Imperial College London

MEng Individual Project

Imperial College London
Department of Computing

Evaluating Transfer Learning Methods for Robot Control Policies

Author:
Rad Ploshtakov

Supervisor:
Dr. Edward Johns

Second Marker:
Dr. Stefan Leutenegger

Date: June 19, 2017

Abstract

Robot control has traditionally been a multidisciplinary problem involving computer vision, planning, kinematics and more. Such systems are comprised of complex stages in a pipeline meant to perceive information via sensors from the outside world, process it in order to extract features and use said features when planning and executing future actions. Each stage presents its own unique challenges and requires different areas of expertise from teams working on the project.

An end-to-end deep learning system for robot control promises to remove the need for these multi-stage pipelines and reduce the overall expertise needed to build and maintain them.

Reinforcement and supervised learning are two different approaches to building such a system. No matter which approach, deep learning's need for large datasets makes it impractical to employ for all but the most well-funded teams which can afford to task multiple robots with learning for a an extended period.

Transfer learning can help tackle this challenge by using information from a domain where data is plentiful and a small sample from a related domain for which data is scarce. In the case of robot control this is usually a case of transferring between a simulation, where samples are quick and easy to generate, and the real world.

This project has two goals. It compares and contrasts reinforcement and supervised learning systems and evaluates three transfer learning techniques in a simple two dimensional environment. The project finds that within the 2D environment a simple reinforcement model performs a better than a more complex supervised one and that Progressive networks are the most powerful of the three transfer learning methods.

Acknowledgements

My thanks to:

- My supervisor Ed for his optimism and help when I needed it
- Slavyan for being a great friend and the nicest person I know
- My family for their endless love and support and especially my mother, Radoslava, for always giving me the extra push I needed. I can't ever repay you.

Contents

Chapter 1

Introduction

1.1 Motivation - the importance of transfer learning

Deep learning techniques produce ever improving results and find application in an always greater set of domains. A major limiting factor in their use to solve problems is the amount of data available to learn from and the amount of time it takes. A lot of today's most powerful models have millions of parameters and require millions of samples to train.

Transfer learning, at its core, is about transferring knowledge gained in domain A to a separate, but similar, domain B. And while the domains are similar, data might be more readily available in domain A than domain B. For example, it is not hard to find countless images of dogs on the internet. Which is great if you want to train a dog classifier. But if you need to train a wolverine (the animal) classifier you will notice that a comparatively limited set of images is available. Yet it is undeniable that both dogs and wolverines have a lot in common. So in theory we should be able to train a neural network to recognize features of dogs and then transfer that knowledge by finding the commonality between dogs and wolverines.

Figure 1.1: Both dogs and wolverine have similar but different body structure

Simulation technology has been steadily improving up to the point that simulated environments today are very close in fidelity to the real world. They are not indistinguishable yet and it is an exponential climb in computational resources to reach parity with reality. However, they are close enough for transfer learning to bridge the gap such that models can be trained on data provided by the simulation and then 'transferred' for application in the real-world.

By training a model on data provided by a simulation and then using transfer to bridge the gap a new set of problems become available for application of deep learning. Domains where data is scarce like training an image classifier without a large, labelled dataset of what you are trying to classify. Or domains where training would be dangerous. You absolutely do not want to use reinforcement learning to train a wildly swinging robot arm or a self-driving car. Transfer learning makes it possible to do so by training in simulation first and then applying it to the real world.

1.2 The project

This project aims to explore three transfer learning techniques when applied to a simple control problem in order to gain insight into their applicability and evaluate their performance. The techniques are presented in the following papers:

- Progressive Neural Networks[1] by Andrei A. Rusu et al, Google DeepMind
- Adapting Deep Visuomotor Representations with Weak Pairwise Constraints[2] by Eric Tzeng et al, University of California Berkeley and Boston University
- Beyond Sharing Weights for Deep Domain Adaptation[3] by Artem Rozantsev, Mathieu Salzmann and Pascal Fuam, EPFL

A second objective is to observe and compare the performance of reinforcement models learning on their own and supervised models learning optimal actions obtained via inverse kinematics. The networks are evaluated in terms of training behaviour and accuracy. The two approaches are compared on their own within one basic domain and in conjunction with a transfer learning techniques in a number of different domains.

The project is structured around training models to control a simple robot arm in a 2D game. After a model has learned an effective policy for the game we modify the environment it operates in by introducing random noise and changing colours. We do so until the models stop performing effectively in the environment. We then apply the transfer learning techniques and train within the new domains. After training we evaluate the models and compare the results between them.

1.3 Results

The results of this project can be summarised as such:

- The reinforcement learning system is simpler yet perform better than the supervised one
- With enough complexity and training data the supervised system could approach optimal performance, outpacing reinforcement learning at a significantly higher training cost
- Progressive neural networks when applied to reinforcement learning adapt remarkably well to a number of different domains
- The remaining techniques only produce improved performance in a single domain but out of those tested it is the most important one for robot control

Chapter 2

Background

2.1 Artificial neural networks

Artificial neural networks (ANNs) are computational models inspired by biological brains. A neural network is comprised of many individual neurons typically arranged in a directed graph with multiple layers where a neuron in layer i has an incoming edge from some or all neurons in layer $i-1$ and an outgoing edge to all neurons in layer $i+1$. A typical network has at least three layers - an input layer, an output layer and a so-called hidden layer between them. The weighted edges of the graph allow the state of one neuron to affect the state of a connected neuron - usually expressed as an activation function. This activation function takes as input the value of all neurons connected to the target neuron and outputs the value of the target neuron. For example, an identity activation function $f(x) = x$ is the most commonly used function to connect neurons between layers:

$$x_j = \sum_{i}^{N} x_i w_i + b \tag{2.1}$$

The value of neuron x_j in layer j can be expressed as a sum of the values of all neurons x_i in layer $j-1$ weighted by their individual connections to x_j and a bias term.

2.1.1 Backpropagation

The network's can learn via backpropagation. This is the process of updating the weights of connections between neurons proportionally to the gradient of the network's loss function. A loss functions encodes the misalignment of the network's desired outputs and its actual outputs given identical inputs. Obtaining a network's output is called a forward pass. It is performed by providing values to its input layer. The values then propagate forward layer by layer, obeying the activation functions until the forward propagation stops at the output layer. The loss function is then applied between the values of the output neurons and their desired values. If the desired values are provided by the network's creator the type of learning is called supervised learning.

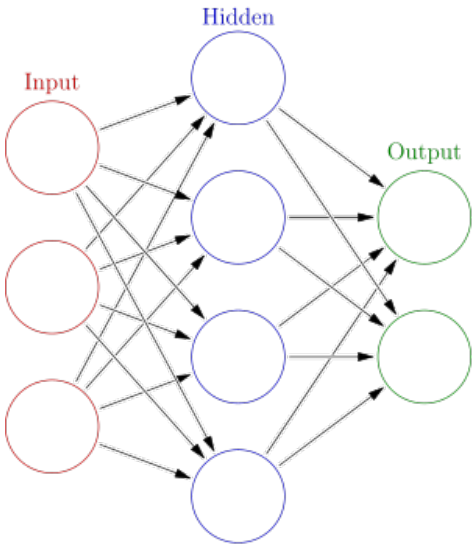


Figure 2.1: An artificial neural network[4]

The output of the loss function is called the error term E. The goal of the network is to minimize E by way of backpropagation. We can express the value of output neurons as a combination of the provided inputs and the weight at each layer. Therefore, we can compute the derivative of E with respect to every weight in the network using the chain rule:

$$\frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta o_j} \frac{\delta o_j}{\delta net_j} \frac{\delta net_j}{\delta w_{ij}} \tag{2.2}$$

Where E is the error, o_j is the output of neuron j, net_j is the input coming in from all neurons to o_j and w_{ij} is the weight between neurons o_j and o_i. This derivative can be applied in a chain to all connected layers. We can therefore express the derivative of E with respect to every parameter of the network. These derivatives are the reason why the activation functions of neurons must be differentiable.

Once we have the derivative of the error term with respect to a weight, we can use it to adjust the weight.

Gradient descent

A common and straightforward way to do that adjustment is Gradient descent. If w is the value of the weight at time t and E is our error function. Then gradient descent will update the value of w to:

$$w_{t+1} = w_t - \eta \frac{\delta E}{\delta w_t} \tag{2.3}$$

Where η is the learning rate parameter which controls the speed of learning. Gradient descent calculates the error term with respect to the entire dataset. This can cause issues when datasets become too large to fit in memory.

Stochastic gradient descent

Stochastic gradient descent (SGD) - also known as "on-line" gradient descent - is similar to ordinary gradient descent. SGD is used when the sample set has to be split in a number of batches. The updates to weights which SGD performs after every batch are therefore approximations of the true updates which would be calculated over the entire set by GD.

Adam

Adam[5] is a first-order stochastic gradient-based optimisation algorithm. Adam computes separate adaptive learning rates for all parameters from estimates of the mean and variance of the gradients. Adam builds on previous work introduced in RMSProp[6].

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$: Initialize mean vector
$v_0 \leftarrow 0$: Initialize variance vector
$t \leftarrow 0$: Initialize timestep
while θ_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \delta_\theta f_t(\theta_{t-1})$ - Get the gradients
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ - Update biased mean estimate
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ - Update biased variance estimate
  $\hat{m^t} \leftarrow m_t / (1 - \beta_1^t)$ - Compute bias-corrected mean estimate
  $\hat{v^t} \leftarrow v_t / (1 - \beta_2^t)$ - Compute bias-corrected variance estimate
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ - Update parameters
end while
return θ_t

Figure 2.2: Pseudo code for Adam presented introduced in the paper[5]. α, β_1, β_2 and ϵ are all hyperparameters provided at start time.

The algorithm enjoys several benefits. The adaptive learning rate makes it easier to avoid oscillation while looking for an optimum. It works well with on-line learning thanks to the estimates which is useful in scenarios with many mini-batches such as the reinforcement setting described further in this report. Another benefit to Adam is that it is widely available in existing deep-learning frameworks.

2.1.2 Activation functions

The identity activation function $f(x) = x$ is useful when connecting neurons. However, most relationships in nature are not linear which is why other activation functions are used in order to introduce non-linearity to the system.

Softmax

The Softmax function squashes down a vector's values such that they sum to 1 which makes it useful as a representation of probability.

$$\sigma(x)_j = -\frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for j = 1,...,K} \tag{2.4}$$

Usually applied to the output layer of a classifier network, where every element of the vector is the probability of the given class.

Rectified linear units

Rectified linear units (ReLu) are another tool by which non-linearity can be introduced in a system:

$$f(x) = \max(0, x) \tag{2.5}$$

ReLu is usually applied after convolutional layers. ReLu is easier to compute than than functions like $tanh$ or $sigmoid$ because of its simplicity which leads to increased performance when training.

2.1.3 Loss functions

Loss functions determine how we measure the 'gap' between our network's desired output and its actual output. Within this project we use two types of functions: Mean squared error and Cross entropy

Mean squared error

Mean squared error(MSE) is the simplest loss function:

$$E = \frac{1}{2N} \sum_{i=1}^{N} \|y(x_i) - y'(x_i))\|^2 \tag{2.6}$$

Where n is the number of samples, x is an individual sample, $y(x)$ is the predicted network output and $y'(x)$ is the desired output. Dividing the mean by 2 is done for convenience when calculating the gradient of E.

Cross entropy

Networks which utilise Cross entropy provide their outputs as a vector of probabilities summing up to one via the use of a softmax activation function.

$$H(p, q) = -\sum_i p_i \log q_i \tag{2.7}$$

$$E = -\frac{1}{N} \sum_{i=1}^{N} H(y(x_i), y'(x_i)) \tag{2.8}$$

Where $H(p,q)$ is the cross entropy between vectors p and q. p is the correct probability vector, and q is the predicted distribution by the current model. In the case of an object belonging to only one class p is one hot i.e. all of its elements are 0 except for the one at the corresponding class index which is 1.

2.2 Deep neural networks

A deep neural network has similar structure to a regular neural network. The difference comes primarily from the increased number of interconnected hidden layers placed between the input and output layers - Figure 2.3. The increased number of hidden layers allows a deep network to take advantage of the hierarchical structure of many problems. Each hidden layer encodes higher level features composed from the relatively lower level features present in the layer before it. For example the first hidden layer of a network which recognizes objects present in an image can learn to recognize a configuration of pixels which together form an edge, the layer after it can take those edges and learn to recognize simple shapes like a square or a triangle and so on. Each layer building on the features of the previous one.

