and then transfer the knowledge to the robot in real world, which is called *sim-to-real*.

Learning in simulation is especially promising for building on recent results using deep reinforcement learning to achieve human-level performance on tasks like Atari [35] and robotic control [27]. However, discrepancies between physics simulators and the real world make transferring behaviors from simulation challenging. Some recent work has received good results on the overcoming the reality gap. Robustness from sim to real can be improved by injecting noise [50], using domain randomization [57], domain adaptation [52], Although not explicitly using real-world data for training, these methods have been shown effective to increase the success rate of sim-to-real transfer

1.1. Aim of the Thesis

The final goal of the thesis is combined with the above methods to optimize the pose estimation via transfer learning, for improving the robustness of the detector from sim to real. And further optimization will be performed for higher estimation accuracy. For performing robotic learning in a physics simulator, the simulation scene is built based on Unity3D rendering engine. The deep neural network models were then built and optimized to obtain a accurate object detector. To analyze the effects of different rendering conditions on accuracy and robustness of detector, various simulation scenes were built. Finally, the estimation accuracy is further improved by using a dual-camera system.

In order to achieve the final goal, the following sub-goals are pursued:

- Set up a simulation environment for the experiment with different function modules, which can achieve various rendering effects.
- Design a deep neural network model for the task and optimize its performance.
- Compare the effects of different rendering conditions on prediction accuracy and robustness.
- Further optimization through dual-camera system.
- Evaluate the performance of detector by performing robot grasping experiment.

1.2. Contributions

In this thesis, in order to optimize the pose estimation based on sim-to-real transfer, a complete pipeline has been built, which starts from building a simulation environment to detector optimization and finally to robotic experiments. The main contributions are:

- The pipeline laid the foundation for further research based on transfer learning. The research focusing on multi-objects estimation and robot control can be extended on our pipeline.
- The simulation environment in the experiment can be used for other deep learning learning tasks. The rendering environment consists of different functional modules, which can be used for other projects by modifying the functions.
- Verification of the feasibility of transfer learning for our pose estimation task.

1.3. Outline

According to the stated main goals, the remainder of this thesis is structured as outlined in the following:

Chapter 2 gives an overview of the state-of-the-art. The related works in object detection and pose estimation in robotics are presented. The it describe the methods for overcoming the reality gap, e.g. domain adaptation and domain randomization.

Chapter3 introduces the technical background of thesis. Section 3.1 gives the basic knowledge of interaction interface and scene rendering in Unity3D. Section 3.2 describes the principle and structure of the deep neural network. Several well-known convolutional neural network models are introduced in section 3.3.

The pipeline architecture design and functionality explanation of different modules is presented in Chapter 4. Section 4.1 describes the structure of overall pipeline. Section 4.2 describes the different function modules of simulation environment. The structure design and training process are introduces in section 4.3. Finally, the evaluation module and the data collection process in real world are presented.

Chapter5 introduces the experimental process and results analysis. Section 5.1 describes the scene settings in both real world and simulation environment. Model optimization

and rendering scene optimization are mentioned in section 5.2 and 5.3. In Section 5.4 the dual-camera system was used to optimize the estimation accuracy. Finally, the robotic experiments are introduced in section 5.5.

In Chapter 6, conclusions are drawn and potential future works are presented.

2. State of the art

This chapter describes the current state of the art of our work. We did a comprehensive literature research on the topic of object detection and pose estimation for robotics and different approaches to bridge the reality gab, which means the barrier to using simulated data on real robots. We describe the results of this literature research in the following sections. In Section 2.1, we reviews different methods which used on object detection and pose estimation for robotics. In Section 2.2, we focus on the approach domain randomization to bridge the reality gap. In Section 2.3, we give an overview of the domain adaptation which are used for training models to a previously unseen target domain. Section 2.4 gives a summary to the these approaches.

2.1. Object detection and pose estimation for robotics

Object detection and pose estimation for robotics is a well-studied problem in the literature. Recent approaches typically involve offline construction or learning of a 3D model of objects in the scene (e.g., a full 3D mesh model [56] or a 3D metric feature representation [9]). At test time, features from the test data (e.g., Scale-Invariant Feature Transform [SIFT] features [15] or color co-occurrence histograms [11] are matched with the 3D models (or features from the 3D models). For example, a black-box nonlinear optimization algorithm can be used to minimize the re-projection error of the SIFT points from the object model and the 2D points in the test image [8]. Most successful approaches rely on using multiple camera frames [7] or depth information [56]. There has also been some success with only monocular camera images [8].

These traditional approaches require less extensive training and take advantage of richer sensory data, allowing them to detect the full 3D pose of objects (position and orientation) without any assumptions about the location or size of the surface on which the objects are placed. However, in this thesis we aims to avoids the challenging problem of 3D reconstruction, and employs a simple, easy to implement deep learning-based pipeline that may scale better to more challenging problems.

2.2. Domain Randomization

Domain randomization is a complementary class of techniques for adaptation that is particularly well suited for simulation. With domain randomization, discrepancies between the source and target domains are modeled as variability in the source domain. Randomization in the visual domain has been used to directly transfer vision-based policies from simulation to the real world without requiring real images during training [44], [57].

Sadeghi and Levine [44] trained vision-based controllers for a quadrotor using only synthetically rendered scenes, and Tobin et al. [57] demonstrated transferring image-based object detectors. Unlike previous methods, which sought to bridge the reality gap with high fidelity rendering [21], their systems used only low fidelity rendering and modeled differences in visual appearance by randomizing scene properties such as lighting, textures, and camera placement. The approach in [57] does not rely on precise camera information or calibration, instead randomizing the position, orientation, and field of view of the camera in the simulator. Whereas approach in [44] chooses textures from a dataset of around 200 pre-generated materials, most of which are realistic, the approach in [57] use only non-realistic textures created by a simple random generation process. Figure 2.1 shows the rendering scene by Tobin et al. [57] in their work.

Figure 2.1.: Rendered low-fidelity images with random camera positions, lighting conditions, and non-realistic textures in [57]

2.3. Domain adaptation

Another way to cross the reality gap is to learn from both simulation and real-world data. The computer vision community has devoted significant study to the problem of adapting vision-based models trained in a source domain to a previously unseen target domain (see, e.g., [20], [19], [32]). A variety of approaches have been proposed, including re-training the model in the target domain (e.g., [61]), adapting the weights of the model based on the

statistics of the source and target domains (e.g., [30]), learning invariant features between domains (e.g., [58]), and learning a mapping from the target domain to the source domain (e.g., [55]). Researchers in the reinforcement learning community have also studied the problem of domain adaptation by learning invariant feature representations [16], adapting pretrained networks [42], and other methods. See [16] for a more complete treatment of domain adaptation in the reinforcement learning literature.

It is often faster to fine-tune a controller learned in simulation than to learn from scratch in the real world [10, 22]. In [13], the authors use a variational autoencoder trained on simulated data to encode trajectories of motor outputs corresponding to a desired behavior type (e.g., reaching, grasping) as a low-dimensional latent code. A policy is learned on real data mapping features to distributions over latent codes. The learned policy overcomes the reality gap by choosing latent codes that correspond to the desired physical behavior via exploration.

Domain adaptation has also been applied to robotic vision. Rusu et al. [43] explore using the progressive network architecture to adapt a model that is pretrained on simulated pixels, and find it has better sample efficiency than fine-tuning or training in the real-world alone. In [59], the authors explore learning a correspondence between domains that allows the real images to be mapped into a space understood by the model. While both of the preceding approaches require reward functions or labeled data, which can be difficult to obtain in the real world, Mitash and collaborators [34] explore pretraining an object detector using realistic rendered images with randomized lighting from 3D models to bootstrap an automated learning process that does not require manually labeling data and uses only around 500 real-world samples.

2.4. Overcoming the reality gap

Previous work on leveraging simulated data for physical robotic experiments explored several strategies for bridging the reality gap. We summarize it as follows:

One type of approach is to make the simulator closely match the physical reality by performing system identification and using high-quality rendering. Though using realistic RGB rendering alone has had limited success for transferring to real robotic tasks [21], incorporating realistic simulation of depth information can allow models trained on rendered images to transfer reasonably well to the real world [38]. Combining data from high-quality simulators with other approaches like fine-tuning can also reduce the number of labeled samples required in the real world [40]. Unlike these approaches, the other type allows the use of low-quality renderers optimized for speed and not carefully matched to real-world textures, lighting, and scene configurations. And it requires no additional training on real-world data [57].

For a better result by overcoming the reality gap, the two types of approaches can be combined.

3. Technogical Fundations

This chapter introduces technological foundations fundamental for the thesis, which includes methods, models and tools that are used later in the thesis. Section 3.1 focus on the basic principle of Unity3D, which is used to build simulation environment for generating synthetic images. Section 3.2 shortly explain the essential knowledge of deep learning, especially convolution neural network(CNN), which plays the basic role for our object detector. In Section 3.3 several deep neural network models with excellent performance in the field of computer vision are introduced.

3.1. Unity3D background

Unity3D[1] is a cross-platform game engine developed by Unity Technologies. Due to the motivation of transfer learning from sim to real, a simulation scene need to be built for generating large numbers of labeled image data. With Unity3D engine we are able to create virtual objects in 3D and render images based on scripting API in C# . These features shows that Unity3D is suitable for our rendering requests and can simplify the works during modeling.

3.1.1. User interface

Unity provides a basic graphical interface to build the simulation environment, which includes five basic windows.

- **The Scene View:** where user work with game objects, including models, lights and colliders, to construct the user's scenes.
- **The Game View:** where user can preview and play simulation scene as a work in progress as user develop it.
- **The hierarchy window:** All of the game objects in the open scenes are listed by the hierarchy window in hierarchical order.
- **The project window:** where user imports, stores and edits the Asset files.
- **The inspector window:** It is context sensitive and displays all the properties of any selected game object, asset or setting.

The most common and useful windows of editor interface are shown in figure 3.1:

All the modeling works can be done in the graphical user interface. A general method of modeling are divided into 4 steps. Firstly a new default model needs to be created

[1] https://unity3d.com/de/unity

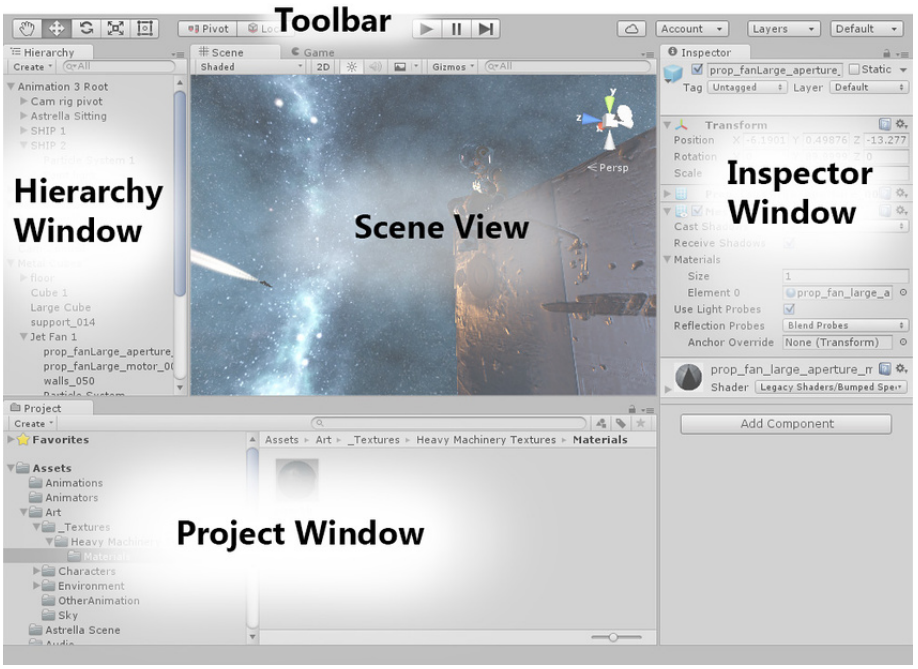

Figure 3.1.: The most common and useful windows in Unity3D. The main editor window is made up of tabbed windows which can be rearranged, grouped, detached and docked.

in Unity or imported from other 3D modeling software. This new model will appear in hierarchy window with other existent objects. Here the hierarchical relations can be adjusted. Secondly the dimensions and positions of new created model can be modified in inspector window, and new components (e.g. C# scripts, Mesh Renderer, etc.) can be added into the model. Then in scene window a arbitrary perspective could be set, and according to the local coordinate system the position relationship among different objects can be adjusted. Finally when all items are correctly set and all needed components are added on them, the simulation scene can be played and the progress can be monitored in game view window in real time.

3.1.2. Basic Concepts

Coordinate System

The world coordinate system is left handed (as Direct X) where x positive axis is right, y positive is up and z is positive into the screen, which is shown in figure 3.2.

- **Screen coordinate system** is bottom-up: (0,0) at bottom-left corner and (pixelWidth-1,pixelHeight-1) at right-top; x axis is positive right and y is positive up. The z position is in world units from the camera.
- **Viewport coordinate system** is normalized and relative to the camera, so the bottom-left point is (0,0), the top-right is (1,1). The z position is in world units from the camera.
- **UI coordinate system** is top-down: the y coordinate varies from zero at the top edge of the window to a maximum at the bottom edge of the window. The upper-left point is (0,0); the bottom-right is (1,1).

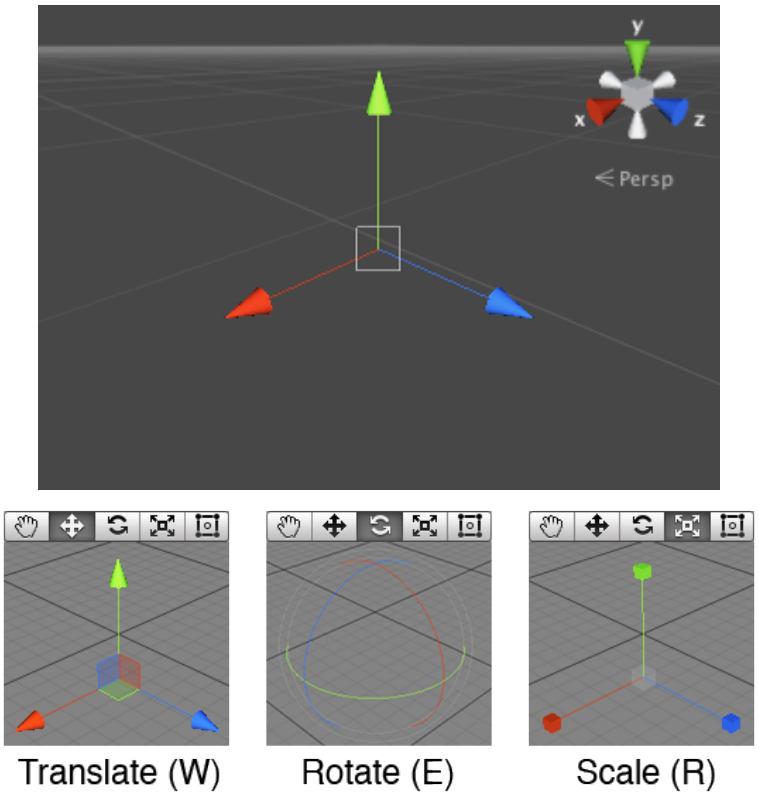

Figure 3.2.: Coordinate system in Unity3D. Unlike the right-handed coordinate system in robotics, the left-hand coordinate system is used in Unity3D.

There are many functions to convert between all these different coordinate systems

Scene Construction

To build a basic scene in Unity 3D, some important core concepts need to be discussed, including *Game Object*, *Component*, *Script* and etc. They form the main basic elements of scene and various functions and effects can be realized with them. Normally, the *Component* and *Script* are sub-elements of the *Game Object*.

Scenes:

Scenes contain the objects of simulation project. They can be used to create a main menu, individual levels, and anything else. Each unique Scene file is a unique level. In each Scene, the environments, obstacles, decorations will be placed, and the project are designed and built in pieces.

A new Unity project will show a new Scene. The scene will be empty except for default objects- either an orthographic camera, or a perspective camera and a directional light. An new empty scene is shown in figure 3.3.

GameObjects:

The GameObject is the most important concept in the Unity Editor. Every object in project is a GameObject. This means that everything which can be thought of to be in the project has to be a GameObject. However, a GameObject can't do anything on its own; The

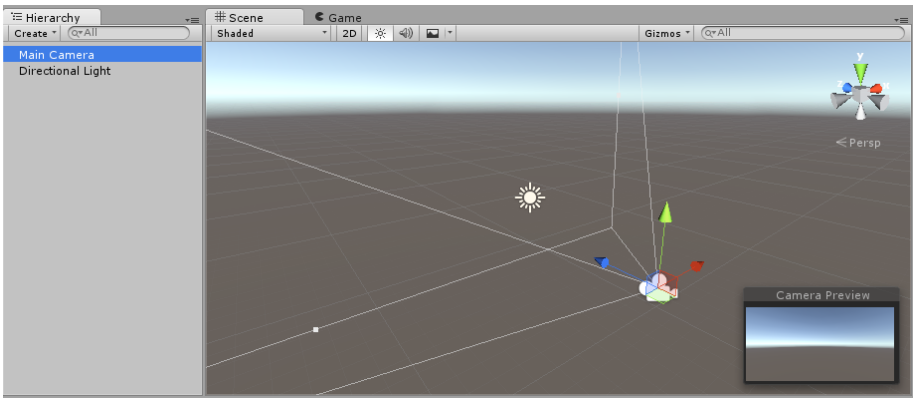

Figure 3.3.: A new empty Scene, with the default 3D objects: a Main Camera and a directional Light.

properties need to be given before it can become a character, an environment, or a special effect.

A GameObject is a container; pieces are added to the GameObject container to make it into a character, a light, or whatever needed in project. Each added piece is called a component. Depending on what kind of object need to be created, different combinations of components can be added to a GameObject. A GameObject could be thought of as an empty cooking pot, and components are different ingredients that make up the recipe of project.

Component:

Components are the nuts & bolts of objects and behaviors in a game. They are the functional pieces of every GameObject.

A GameObject is a container for many different Components. By default, all GameObjects automatically have a Transform Component. This is because the Transform dictates where the GameObject is located, and how it is rotated and scaled. Without a Transform Component, the GameObject wouldn't have a location in the word.

Script:

Scripting is an essential ingredient in all projects. They can be used to create graphical effects, control the physical behavior of objects and even implement a custom automatic system in the project. In next subsection, basic knowledge of scripting will be introduced to show how the script runs in unity and which basic functions are included in it.