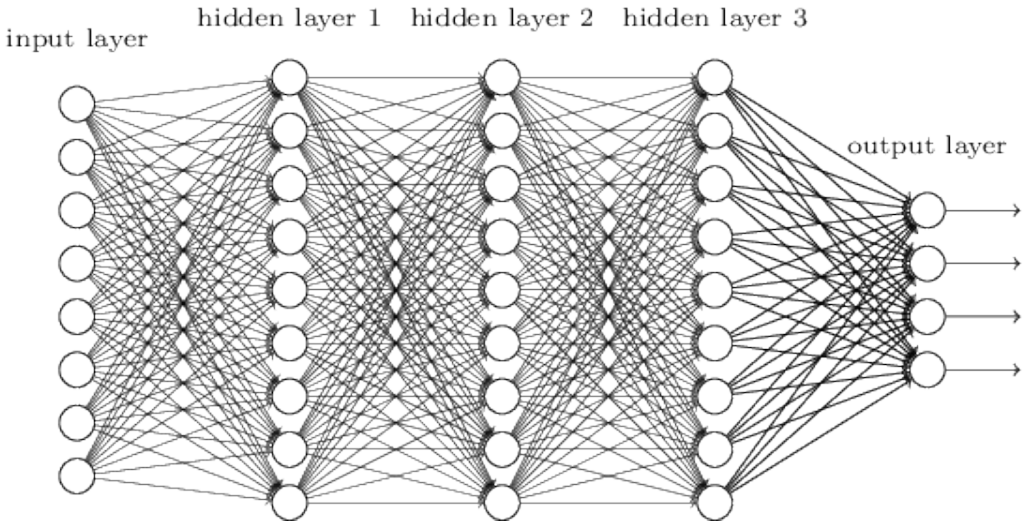


Figure 2.3: A deep neural network[7]

2.2.1 Convolutional neural networks

Convolutional neural networks gained traction in 2012 when they significantly outperformed the competition in the annual NIPS Image recognition competition[8]. These nets are designed to process data which comes in the form of tensors of rank 3. The most well-known application is with 3D arrays containing pixel intensities for the three colours of an image[9].

Typically a convolutional network is structured as a series of stages. The first few stages are composed of two types of layers: convolutional layers with ReLu activation functions and pooling (subsampling) layers - Figure 2.4. Units in a convolutional layer are organized in feature maps, within which each unit is connected to local patches in the feature maps of the previous layer through a set of weights called

a filter. The result of this local weighted sum is then passed through a non-linearity such as a ReLU to remove any negative values.

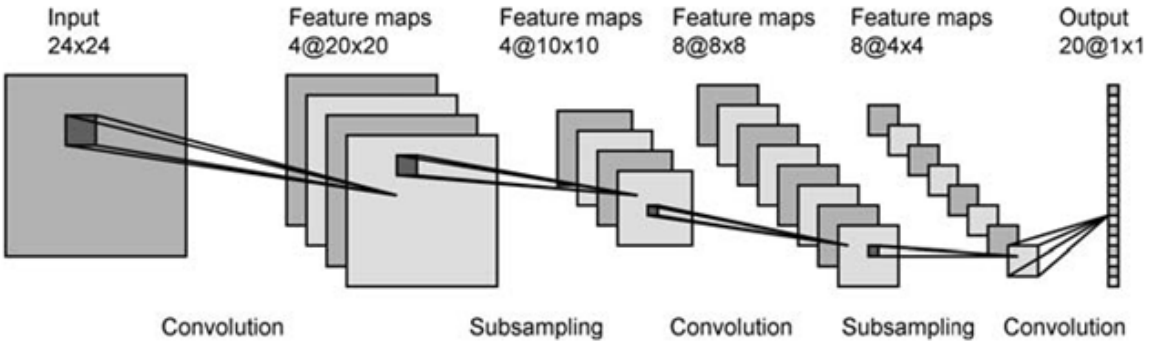


Figure 2.4: A convolutional network[10]

The role of the convolutional layers is to detect local clusters of features from the previous layers while the pooling layers eliminate small perturbations in the locality of the features by treating nearby locations as the same.

Convolution and filters

In the context of images convolution is the process of adding the neighbours of a pixel weighted by a filter (also known as kernel) and outputting the sum. Convolutions have been widely used in image processing and computer vision systems for a long time. Useful for a number of image enhancing techniques such as blurring or sharpening they have only recently made their way to Deep learning.

Filters are the vectors of weights with which the output is convolved. Different filters are able to recognize different features such as horizontal or vertical edges, or particular colours if they are of importance to the structure of the problem. Filters are not manually set but are learned during the training of the network.

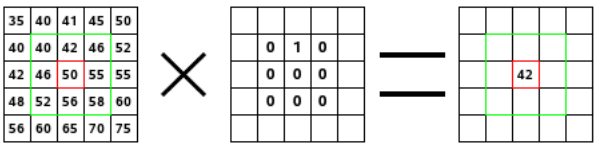


Figure 2.5: A simple 3x3 filter which when applied returns the value of the pixel directly above the target.[11]

Filters are applied in a grid-like manner to an image. Every application's central point is separated by an integer stride. The size of the output feature map depends on the size of the kernel and the length of the stride. E.g a 3x3 filter with a stride of 3 applied on a 30 by 30 image with no padding will produce a 10x10 feature map.

Pooling layer

Pooling layers are sometimes applied after a convolutional layer in order to reduce the size of the representation, reduce the number of parameters and increase the the resistance to small perturbations such as random noise in an image. The most common type of pooling is max pooling which when applied to a grid of pixels picks outputs the one with largest value.

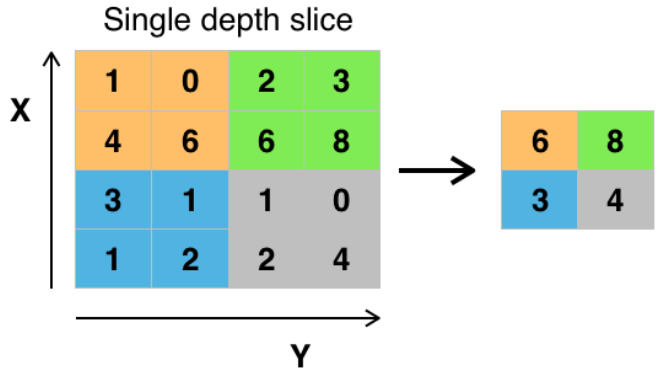


Figure 2.6: Max pooling with a 2x2 filter and a stride of 2[12]

2.3 Reinforcement learning

Reinforcement learning is an area of machine learning concerned with the way an agent learns to interact with its environment in order to achieve a specific goal. A typical reinforcement learning system - Figure 2.7 - will represent its agent and the environment as entities having a specific state with a number of actions available for the agent to perform based on these states. Upon executing an action a the agent and/or the environment change states the agent receives a reward r which can be positive or negative. The method by which an agents chooses action a in a state s is called a policy π.

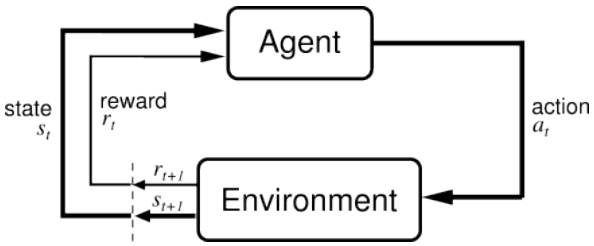


Figure 2.7: A simple reinforcement learning system[13]

Typically agents aim to maximize their reward R_t starting at state t and finishing at some final state T.

$$R_t = r_{t+1} + r_{t+2}...r_T \tag{2.9}$$

Sometimes agents are expected to operate continuously i.e. $T = \infty$. In addition to being computationally out of reach, simple addition of the rewards at

every step will place the same value on rewards which are easily obtainable in a few steps and rewards requiring thousands of actions. In reality, unless the system is completely known and deterministic the farther away a potential reward is the greater the risk that conditions change and prevent the agent from obtaining it. In order to mitigate these factors a discount rate $0 \leq \gamma \leq 1$ is introduced. The total return becomes:

$$R_t = \sum_{i=0}^{N} \gamma^i r_{t+i+1} \tag{2.10}$$

γ discounts rewards farther into the future compared to more immediate rewards. Tweaking the value of γ can transform the agent from an incredibly-short sighted one to one which is vulnerable to changing conditions.

In order to make decisions an agent has to be able to assign a value to being in a state s. The value can be measured in two ways - a state-value or an action-value. The state value for a state s under policy π is the following:

$$V^{\pi}(s) = E_{\pi}(R_t|s_t = s) \tag{2.11}$$

Where $E_{\pi}(R_t|s_t = s)$ is the expected value of following the policy π. The action-value for a state s under policy π evaluates the result of taking action a in state s and then continuing to follow policy π:

$$Q^{\pi}(s,a) = E_{\pi}(R_t|s_t = s, a_t = a) \tag{2.12}$$

Reinforcement learning is searching for an optimal policy π^* such that:

$$V^{\pi^*}(s) \geq V^{\pi}(s) \quad \forall s \tag{2.13}$$

With an optimal policy π^* we can also define an optimal state and action value functions:

$$V^*(s) = max_{\pi} V^{\pi}(s) \tag{2.14}$$
$$Q^*(s,a) = max_{\pi} Q^{\pi}(s,a) \tag{2.15}$$

2.3.1 Q-learning

Q-learning is a way to approximate an optimal policy by approximating Q^*. An experience is defined to be (s,a,r,s') - the agent starts off in state s, performs action a, receives reward r and transitions into state s'. The value of $Q(s,a)$ is then updates:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r + \gamma \max_a Q(s_{t+1},a_t) - Q(s_t,a_t)] \tag{2.16}$$

This approach effectively learns by observing a number of episodes and converging to an optimal solution.

2.3.2 Deep Q network

An important milestone in the area of reinforcement learning is DeepMinds' Deep Q network (DQN)[14]. A combination of a deep convolutional neural network and Q-learning, DQNs learned to play a number of Atari games at a better-than-human level while receiving as input only the images from the game and the current score.

There are two key ideas behind DQN. First they approximate the Q function by way of a convolutional neural network. The network is trained to take in a preprocessed image $\phi(x)$ and output the predicted future reward for all possible actions (corresponding to button presses on an Atari controller). Second is their use of the 'Experience replay' technique where the algorithm keeps the N most recent experiences to be used in training. Each experience $e_t = (s_t, r_t, a_t, s_{t+1})$ consists of a state s_t, action a_t taken in that state, reward r_t for performing said action and resulting state s_{t+1}. Mini-batches of experiences are randomly sampled for training. Experience replay removes correlations between observed sequences thereby preventing overfitting.

The DQN algorithm

The algorithm consists of two nested loops - an outer loop iterating over episodes and an inner loop performing a number of actions N, recording the experiences resulting from those actions and then sampling and training on a mini batch of previous experiences. The agent selects actions to perform based on an ϵ-greedy policy. At the start of training ϵ is 1 and it gradually decays with time until it reaches a minimum - Figure 2.8.

The Q network in the paper is actually two networks - Q and $\hat{Q}$ where network $\hat{Q}$ produces predicted rewards to be stored as part of an experience tuple. Every C steps the $\hat{Q}$ network's weights are reset to those of network Q. The introduction of the second network helps prevent oscillation or divergence when compared to an online-only Q network. In the context of a simple 2D robot arm-game, divergence is not a big issue and can be avoided by making a number of checkpoints while training and selecting the best one. For this reason the DQN implementation for this project does not use $\hat{Q}$ but only Q for faster learning.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, M **do**
Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
for t=1, T **do**
With probability ϵ select a random action a_1 otherwise select $a_1 = arg\,min_a Q(\phi(s_t), a; \theta)$
Execute action a_t in emulator and observe reward r_t and image x_{t+1}
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_d \hat{Q}(\phi_{j+1}, d; \theta^-) & \text{otherwise} \end{cases}$
Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameter θ
Every C steps reset $\hat{Q} = Q$
end for
end for

Figure 2.8: Deep Q-learning with experience replay algorithm[14]

2.4 Transfer learning

Learning something in one domain and transferring that knowledge has been an area of research for a number of years[15]. Transfer learning techniques have recently been developed in the realm of neural networks.

2.4.1 Transferring by substituting the output layer

Within a convolutional neural network virtually all layers except the final output are dedicated to extracting features. The simplest way to do transfer learning is to take advantage of a pre-trained network's feature extracting layers by obtaining their output - $f(x)$ where x is the new dataset of images and $f(\cdot)$ is the network's intermediate output right before the final layer. Once $f(x)$ is obtained, simply train the classifier on $f(x)$ with the desired labels. However this approach to transfer learning is applicable only to domains which are extremely similar. More differing domains might require different sets of high- or low-level features for classification - and since the feature extraction remains the same, training a new classifier will be ineffective.

2.4.2 Finetuning

One step further from retraining the final layer is finetuning. As before this approach uses a pre-trained network in a source domain and adapts its output layer to fit the

target domain. However, finetuning also performs backpropagation over the entire network, including feature-extracting layers. Usually backpropagation is performed with a reduced learning rate and some layers might be 'frozen' preventing them from updating their parameters. The hope is that a majority of the features extracted by the pre-trained network will be kept and only a small set of new features will have to be identified. For example, a large part of features, especially low-level ones like edges and simple shapes need not change a lot if at all. Finetuning is currently the most common approach to transfer as Convolutional networks exhibit a good level of knowledge transfer even with a relatively simple technique[16].

2.5 Inverse kinematics

Inverse kinematics gives answer to the question: "Given a desired location for the tip of the robotic arm, what should the angles of the joints be for the arm's tip to be that position?" There is usually more than one solution and inverse kinematics can be a difficult problem to solve. However, in our 2D environment, the necessary angle equations get reduced to a simple trigonometric problem.

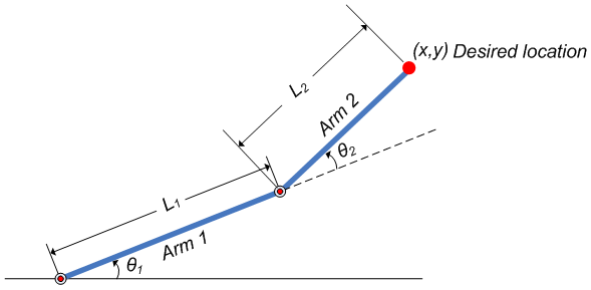


Figure 2.9: Inverse kinematics applied to a one-joint system will give you angles θ_1 and θ_2[17]

2.6 PyGame

PyGame[18] is a cross-platform python module designed for writing simple 2D games. A PyGame project relies on the concept of bitmap 'surfaces' which are Bit blit together. A useful feature of PyGame is that it allows direct access the bitmap surfaces as NumPy arrays which can be fed directly to a convolutional network.

Another benefit to PyGame is that it can run without a GUI over ssh, simplifying the use of multiple DoC machines for training.

2.7 Maximum mean discrepancy

Maximum mean discrepancy[19] (MMD) is a test statistic which measures the distance of two sets after mapping them to a Reproducing Kernel Hilbert Space[1] (RKHS).

[1] `https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space`

The measure itself is straightforward:

$$MMD^2(\mathbf{x}, \mathbf{y}) = \Big\| \sum_{i=1}^{N} \frac{\phi(\mathbf{x}_i)}{N} - \sum_{j=1}^{M} \frac{\phi(\mathbf{y}_j)}{M} \Big\|^2 \tag{2.17}$$

Where $\mathbf{x}$ and $\mathbf{y}$ are the feature representations of our datasets and N and M are the corresponding number of samples. $\phi(\cdot)$ denotes the mapping to RKHS which is typically unknown.

MMD allows us to measure how close the feature representations of images from different domains are in the Domain Adaptation model described further in this report. Intuitively, we want MMD to be small as possible - i.e. our models to represent features of images from both a simulation and reality in a similar manner.

2.7.1 Kernel matrices

In most cases $\phi(\cdot)$ will be an unknown function. To avoid this, MMD in this project is computed via the kernel trick, where a standard Radial Basis Function (RBF)[2] kernel is chosen:

$$k(u, v) = \exp(-\|u - v\|^2/\sigma) \quad \text{The RBF kernel function} \tag{2.18}$$

Therefore we can rewrite MMD^2 as:

$$MMD^2 = \sum_{i,i'} \frac{k(\mathbf{x}_i, \mathbf{x}_{i'})}{N^2} - 2 \sum_{i,j} \frac{k(\mathbf{x}_i, \mathbf{y}_j)}{NM} + \sum_{j,j'} \frac{k(\mathbf{y}_j, \mathbf{y}_{j'})}{M^2} \tag{2.19}$$

[2] https://en.wikipedia.org/wiki/Radial_basis_function_kernel

Chapter 3

Models

This chapter introduces the three different models used to achieve transfer and any modifications done to them.

3.1 Progressive neural networks

Progressive networks[1] rely on the concept of columns where each column is a whole neural network with layers $h_i^{(j)}$ where j is a column an i is a layer in that column. Columns are trained sequentially on similar, but separate, tasks. Whenever a column has finished training with some final parameters Θ it is frozen - the parameters can no longer be updated via backpropagation. In addition to connections between layers h_i and h_{i-1} within a column, all new columns have lateral connections between the output of the corresponding layers in the columns before it. A layer - $h_i^{(j)}$ - receives input both from the previous layer in the columns $h_{i-1}^{(j)}$ and the previous layer from the previous column $h_{i-1}^{(j')}$ where $j' \in (1, ..., j-1)$. This generalizes to the following layer equation:

$$h_i^{(k)} = f\left(W_i^{(k)} h_{i-1}^{(k)} + \sum_{j<k} U_i^{(k:j)} h_{i-1}^{(j)}\right) \tag{3.1}$$

Where $f(x)$ is the layer's activation function, $W_i^{(k)}$ is the weight matrix for layer i of column k, $U_i^{(k:j)}$ are the connections between layer $i-1$ of column j and layer i of column k and h_0 is the network input. A sample network with 3 columns can be seen in Figure 3.1.

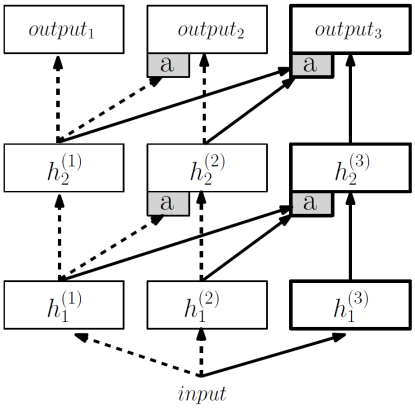


Figure 3.1: A progressive network with 3 columns

An advantage of progressive networks over typical fine-tuning approaches is that they do not discard previously learned features in older columns since they are frozen. This allows a progressive networks to retain their performance on all trained domains. However, that comes at the cost of increasing the numbers of parameters of the network with the addition of new columns for every task.

In order to avoid the curse of dimensionality when dealing with multiple columns progressive nets can replace the lateral connections between columns with an adapter. If $h_{i-1}^{(<k)} = [h_{i-1}^{(1)}...h_{i-1}^{(j)}...h_{i-1}^{(k-1)}]$ is the vector of all laterally layers in previous columns we want to replace it with a single hidden layer multilayer perceptron (MLP). Before $h_{i-1}^{(<k)}$ is provided as an input to the MLP it is multiplied by a vector to adjust for the different scales of the inputs. As the index k grows the adapter becomes more necessary. When we use an adapter, $h_i^{(k)}$ becomes:

$$h_i^{(k)} = f\left(W_i^{(k)} h_{i-1}^{(k)} + U_i^{(k:j)} \sigma(V_i^{(k:j)} \alpha_{i-1}^{(<k)} h_{i-1}^{(<k)})\right) \tag{3.2}$$

Where α is the scaling vector and $V_i^{(k:j)}$ is the projection matrix representing the MLP. In this project we will not be using networks with more than two columns and will not need the adapter.

3.2 Domain alignment network with weakly supervised pairwise constraints

This approach relies on two general ideas. First - bringing closer in feature space the source domain, where data is easily available, and the target domain, where obtaining training data is hard. Second pairing up examples from the source and target domains which represent similar states e.g. an image from a simulation depicting a robot arm in a specific position and a real-world image depicting the arm in the same configuration. These two examples should fall close to one another in the space of the final representation θ_{repr}.

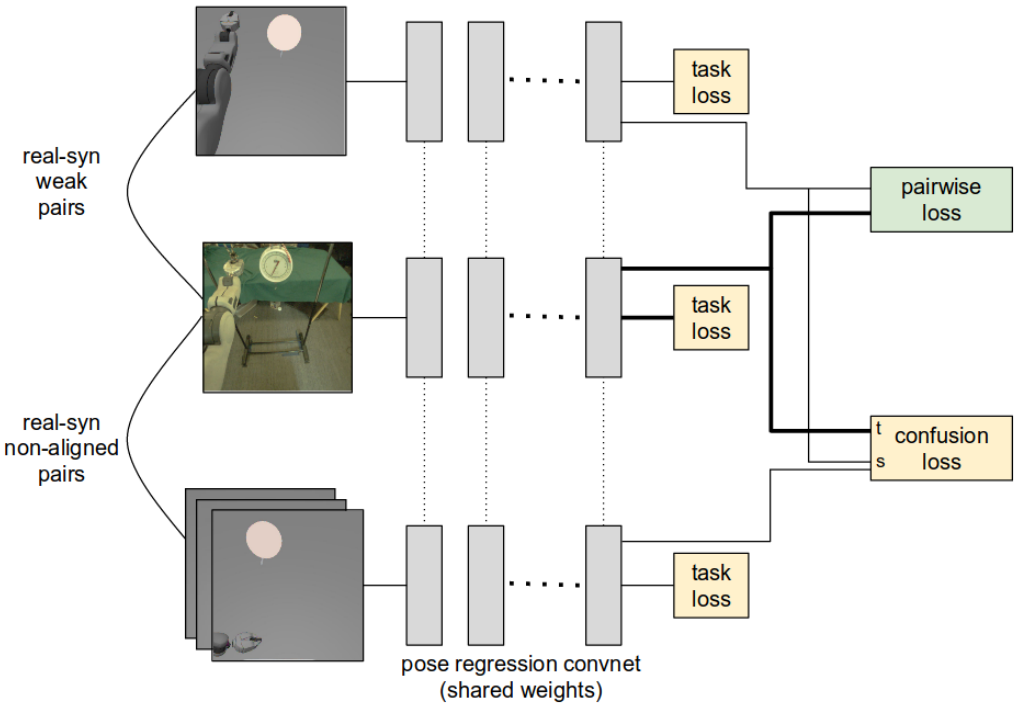


Figure 3.2: The network computes three types of losses: Typical classification losses, a domain confusion loss and a pairwise loss.[2]

3.2.1 Domain confusion

When trying to align the two domains the model trains a domain classifier θ_D which tries to correctly classify images from the source and target data sets. Parallel to that the loss function $\mathcal{L}_{conf}$ learns a representation θ_{repr} so that the domain classifier cannot distinguish between the two domains in features space.

$$\mathcal{L}_{conf}(x_S, x_T, \theta_D; \theta_{repr}) = \sum_{x \in (x_S \cup x_T)} \sum_{d} \frac{1}{D} \log q_d(x, \theta_D, \theta_{repr}) \tag{3.3}$$

Where q is the domain classifier activations.

$$q_d(x, \theta_D, \theta_{repr}) = \text{softmax}(\theta_D^T f(x; \theta_{repr})) \tag{3.4}$$

3.2.2 Pairwise loss

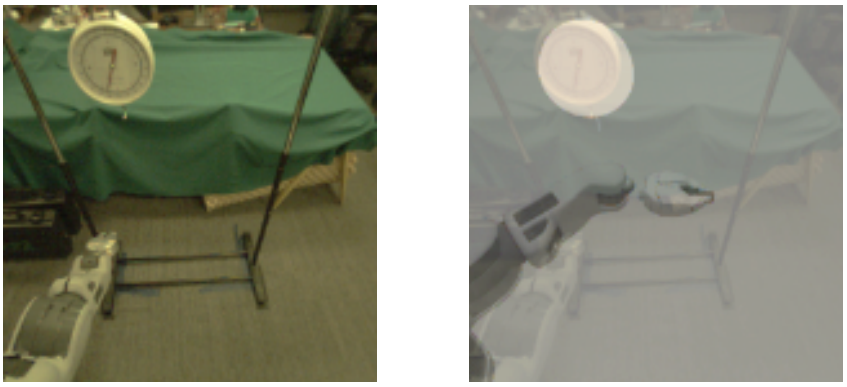
The pairwise loss function $\mathcal{L}_{pairwise}$ is defined as the distance between two corresponding examples from the source and target domains in the feature space of θ_{repr}:

$$\mathcal{L}_{pairwise}(x_S, x_T; P, \theta_{repr}) = \sum_{(i,j) \in P} \left[\frac{1}{2} \rho\left(x_S^{(i)}, x_T^{(j)}; \theta_{repr} \right)^2 \right] \tag{3.5}$$

Where P is the set of weak pairs between images from source and destination and ρ is the Euclidean distance in the feature space:

$$\rho\left(x_S^{(i)}, x_T^{(j)}; \theta_{repr}\right) = \left\| f(x_S^{(i)}; \theta_{repr}) - f(x_T^{(j)}; \theta_{repr}) \right\|_2 \tag{3.6}$$

It is important to note that the pairing P and the representation θ_{repr} depend on each other which prevents us from optimising them at the same time. To avoid this problem the authors of the paper first learn a representation θ_{repr} from only the examples in the source domain while also minimising $\mathcal{L}_{conf}$. Once the source-only model is trained θ_{repr} is used to map both source and target images to the same space where a nearest neighbour algorithm can determine the weak pairs P. An example weak pair - Figure 3.3.