3.1.3. Script Overview

Unity3D supports the C# programming language, which is an industry-standard language similar to Java or C++.

A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called MonoBehaviour. A class is like a kind of blueprint for creating a new Component type that can be attached to GameObjects. The core elements to realize different effects are the Event Functions. A script in Unity is not like the traditional idea of a program where the code runs continuously in a loop until it

completes its task. Instead, Unity passes control to a script intermittently by calling certain functions that are declared within it. Once a function has finished executing, control is passed back to Unity. These functions are known as event functions since they are activated by Unity in response to events that occur during gameplay. Unity uses a naming scheme to identify which function to call for a particular event. There are four basic types of events as shown in figure 3.4:

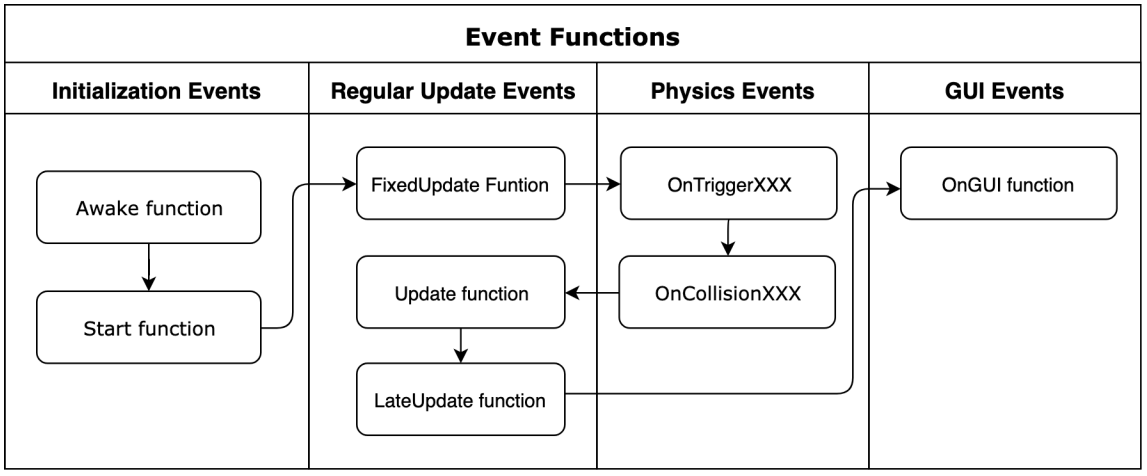

Figure 3.4.: Event function and execution order in Unity. For scene rendering, each event function is executed in order. Regular update and physics events execute in a loop, refresh the tasks in real-time

Regular Update Events:

- Update function: The Update function is the main place for making changes to position, state and behavior of objects in the scene just before each frame is rendered. Update is called before the frame is rendered and also before animations are calculated.
- FixedUpdate function: The physics engine also update in discrete time steps in a similar way to the frame rendering. A separate event function called FixedUpdate is called just before each physics update.
- LateUpdate function: It is also useful sometimes to be able to make additional changes at a point after the Update and FixedUpdate functions have been called for all objects in the scene and fter all animations have been calculated.

Listing 3.1 Regular Update Events Functions

```
1 void Update() {
2     float distance = speed * Time.deltaTime * Input.GetAxis("
          Horizontal");
3     transform.Translate(Vector3.right * distance);
4 }
5
6 void FixedUpdate() {
7     Vector3 force = transform.forward * driveForce * Input.GetAxis("
          Vertical");
8     rigidbody.AddForce(force);
9 }
```

```
10
11 void LateUpdate() {
12     Camera.main.transform.LookAt(target.transform);
13 }
```

Initialization Events:

- Start function: It is called before the first frame or physics update on an object.
- Awake function: The Awake function is called for each object in the scene at the time when the scene loads. All the Awakes will have finished before the first Start is called. This means that code in a Start function can make use of other initializations previously carried out in the Awake phase.

GUI event:

Unity has a system for rendering GUI controls over the main action in the scene and responding to clicks on these controls. This code is handled somewhat differently from the normal frame update and so it should be placed in the OnGUI function, which will be called periodically.

Listing 3.2 GUI Events Functions

```
1 void OnGUI() {
2     GUI.Label(labelRect, "Game Over");
3 }
```

Physics events:

The physics engine will report collisions against an object by calling event functions on the object's script.

Listing 3.3 Physics Events Functions

```
1 void OnCollisionEnter(otherObj: Collision) {
2     if (otherObj.tag == "Arrow") {
3     ApplyDamage(10);
4     }
5 }
```

3.2. Deep Learning and Convolutional Neural Network

In this section a introduction to neural network and deep learning are presented with the aim of enabling the reader to build up a precise understanding of the concepts and ideas as well as of the thesis' context.

3.2.1. Machine Learning

Learning algorithms are widely used in computer vision applications. Before considering image related tasks, we are going to have a brief look at basics of machine learning.

Machine learning has emerged as a useful tool for modeling problems that are otherwise difficult to formulate exactly. Classical computer programs are explicitly programmed by hand to perform a task. With machine learning, some portion of the human contribution is replaced by a learning algorithm [14]. As availability of computational capacity and data has increased, machine learning has become more and more practical over the years, to the point of being almost ubiquitous.

A practical view of a machine learning system is depicted in Figure 3.5. The process is split into two phases. In the first phase, the machine learning algorithm is used to learn from the training data, and the second phase is the prediction. The training data could be labeled images of vehicles with the task to predict the pose. The machine learning algorithm learns a model on these data and this model can then used to predict unseen images of the same task. This prediction is happen in the second phase and apply only the learned model. In our example, the learned model get images of a vehicle and must predict the pose.

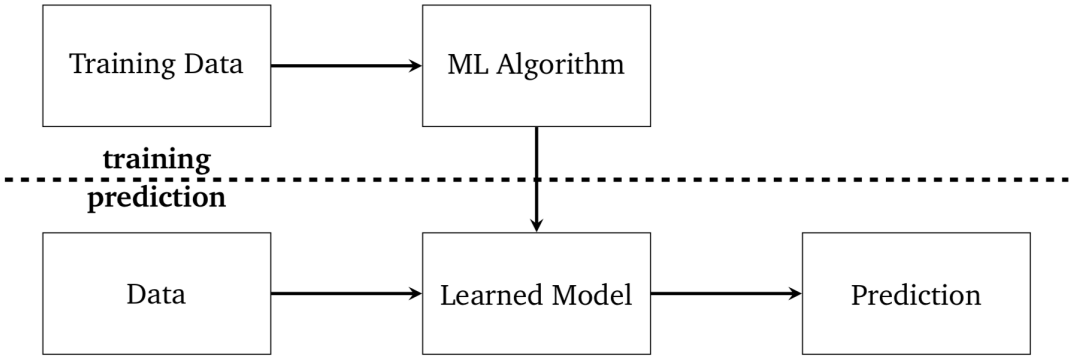

Figure 3.5.: Workflow of a machine learning problem. The data will be used to train a model with a machine learning algorithm. Then, in the prediction phase, the learned model can be used to generate a prediction.

Types of Learning Problem

In machine learning it can be distinguished between different learning problems. The main learning problems are:

- **Supervised Learning:** A typical way of using machine learning is supervised learning [4]. A learning algorithm is shown multiple examples that have been annotated or labeled by humans. For example, in the object detection problem we use training images where humans have marked the locations and classes of relevant objects. After learning from the examples, the algorithm is able to predict the annotations or labels of previously unseen data.

- **Unsupervised Learning:** In unsupervised learning, the algorithm attempts to learn useful properties of the data without a human teacher telling what the correct output should be. Classical example of unsupervised learning is clustering [4]. More recently, especially with the advent of deep learning technologies, unsupervised preprocessing has become a popular tool in supervised learning tasks for discovering useful representations of the data [2].
- **Reinforcement Learning:** In reinforcement learning the machine learning algorithm interact with an environment and must reach a certain goal. This could be to learn to playing a game, or to driving a vehicle in a simulation. The algorithm gets only information how good or bad he has interact with the environment. For example in learning to play a game, could this information the winning or losing of a game.

Types of Learning Tasks

Classification and regression are the most important task types [4]. In classification, the algorithm attempts to predict the correct class of a new piece of data based on the training data. In regression, instead of discrete classes, the algorithm tries to predict a continuous output.

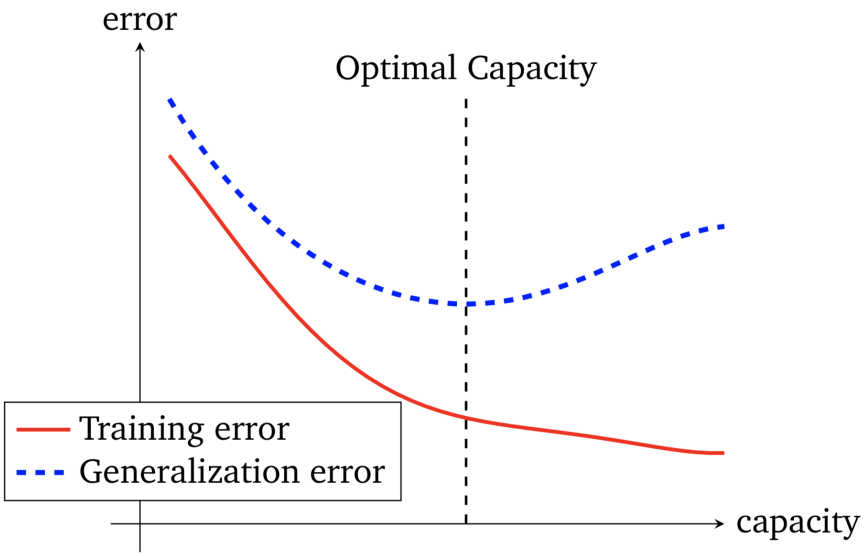

Figure 3.6.: Shows typical curves for training and generalization error in dependency of the capacity of the model. Optimal capacity is reached at the minimal generalization error. Left of the optimal capacity is the model underfitting. On the right side of the optimal capacity is the model overfitting. The generalization error has typical a U-shaped curve.

Under- and Overfitting

A machine learning algorithm must perform well on unseen data. The ability to perform well on unseen data is called generalization. The generalization error is measured on a

test set. If a model perform not well on unseen data, then there are two reasons. Either the model has not enough capacity and underfit the underlying function, or it has too much capacity and overfit the underlying function. If a model is underfitting, then will be the training error and also the test error high. If the model is overfitting, then is the training error low and the test error high. The goal is to find a model, which has a low generalization error. The typical curves for training and generalization error are depicted in Figure 3.6.

In Figure 3.7 are three diagrams depicted, which shows the same noisy sampling of a sinus function. The samples are used learn a model, which describes the underlying function. The left diagram learned a model with a low capacity. So the training error is high and a test set would be also produce a high test error. In the center is depicted a model with a higher capacity. This shows a good approximation of the underlying function. Nevertheless the model has a training error, because we sampled the sinus with noise. But this model will have on a test set the best generalization error compared to the other two models. The third diagram shows a model with a high capacity. The training error will be low, but the learned model is not a good approximation of the sinus function, which would result on a test set with a high error.

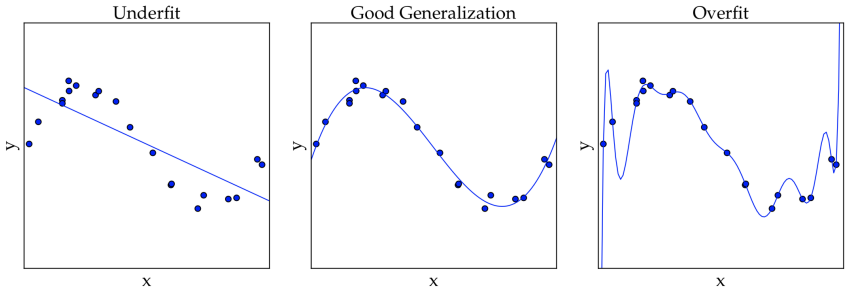

Figure 3.7.: Theses three diagrams show three different models, which tries to fit the sampled points. The sampled points are sampled from a noisy sinus function. The models are described by a polynom of degree 1,4,15. The left model underfits the underlying function. The model in the center is a good approximation of the underlying function. The right model overfits the underlying function. It has the smallest error to fit the sampled points, but on unseen samples, it will be provide a bad error [14].

3.2.2. Neural Network Basis

Neural networks are a popular type of machine learning model. A special case of a neural network called the convolutional neural network (CNN) is the primary focus of this thesis. Before discussing CNNs, we will discuss how regular neural networks work.

Biological Motivation and Neural Network Architecture

The basic computational unit of the brain is a neuron. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately

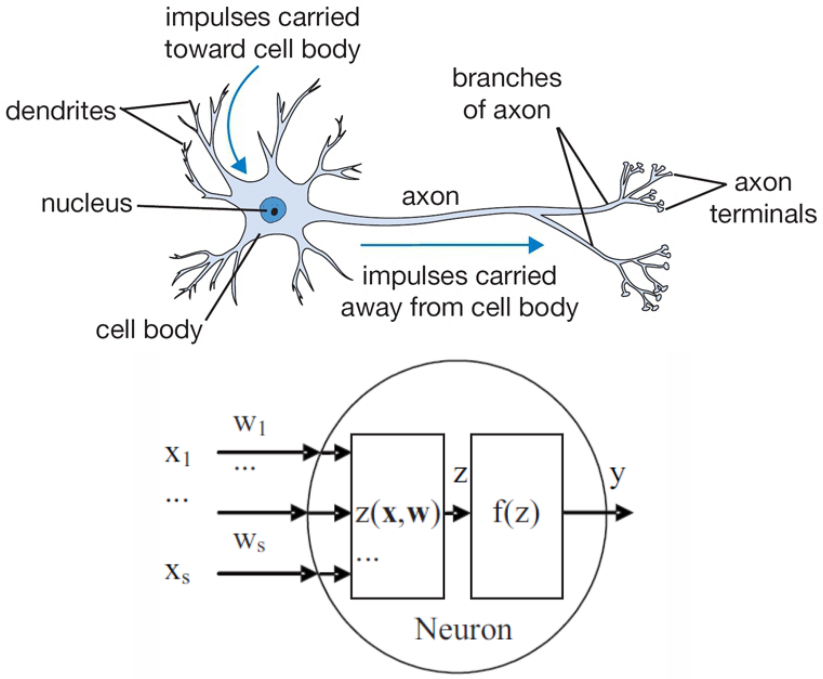

Figure 3.8.: a single Neuron example. At the top is the biological structure, and at the bottom is the simplified mathematical model.

10^{14}-10^{15} synapses [31]. The diagram below shows a cartoon drawing of a biological neuron and a common mathematical model.

An artificial neuron based on the McCulloch-Pitts model is shown in Figure 3.9 [4]. The neuron k receives m input parameters x_i. The neuron also has m weight parameters W_i. The weight parameters often include a bias term that has a matching dummy input with a fixed value of 1. The inputs and weights are linearly combined and summed. The sum is then fed to an activation function f that produces the output y of the neuron:

$$y_k = f(z_k) = f\left(\sum_{j=0}^{s} w_{kj} x_j\right) \tag{3.1}$$

The above neuron composed of multiple elements, which have different features and play various role in the neuron networks. Besides the inputs and outputs, the weights and activation function are the most important parts, which decide the basic function of a neuron. The network consists of connections, each connection transferring the output of a neuron i to the input of a neuron j. In this sense i is the predecessor of j and j is the successor of i, each connection is assigned a weight W_{ij}

Activation Functions

The activation function of a node defines the output of that node given an input or set of inputs. To approximate nonlinear function each neuron has to perform nonlinear transformation of its input, which is done with activation function $f(z)$. There are several different commonly used activation functions. Its usage depends on the type of network and also on the type of layer in which they operate. An example of different activation functions are shown in figure 3.9.

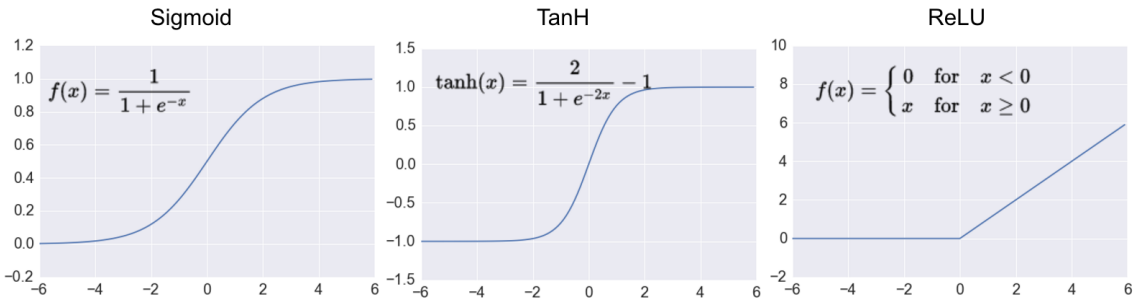

Figure 3.9.: Most popular activation functions: Sigmoid, Tanh, ReLU

- **Sigmoid:** One of the oldest and historically most commonly used activation function is sigmoid function. It is defined by

$$g(x) = \frac{1}{1+e^{-x}} \tag{3.2}$$

  It takes a real value and squashes it between 0 and 1. However, when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Thus, the backpropagation algorithm fail at modifying its parameters and the parameters of the preceding neural layers.

- **Hyperbolic Tangent:** The TanH non-linearity has the following mathematical form:

$$y = \tanh(-x) \tag{3.3}$$

  It squashes a real-valued number between -1 and 1. However it has the same drawback than the sigmoid.

- **Rectified Linear Unit:** The ReLU has the following mathematical form:

$$y = max0, x \tag{3.4}$$

  The ReLU has become very popular in the last few years, because it was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions due to its linear non-saturating form [23]. In fact, it does not suffer from the vanishing or exploding gradient. An other advantage is that it involves cheap operations compared to the expensive exponentials. However, the ReLU removes all the negative information and thus appears no suited for all datasets and architectures.

- **Leaky RelU:** To enlarge the range of the ReLU in negative direction, Leaky ReLU uses a small slop(usually 0.01), hence the Leaky ReLU can tend to infinity in both directions:

$$y = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases} \tag{3.5}$$

  where a is a small constant.

Structure of a shallow Neural Network

There are many classes of neural networks and these classes also have sub-classes, e.g. feedforward, lateral and feedback connections. Here the most used ones will be listed - Feedforward neural network, which is shown in figure 3.10.