(a) A real image **(b)** A simulated pair image

Figure 3.3: A weak pair

The model's weights are shared between the source and target domain samples i.e. θ_ϕ and θ_{repr} are the same for both classification functions. In turn the total loss function of the model is:

$$\mathcal{L}(x_S, \phi_S, x_T, \phi_T, P, \theta_D; \theta_\phi, \theta_{repr}) = \tag{3.7}$$

$$\mathcal{L}_\phi(x_S, \phi_S; \theta_\phi, \theta_{repr}) + \mathcal{L}_\phi(x_T, \phi_T; \theta_\phi, \theta_{repr}) \tag{3.8}$$

$$+\lambda \mathcal{L}_{conf}(x_S, x_T, \theta_D; \theta_{repr}) \tag{3.9}$$

$$+\upsilon \mathcal{L}_{pairwise}(x_S, x_T; P, \theta_{repr}) \tag{3.10}$$

Where λ and υ tweak how strongly the additional constraints are enforced. Before the full error can be minimized the set of pairings P between the source an target domains must be established. We do so by first minimizing the action classifier on only source images and domain confusion as described in Figure 3.4.

This model is semi-supervised. While action labels are available from the simulated source domain, they are not given in the target real domain. The algorithm overcomes this issue by assigning the action label of an image from the source domain to the closest weak-paired image from the target domain.

Collect x_s source domain images with labelled object pose
Collect x_t target domain images
Minimize $\mathcal{L}_\phi(x_S, \phi_S; \theta_\phi, \theta_{repr}) + \lambda \mathcal{L}_{conf}(x_S, x_T, \theta_D; \theta_{repr})$ with respect to $\theta_\phi, \theta_{repr}$
for $x_T^{(j)}$ in x_T **do**
    $i^* = arg\,min|f_{conv1}(x_s^{(i)}; \theta_{repr}) - f_{conv1}(x_s^{(j)}; \theta_{repr})|_2$
    Add (i^*, j) to P
end for
Minimize $\mathcal{L}(x_S, \phi_S, x_T, \phi_T, P, \theta_D; \theta_\phi, \theta_{repr})$ with respect to $\theta_\phi, \theta_{repr}$

Figure 3.4: Establishing the set of weak pairs P[2]

3.3 Domain adaptation network

The final approach focuses on a two-stream solution where one network is trained on the source domain and another network is trained on the target domain simultaneously.

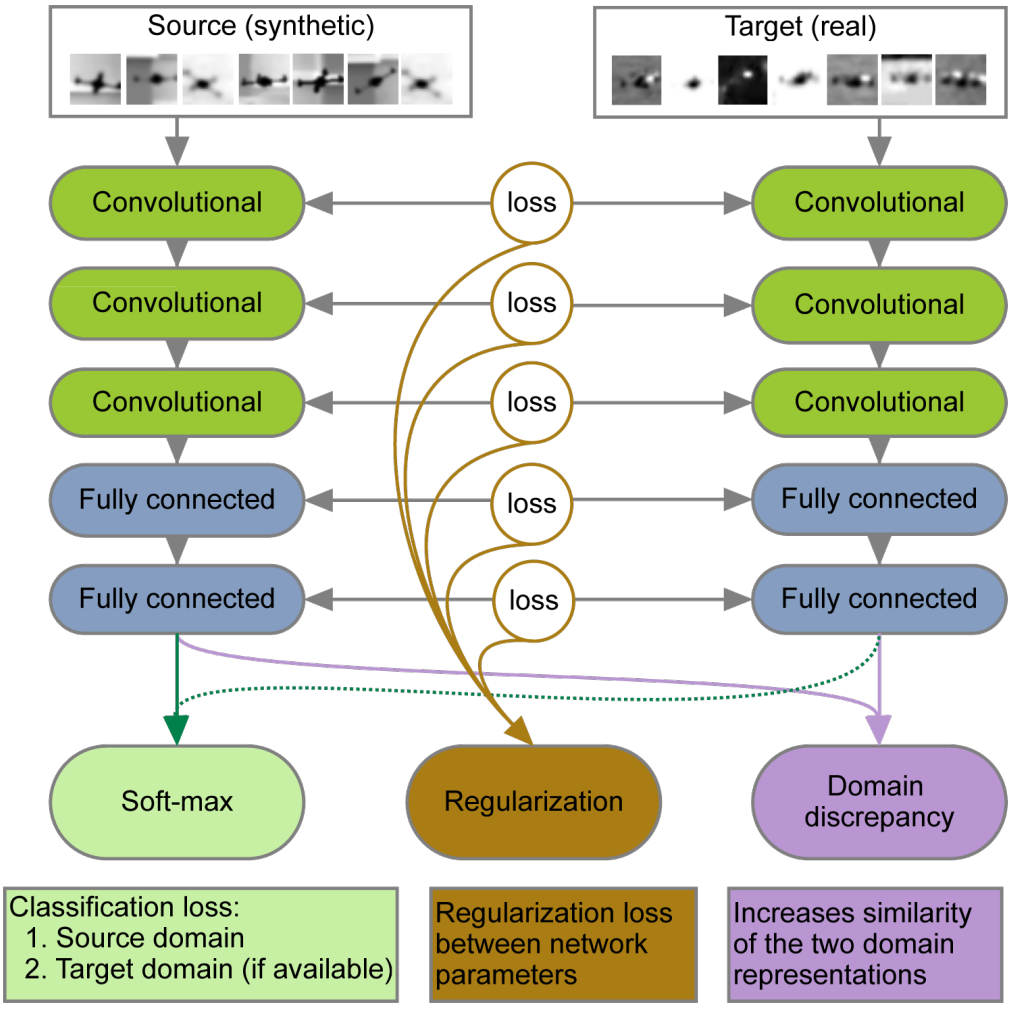


Figure 3.5: The two stream framework.[3]

3.3.1 Layer weights regularization

A loss function is imposed on the distance between weights of corresponding layers in the two networks. These weight regularizers prevent overfitting on the target domain when there are only a few examples to train on. For example an L_2 norm:

$$r_w(\theta_j^s, \theta_j^t) = \|a_j\theta_j^s + b_j - \theta_j^t\|_2^2 \tag{3.11}$$

The authors chose to not penalize linear transformation of the weight in order to better model the domain shift between source and target. Scalars a_j and b_j are therefore learned at training time.

3.3.2 Maximum Mean Discrepancy

The authors also introduce a Maximum Mean Discrepancy[19] loss between the distributions of the representations of the source and target sample sets. Since the two domains share common key features their representations at the layer before the classification layer should have similar distributions. This similarity is the mean discrepancy measured by MMD. The whole loss function takes the following the form:

$$L(\theta^s, \theta^t|\mathbf{X}^s, Y^s, \mathbf{X}^t, Y^t) = L_s + L_t + L_w + L_{MMD} \tag{3.12}$$

$$L_s = \frac{1}{N}\sum_{i=1}^{N^s} c(\theta^s|x_i^s, y_i^s) \tag{3.13}$$

$$L_t = \frac{1}{N}\sum_{i=1}^{N^t} c(\theta^t|x_i^t, y_i^t) \tag{3.14}$$

$$L_w = \lambda_w \sum_{j\in\omega} r_w(\theta_j^s, \theta_j^t) \tag{3.15}$$

$$L_{MMD} = \lambda_u r_u(\theta^s, \theta^t|\mathbf{X}^s, \mathbf{X}^t) \tag{3.16}$$

Where $c(\theta^s|x_i^s, y_i^s)$ and $c(\theta^t|x_i^t, y_i^t)$ are the classification losses for each domain, $r_u(\theta^s, \theta^t|\mathbf{X}^s, \mathbf{X}^t)$ is the MMD, $r_w(\theta_j^s, \theta_j^t)$ is the distance between a source and target layer's weights and λ_w and λ_u are scalar parameters In this paper the classification loss was cross-entropy applied to a softmax output layer.

Chapter 4

Implementation

The models in this project were implemented either in Tensorflow[20] or Keras[21], a high-level Tensorflow fronted. Keras enabled fast model creation and experimentation while Tensorflow allowed for the implementation of the more complex losses within models. Models were run in a small two dimensional environment.

4.1 The environment

The environment is a simple 2D game, implemented in PyGame, featuring an arm with two joints anchored in the middle of the frame. The objective of the game is for the arm's gripper to reach a target whose location is randomly chosen as long as it is within reach. As inputs it receives one of eight possible actions corresponding to all possible combinations of clockwise and anti-clockwise joint rotations.

The game can scale to multiple resolutions, however a small 40 by 40 resolution was chosen in order to reduce the models' size and accelerate training and testing.

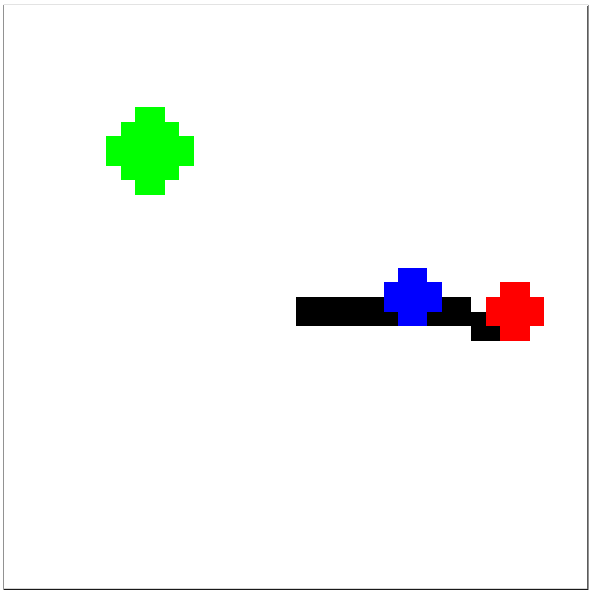


Figure 4.1: A scaled up version of the 40 by 40 game. The red end must reach the green target.

The game created some issues for the learning models. Because of the relatively small resolution some changes to the arm angle values would not be reflected in a change of state. Therefore the models could perform an action and receive no visual update of the state. In order to prevent that at a game size of 40x40 the arm angles change by 12 degrees at every step.

4.1.1 Modifying the environment

In order to establish the need for transfer the environment's characteristics can be changed at will. Different colours for objects and background, shapes of targets and random noise can be added to the images which the environment provides to the models. After some informal testing, three different environments were chosen, each modifying the base game in a fundamentally different way.

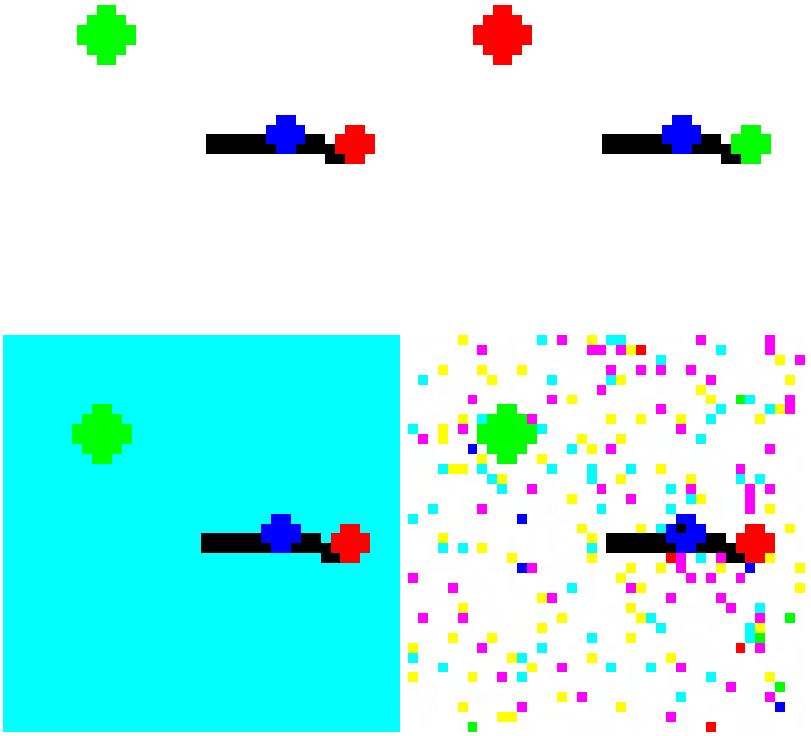


Figure 4.2: Clockwise from top left - Normal game, swapped target and gripper colours, Gaussian noise with $\sigma = 0.6$ and cyan background

4.2 Inverse Kinematics Engine

The inverse kinematics engine generates action labels for use with our supervised models. The engine takes a game's inner state as input and produces optimal actions for the robot's gripper to reach the object. It does so by calculating angles θ_1 and θ_2 for the joints using trigonometry[22].

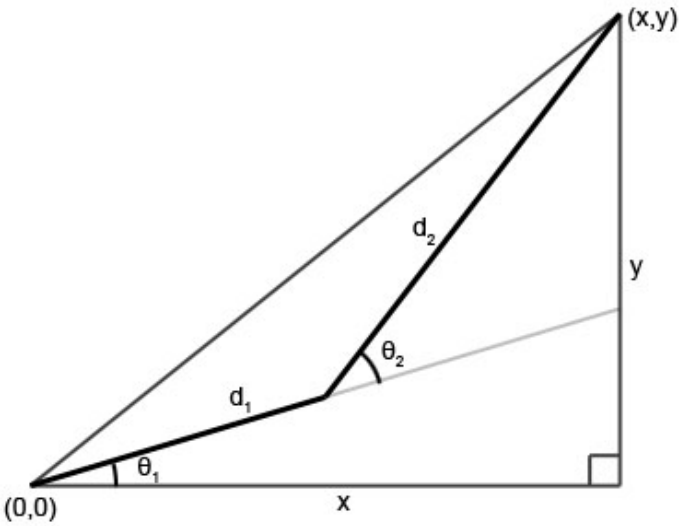


Figure 4.3: The trigonometric problem.

The equations for angles θ_1 and θ_2 are:

$$dist = \sqrt{(x^2 + y^2)} \tag{4.1}$$

$$\theta_1 = \pm\Big(\arccos(\frac{x}{dist}) \pm \arccos(\frac{dist^2 + d_1^2 - d_2^2}{2d_1 dist})\Big) \tag{4.2}$$

$$\theta_2 = \mp\arccos(\frac{d_1^2 + d_2^2 - dist^2}{2d_1 d_2}) \tag{4.3}$$

The plus or minus signs are dependent on which quadrant the target is in.

4.3 Q-Learning setup

The Q-learning process consists of three main parts:

- Initializing the experience replay memory with random actions
- Generating new experiences by performing the highest future reward action according to the network and recording them in the memory
- Randomly sampling from the experience memory and training in mini-batches after every N actions

The two last steps repeat for a large number of frames - more than 200000.

Rewards to the agents are given upon entering concentric rings around the target. Entering an inner ring awards a greater reward while leaving a ring incurs a penalty. A significant reward is given upon direct contact with the target. Multiple experiments with distance-based rewards were ineffective as the agents would not hit enough targets in its exploration stages and instead chose to oscillate between a small number of position, each wielding a small reward. Both the vanilla Q-network and progressive networks built on top of it are trained using the same framework.

4.4 Keras models

The foundational Q network and the progressive networks which build on top of it were implemented in Keras mainly for its simplicity and ease-of-use.

4.4.1 Q network

The Q network is a very small CNN. It has only one convolutional ReLu layer with 8 filters of size 3 by 3 and stride of 3. Following that is a hidden layer with 16 neurons and an output layer with 8 neurons - each representing the expected future reward for each of the 8 possible actions. The network was minimising a Mean Squared Loss using Adam with a learning rate of $lr = 0.00001$. A relatively small learning rate was chosen since DQN performs a lot of mini-batch updates throughout its training.

The small number of network parameters drastically sped up the DQN process, which was important as on average the best-performing networks required at least 3 million frames of training.

After training the network, we can inspect the output of the network's convolutional layer. As expected it has learned filters for the target (second from the top in the left column), blue joint(top, right column) and red gripper(third from the top, right column).