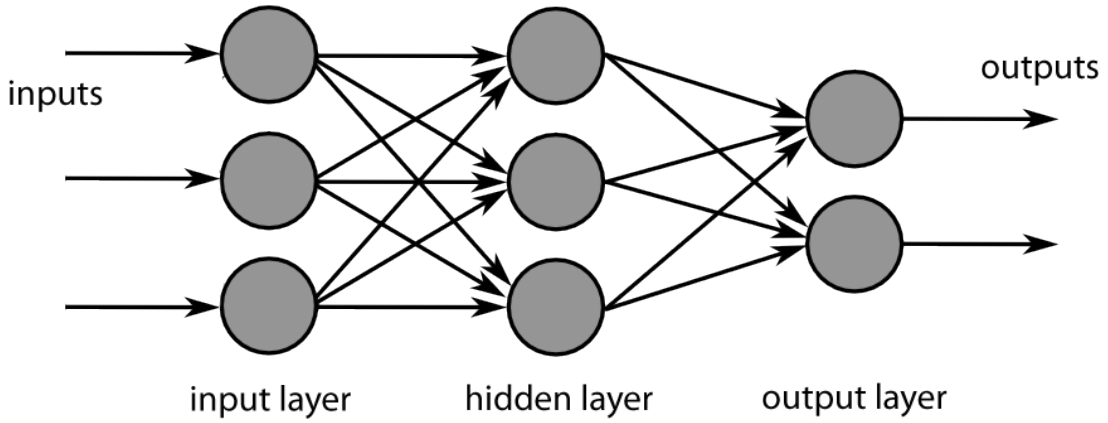

Figure 3.10.: A feedforward Multi-layer perceptron(MLP)

A feedforward neural network can consist of three types of layers:

- **Input Layer:** No computation is done here within this layer, they just pass the information to the next layer (hidden layer most of the time).
- **Hidden Layer:** In Hidden layers is where intermediate processing or computation is done, they perform computations and then transfer the weights (signals or information) from the input layer to the following layer (another hidden layer or to the output layer). It is possible to have a neural network without a hidden layer, which is called Single layer Perceptron.
- **Output Layer:** Here an activation function can be used to maps to the desired output format (e.g. softmax for classification) .

The feedforward neural network was the simplest type of artificial neural network devised, where connections between the units do not form a cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes.

Loss Functions and Optimization Problem

Loss function is an important part in artificial neural networks, which is used to measure the inconsistency between predicted value $\hat{y}$and actual label y. It is a non-negative value, where the robustness of model increases along with the decrease of the value of loss function. The aim of supervised learning is to minimize the overall cost, thus optimizing the correlation of the model to the system that it is attempting to represent. An optimization problem seeks to minimize a loss function.

For the regression problem, several loss functions are commonly used:

- **Mean Squared Error:** Mean Squared Error (MSE), or quadratic, loss function is widely used in linear regression as the performance measure, and the method of minimizing MSE is called Ordinary Least Squares (OSL), the basic principle of OSL is that the optimized fitting line should be a line which minimizes the sum of

distance of each point to the regression line, i.e., minimizes the quadratic sum. The standard form of MSE loss function is defined as:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \tag{3.6}$$

where $(y^{(i)} - \hat{y}^{(i)})$ is named as residual, and the target of MSE loss function is to minimize the residual sum of squares.

- **Mean Absolute Error:** Mean Absolute Error (MAE) is a quantity used to measure how close forecasts or predictions are to the eventual outcomes, which is computed by:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right| \tag{3.7}$$

where $|\cdot|$denotes the absolute value. Albeit, both MSE and MAE are used in predictive modeling, there are several differences between them. MSE has nice mathematical properties which makes it easier to compute the gradient. However, MAE requires more complicated tools such as linear programming to compute the gradient. Because of the square, large errors have relatively greater influence on MSE than do the smaller error. Therefore, MAE is more robust to outliers since it does not make use of square. On the other hand, MSE is more useful if concerning about large errors whose consequences are much bigger than equivalent smaller ones. MSE also corresponds to maximizing the likelihood of Gaussian random variables.

Gradient Descent

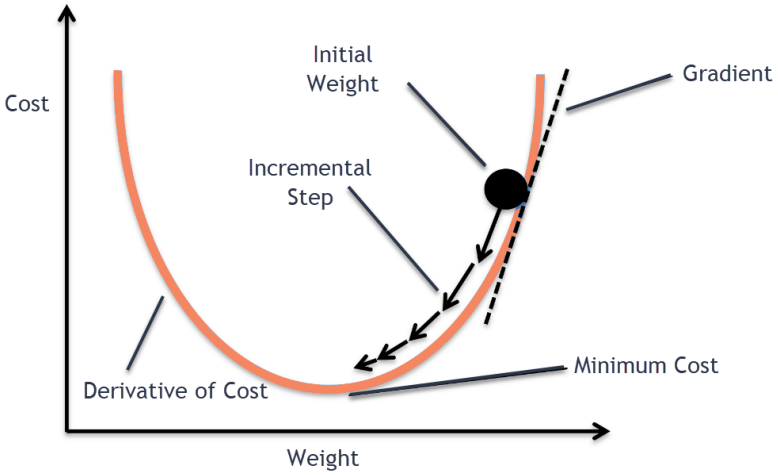

Figure 3.11.: Gradient descent to find the minimum, with incremental steps the weights are approaching local optima

The gradient descent method, also known as the method of steepest descent, is an iterative method for unconstrained optimization that takes an initial point x_0 and attempts to sequence converging to the minimum of a function $f(x)$ by moving in the direction of the negative gradient $(-\nabla f(x))$. In order to find a true minimum, the method requires a sufficiently smooth, convex function $f(x)$ [46]. The step size is usually decided by a line

search method, which seeks a sufficient decrease at each iteration for better convergence [36]. The standard form of a gradient descent is:

$$w_{k+1} = w_k - \alpha \frac{\partial \mathcal{L}}{\partial w_k} \tag{3.8}$$

where w are the learning parameters and α is the learning rate

If the steps it takes are too big, it maybe will not reach the local minimum because it just bounces back and forth between the convex function of gradient descent. If the learning rate is set to a very small value, gradient descent will eventually reach the local minimum but it will maybe take too much time, which is show in figure 3.12.

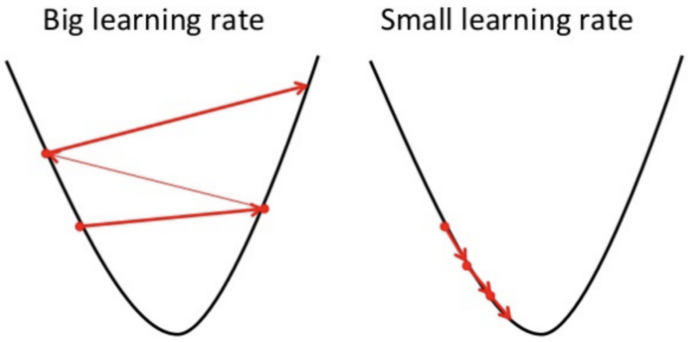

Figure 3.12.: Influence of big and small learning rate. Too big learning rate result in bounces around the local optima, however too small value cause low efficiency

There are three popular types of Gradient Descent, that mainly differ in the amount of data they use.

- **Batch Gradient Descent:**

  Batch Gradient Descent, also called vanilla gradient descent, calculates the error for each example within the training dataset, but only after all training examples have been evaluated, the model gets updated. This whole process is like a cycle and called a training epoch.

  Advantages of it are that it's computational efficient, it produces a stable error gradient and a stable convergence. Batch Gradient Descent has the disadvantage that the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. It also requires that the entire training dataset is in memory and available to the algorithm [14].

- **Stochastic Gradient Descent:**

  Stochastic gradient descent (SGD) in contrary, does this for each training example within the dataset. This means that it updates the parameters for each training example, one by one. This can make SGD faster than Batch Gradient Descent, depending on the problem. One advantage is that the frequent updates allow us to have a pretty detailed rate of improvement. SGD can lead to "zig-zagging" behavior, where the single data point estimate of the gradient keeps changing direction and does not approach the minimum directly [14].

The thing is that the frequent updates are more computationally expensive as the approach of Batch Gradient Descent. The frequency of those updates can also result in noisy gradients, which may cause the error rate to jump around, instead of slowly decreasing.

- **Mini Batch Gradient Descent:**

  Mini-batch Gradient Descent is the go-to method since it's a combination of the concepts of SGD and Batch Gradient Descent. It simply splits the training dataset into small batches and performs an update for each of these batches. Therefore it creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent [60].

  Common mini-batch sizes range between 50 and 256, but like for any other machine learning techniques, there is no clear rule, because they can vary for different applications. Note that it is the go-to algorithm when you are training a neural network and it is the most common type of gradient descent within deep learning.

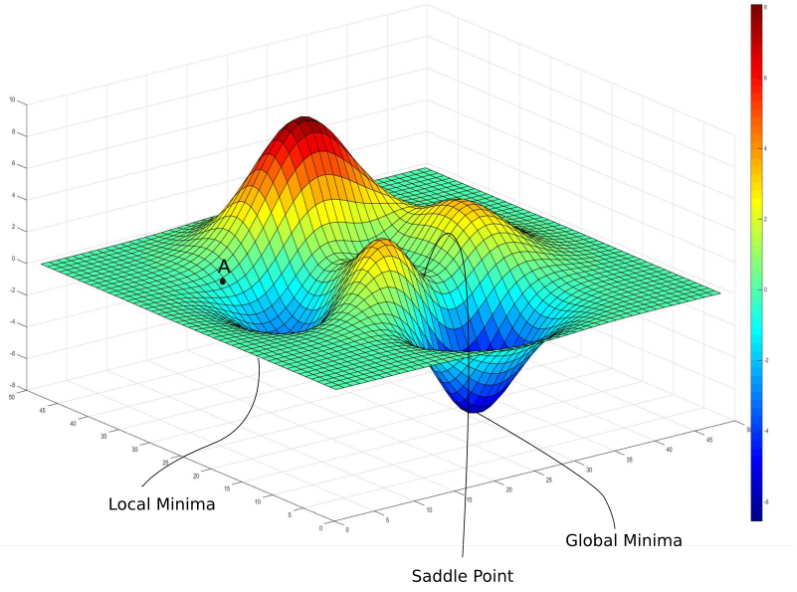

Figure 3.13.: Local minima and global minima

Back Propagation

A neural network is trained by selecting the weights of all neurons so that the network learns to approximate target outputs from known inputs. It is difficult to solve the neuron weights of a multi-layer network analytically. The *back-propagation algorithm* [4, 14],provides a simple and effective solution to solving the weights iteratively.

The backpropagation is an algorithm to propagate back the error from the loss function through the network to compute the gradient $\frac{\partial \mathcal{L}}{\partial w_{ij}}$

In the first phase of the algorithm, an input vector is propagated forward through the neural network. Before this, the weights of the network neurons have been initialized to some values, for example small random values. The received output of the network is compared to the desired output (which should be known for the training examples) using

a loss function. The gradient of the loss function is then computed. This gradient is also called the error value. When using mean squared error as the loss function, the output layer error value is simply the difference between the current and desired output.

The error values are then propagated back through the network to calculate the error values of the hidden layer neurons. The hidden neuron loss function gradients can be solved using the chain rule of derivatives. Finally, the neuron weights are updated by calculating the gradient of the weights and subtracting a proportion of the gradient from the weights, which known as gradient descent we mentioned in previous chapters. This ratio is called the *learning rate* [4]. The learning rate can be fixed or dynamic. After the weights have been updated, the algorithm continues by executing the phases again with different input until the weights converge.

3.2.3. Deep Learning and Convolutional neural networks

Difference Between Machine Learning and Deep Learning

Deep learning is a specialized form of machine learning. A machine learning workflow starts with relevant features being manually extracted from images. The features are then used to create a model that categorizes the objects in the image. With a deep learning workflow, relevant features are automatically extracted from images. In addition, deep learning performs "end-to-end learning" – where a network is given raw data and a task to perform, such as classification, and it learns how to do this automatically.

Another key difference is deep learning algorithms scale with data, whereas shallow learning converges. Shallow learning refers to machine learning methods that plateau at a certain level of performance when you add more examples and training data to the network.A key advantage of deep learning networks is that they often continue to improve as the size of your data increases.

Overview of Convolutional Neural Networks

Convolution Neural Networks (ConvNets) are specialized Artificial Neural Networks. ConvNets are well suited for the processing of images, but the same concepts are adaptable for other fields, like audio or video. In this section, we describe the ConvNets for the processing of images.

A ConvNet consists of multiple convolution and pooling layers. At the end follows normally a fully connected layer. A pooling layer is applied after one or multiple convolution layers. The convolution layers have the task to extract useful features from the input, which results in multiple feature maps. The pooling layer reduces the spatial size of these feature maps.

Convolution layer

A convolution layer consists of units as well as a normal neural network, but with a different arrangement and a different connectionism of the units. The main differences to the neural networks are: A convolution layer consists of units as well as a normal neural

network, but with a different arrangement and a different connectionism of the units. The main differences to the neural networks are:

- Three dimensional arrangement of the unit, instead of 1 dimension
- Weight sharing
- Local connectivity

The three dimensional arrangement comes from the image. A colored image has normally three channels (red, green, blue), and every channel is described by a two dimensional matrix. Therefore, the input of the ConvNet is a three dimensional matrix. The output of a convolution layer is again a three dimensional matrix, with feature maps of two dimensions and this x times for the number of filters for this layer. Every filter produces a feature map.

Weight sharing means that the same weights are used for multiple output units. Through this, the ConvNet gets the property, that the features are invariant against translation. This means, that a feature can be found on the complete input. To compare this with the Sobel filter, the weights are the filter, and this filter will be used on the complete input to generate the output.

Local connectivity means that that not all units of the input are connected with the output unit. The size of the local connectivity is described by the kernel size.

In the Convolutional layers image can be filtered using the convolution operation [33]. A convolution kernel, or filter, describes how each pixel will be influenced by its neighbors. For example, a blurring filter will take the weighted average of neighboring pixels so that large differences between pixel values are reduced. By using the same source image and changing only the filter, one can produce effects such as sharpening, blurring, edge enhancing, and embossing.

Convolution algorithms works by iterating over each pixel in the source image. For each source pixel, the filter is centered over the pixel, and the values of the filter multiply the pixel values that they overlay. A sum of the products is then taken to produce a new pixel value. Figure 3.14 provides a visual representation for this algorithm.

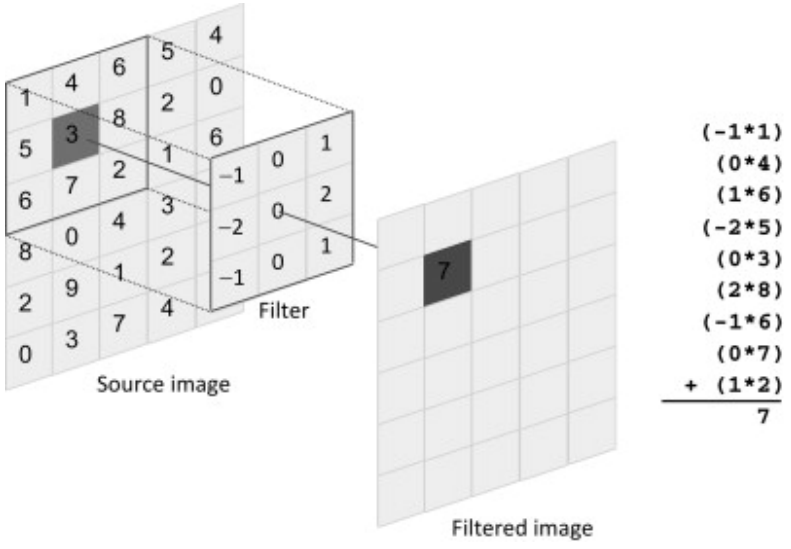

Figure 3.14.: Convolution filter

The discrete convolution operation between an image f and a filter matrix g is defined as:

$$h[x,y] = f[x,y] * g[x,y] = \sum_n \sum_m f[n,m]g[x-n][y-m] \tag{3.9}$$

In effect, the dot product of the filter g and a sub-image of f (with same dimensions as g) centred on coordinates x, y produces the pixel value of h at coordinates x, y [14]. The size of the receptive field is adjusted by the size of the filter matrix. Aligning the filter successively with every sub-image of f produces the of output pixel matrix h. In the case of neural networks, the output matrix is also called an feature map [14](or an activation map after computing the activation function). Edges need to be treated as a special case. If image f is not padded, the output size decreases slightly with every convolution.

A set of convolutional filters can be combined to form a *convolutional layer* of a neural network [12]. The matrix values of the filters are treated as neuron parameters and trained using machine learning. The convolution operation replaces the multiplication operation of a regular neural network layer. Output of the layer is usually described as a volume. The height and width of the volume depend on the dimensions of the activation map. The depth of the volume depends on the number of filters.

Since the same filters are used for all parts of the image, the number of free parameters is reduced drastically compared to a fully-connected neural layer [25]. The neurons of the convolutional layer mostly share the same parameters and are only connected to a local region of the input. Parameter sharing resulting from convolution ensures translation invariance. An alternative way of describing the convolutional layer is to imagine a fully-connected layer with an infinitely strong prior placed on its weights [14]. This prior forces the neurons to share weights at different spatial locations and to have zero weight outside the receptive field.

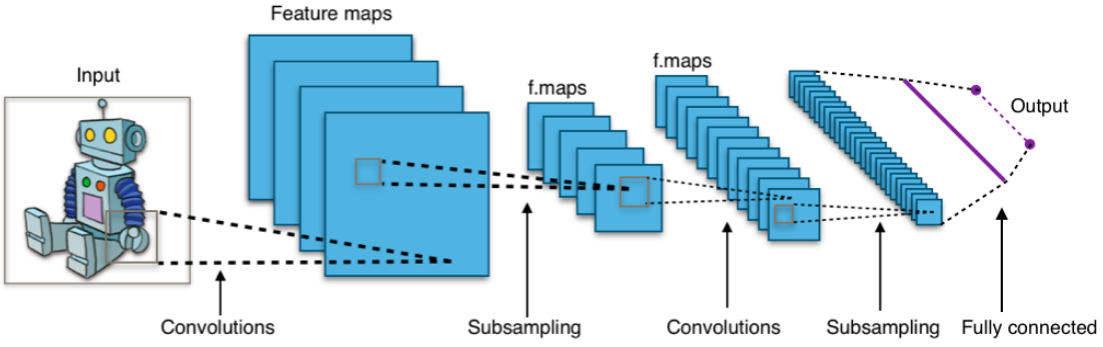

Figure 3.15.: An example of a convolutional network

Successive convolutional layers (often combined with other types of layers, such as pooling described below) form a *convolutional neural network*(CNN). An example of a convolutional network is shown in figure 3.15. The back-propagation training algorithm, described in previous section, is also applicable to convolutional networks [14]. In theory, the layers closer to the input should learn to recognize low-level features of the image, such as edges and corners, and the layers closer to the output should learn to combine these features to recognize more meaningful shapes [12].