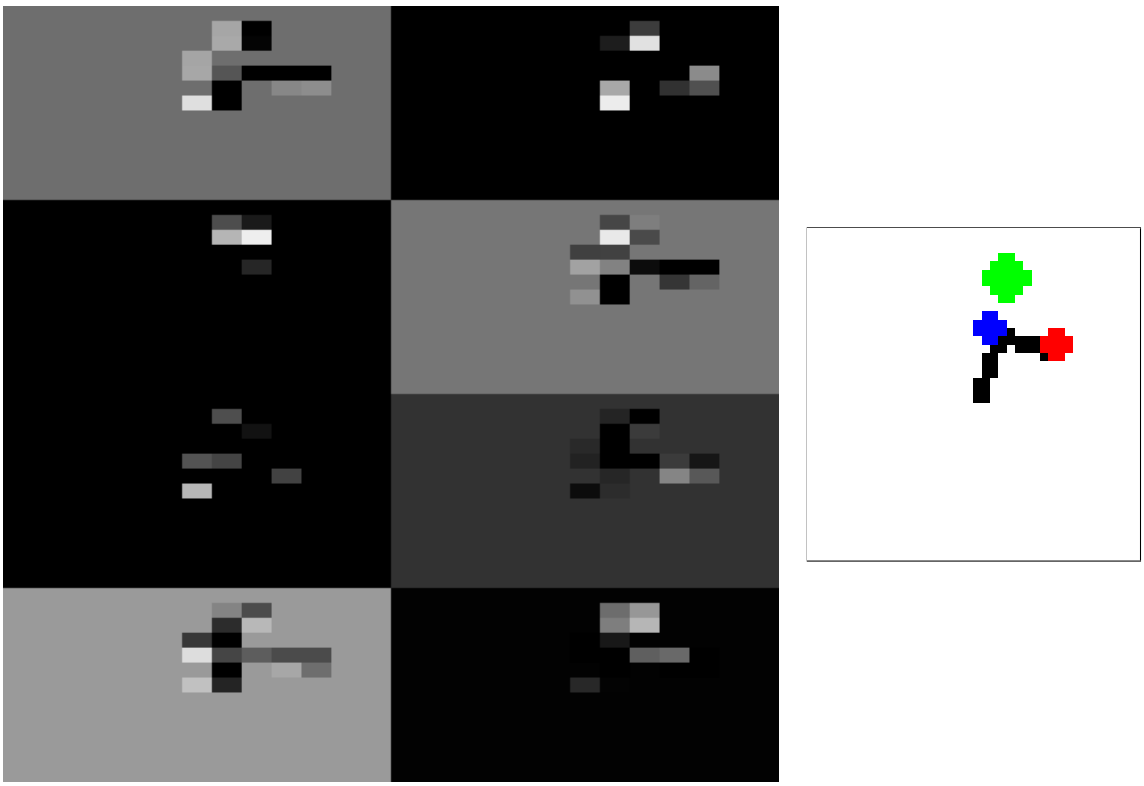


Figure 4.5: Output from the trained network's convolutional layer

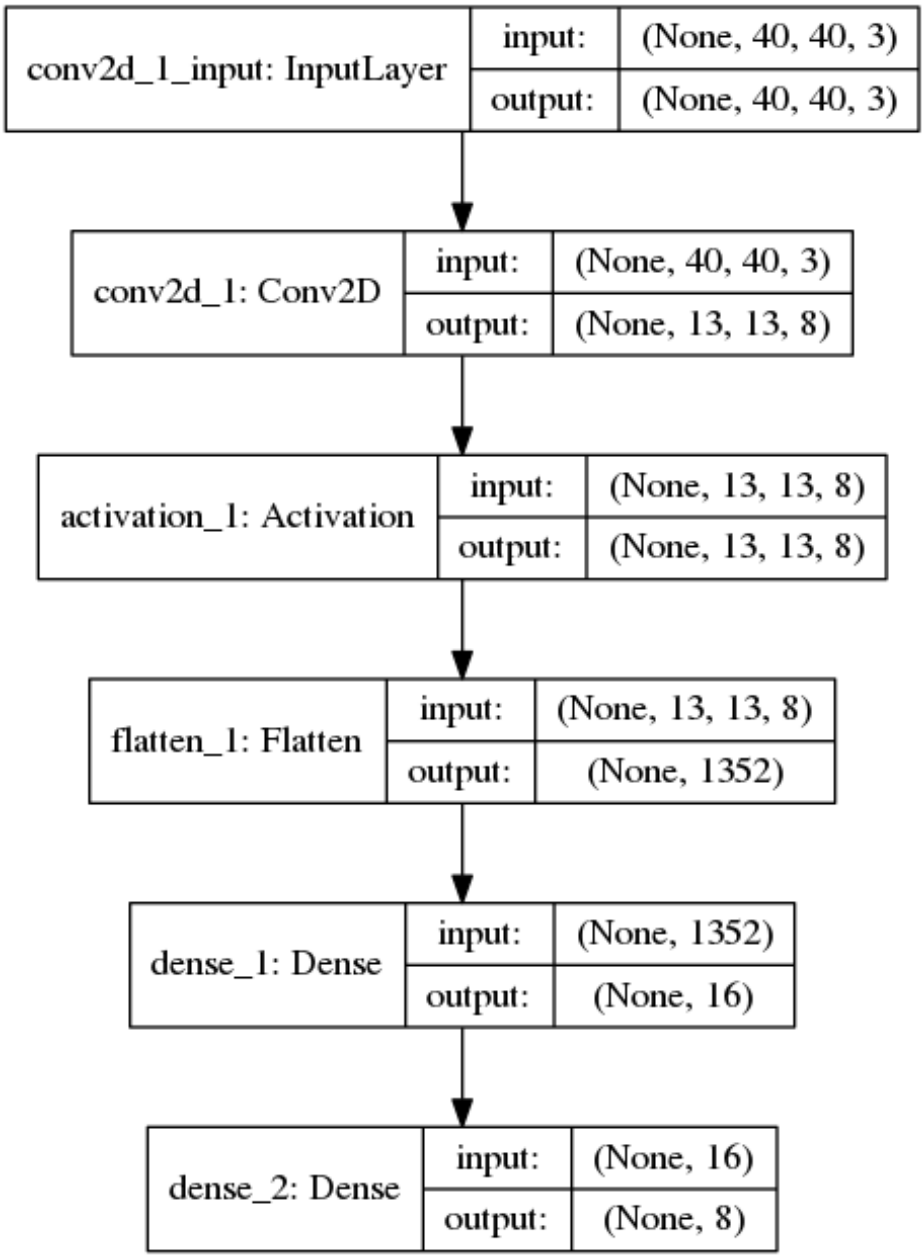


Figure 4.4: Structure of the DQN network in Keras

4.4.2 Progressive network

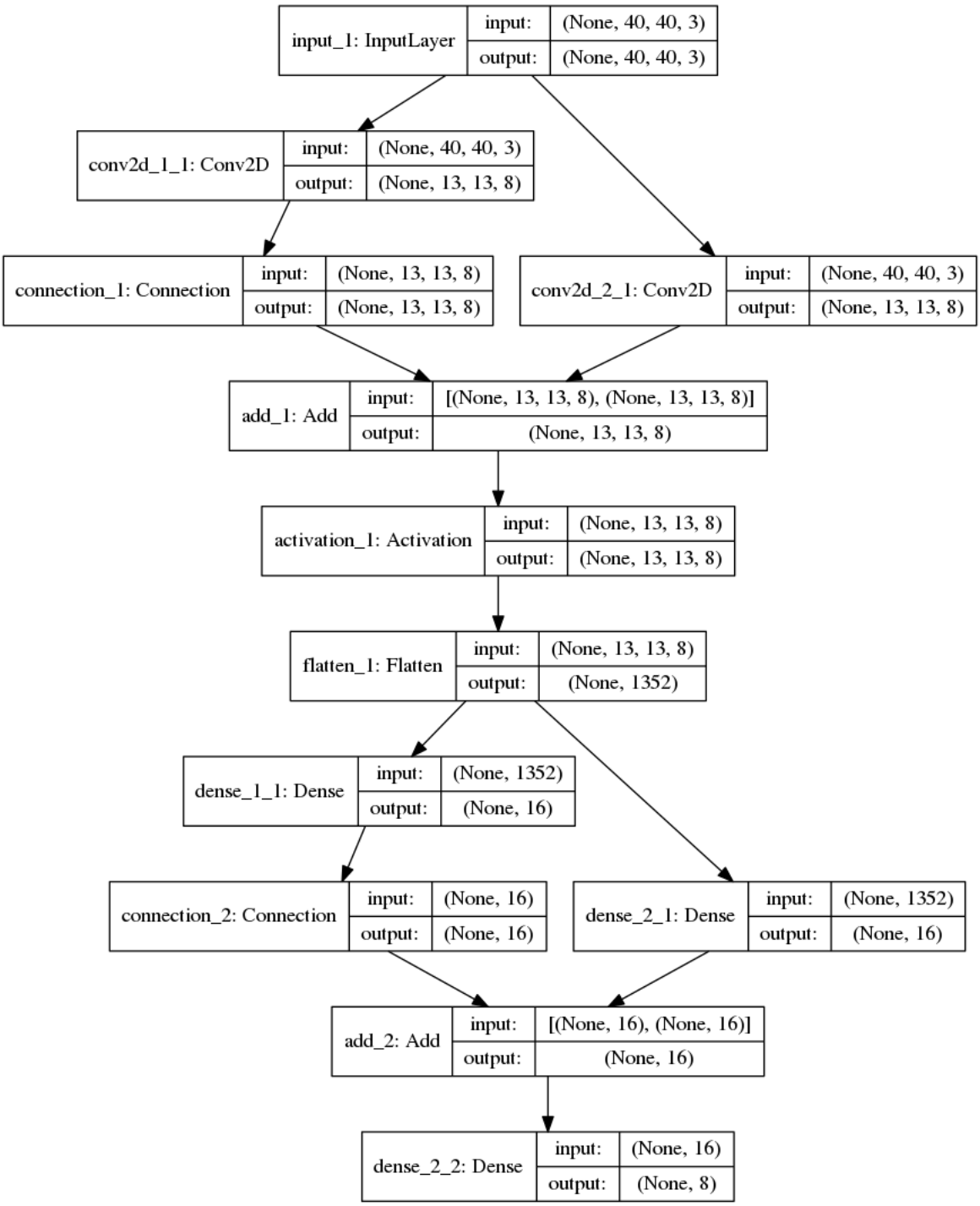


Figure 4.6: Structure of a progressive network with one layer in Keras

Each column of the progressive network has the same set of hyperparameters as the basic DQN model. The only difference is that each layer of a column is connected to the corresponding layer in the column before it with a custom Connection layer which acts as a per element weight matrix for the outputs of the previous column's layers. The output of the Connection layer is then added to the output of the current

column's own layer.

A two column network is trained for each of the three modified environments for a total of three progressive networks. The first column is always the base DQN network and the second column is trained within the modified environment. Instead of three two-column networks we could have trained one four-column network, where the last three columns were each trained on a different environment. This was not done because adding a third and fourth columns would have severely slowed down training performance for two reasons. First, with the increased number of parameters each subsequent column would train slower than the one before it. While not exponential, this slowdown would be noticeable. Second, and more important, the columns have to be trained in sequence, which prevents us from training the three columns at the same time on multiple machines.

4.5 Tensorflow models

The Domain adaptation and Domain alignment networks were defined using Tensorflow as their architecture and especially loss functions were atypical.

Both models are designed to work in a supervised environment. Unlike progressive networks, utilising them in a reinforcement setting is not straightforward since their main objective is adapting to a domain for which we have a very limited number of samples. In the case of reinforcement learning there are indeed less samples to train on in target domain than the source. But this is a positive side-effect of transfer learning working by reducing the time needed to converge.

4.5.1 Supervised CNN

The base network on top of which both transfer learning models are built is a CNN made of two convolutional layers with ReLu activations, one hidden dense layer and a softmax output layer. Both convolutional layers have 8 filters. The first layer has a kernel size of 5 by 5 and a stride of 1 while the second layer has a kernel size of 3 by 3 and a stride of 1. The stride is deliberately small, as we do not want to skip over any pixels. The hidden layer has 256 units and the output layer has 8 - corresponding to the possible actions of the agent - Figure 4.7. The network does not use pooling layers because at a low resolution every pixel is important. All supervised networks are trained using Adam and learning rate $lr = 0.0005$ which provides a good balance between convergence speed and oscillation tolerance.

Instead of predicting the future reward like DQN the output layer represents the probabilities of the next best action according to the policy learned under the supervision of the inverse kinematics engine.

This network is more complex than the one utilised in the reinforcement models. Initially, it had an identical configuration to the one used in the DQN model. However that network was not able to achieve over 90% accuracy on even a relatively small sample size of 3000. The network was too shallow and lacked the necessary expressiveness.

Adding an additional convolutional layer focuses on the image. After training we can see that the first layer of the deeper network successfully manages to

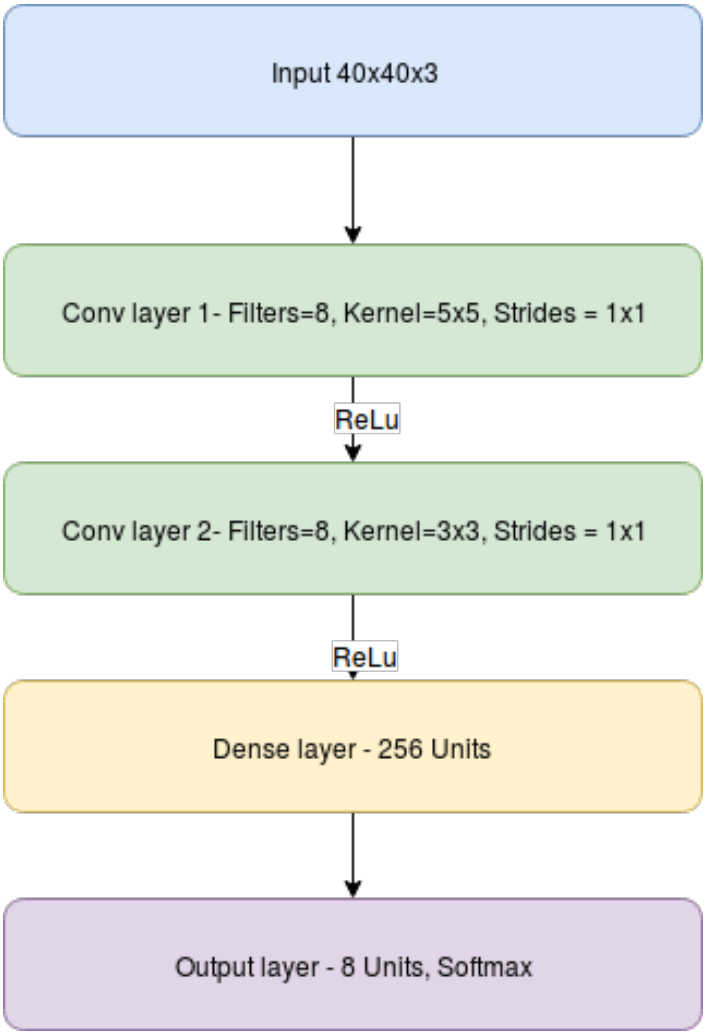


Figure 4.7: The basic Convolutional network used in supervised models

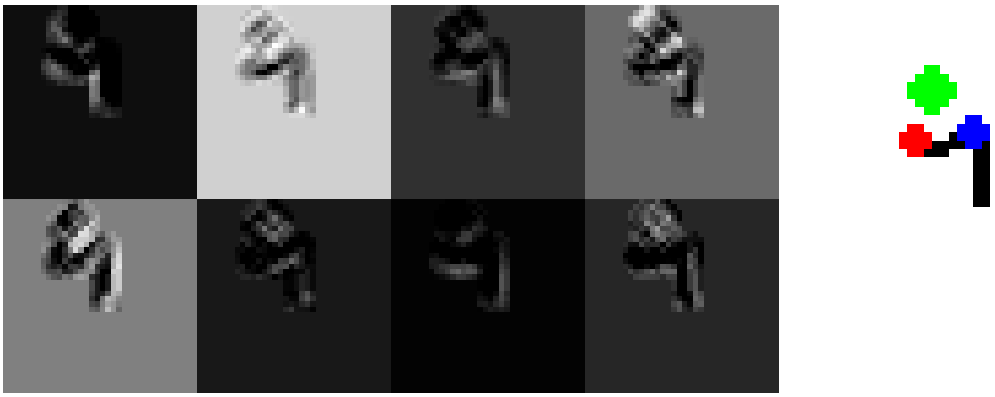


Figure 4.8: Output from a trained network's first convolutional layer

isolate the key objects - the robot and target - from the background, allowing the subsequent layer to pick out higher-level features - Figure 4.8.

4.5.2 Domain alignment network

Domain alignment networks share layers between source and target domains. Training proceeds in two steps. First the network minimizes the classification loss on the target domain only while also minimizing the domain confusion loss. After convergence, we pair up images from the target domain to images in the source domain whose representation after the second convolutional layer is closest to theirs (Figure 4.9). Then we proceed with minimising the total loss.

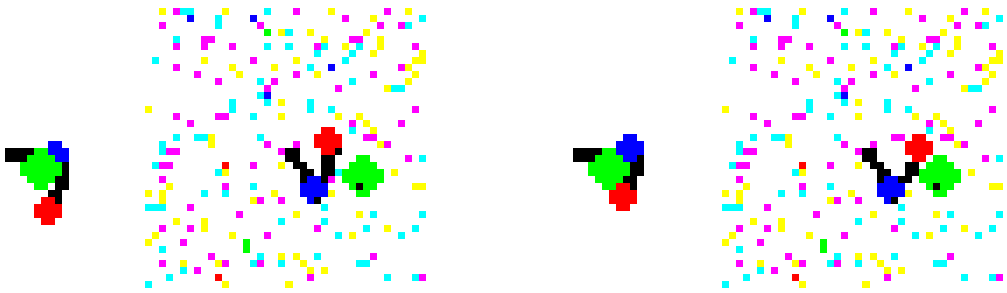


Figure 4.9: Weak pairs selected by the alignment network. Likely due to the proximity between the three coloured points.

4.5.3 Domain adaptation network

The Domain adaptation network is a two stream network where the source and target domain streams do not share layers but instead have a loss placed on the distance between their weights. A custom Radial basis function kernel function was implemented in order to compute the Maximum Mean Discrepancy via the kernel trick. During training the network first trains on the source domain until convergence and afterwards it trains using data from both source and target domains. The first part of training can be skipped if a pre-trained network is provided.

Chapter 5

Evaluation

The evaluation strategy for all models is identical. First, both a DQN and a supervised model are trained on the basic version of the game in order to obtain a baseline performance benchmark. Afterwards the models are tested in the modified environments and evaluated again with the expectations to observe some significant degradation in performance. In the case where the environment is polluted with random noise, various levels of noise were tested until performance was noticeably below the baseline level.

For each of the three modified environments a new transfer model was trained using each of the three different approaches for a total of 9 different models.

The different agents and environments were tested over 1000 episodes, where each episode is limited to 100 actions before moving the target to a new location, resetting the game and attempting again. Typically agents perform the optimal actions as provided by their neural networks. However, 5% of the time, the agents would perform a randomly selected action instead. This prevents agents from oscillating between two states.

5.1 Baseline levels

5.1.1 DQN

The baselines DQN model trained on 5 million frames performs quite well, especially considering how simple its layers are. With almost 90% success rate of hitting a target within 100 actions, it is twice as successful as an agent which performs actions randomly. However, where the difference in performance becomes clear is when you compare the average number of actions required to hit a target. The random agent takes on average 35, the DQN - 12.

As expected the baseline agent's performance noticeably degrades when placed in a different environment. The worst case being the reversing of colours between the arm and target, where the number of targets hit is reduces by 36% - Figure 5.3. This is reasonable as the most-important features in the non-modified version of the environment are the green target and red arm gripper.

Average distance travelled per hit is also affected but not as significantly - 26% more moves on average when evaluated in an environment with a cyan background colour - Figure 5.13.

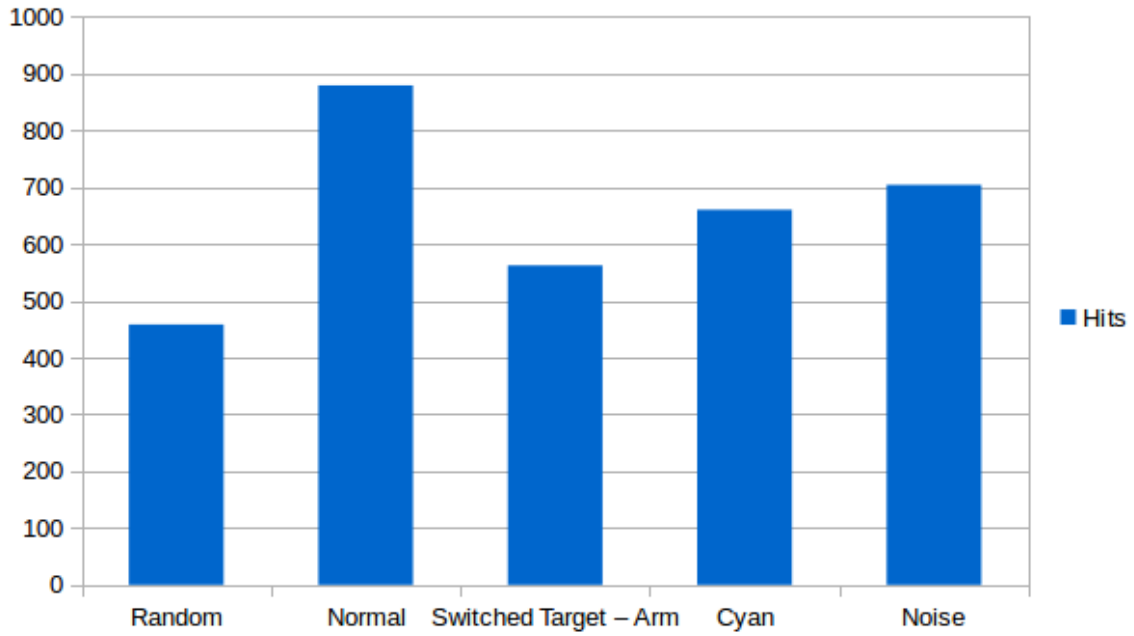


Figure 5.1: Number of targets hit by the base DQN model performing in different environments. Random agent performance provided for comparison.

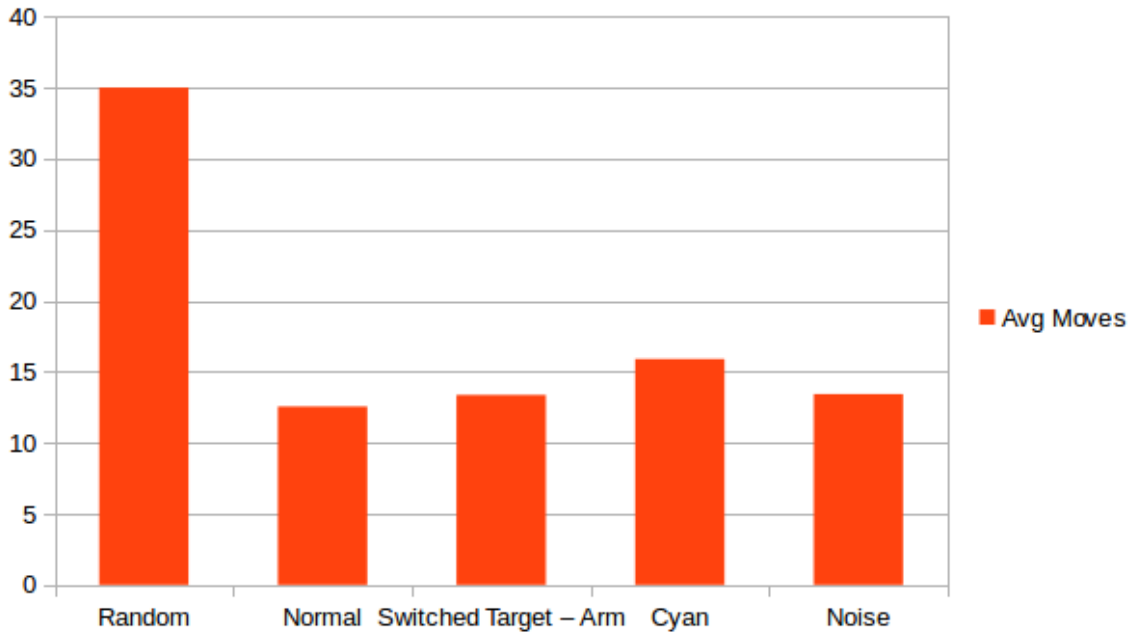


Figure 5.2: Number of average action taken by DQN to hit a target. Random agent performance provided for comparison.

Overall we can observe that the baseline DQN model performs well and changes to the environment, while quite detrimental to performance are not catastrophic.

5.1.2 Supervised model

The base supervised model which has been provided optimal action labels by the inverse kinematics engine performs worse than the DQN model when operating both in a normal environment and a modified environment. After achieving over 99% accuracy on a train set of 10000 image-action pairs, the supervised network performs poorly in an actual scenario. By only hitting 600 out of 1000 targets it performs just 30% better than an agent which chooses its actions randomly.

The supervised model also does not generalise too well when placed in a different environment. At best it performs as well as a random agent and, in the case of a cyan-coloured background, it performs significantly worse than a random agent. Whenever the supervised agent hits a target it doesn't do so efficiently averaging between 25 and 30 actions per target hit which is slightly better than a fully random agent.

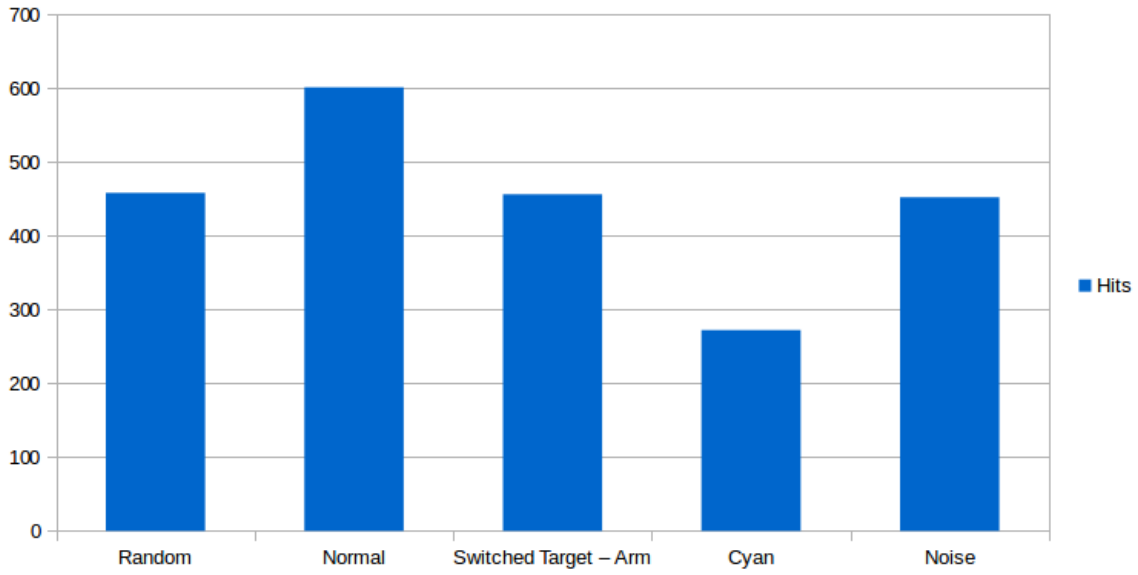


Figure 5.3: Number of targets hit by the base supervised model performing in different environments. Random agent performance provided for comparison.

My hypothesis is that there are two reasons for the sub-par performance. First, the small resolution of the environment combined with the precision of the inverse kinematics actions gives rise to a situation where two states may vary by a single pixel but the suggested actions for the agent are different. Tackling this problem would require massive increasing the training dataset in order to ensure as much states as possible are present in it. While this isn't much of a problem by itself as we can quickly generate as much data as needed it gives rise to the second reason for the lacklustre results. When the model is trained with bigger datasets of 50000 states, it exhibits the tendency to alternate between to states, moving from one to

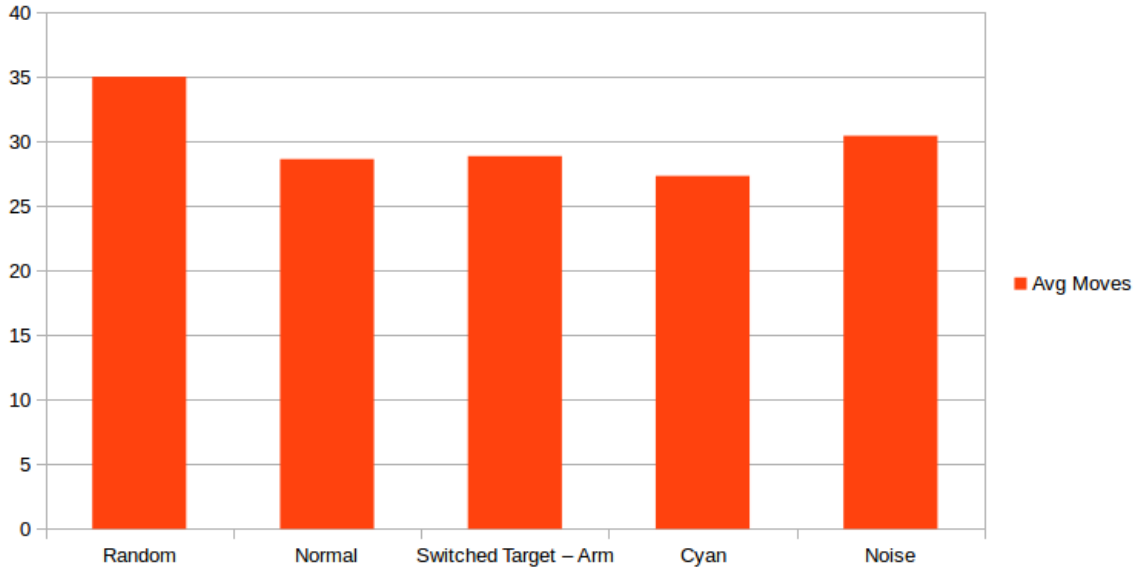


Figure 5.4: Number of average action taken by the supervised model to hit a target. Random agent performance provided for comparison.

the other and never closer to the target. This is why even when training with 20 times as much data the performance of the resulting agent is barely better. Figure 5.5 shows the performance of agents trained with only 500 samples. Notice that in the two static environments - the one where the target and gripper colours are switched, and the one with a cyan background their performance is very close to 600 - the performance of a model trained on 10000 samples.

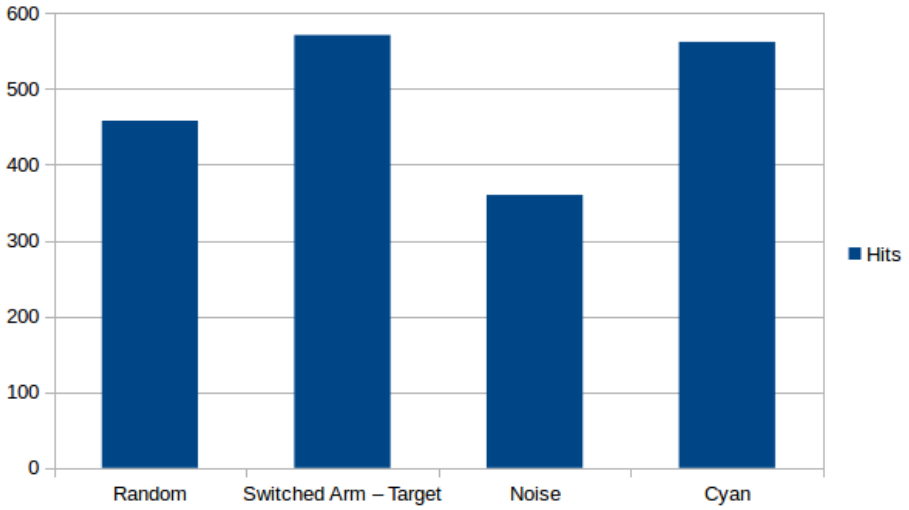


Figure 5.5: Even when trained on only 500 images, the agents performance is close to that of the one trained on 10000 images.

This tendency to oscillate is caused by the model's inability to encode the relatively complex behaviour of the arms when controlled by inverse kinematics. The model is not deep enough to learn the high-level reasoning. A deeper model would be able to learn the correct behaviour but combined with the first issue of every pixel

mattering, this would increase training time exponentially.

In comparison, the reinforcement agent does not try to learn the optimal yet too complex behaviour of IK. This results in more actions than the absolute minimum but the DQN model rarely gets stuck, resulting in an overall better performance.

5.2 Transfer models

5.2.1 Progressive networks

Training

By reusing the output of some or all previous layers, Progressive networks improve training performance on new domains. This is easily observable in the cases when our environment's background is changed or noise is added.

The progressive network training in a cyan-coloured environment starts off performing slightly worse than the pure DQN model. However, it declines much more steadily over the first million frames compared to the baseline algorithm. By frame 800000 the progressive networks has reached its optimal performance and its performances begins to flatten and slightly oscillate.