Pooling and stride

The pooling layer is normally applied after some convolution layers. It has the function to reduce the spatial size of the feature maps. This reduce the amount of parameters and the computation time, because the next layer get a smaller version of the feature maps. In other words, it down samples the feature maps. With one pooling with the stride size of 2×2, the spatial dimension of the feature maps is reduced of 75%. The pooling works independently on every feature map and has no activation function or weights to learn. Furthermore, the pooling layer helps to be invariant to small translations of the input, because a small translation has mostly no change on the values of the outputs of the pooling layer.

The most common pooling layers are the max- and average-pooling. Both pooling layers take as hyperparameters the filter size and the stride. The max-pooling computes the maximum, which lies in the receptive field of the filter, and the average-pooling computes the average. An example for both poolings are depicted in figure 3.16. As recommend values for the filter size and the stride are usually for both the size of 2×2. A stride of 1×1 has no down sampling character, and therefore it is not common. A higher size for the stride has normally a negative effect, because the reduction is to much (the filter size has at the minimum the same size like the stride). A filter size of 3×3 and a stride of 2×2 has sometimes a performance gain [23].

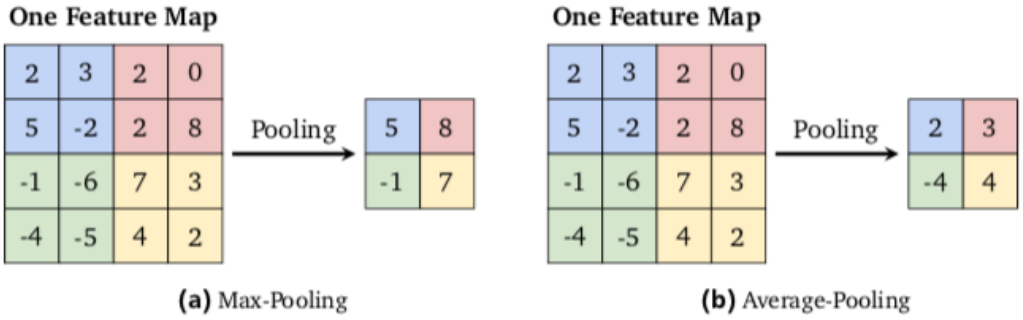

Figure 3.16.: Example for the max-pooling and the average pooling with a filter size of 2×2 from the Stanfrod Lecture CS231n [28]

If the feature map is not divisible by the stride without remainder, then it will be cut off the last row or column, and we lost information. In this case, it is better to add a padding. Another approach to avoid this problem is to use an input size of power of two for the network. And add to all convolutions a padding. So will be all feature maps divisible by the stride size, if the stride was 2×2.

There are further approaches for the pooling, like to use a pooling with max+average. Another approach is to use no pooling, and instead to use a convolution with a stride [48]. This reduce also the size of the feature maps, and has the benefit, that a further non-linearity is used.

Additional layers

The convolutional layer typically includes a non-linear activation function, such as a rectified linear activation function. Activations are sometimes described as a separate layer between the convolutional layer and the pooling layer.

Some systems, such as [47], also implement a layer called local response normalization, which is used as a regularization technique. Local response normalization mimics a function of biological neurons called lateral inhibition, which causes excited neurons to decrease the activity of neighboring neurons. However, other regularization techniques are currently more popular and these are discussed in the next section.

The final hidden layers of a CNN are typically fully-connected layers [4]. A fully-connected layer can capture some interesting relationships parameter-sharing convolutional layers cannot. However, a fully-connected layer requires a sufficiently small data volume size in order to be practical. Pooling and stride settings can be used to reduce the size of the data volume that reaches the fully-connected layers. A convolutional network that does not include any fully-connected layers, is called a *fully convolutional network*(FCN) [39].

If the network is used for classification, it usually includes a softmax output layer [4], The activations of the topmost layers can also be used directly to generate a feature representation of an image. This means that the convolutional network is used as a large feature detector [25].

Prevent Overfitting

A classifier trains a model which predicts the training set labels well, but fails to generalize the problem at hand well enough to predict the labels of the test setwith satisfactory accuracy. This phenomenon is called overfitting [4].

Data augmentation

Overfitting can be reduced by increasing the amount of training data. When it is not possible to acquire more actual samples, data augmentation is used to generate more samples from the existing data [14]. For classification using convolutional networks, this can be achieved by computing transformations of the input images that do not alter the perceived object classes, yet provide additional challenge to the system. The images can be, for example, flipped, rotated or subsampled with different crops and scales. Also, noise can be added to the input images [14].

Batch Normalization

The training of a network changes the weights on every layer. This change has the effect, that the input distribution changes during updates of previously layers. The authors of batch normalization [**?**] this effect internal covariate shift. To counteract the change of the distribution during the learning, they introduced the batch normalization. Every mini-batch, which is inserted in the network, will be on every layer normalized on the input of

the previously layer. The formula for the normalization is the following

$$\mu \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.10}$$

$$\sigma^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu) \tag{3.11}$$

$$\tilde{x} \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{3.12}$$

where μ and σ describe the mean and the standard deviation, $\tilde{x_i}$ is the normalized value of the input. The value m describe the size of the mini-batch. The value $\tilde{x}$ has then a mean of 0 and a standard deviation of 1. The batch normalization will be applied after the linear combination of a unit. The value $\tilde{x}$ will be scaled and shifted.

At test time, μ and σ are replaced with the average, which is collected during the training. So it is possible to predict a single example, without to have a minibatch.

Through the applying of batch normalization, the learning rate can be increased, and this results in a faster training. Furthermore, the accuracy is increasing compared to the same network without batch normalization [**?**].The batch normalization helps the activation function ReLU to learn faster and have a better performance [**?**].

Dropout

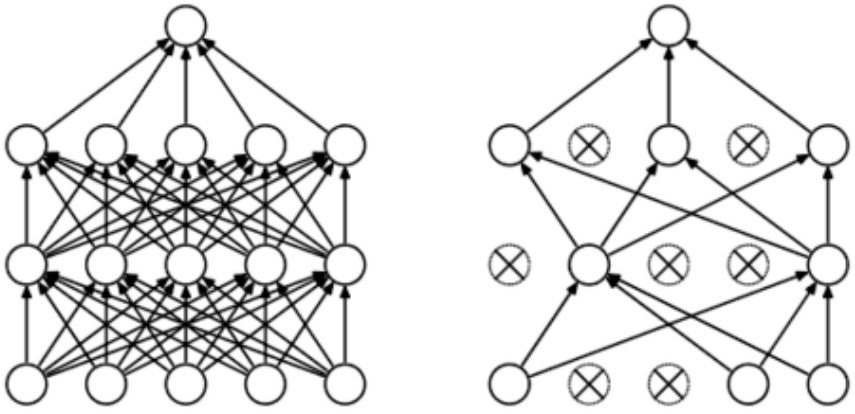

Figure 3.17.: Neural network with dropout *Left:*A standard neural network with 2 hidden layers. *Right:*An example of a thinned net produced by applying dropout to the network [49]

In CNNs or more generally machine learning overfitting is a syndrome, that hinders a model from showing generalized behavior. In simple terms the model performs good only on the training data-set, with which it is trained. Dropout is a technique for addressing this problem [49]. The key idea is to randomly drop units from the neural network during training. This prevents nodes from co-adapting too much and being dependent on other nodes. This causes the system not to depend too much on any single neuron or

connection and provides an effective yet computationally inexpensive way of implementing regularization [14]. In convolutional networks, dropout is typically used in the final fully-connected layers [47]. A neural network with dropout is shown in figure 3.17.

3.3. Famous Network Models in field of Computer Vision

Several designs of CNN architectures have already been created, some of them will be described here.

3.3.1. VGGNet

The VGG network architecture was introduced by Simonyan and Zisserman in their 2014 paper, *Very Deep Convolutional Networks for Large Scale Image Recognition.* [47]. It makes the improvement over AlexNet by replacing large kernel-sized filters with multiple 3×3 kernel-sized filters one after another.
The architecture depicted in figure 3.18 is VGG16. The input to cov1 layer is of fixed

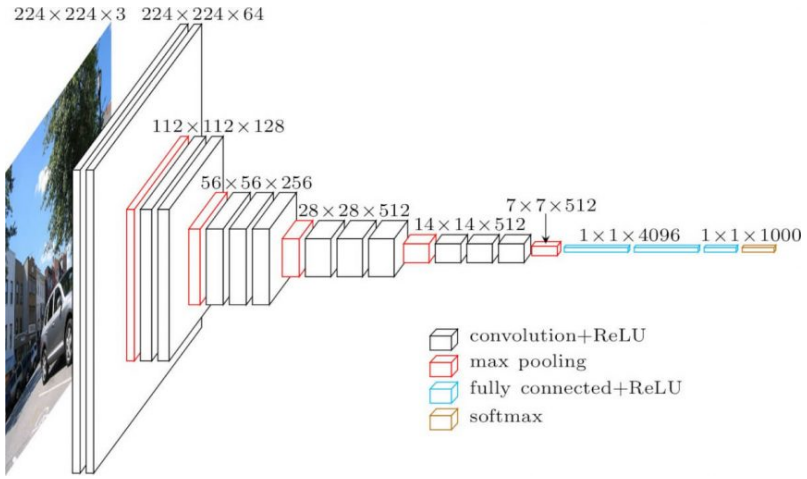

Figure 3.18.: VGG-16 Architecture

size 224 x 224 RGB image. The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations, it also utilizes 1×1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third

performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.

All hidden layers are equipped with the rectification (ReLU) non-linearity. It is also noted that none of the networks (except for one) contain Local Response Normalisation (LRN), such normalization does not improve the performance on the ILSVRC dataset, but leads to increased memory consumption and computation time.

3.3.2. GoogLeNet/Inception

The *Inception* micro-architecture was first introduced by Szegedy et al. in their 2014 paper, *Going Deeper with Convolutions* [54]

While VGG achieves a phenomenal accuracy on ImageNet dataset, its deployment on even the most modest sized GPUs is a problem because of huge computational requirements, both in terms of memory and time. It becomes inefficient due to large width of convolutional layers.

In a convolutional operation at one location, every output channel is connected to every input channel, and so it can be called a dense connection architecture. The Inception architecture builds on the idea that most of the activations in a deep network are either unnecessary(value of zero) or redundant because of correlations between them. Therefore the most efficient architecture of a deep network will have a sparse connection between the activations, which implies that all output channels will not have a connection with all the input channels.

So GoogLeNet devised a module called inception module that approximates a sparse CNN with a normal dense construction, which is shown in figure 3.19. Since only a small number of neurons are effective as mentioned earlier, the width/number of the convolutional filters of a particular kernel size is kept small. Also, it uses convolutions of different sizes to capture details at varied scales. Another salient point about the module is that it has a so-called bottleneck layer(1X1 convolutions in the figure). It helps in the massive reduction of the computation requirement as explained below. Another change that GoogLeNet

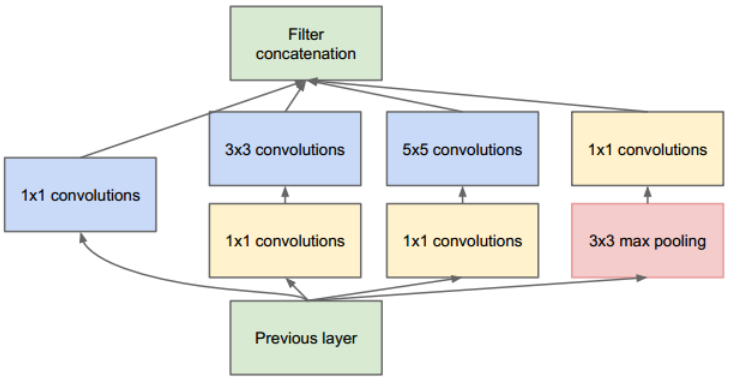

Figure 3.19.: Inception module with dimension reductions

made, was to replace the fully-connected layers at the end with a simple global average

pooling which averages out the channel values across the 2D feature map, after the last convolutional layer. This drastically reduces the total number of parameters. This can be understood from AlexNet, where FC layers contain approx. 90% of parameters. Use of a large network width and depth allows GoogLeNet to remove the FC layers without affecting the accuracy.

3.3.3. Residual Networks

During each iteration of training a neural network, all weights receive an update proportional to the partial derivative of the error function with respect to the current weight. If the gradient is very small then the weights will not be change effectively and it may completely stop the neural network from further training. The phenomenon is called vanishing gradients [18].

To overcome this problem, Microsoft introduced a deep residual learning framework [17]. Instead of hoping every few stacked layers directly fit a desired underlying mapping, they explicitly let these layers fit a residual mapping. The formulation of $F(x) + x$ can be realized by feedforward neural networks with shortcut connections. Shortcut connections are those skipping one or more layers shown in figure 3.20. The shortcut connections perform identity mapping, and their outputs are added to the outputs of the stacked layers. By using the residual network, there are many problems which can be solved such as:

- ResNets are easy to optimize, but the "plain" networks (that simply stack layers) shows higher training error when the depth increases.
- ResNets can easily gain accuracy from greatly increased depth, producing results which are better than previous networks.

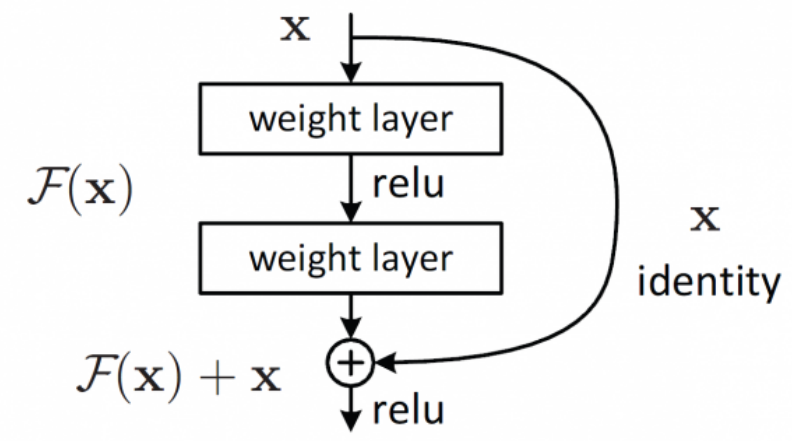

Figure 3.20.: Shortcut connection

Residual network [17] based on the above plain network, a shortcut connection is inserted which turn the network into its counterpart residual version. The identity shortcuts $F(xW + x)$ can be directly used when the input and output are of the same dimensions. When the dimensions increase, it considers two options:

- The shortcut performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no additional parameter.

- The projection shortcut in $F(xW + x)$ is used to match dimensions (done by 1×1convolutions).

For either of the options, if the shortcuts go across feature maps of two size, it performed with a stride of 2.

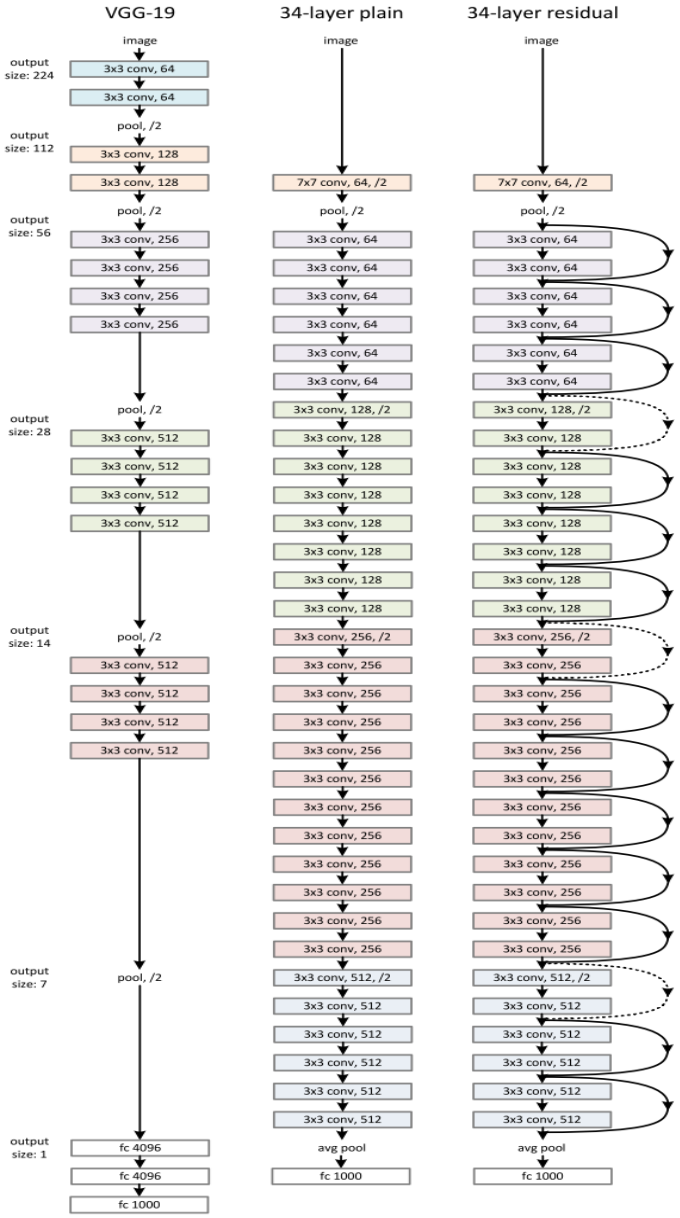

Figure 3.21.: Example network architectures for ImageNet. Bottom: the VGG-19 model [47] (19.6 billion FLOPs) as a reference. Middle: a plain network with 34 parameter layers (3.6 billion FLOPs). Top: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. Table 1 shows more details and other variants [17].

Even though ResNet is much deeper than VGG16 and VGG19, the model size is actually substantially smaller due to the usage of global average pooling rather than fully-connected layers.

4. Methode and Pipeline Design

The remainder of this section describes the specific domain randomization and neural network training methodology we use. And we will walk through the pipeline and explain how different modules interact with each other. Firstly an overview of the pipeline architecture is presented, and the functionality of different modules will be introduced. Then the pipeline about the construction and training of neural network model will be described. Lastly the details of evaluation methods are given.

4.1. Pipeline Architecture

Given some objects of interest s_{ii}, our goal is to train an object detector $d(I_0)$ that maps a single camera frame I_0 to the Cartesian coordinates $(x_i, y_i, \theta_i)_i$ of each object. In addition to the objects of interest, our scenes sometimes contain distractor objects that must be ignored by the network. Our approach is to train a deep neural network in simulation using domain randomization.

The entire pipleline consists of four parts, namely the simulation environment module, the model training module, the real world module and the evaluation module. The simulation environment is built based on unity3D and is used to generate simulation data needed for model training. The model training module is used to preprocess the training data, then build and train the CNN model. The acquisition of real data is implemented in the real world module, and the prediction effect of them model trained by pure simulation data can be evaluated by real data. The evaluation module can be used to analyze the performance of the model on simulation data, and can also be used to analyze the performance in the real world, that is, the CNN model trained on the simulation data predicts the 3D pose of real world image and identifies error between the actual pose of the object. The structure of the entire pipeline is shown in figure 4.1. The final goal is to achieve better object recognition and detection accuracy by adjusting the structure of the CNN model and the rendering effect of the simulation environment.

4.2. Domain randomization in Unity3D

The purpose of domain randomization is to provide enough simulated variability at training time such that at test time the model is able to generalize to real-world data. We randomize the following aspects of the domain for each sample used during training:

- Number and shape of distractor objects on the table
- Position and texture of all objects on the table

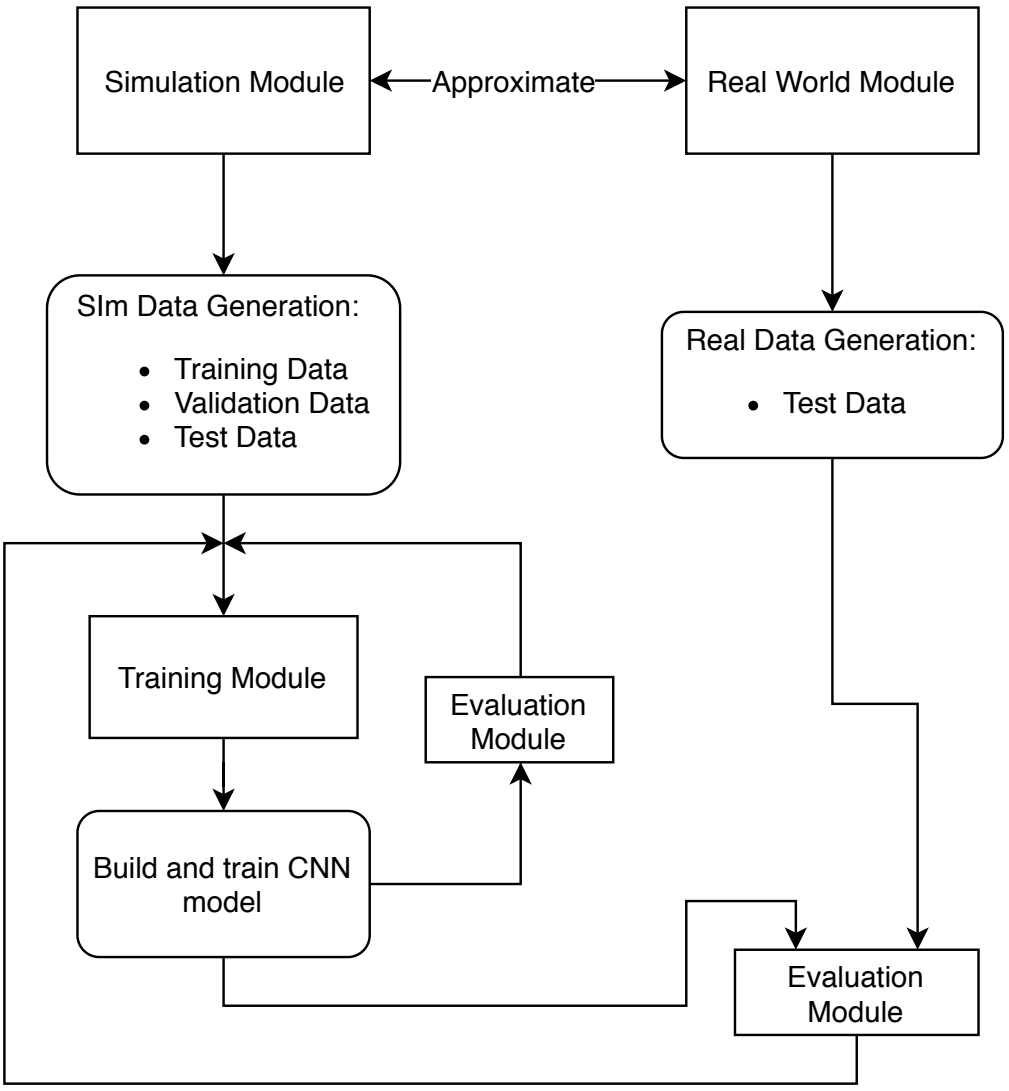

Figure 4.1.: Pipeline Structure

- Textures of the table, floor, skybox, and robot
- Position, orientation, and field of view of the camera
- Number of lights in the scene
- Position, orientation, and specular characteristics of the lights
- Type and amount of random noise added to image

Since we use monocular camera image from an uncalibrated camera to estimate object positions, we fix the height of the table in simulation, effectively creating a 3D pose estimation task. Random textures are chosen among the following:

- A random RGB value
- Additional pattern

The textures of all objects are chosen uniformly at random – the detector does not have access to the color of the object(s) of interest at training time, only their size and shape. We render images using the Unity3D's [1] built-in renderer. This renderer is not intended

to be photo-realistic, and physically plausible choices of textures and lighting are not needed.

Between 0 and 10 distractor objects are added to the table in each scene. Distractor objects on the floor or in the background are unnecessary, despite some clutter (e.g., cables) on the floor in our real images.

Our method avoids calibration and precise placement of the camera in the real world by randomizing characteristics of the cameras used to render images in training. We manually place a camera in the simulated scene that approximately matches the viewpoint and field of view of the real camera. Each training sample places the camera randomly within a $(10 \times 5 \times 10)$ cm box around this initial point. The viewing angle of the camera is calculated analytically to point at a fixed point on the table, and then offset by up to 0.1 radians in each direction. The field of view is also scaled by up to 5 % from the starting point.

According to the previous introduction, the purpose of the whole experiment is to explore the performance of of the CNN model trained by pure simulation data and how to improve the recognition accuracy and ability of the model by adjusting the model structure and parameters and changing the rendering effect in the simulation environment. Therefore, the construction of the simulation environment is a very important part of the experiment. We require that the simulation environment have the following basic characteristics:

1. The rendering effect can be modified in real time, because the training of the CNN model relies on a large amount of training data, so in the data generation process, in order to obtain images with different rendering effects to fit the changes of the environmental parameters in the real world, the rendering must be able to be modified in real time during the image data generation process.

2. The positional parameters of target objects in the simulation environment are variable in real time. Since the purpose of the experiment is to predict the 3D pose of the target object, the simulation data needs to be contain a large amount of different pose information in the simulation scene for training the CNN model. Therefore the CNN model can learn a lot of different information, which is beneficial to the generalization effect of the model, so as to get better performance in the real word.

3. The environment need to generate the pose information of the object. As a supervised learning model, the CNN model requires effective label data by each image data for model training. In the process of collecting real world data, the labeling task for each image is very cumbersome and time-consuming, which often requires a lot of manpower. In order to solve this problem, the simulation environment should include the 3D pose information corresponding to the target object of each sample, and the label information can be directly output corresponding to the image.

Based on the above requirements, we chose Unity3D as the rendering engine for the simulation environment. As described in section 3, Unity3D is a commercial game engine with good user interface that is easy for operators to render and model. The graphics rendering engine of Unity3D can meet the requirements of image rendering task in our experiment. At the same time, Unity3D has a rich set of advanced APIs. By pre-writing script code, we can achieve the required rendering effects and settings.

4.2.1. Scene Construction

This step can be done directly in the graphical interface window of Unity3D. First we need to import the model into the scene, it can be generated in other modeling software and then imported into Unity3D, or we can directly generate a simple geometry model in Unity. Then other components in the scene need to be added, such as cameras, different light sources, and environmental components. The entire scene is built similar to the modeling process in general modeling software. After all required scene components have successfully appeared, we can adjust the relative positional relationship of components to generate the desired scene. A completed simulation scenario is shown in figure 4.2. After the scene is built, the next step is to implement the required functionality through different functional module written in scripts.

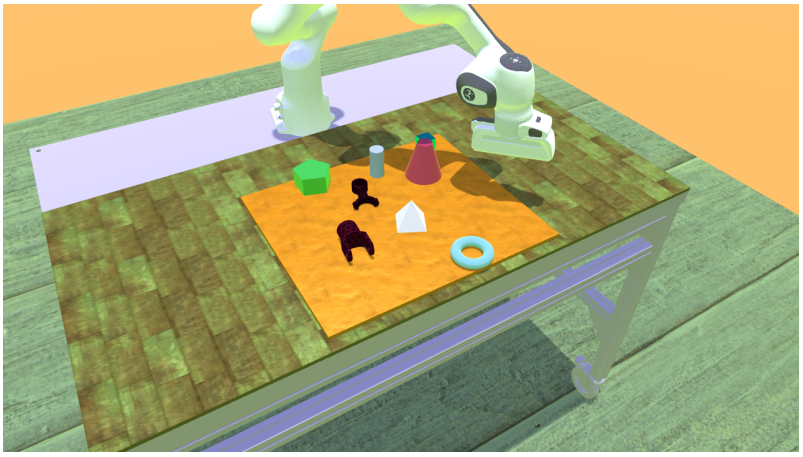

Figure 4.2.: An example of Simulation Scene

4.2.2. Object Randomization Module

As mentioned before, in order to make the rendered objects have different rendering effects, and randomly generate and output different pose in a certain area, we have written the function module in script called object randomization module.

The main function:

1. Select the target object that appears in the scene and randomly generate interfering objects in each sample
2. Randomly render the color and texture of the object appearing in the image
3. Randomly generate the pose of target and interfering objects, and record the pose information in each sample

Function details:

First, some basic parameters will be set, eg. data output address, the number of images to be generated. And the target object and the interfering objects need to be selected from the candidate objects list. After all the basic parameters have been read in, the program begins to prepare to render the image frame by frame. The program randomly determines the number of objects that appear in the frame of frame, including the target object and

the interfering object. When the number of objects of current frame image is determined, the individual objects are rendered one by one. Firstly for the target object, it's pose and texture will be randomly generated. After that, the interfering objects will be rendered under the conditions that avoid collision with previously generated objects. When all the objects in current frame have been rendered, the pose information of target object will be stored. Next, it is judged whether the preset number of images is reached, if the number of image is not reached, the rendering process is continued in a loop. If all the images have been rendered, the rendering loop will be terminated and the pose information of target object of each frame will be packed into .json[1] file and output. As shown in figure 4.3, the function of the object randomization module is implemented by looping. After each re-entry loop, the new image will be randomly rendered, at the same time, the pose information corresponding to current frame will be stored.

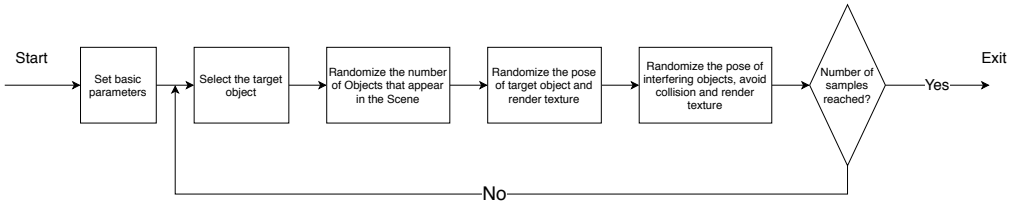

Figure 4.3.: Object Randomization Module

APIs Summary:

1. Target selector: Select the target object from the candidate objects
2. Interfering option: Select whether the interfering objects appear or not
3. Texture option: Select whether to add extra texture to the target object, or perform solid color rendering on the target object
4. Target color option: Select the HSV color rendering range of the target object, it can be selected to render the object around actual color or render full randomly

All APIs are integrated into the Unity3D user interface via scripts, which can be selected directly via input column or down menu, and the functionality of the APIs can be further extended according to requirements to achieve different functional combinations. Figure 4.4 shows what the currently written APIs look like in the Unity3D control panel.

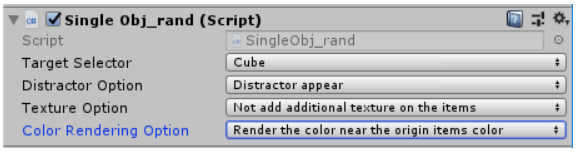

Figure 4.4.: APIs for Object Randomization Module

4.2.3. Environment Randomization Module

According to the method of domain randomization, in addition to the random rendering of target object and the interfering objects in the scene, it is necessary to randomly render

[1] JSON (JavaScript Object Notation): https://www.json.org/

the environment of the entire scene. In our experiments, the scene mainly include the floor, table, robot, ambient light and the skybox.

The main function:

1. Randomly render the floor, table, skybox
2. Randomly render the ambient light with different numbers, lighting angles and spectrum

Function details:

The environment of the scene mainly includes the floor, table, ambient light and skybox. The program first determines the environmental objects in scene and associates them with the objects defined in scripts. Then according to the setting parameters to determine whether the environment object is rendered with additional complex textures, otherwise it will be randomly rendered with a solid color. After that, the ambient light source in the scene will be rendered. The light source is mainly divided into directional light and point light. The directional light is the main light source in the environment, it is used to simulate the sunlight or the main illumination source in the real world. The number of the directional light and the lighting angle will be randomly rendered. The spectrum of directional light can be selected as daylight or full randomly color. Finally, the number and location of point light are randomly rendered, which is used to simulate the effects of reflected light in the real world. Figure 4.5 describes the implementation of environment randomization.

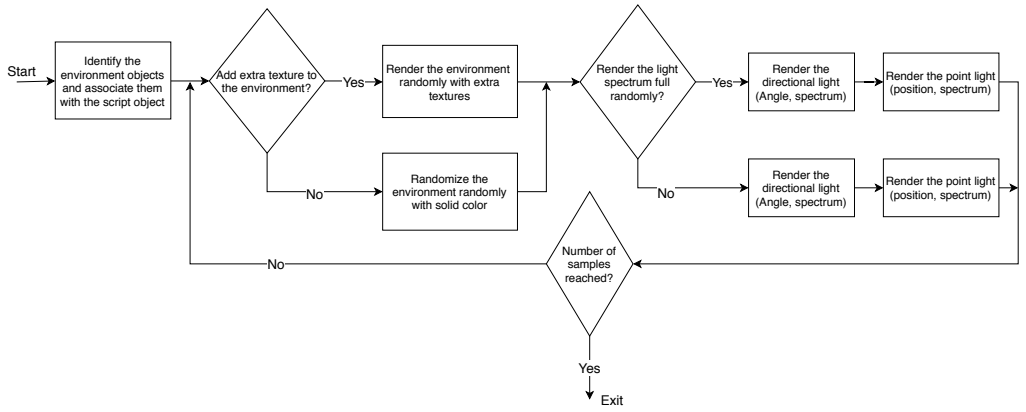

Figure 4.5.: Environment Randomization Module

Api summary:

1. Texture option: Whether to render environment with additional complex texture, or with solid color
2. Num of directional light: Determine the number of directional light added to the scene
3. Spectrum option: Select the spectrum of the light source, which can be selected to simulate daylight or render on full random color
4. Num of point light: Determine the number of point lights in the scene

APIs are also integrated into the Unity3D interface through the setup script, as shown in figure 4.6.

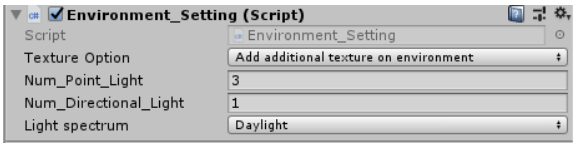

Figure 4.6.: APIs for Environment Randomization Module

4.2.4. Camera Randomization Module

In the scene rendering process, the image data we need is obtained from the perspective of the scene camera, so we need special camera function module to control the position and direction of the camera angle to simulate the pose of the camera in real world. The parameters of the camera mainly include external parameters and internal parameters. The external parameters describes the positional relationship of the camera coordinate system with respect to the word coordinate system, while the internal parameters describes the transformation between image plane coordinates and pixel coordinates, and also the geometric distortion introduced by the optics. [62]. Due to the certain error in the installation of the camera and in the internal structure, camera calibration is often required before using it in the real world. For camera calibration Zhang' s method can be used. For our experiment this error will cause a certain deviation between the simulated image and the real world image. In order to avoid the camera calibration caused from this error, we will randomize the characteristics of the camera in the rendering process.

The main function:

1. Place the camera with a initial pose and start it with default initial parameters, which include focus length and field of view (FoV).
2. In each frame randomize the camera' s position in a small range of area and also the viewing angle and FoV.
3. Control the camera to save the image after each frame is rendered

Function details:

The camera' s initial position and parameters will be read in, then the camera can be set by script to match the state of the camera in real world. In each sample, the camera is randomly moved in a predetermined small box area, which is placed around the initial position of the camera. After that setting the view angle of camera point to the center of the table analytically, and then a deflection will be set in each direction. The camera' s FoV will be randomly scaled in a small range. After completing the randomization of camera, the image storage process will wait until the rendering process is complete. The entire process is shown in figure 4.7.

Api summary

1. Initial Pose: Set the initial pose of camera
2. Max Placement Error: Set the maximum random translation error range in the x, y, z directions. That is, the box area around the initial point mentioned before.
3. Max Angle Error: Set the maximum deflection error range of the view angle to simulate the error caused by installation.
4. Enable Second Camera: Whether enable second camera or not.

The APIs is integrated into the Unity3D control panel, as shown in figure 4.8.

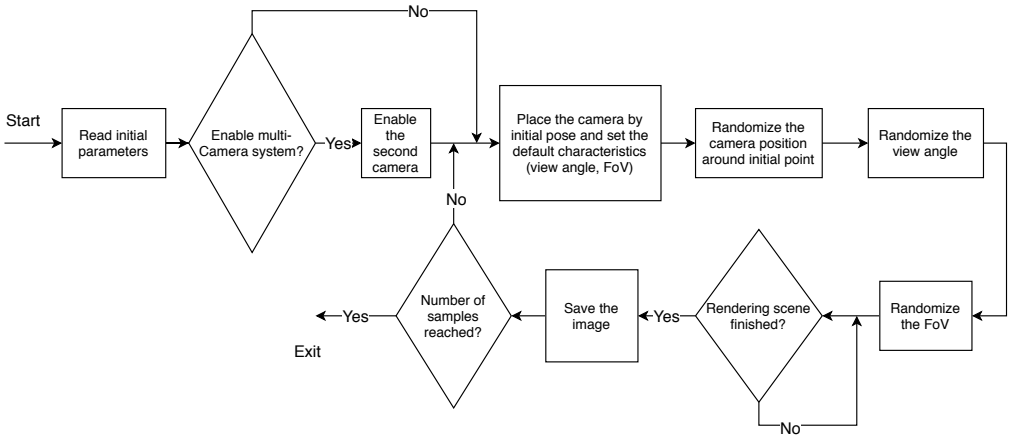

Figure 4.7.: Camera Randomization Module

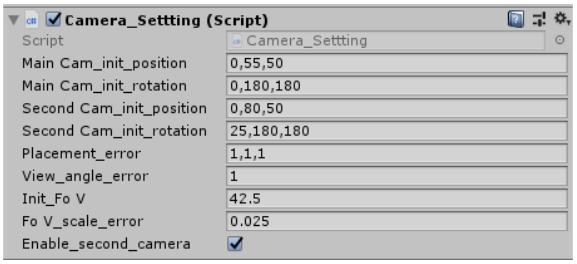

Figure 4.8.: APIs for Camera Randomization Module

4.2.5. Robot-arm Randomization Module

The robot arm in the simulation scene can be set with a random grab gesture to fit the occlusion on the view area during the capture process in real world. Depending on the requirement , we can select whether the robot arm randomly generates the grab gesture or keep in the initial state during the rendering process.

The main function:

1. Control the angle of each joint of the robot arm to randomly generate different grab gesture.

Figure 4.9 shows the effect of the robot arm in different grab gesture.

4.2.6. Common Setting Module

Some basic globe parameters can be entered manually directly through the parameters setting module, or read from a pre-written XML file.

The main function:

1. Read in global parameters which are manually entered
2. Read the parameters from the XML file

Api summary:

1. Parameter entry: manually enter the global parameter, or read in from the xml

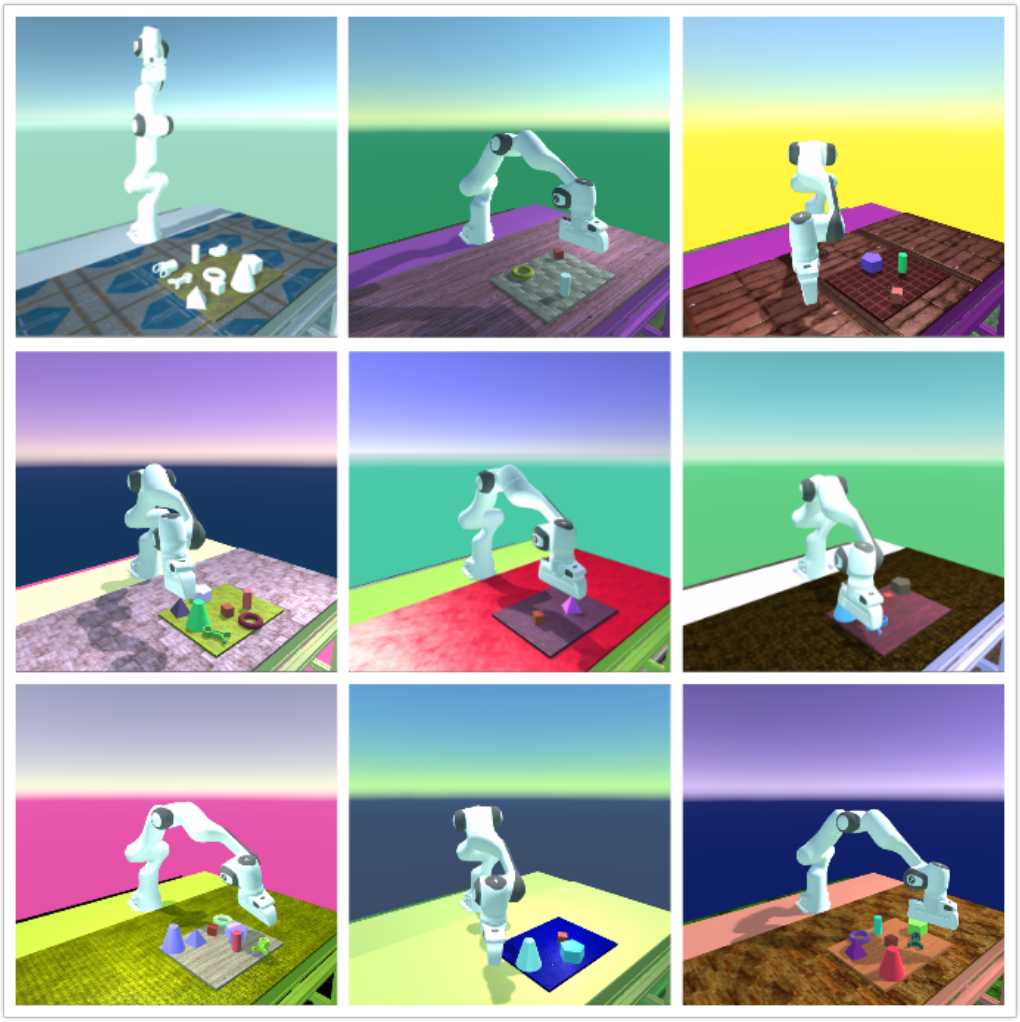

Figure 4.9.: Different grab gestures of robot arm

2. Manually entered parameter content (storage address, picture resolution)

3. Total Num of Images: set the number of images to be rendered

4. Enable save images: Whether save images or not

4.3. Model architecture and training

Based on the simulation environment introduced in the previous section, we obtained a large number of image data with labels. Each sample is randomly rendered based on domain randomization through the unity engine. The next step is to use the simulation data to train the CNN model. In this Section, the Setup of the model structure and the training process of it will be introduced.

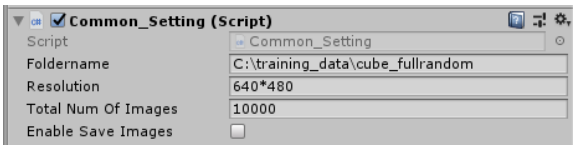

Figure 4.10.: APIs for Common Setting Module

4.3.1. Environment set up

Keras [2] is the recommended library for beginners, since its learning curve is very smooth compared to others, and at the moment it is one of the popular middleware to implement neural networks. Keras is a Python [3] library that provides, in a simple way, the creation of a wide range of Deep Learning models using as backend other libraries such as TensorFlow [4], Theano [5] or CNTK [6]. It was developed and maintained by François Chollet, an engineer from Google, and his code has been released under the permissive license of MIT. Also an important thing is that Keras is included in TensorFlow as a API. Although Keras is currently included in Tensorflow package, but can also be used as a Python library.

Our model training environment is based on Keras, because Keras provides a number of advanced APIs that make it easy for us to build CNN models quickly and modify the structure of the model. From Keras we can directly import many popular networks with good performance in the field of computer vision, such as VGG-net, Resnet. At the same time, Keras provides an interface that can import pre-trained weights of these neural networks based on ImageNet [7], so we can import the models with the initial weights, and then modify the structure of the neural network by modifying the fully connected layers and output layer to get the model we want. This enable us to implement fast experiments to doing good research.

4.3.2. Data Preprocess and import

For the training of the neural network model, the first step is the preprocessing and import of training data. Since the data obtained from Unity3D are separate images und a file in json format which has stored label information. If the training data are directly read into the neural network, the image data need to be processed in real time, which will greatly consume the GPU and CPU resources of system. In this case the training efficiency will be reduced. So we need to preprocess and manage the large amounts of training data, which is very import for the training efficiency.

The Hierarchical Data Format(HDF5)

The first thing we need to do is to convert the training data that was stored in raw-image format into an array-like format, which can be read directly by the neural network model.

[2] https://keras.io/
[3] https://www.python.org/
[4] https://www.tensorflow.org/
[5] http://deeplearning.net/software/theano/
[6] https://github.com/Microsoft/CNTK
[7] http://www.image-net.org/

Here HDF5 is chosen to store the image data, because it has good hierarchical structures for data storage, and data are stored in an array-like form.

The Hierarchical Data Format version 5 (HDF5) [8] is a popular format for storing and exchanging unstructured data such as images, videos or volumes in raw format for use in various areas of research or development. It is supported by many programming languages and APIs and is therefore becoming increasingly popular. This also applies to storing data for use in machine learning workflows.

A HDF5 file consists of two major types of objects: Datasets and groups. Datasets are multidimensional arrays of a homogeneous type such as 8-bit unsigned integer or 32-bit floating point numbers. Groups on the other hand are hierarchical structures designed for holding datasets or other groups, building a file system-like hierarchy of datasets. Additionally, groups and datasets might have metadata in the form of user-defined attributes attached to them. Python supports the HDF5 format using the h5py package. This package wraps the native HDF C API and supports almost the full functionality of the format, including reading and writing HDF5 files.

Image Preprocessing

The format of the image data obtained from the Unity3D is JPG (Joint Photographic Experts Group) [9] with a default resolution of 640×480. The default input size of the CNN model is $224 \times 224 \times 3$, where 224 represent the length and width of the image. Because we use color image, each image data will have 3 channels to store the r,g,b values. Therefore, we need tor resize the image and store it in the array-like form, and store the label information corresponding to each image. Here we use the dataset and group methods to classify the different type of data for storage.

We classify the dataset into 2 groups, image and label. By creating different groups, the image data and the label data are respectively written into each group in array-like form. In each group, the corresponding entire data will form a whole dataset. For example, there are N images and labels in the dataset, which are jpg format with a resolution of 640×480. After importing the images and labels into the HDF5 file we get two groups of dataset, image and label. In the image group, the shape of dataset is $N \times 224 \times 224 \times 3$. Which means the image data are downsized to 224×224, and in the dataset exist N images. Meanwhile the size of the dataset in label group is $N \times 3$, because each label corresponds to the image has 3 label information, which are position coordinate along the axis x and y, and the rotation θ around the z axis. In this way, by storing the image in HDF5 file, we get the array-like dataset. Figure 4.24 shows a example of the data structure in HDF5 file. Next we will design how to read the data into the training model.

Data Import For the training of CNN models, mini-batch training [29] will be used. As mentioned in Chapter 2, mini-batch training allows for a more robust convergence and helps avoid local minima. The batched updates provide a computationally more efficient process than stochastic gradient descent, which benefits from vectorization. Not only that, the batching allows both the efficiency of not having all training data in memory and algorithm implementations, which fits in the memory easily.

In this case, we use data generator to get the batch training data from the dataset. The generator is run in parallel to the model for efficiency. Through the generator the number of

[8] https://www.hdfgroup.org/solutions/hdf5/
[9] https://jpeg.org/about.html

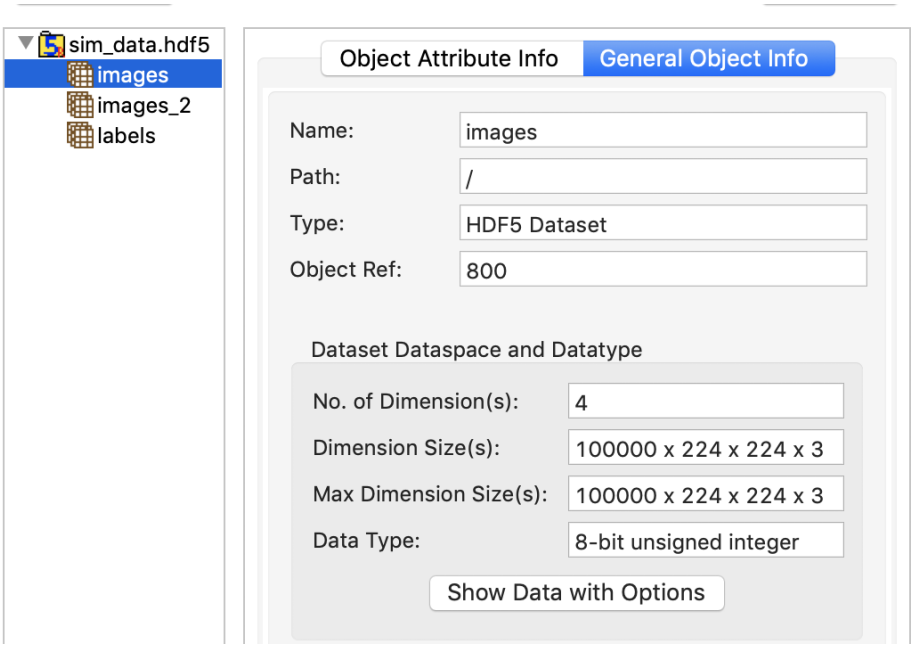

Figure 4.11.: An example of a hdf5 file with three groups, including images, image_2 and labels. The size of the dataset in images is $100000 \times 224 \times 224 \times 3$, which means 100000 RGB images with size of 224×224 are stored in the dataset.

data obtained from the dataset can be controlled by giving the batch size parameter. That means the training process can be implemented under different batch sizes, and through experiment we can choose a batch size to achieve faster convergence speed.

4.3.3. Architecture design

we parametrize our object detector with a deep convolutional neural network. As mentioned in the previous chapters, several well-known CNN models such as VGG-16 [47], ResNet50 [17], Inception V3 [**?**], have very good performance on the variable computer vision tasks. Similar to the work in [57], in our experiment we will use the modified version the these models. By retaining the convolutional layers of the model, but modifying the structure of the fully connected layers, a new model can be established. Like [57] dropout will be removed from the model. Figure 4 shows a modified VGG-16 model by modifying its fully connected layers. In this model, ReLU nonlinearities are used in each convolutional layers. And we reduce the size of the fully connected layer to 256 and 64, and modify the output layer to 3 outputs, which predict the 3D pose of the object of interest.

Similar to the model above, the modified ResNet50 model and Inception V3 model are shown in figure 4. In the experiment, the performance of each model will be verified, the goal is to select the model with best performance and this model will be used to verify the impact of different rendering conditions. Finally, we the improve the recognition accuracy in the real world by analyzing the optimal rendering conditions and model structure.

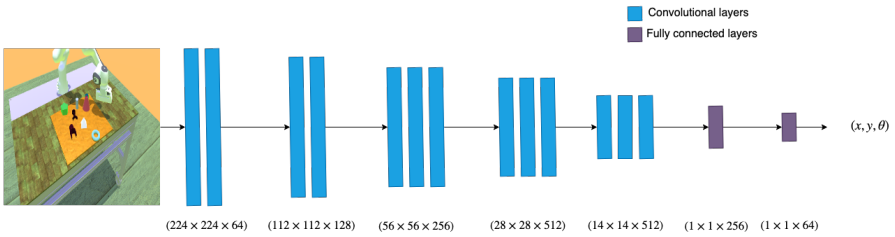

Figure 4.12.: The initial model architecture of VGG-16. Each vertical bar corresponds to a layer of the model. ReLU nonlinearities are used throughout, and max pooling occurs between each of the groupings of convolutional layers. The input is an image from an external webcam downsized to (224 × 224) and the output of the network predicts the (x, y, θ) coordinates of object(s) of interest.

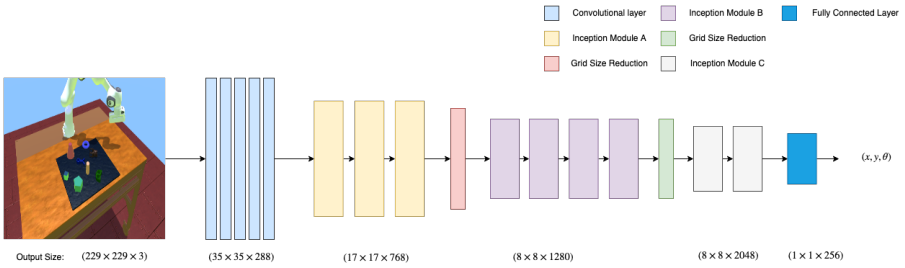

Figure 4.13.: The initial model architecture of Inception-v3. Each vertical bar corresponds to a layer or module of the model. The input is an image from an external webcam downsized to (224 × 224) and the output of the network predicts the (x, y, θ) coordinates of object(s) of interest.

4.3.4. Multicamera system

Once we have a single-image object detector, we can combine their outputs to obtain improved detection performance. Roughly speaking, if we receive multiple observations of the same scene but from different angles, then even if these observations are correlated it is quite possible that the combined beliefs derived from all of the observations will be better than any single observation[6].

In our setting, we assume that we are given multiple different images of the same scene where the viewpoints from which the images were captured are separated by a modest baseline (similar to what could be achieved with multiple cameras on a robot, or by a robot that can move itself or the cameras a short distance). We'll assume from now on that we have just two cameras, but the same technique can straight-forwardly be applied to any number of observations.

For the multi-cam system, since there are two image as input data of the training model in each frame, we use two convolutional layer models in parallel to process the input data. That means two models learn the information of the two image at the same time. After the convolutional layers, the two model are fed to a fully connected layer to fuse the information from the two images. A example of the model is shown in figure 4, in the model all layers use ReLU [37] activation function. The 3 outputs in the last layer predict the 3D of target object.

4.3.5. Model training

Loss function The final output of our model are position and rotation, which includes two different units, one of them is the length unit of the position prediction, and the other is the angle unit for predicting the rotation. For each part of prediction L2 loss will be used, that means there are two Loss $L(x, y)$, $L(\theta)$ here, corresponding to position and rotation. Since the units of the two Loss function are different, for a better training effect we use a coefficient to correct the $L(\theta)$. The formula is as follow:

$$L(total) = L(x, y) + Coef * L(\theta) \tag{4.1}$$

Optimization Gradient descent is on of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. As we mentioned in the previous section, there are 3 variants of gradient descent, which differ in how much data we use to compute the gradient of the loss function:

- Stochastic gradient descent (SGD)
- Batch gradient descent
- Mini-batch gradient descent

In the early training process, we used the mini-batch gradient descent to get better and faster convergence, and the use SGD to achieve better convergence.

Based on the gradient descent method, there are other types of optimizers that are used for training process, which can solve some of the problems that normal SGD may encounter, and some of them that are widely used are: Momentum, RMSprop, Adam. The work in [57] used the Adam optimizer to prediction the position of target object, and achieved good results. We also use Adam to train the model and predict position and rotation.

In the first step of experiment different learning rates will be set for the optimizer. Combining the previous hyperparameters such as batch size and model structure, all these hyperparameters wil be adjusted to get the model with best performance on the simulation dataset.

4.4. Evaluation Module and real data acquisition

The evaluation module is very important for the whole experiment, because its functions is to judge the quality of the experiment. A good evaluation model can intuitively reflect the performance of the model and provide direction for the improvement of the model. In this Section, firstly the evaluation on simulation dataset is introduced, its results are used to optimize the CNN model for a better performance. Then the evaluation on real image is involved, the results of it are used to analyze the effects of different rendering conditions on the prediction accuracy. Finally, the metrics used for evaluation are mentioned.

4.4.1. Evaluation in simulation data

According to the previous introduction, the model training is based on simulation dataset, and no real dataset will be added. In the training process, we need to evaluate the performance of each model by analyzing the prediction accuracy on the sim-dataset.

In order to evaluate the model, the simulation dataset is divided into 3 parts, namely training-set, validation-set and test-set, where training-set and validation-set are used to train the model, so that the CNN model learns from the data to predict the 3D pose of the target object. Once the training process complete, test-set will be used to evaluate the performance of the trained model. The 3 dataset are randomly divided from the original dataset according to the ratio of 0.6, 0.2, 0.2, the contents of which differ from each other.

4.4.2. Evaluation in real world data

After the CNN model is trained on simulation dataset, the performance of the model will be verified under the real dataset. The goal of this step is to compare the performance of each model trained under different rendering conditions, and achieve better prediction accuracy in the real world.

Through the previous work, we got sim dataset with variable rendering conditions, according to these conditions the dataset are divided into different classes. The model are trained by the dataset under these classes, that means from each class we got a evaluation result, and from these results we could analyze, under which rendering condition a better prediction accuracy can be achieved, and we select the best rendering condition to implement the further experiment.

4.4.3. Metrics for evaluation

There are different metrics for evaluating models. Generally, we divide metrics into two categories for different problems, regression metrics and classification metrics. For classification problems the popular metrics are:

- Categorical Accuracy
- Top k Categorical Accuracy

For regression problems, these metrics are used in general:

- **Mean Absolute Error (MAE or mae)**

  Mean Absolute Error is the average of the difference between the Original Values and the Predicted Values. It gives us the measure of how far the predictions were from the actual output. However, they don't gives us any idea of the direction of the error i.e. whether we are under predicting the data or over predicting the data. Mathematically, it is represented as :

$$MAE = \frac{1}{n}\sum_{i=1}^{n} \mid \hat{Y}_i - Y_i \mid \tag{4.2}$$

- **Mean Squared Error (MSE or mse)**

  Mean Squared Error(MSE) is quite similar to Mean Absolute Error, the only difference being that MSE takes the average of the square of the difference between the original values and the predicted values. The advantage of MSE being that it is easier to compute the gradient, whereas Mean Absolute Error requires complicated

linear programming tools to compute the gradient. As, we take square of the error, the effect of larger errors become more pronounced then smaller error, hence the model can now focus more on the larger errors.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{4.3}$$

In our experiments, the prediction of the 3D pose of an object can be understood as a regression problem, so we can use MSE and MAE as metrics to evaluate the results. In addition, we use the error distribution to analyze the prediction results.

Error distribution We use the error distribution to visually reflect the distribution of position error and rotation error, and their relationship. In the distribution diagram shown in figure 4, the x-axis represents the distribution of position error in different length intervals, the y-axis represents the distribution of rotation errors in different angular intervals, and the z-axis represents the number of samples.

4.4.4. Real data acquisition

In the real world, the camera is fixed at the end-effector of the robot. We set a initial pose of the robot, after each grab, the robot will return to the initial pose, which ensure that images are taken from roughly the same position and view angle each time. That means the camera position remains constant across all images. Since we have added camera randomization to the simulation environment, the camera installation error can be learned by the CNN model, so additional camera calibration is not required. Moreover we did not control for lighting conditions or the rest of the scene around the table (e.g., all images contain part of the robot and tape and wires on the floor).

We acquire the real data with the help of robot. The principle of this method is to obtain the pose information of the object by controlling the robot to grasp and place the object of interest, which is shown in figure 4.12.

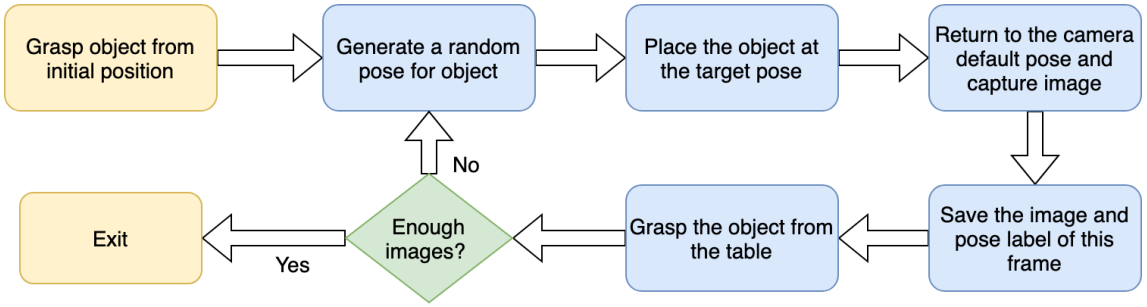

Figure 4.14.: The flowchart of image acquisition process. The program is executed in a loop, each time after the image is acquired, a new pose is generated for the next step.

First, the robot controls the gripper to grab the object at a initial position. After the object is successfully grabbed, the robot places the object at a target position, which is randomly generated within the desktop range. Then robot the end-effector returns to the default position and camera shoot a image. The position information corresponds to the image

will also be recorded. Next, the robot will grab the target object again, and place the object in a new position. These steps will be repeated until the required amount of data is reached. Figure 4.13 shows the robot gestures during the data acquisition process.

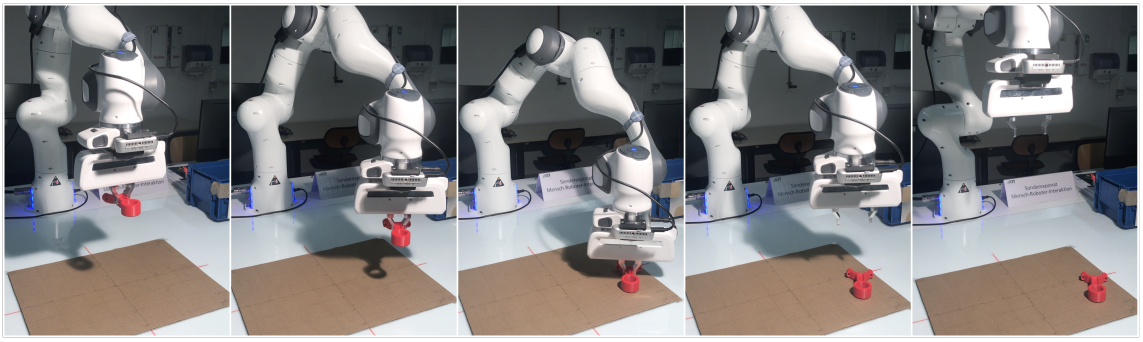

Figure 4.15.: The process of image acquisition through the robot. The robot places the object at the target location and then returns to the camera's default position to get the image.

In the whole process, only the number of required images need to be input, and the data acquisition can be realized automatically, which simplifies the process of data collection.

5. Implementation

In this Chapter the details of the experimental process will be specifically introduced. According to the pipeline introduced in the previous section, the experiment is mainly divided into 3 parts, the model optimization, rendering optimization and method optimization by using multiple camera as input data. Section 5.1 specifies the process and result of model optimization, which is based on simulation data implemented. Section 5.2 describes the process, that we train the models with images rendered by different simulation conditions, and verify the performance of the models in real world, which aims to analyze the impact of different rendering conditions to get a best rendering environment. The section 5.3 aims to improve the recognition accuracy by using further optimization method, here the multi-camera system are used in our experiment to explore whether the recognition accuracy can be improved by increase the information inputs.

5.1. Scene Setup

Building the experiment scene is the first step of our work. Aiming to train the detector for pose estimation of object in simulation environment, we first need to construct the scene in real world, which is the basis of simulation.

5.1.1. Scene in Real World

The experiment scene for pose estimation is set on the workbench of *Franka Emika Panda*[1]. Within a thing pad on the table the objects are located, which are expected to be detected in the experiment. Camera is placed at the end effector of the robot. A initial distance of 50 cm is set from the camera to table. Each time the images will be captured from the initial position. We did not control for lighting conditions or the rest of the scene around the table. The scene built is shown in figure5:

[1] https://www.franka.de/panda/

5.1.2. Scene in Simulation

After the scene is built in real world, we create the corresponding models in Unity3D. The architectural relationship will be rebuild in simulation environment. Then we add ambient light into the scene to simulate realistic lighting conditions. The required function modules are also added into the scene for rendering different images. The simulated scene is shown in figure 5

5.2. Model Optimization

The models used in the experiments were modified VGG-16, ResNet-50 and Inception-v3 networks, which are described in Chapter 4. The main purpose of this part of experiment is to compare the performance of these three base models and then modify the structure and hyperparameters of the model with best performance on our task. The optimized model can be used in further experiment.

5.2.1. Base Model

We chose VGG-16, ResNet-50 and Inception-v3 as candidate base models for our experiment, because they all have good performance in image process tasks in the field of computer vision. Firstly, we need to select one model with best performance on our 3D pose estimation task from these three. in [5] analyzed the performance of different deep neural network models for practical applications. Figure 5.1 shows Inception-v3 hat the best Top-1 accuracy in these three models, and ResNet-50 performs better than VGG-16. The amount of operations required for a single forward pass of different models are shown in figure 5.2. The parameters of Inception-v3 and ResNet-50 are significantly less than VGG-16, so they consume less computation in training process and have higher efficiency. By comparison, it can be concluded that the performance of the Inception-v3 is better than the other two networks in terms of Top-1 accuracy and training efficiency.

Next, we will verify the performance of the three models on our 3D pose estimation task, and demonstrate whether the results are consistent with [5]. Then the model with best performance will be selected for further experiments. In this experiment, the three models will be set to the same hyperparameters while keeping the structure of the convolutional layers unchanged, and the fully connected layers will be downsized to 256. The table 5.1 shows the summary of the hyperparameters used in this experiment.

Result

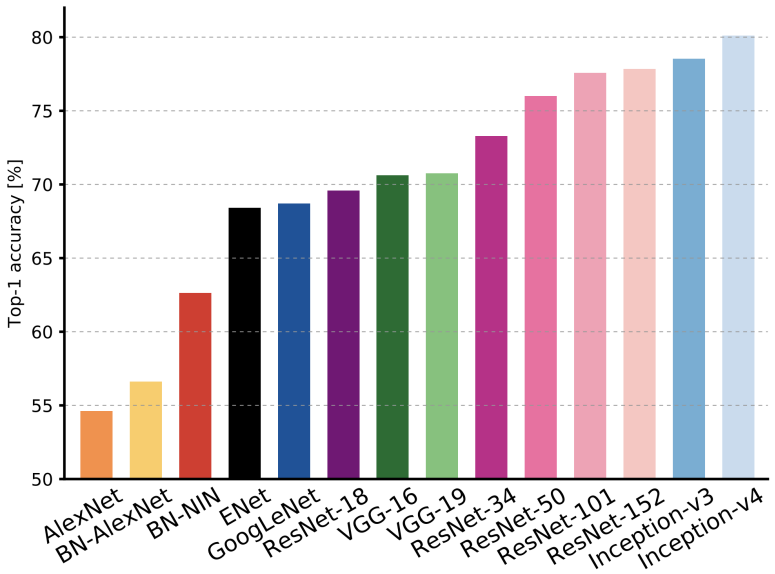

Figure 5.1.: Top1 *vs.* network. Single-crop top-1 validation accuracies for top scoring single-model architectures.[5]

Fully connected layers size	Batch size	Learning rate	Learning rate decay	Epochs
256	32	0.0001	0	60

Table 5.1.: Hyperparameters use to verify performance of VGG-16, ResNet-50 and Inception-v3

In figure 5.4, the validation loss of different modified models can be seen. From the results in the figure we find that the validation loss of the modified Inception-v3 is finally smaller than the other two models, that is, the error of the pose estimation is smaller and the accuracy is higher. This conclusion is consistent with the analysis in [5], so we will use Inception-v3 as the base model for the further experiment.

5.2.2. Hyperparameters of Model Structure

The goal of structure optimization is to improve the accuracy of our detector by modifying the model structure. According to the result from previous section, a modified Inception-v3 is set as the initial structure of our model, which is shown in figure 5.3. In the experiment the standard convolutional layers of Inception-v3 will be used, that means we don't change the structure of these layers. However, we change the size of the fully connected layer and remove Dropout after the average pooling layer.

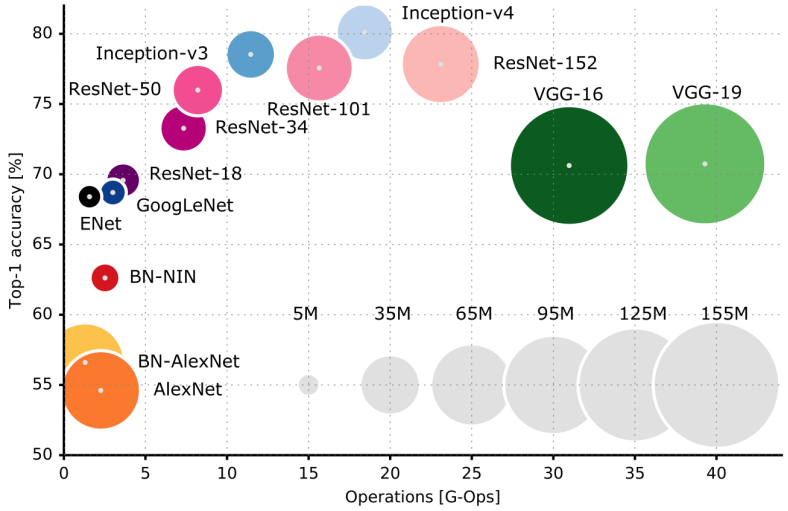

Figure 5.2.: Top1*vs.* operations, size $\propto$ parameters.Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 params. Both these figures share the same y-axis, and the grey dots highlight the centre of the blobs. [5]

Batch size	Learning rate	Learning rate decay	Epochs
32	0.0001	0	60

All the hyperparameters will be set as the same in each modified structure, which are summarized in table 5.2. For the size of fully connected layer, we choose 64, 128, 256, 512 as the candidate parameters, and train the model with different sizes separately. And we analyze the performance of different models on the validation set of simulation data.

Here the validation loss and mean absolute error are used to evaluate the results. Both values represent the error between the predicted pose and the real pose. The mean absolute error describes the average error as mentioned before. However, the loss function is based on $L2loss$ (mean squared error), which is more sensitive to outliers, so we combine both results to judge the performance of the model. That means, the smaller the result of these two values, the better the performance of the model.

Result

After training, we obtain the curve of validation loss and mean absolute error as shown in figure 5.4. The x-axis indicates the training epochs, and the y-axis indicates the corresponding validation loss and MAE.

It can be seen from the curve variation in the figure that the model trained with size of 256 for the fully connected layer performs best. The validation loss and MAE are smaller than those of other size. That means, the model has a smaller average error, and the distribution of errors is more concentrated [2].

5.2.3. Hyperparameters of Training Process

In this section we aim to select the appropriate hyperparameters to optimize the training process. According to the experiments in the previous sections, the modified Inception-v3 was chose for hyperparameter tuning, because it has a better performance for our pose estimation task.

The main hyperparameters of this part include learning rate, learning rate decay, batch size and epochs. We try to select larger batch size in the early stage of the experiment to make full use of the graphics resources and improve the convergence speed. By tuning the learning rate and decay, the training result can converge quickly and avoid local optimal. And a appropriate epoch can prevent underfitting and overfitting.

We perform the experiment by evaluating combinations of different learning rates, decay and batch size. We divide the experiment into two steps. The first step is to select the best performing combination of learning rate and decay under the same batch size setting. Then different batch sizes can be examined based on the selected learning rate and decay to determine the batch size with best performance.

Learning rate and Decay

The types of combinations we designed are shown in table 5.5.

Result
The models are trained based on the combination of parameters above. Figure 5.5 shows the training results.

t can be seen from the figure that the model performs best when the learning rate is set to 0.001 and the decay to 0.001.

[2] Since the hyperparameters are not set to be optima at this time, and the training set is not rendered with the best rendering conditions, both the validation loss and MAE are larger than the final result.

Batch size

The types of combinations we designed are shown in table 5.6.

Result We train the model with different batch sizes and get the results shown in figure 5.4.

When batch size is set 8, the model will diverge and can not converge because the batch size is too small. When the batch size is 16 or 32, the model converges quickly and performs better, which gain the smaller error of the pose estimation. With the further increasing batch size the convergence of the model is slightly worse until 50 and 64. Based on the results above, we set the batch size to 32. In the case of ensuring better convergence of the model, the larger batch size can make full use of the graphics resources and improve the training efficiency.

5.3. Rendering Optimization

In the section we will generate die simulation scenes under different rendering conditions and use these data to train the models separately. The models will be then examined in the real world to evaluate the effects of these different rendering conditions. As mentioned in the previous chapter, we render the scenes with the method of domain randomization. The aspects of domain include:

- Number and shape of distractor objects on the table
- Position and texture of all objects on the table
- Textures of the table, floor, skybox, and robot
- Position, orientation, and field of view of the camera
- Number of lights in the scene
- Position, orientation, and specular characteristics of the lights
- Type and amount of random noise added to image

5.3.1. Pre-analysis

It can be seen that the range of randomized domains here is large, many aspects may affect the performance of the final training result. In order to simplify the research, we selected several aspects of domain in our experiment, which we hypothesize may have a

Num.	Selected aspects of the domain
1	Additional textures on objects
2	Color mode on objects
3	Additional textures on Environment
4	Random noise

Evaluation Type Num.	Color mode on target object	Addition textures on objects	Additional te
1	Full randomly	On	
2	Full randomly	On	
3	Full randomly	Off	
4	Full randomly	Off	
5	Around real color	On	
6	Around real color	On	
7	Around real color	On	

great influence on the training results. The selected aspects of the domain are summarized in table 5:

We generate different rendered scenes with combination of the selected aspects of domain. The rest aspects of the domain are rendered full randomly in each sample, such as the position and number of the lights and the number of distractor objects, etc. Table 5.5 shows the combination of different rendering conditions.

For the target object it can be set whether to perform a full random rendering, or render it around the real color. It can also be set whether to add additional complex textures on the object and to the environment, or they will be rendered in solid color. Here the environment includes the items in the scene except the target object and distractor objects, such as tables, floor and etc. The rendered images can be added with additional noise.

5.3.2. Scene Rendering

Each type of scenes can be rendered in Unity3D. With the functional modules we wrote in Unity3D, the rendering conditions can be controlled to generate the target scenes for the model training. All the types of rendered scenes are shown in figure 5.5:

Each column in the figure corresponds to a target scene. For each scene, the images for two cameras with different perspectives are captured for further experiments. The different rendering conditions between images in figure can be observed. For scene with additional complex texture, the generated images contains more information than those rendered with solid colors. We assume that the scene with texture more capable of simulating the complex real world.

Determined hyperparameter for rendering optimization	
Base model	Inception-v3
Size of fully connected layer	256
Optimizer	Adam
learning rate	0.001
Decay	0.001
Batch size	32
Epochs	60

5.3.3. Model Training

The models will be trained by using the images from different rendered scenes. In the experiment, two types of target object are selected for our task, namely the cube and piston rod. The geometry of cube is relatively simple while the piston rod is more complexly constructed. By comparing the detection accuracy on these two objects, the influence of the complexity of the object shape on the training results. The model of the two objects is shown in figure 5.5:

Through previous work, the model structure and it's hyperparameters are determined for a better training result. In the next experiment these parameters will be used for model training, which are summarized in table 5.5.

5.3.4. Result

The models trained in different rendered scenes will be evaluated in real world. We acquired 300 real data for the pose evaluation of the piston rod, of which in 200 data the distractor objects don't appear and the other 100 data contained the distractor objects with random shape and number. We use the model trained in different scenes to predict the 3D pose of rod piston, and calculate the prediction error. Following results were obtained:

Table II examines the performance of the algorithm under different rendered scenes. By comparing the result of set 1 and set 2 we found that rendering the target object around the real color is beneficial to improve the prediction accuracy. From the results of set 6 and 7, it can be seen that adding additional texture to the target object can also improve the performance of our model, and the model is more robust to the variations of scenes. The accuracy of the prediction is guaranteed to a good level when there are distractor objects in real scene. Similarly, adding additional textures to the environment has the same effect, the model can detect and estimate the pose of target object more accurately. Adding additional random noise did not improve the performance of the model in real world. We consider that by rendering complex lighting conditions and adding background

Detection error for Rod on real images						
Evaluation Type Num.	Target object only			Random noise		
	Position error	Rotation error	Total error	Position error	Rotation error	
1						
2						
3						
4						
5						
6						
7						

Detection error from best rendered scene			
Evaluation types	**Position error**	**Rotation error**	**Total error**
Piston rod			
Cube			

textures, the complexity of the image information is high enough. Various noises are already included, so adding additional noise does not bring a significant improvement for the detection.

Table 1 compares the detection accuracy of models for the cube and piston rod. The results show that the model performs well for the object with complex geometries.

Based on the results, we suggest to add additional complex textures to the objects and background environment of the rendered scene while other aspects of the domain are randomly rendered. When the color feature of the target object is known, the object should be rendered around the real color as exact as possible. It is not necessary to add additional random noises to the rendered scene.

5.3.5. Further Optimization

In this section we aim to further improve the detection accuracy with feasible optimization methods. Firstly, we try to initialize the parameters with pre-trained weights and evaluate the results. Then the improvement of generalization performance by increasing the amount of training data will be discussed.

Pre-trained Weights

Generally, initializing the parameters by loading pre-trained weights before the model training gains a faster convergence and better results with limited amount of training data set. This is also known as transfer learning. Here the performance of the model initialized by two methods will be compared. One model will be initialized by weights obtained by pretraining on ImageNet, the other one by random weights.

Figure 5.13 provides the results of the models trained by these two methods. The model initialized with pre-trained weights has a faster convergence speed and better performance, which gains the smaller detection error.

Training with more data

The neural network model learns the extracts key information from the input data, and then generalizes them to other similar scenes of the same problem. That means the model is able to process the new data based on the obtained knowledge. Generally, using larger training set tends to the better results, while the larger data set contains more information and covers more possible scenes, from which the model learns more.

Therefore, we increase the size of the training set to make it contain more variations of scene, so that it is able to cover the real scenes as much as possible. In the experiment 4 different types of size are used and the results of detection error are presented in figure 5.5.

5.4. Multiple Camera system

This Section describes the experiment for improving the pose estimation accuracy by using dual camera viewpoints. Firstly, the hardware layout of the dual-camera system in real world is introduced. Then the structure design of the deep neural network model and the optimization process are presented. With comparing the training results, the performance of different models will be evaluated.

5.4.1. Scene Modification

The dual camera system will be extended based on the scene of single camera system. In previous setup, a single camera was placed at the end effector of the robot. In order to

increase the input information, the second camera should be placed at a different viewpoint. The information from both viewpoints can be fused together as an input to the neural network.

Based on the design principle above, we fixed the second camera at the midpoint of the edge of table, to align it with the first camera in th y-axis direction of the robot coordinate system. It is set to the same height as the first camera and pointed to the center of the table by adjusting the view angle.

Figure5 shows the layout of the dual camera system in real world and the images captured from different viewpoints.

5.4.2. Model Structure

Since a new viewpoint is added into the system, the structure of corresponding deep neural network model is required to be modified. The main idea is to fuse the information of two networks based on images from the two viewpoints: The images from two viewpoints are input into two Inception-v3 networks, and then the output from convolutional layer of the two networks are concatenated, which pass through fully connected layers and give the final output to predict the pose of object. Following three networks are designed:

In the first model, the two outputs of Inception-v3 pass through an global average pooling(GAP) layer and then concatenated together. The two fully connected layers are downsized to 256, 64.

In the second model, the output of convolutional layer are directly concatenated, and then connected to the global average pooling layer.

In the third model, an additional fully connected layer is added after the concatenated convolutional layer from two networks.

In the three models ReLU nonlinearities are used throughout, and the output of them are the 3D pose of the target object.

Determined hyperparameter for rendering optimization	
Base model	Inception-v3
Size of fully connected layer	256
Optimizer	Adam
learning rate	0.001
Decay	0.001
Batch size	32
Epochs	60

5.4.3. Model Training

For training our detector, we generate the scene by using the rendering conditions with best performance. And we also use the best hyperparameter setting obtained from previous works for the training process. The parameters are summarized in the table.

Before our training, we use the pre-trained weights to initialize the parameters of convolutional layers and use 100K dataset. The training results are shown in figure5.

Based on the training result, we found that model 2 hast the fastest convergence and smaller prediction error. Comparing the best result of single camera, the validation loss and mean absolute error of model 2 are smaller. It performs better on the simulation dataset.

5.4.4. Evaluation in real world

As achieving the progressive training results in the simulation, we need to evaluate whether the detector is also robust in the real world. We use 250 data for evaluation, 200 of which contain only the target object, and in the other 50 data the distractor objects appear. The result is shown in table 5:

As shown in the results, the dual-camera system achieves a improvement for pose estimation accuracy in the both scenes with only target object and with distractor objects.

5.5. Robotic Experiments

To demonstrate the performance of our detector trained in simulation, we evaluated the use of our object detection networks for localizing an object in clutter and performing a prescribed grasp by using the off-the-shelf motion planning software *Moveit*[51].

Detection error for Rod on real images						
Evaluation Type Num.	Target object only			Random noise		
	Position error	Rotation error	Total error	Position error	Rotation error	
1						
2						
3						
4						
5						
6						
7						

Results of robotic experiments			
Color of piston rod	Total trials	Successful trials	Failed trials
Red	50		
Gray	50		

5.5.1. Experiment Setup

We perform the experiment with single camera system. For each grasp, the robot will capture a image from the initial position and obtain a prediction by detector for the 3d pose of the target object. Then the robot grasps the object and returns to the initial position.

Detector In order to ensure the grasping performance, we used the most consistently accurate detector for the experiment.

Grasping Object We used the piston rod as the actual target object in our experiment. In order to test whether the detector is specifically generalized, we prepared two different color objects, red and gray.

Scene Setup To test the robustness of our method to discrepancies in object distributions between training and test time, some of our test images contain distractors placed at orientations not seen during training (e.g., a hexagonal prism placed on its side).

5.5.2. Results

We performed 50 grasping experiments on the piston rod for each color. The results are as follows:

From the results, we found that it was able to successfully detect and pick up the gray piston rod in 48 out of 50 trials, which is included in highly cluttered scenes. Note that the trained detector have no prior information about the color of this gray piston rod, only its shape and size. The results shows that our detector is robust from simulation to real world, and it is able to ignore the previously unseen distractors and detect the target object placed closely to other objects of the same color.

Figure 6 shows examples of the robot grasping trials.


Missing
figure

6. Results

In the last chapter, we will present conclusions that we have drawn from the experiment results. Then some limitations in the proposed pipeline will be discussed and potential future works will be introduced.

6.1. Conclusion

In this thesis, we trained the detector for estimating the object pose in the simulation environment through transfer learning. By optimizing the network model and simulation environment, we achieved the satisfied accuracy and robustness in real world. And through dual-camera system the accuracy of our detector is further improved. Finally we evaluated the performance of our detector by performing robot grasping experiment.

We built a simulation environment with several function modules based on Unity3D for robot training. In each module, we introduced its functionality and explained related algorithms and the frameworks we referred to, followed by some implementation details with the APIs in the module. We studied the performance of different neural network models for our task and optimized the model structure and hyperparameters. Next, we analyzed the effects of different rendering conditions on object estimation accuracy and robustness from sim to real, the optimized the detector from the results. The we further improved the accuracy of the detector by using dual-camera system. Finally, the robot experiment was performed to demonstrate the potential of the detector trained in simulation and transferred to the real world.

The experiment results prove that our detector can identify and estimate the object pose from its shape and size. It is also able to ignore the previous unseen distractors from the real world in highly cluttered scenes, which shows the good robustness.

6.2. Future work

Future directions mainly contains:

- In this thesis, the current work focus on pose estimation for single object, for further research it can be extended to multiple objects via transfer learning.
- The accuracy of object detectors can be improved by other approaches, such as using higher resolution camera frames, or incorporating depth information.
- Robot control and trajectory planning can be integrated in this pipeline. As our key point is mainly focus on the perception problem, the training task on obstacle avoidance and trajectory planning can also be implemented via transfer learning based on the physical engine in Unity3D. And a complete End-to-End pipeline can be established.

Bibliography

[1] Christoph Bartneck, Marius Soucy, Kevin Fleuret, and Eduardo B Sandoval. The robot engine—making the unity 3d game engine work for hri. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 431–437. IEEE, 2015.

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[3] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[4] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[5] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[6] Adam Coates and Andrew Y Ng. Multi-camera object detection for robotics. In *2010 IEEE International Conference on Robotics and Automation*, pages 412–419. IEEE, 2010.

[7] Alvaro Collet and Siddhartha S Srinivasa. Efficient multi-view object recognition and full pose estimation. In *2010 IEEE International Conference on Robotics and Automation*, pages 2050–2055. IEEE, 2010.

[8] Alvaro Collet, Dmitry Berenson, Siddhartha S Srinivasa, and Dave Ferguson. Object recognition and full pose registration from a single image for robotic manipulation. In *2009 IEEE International Conference on Robotics and Automation*, pages 48–55. IEEE, 2009.

[9] Alvaro Collet, Manuel Martinez, and Siddhartha S Srinivasa. The moped framework: Object recognition and pose estimation for manipulation. *The International Journal of Robotics Research*, 30(10):1284–1306, 2011.

[10] Mark Cutler and Jonathan P How. Efficient reinforcement learning for robots using informative simulated priors. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2605–2612. IEEE, 2015.

[11] Staffan Ekvall, Danica Kragic, and Frank Hoffmann. Object recognition and pose estimation using color cooccurrence histograms and geometric modeling. *Image and Vision Computing*, 23(11):943–955, 2005.

[12] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

[13] Ali Ghadirzadeh, Atsuto Maki, Danica Kragic, and Mårten Björkman. Deep predictive policy training using reinforcement learning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2351–2358. IEEE, 2017.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. book in preparation for mit press. *URL¡ http://www. deeplearningbook. org*, pages 204–210, 2016.

[15] Iryna Gordon and David G Lowe. What and where: 3d object recognition with accurate pose. In *Toward category-level object recognition*, pages 67–82. Springer, 2006.

[16] Abhishek Gupta, Coline Devin, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Learning invariant feature spaces to transfer skills with reinforcement learning. *arXiv preprint arXiv:1703.02949*, 2017.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[18] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[19] Judy Hoffman, Erik Rodner, Jeff Donahue, Trevor Darrell, and Kate Saenko. Efficient learning of domain-invariant image representations. *arXiv preprint arXiv:1301.3224*, 2013.

[20] Judy Hoffman, Sergio Guadarrama, Eric S Tzeng, Ronghang Hu, Jeff Donahue, Ross Girshick, Trevor Darrell, and Kate Saenko. Lsda: Large scale detection through adaptation. In *Advances in Neural Information Processing Systems*, pages 3536–3544, 2014.

[21] Stephen James and Edward Johns. 3d simulation for robot arm control with deep q-learning. *arXiv preprint arXiv:1609.03759*, 2016.

[22] J Zico Kolter and Andrew Y Ng. Learning omnidirectional path following using dimensionality reduction. In *Robotics: Science and Systems*, 2007.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[24] Yann LeCun. Yoshua bengio, and geoffrey hinton. *Deep learning. nature*, 521 (7553):436–444, 2015.

[25] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521 (7553):436, 2015.

[27] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1): 1334–1373, 2016.

[28] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. Cs231n: Convolutional neural networks for visual recognition. *University Lecture*, 2015.

[29] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[30] Yanghao Li, Naiyan Wang, Jianping Shi, Jiaying Liu, and Xiaodi Hou. Revisiting batch normalization for practical domain adaptation. *arXiv preprint arXiv:1603.04779*, 2016.

[31] Lyle N Long and Ankur Gupta. Scalable massively parallel artificial neural networks. *Journal of Aerospace Computing, Information, and Communication*, 5(1): 3–15, 2008.

[32] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael I Jordan. Learning transferable features with deep adaptation networks. *arXiv preprint arXiv:1502.02791*, 2015.

[33] David Marr and Ellen Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 207(1167):187–217, 1980.

[34] Chaitanya Mitash, Kostas E Bekris, and Abdeslam Boularias. A self-supervised learning system for object detection using physics simulation and multi-view pose estimation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 545–551. IEEE, 2017.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[36] Jorge J Moré and David J Thuente. Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software (TOMS)*, 20(3):286–307, 1994.

[37] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[38] Benjamin Planche, Ziyan Wu, Kai Ma, Shanhui Sun, Stefan Kluckner, Oliver Lehmann, Terrence Chen, Andreas Hutter, Sergey Zakharov, Harald Kosch, et al. Depthsynth: Real-time realistic synthetic data generation from cad models for 2.5 d recognition. In *2017 International Conference on 3D Vision (3DV)*, pages 1–10. IEEE, 2017.

[39] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[40] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision*, pages 102–118. Springer, 2016.

[41] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[42] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[43] Andrei A Rusu, Mel Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. *arXiv preprint arXiv:1610.04286*, 2016.

[44] Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[45] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[46] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[48] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[49] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[50] Hao Su, Charles R Qi, Yangyan Li, and Leonidas J Guibas. Render for cnn: Viewpoint estimation in images using cnns trained with rendered 3d model views. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2686–2694, 2015.

[51] Ioan A Sucan and Sachin Chitta. Moveit! *Online at http://moveit. ros. org*, 2013.

[52] Baochen Sun and Kate Saenko. From virtual to reality: Fast adaptation of virtual object detectors to real domains. In *BMVC*, volume 1, page 3, 2014.

[53] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural networks for object detection. In *Advances in neural information processing systems*, pages 2553–2561, 2013.

[54] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[55] Yaniv Taigman, Adam Polyak, and Lior Wolf. Unsupervised cross-domain image generation. *arXiv preprint arXiv:1611.02200*, 2016.

[56] Jie Tang, Stephen Miller, Arjun Singh, and Pieter Abbeel. A textured object recognition pipeline for color and depth image data. In *2012 IEEE International Conference on Robotics and Automation*, pages 3467–3474. IEEE, 2012.

[57] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017.

[58] Eric Tzeng, Judy Hoffman, Ning Zhang, Kate Saenko, and Trevor Darrell. Deep domain confusion: Maximizing for domain invariance. *arXiv preprint arXiv:1412.3474*, 2014.

[59] Eric Tzeng, Coline Devin, Judy Hoffman, Chelsea Finn, Pieter Abbeel, Sergey Levine, Kate Saenko, and Trevor Darrell. Adapting deep visuomotor representations with weak pairwise constraints. *arXiv preprint arXiv:1511.07111*, 2015.

[60] Bin Yang, Junjie Yan, Zhen Lei, and Stan Z Li. Craft objects from images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6043–6051, 2016.

[61] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[62] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22, 2000.

Appendix

A. First Appendix Section

List of Figures

List of Tables

Listings

List of Algorithms

```
@mastersthesis{Zhuoyi Han_31. April 2019,
	author = {Zhuoyi Han},
	editor = {M.Sc. Yongzhou Zhang, },
	ipr-thesis = Master Thesis,
	keywords = {Keywords, of, my, Thesis},
	location = {Karlsruhe, Germany},
	month = ,
	pages = ,
	school = {Karlsruhe Institute of Technology},
	title = {Pose Estimation for Industrial Robots
via Transfer Learning from Sim to Real},
	year = {31. April 2019}
}
```