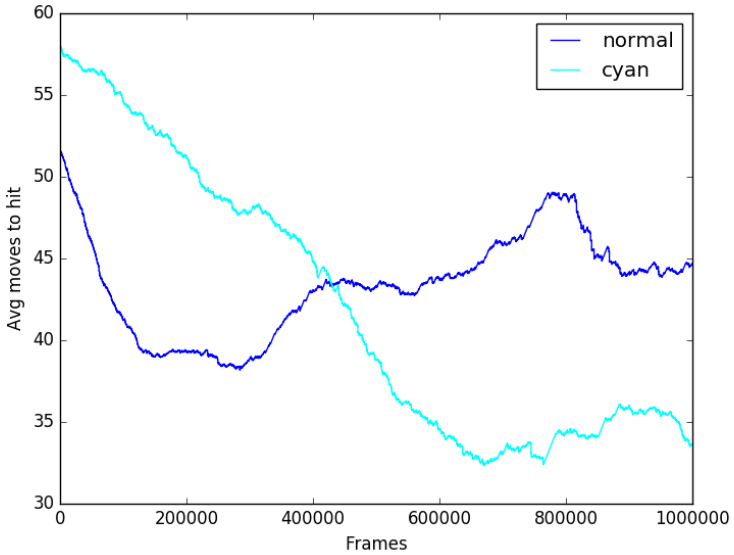


Figure 5.6: The progressive network improves much more steadily than the baseline.

We can observe a similar behaviour when training in a noisy environment. This time due to the inherent nature of noise the progressive model does not improve as steadily and oscillates quite a bit. However, it still converges very quickly and by frame number 800000 begins to oscillate. The model reaches its optimal performance around a million and a half frames, more than 3 times as less than the baseline's 5 million.

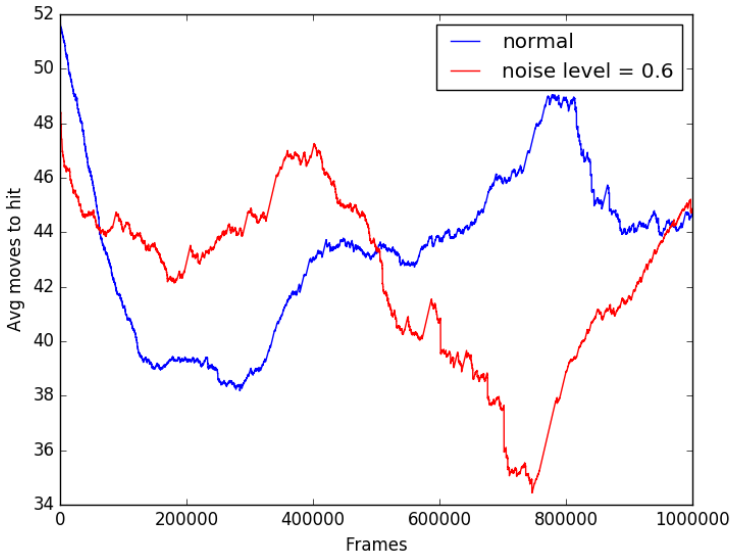


Figure 5.7: The progressive network improves much faster than the baseline.

Switching the target and gripper colours results in a different training scenario. The progressive agent starts off performing much worse than the baseline which is expected. However, after a small period of no improvement it converges extremely quickly, reaching optimal performance in just about 500000 frames where it begins to diverge slightly. This mirrors the results observed in the original paper[1], where the authors inverted the colours of their game.

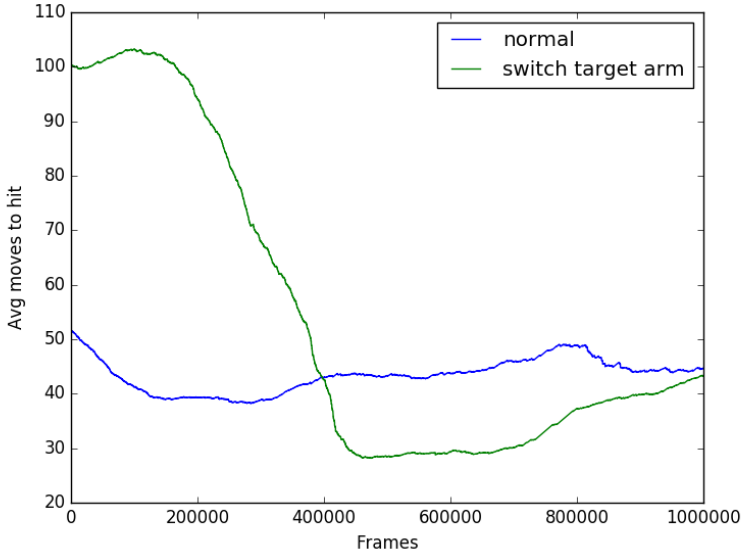


Figure 5.8: Switching the colour of gripper and target initially confuses the agent but it quickly reajusts.

An interesting observation is that during training the baseline model is much more optimistic about the amount it expects to score on average in an episode. In comparison, all three progressive networks were much more conservative.

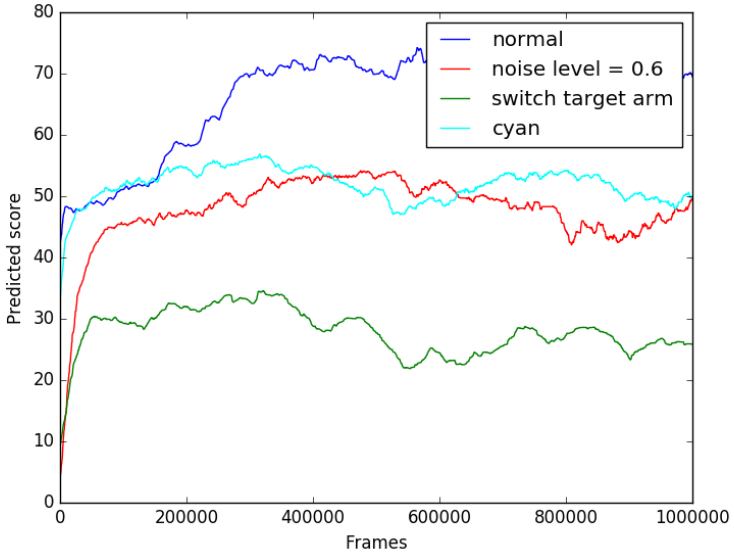


Figure 5.9: The baseline model predicts much higher rewards for itself than the rest. Actual performance indicates otherwise.

The cyan agent and the switched colour agent stop seeing performance gains after a million frames. Beyond one and a half million so does the agent in the noisy environment. Only the baseline agent needs more frames to converge during training. Unfortunately, training of the model in the cyan environment was interrupted before the two millionth frame.

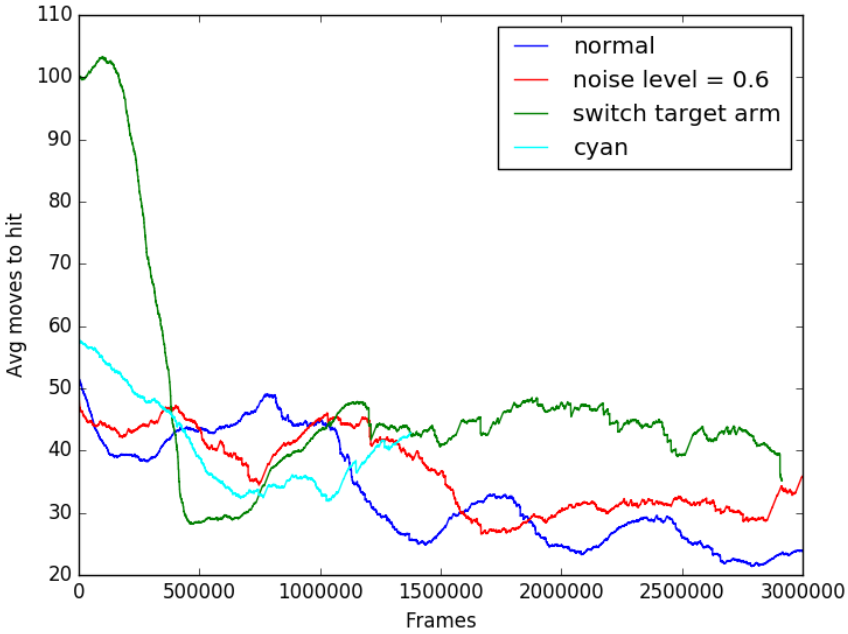


Figure 5.10: All but the baseline model begin oscillating after one and a half million frames.

Accuracy

The reinforcement agents were trained in mini-epochs. Each mini-epoch consists of four frames of action and one mini-batch of training. A million frames correspond to 250000 epochs. During training, a snapshot was saved of the models every 50000 mini-epochs. These snapshots allow us to evaluate the actual performance of the models at different points in their training.

The first snapshot of the agent learning in an environment with a coloured background performs worse than the baseline's performance, achieving accuracy under 50%. At that point it is likely still learning to 'ignore' the extra information provided by the colour. Over the next two checkpoints its accuracy jump to about 90% at which point it stops improving a lot.

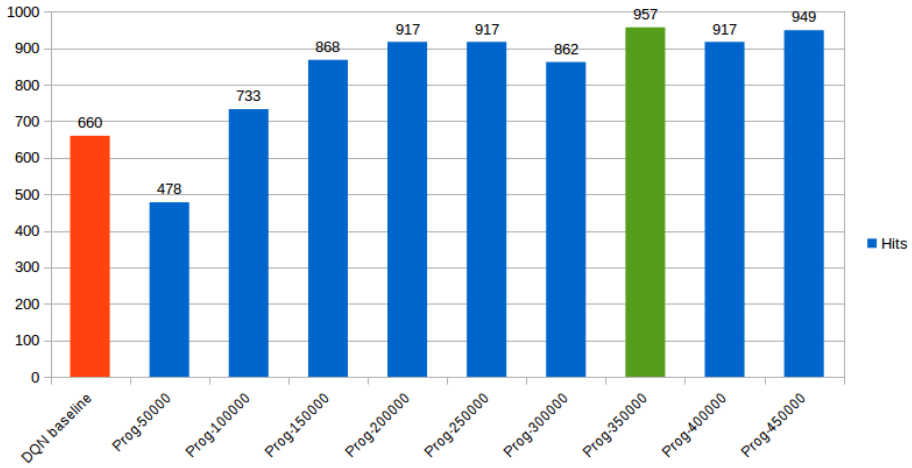


Figure 5.11: Cyan environment. The initial bad performance is quickly corrected.

Switching the target and gripper colours starts us off at the lowest baseline performance of our DQN. Consequently our progressive network does not perform much better in its first snapshot. Its performance continues to increase steadily and after 300000 mini-epochs (1200000 frames) it reaches its optimal performance whereupon it starts oscillating a small amount. The baseline DQN, which forms the first column of the progressive network, is very well stabilised and once the progressive network has learned to correctly switch the target and gripper filter outputs, it can rely on the baseline's immutable dense layer to provide the final feature representation.

The agent trained in the noisy environment also shows similar performance improvement to the previous two. The difference is that it tends to oscillate more than the others, not improving for a number of epoch, then making a leap forward and going back down.

In addition to requiring only a quarter of the frames compared to the baseline DQN in order to reach optimal performance in their own environment, the progressive models all perform better in their respective domains, hitting 950 out of 1000 targets, than the base DQN in the base environment which manages just shy of 900 out of 1000. I.e they not only do progressive nets train faster but they also perform better than the baseline.

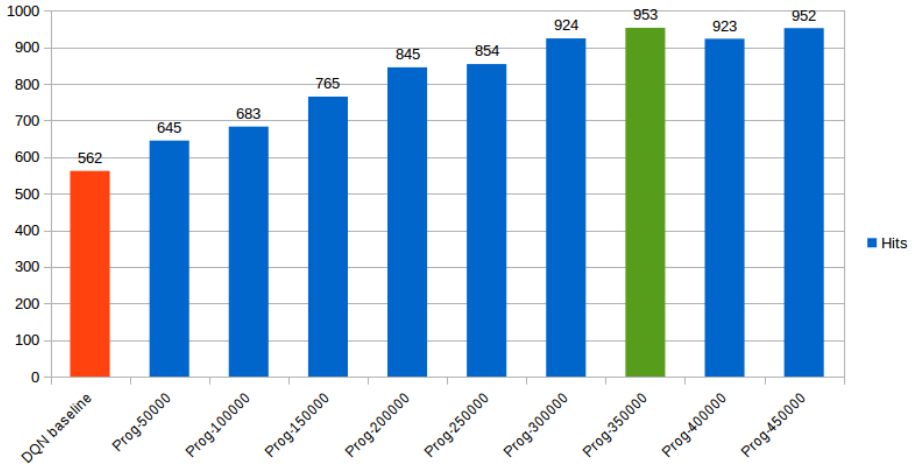


Figure 5.12: Switched target and gripper environment. The model improves until plateauing.

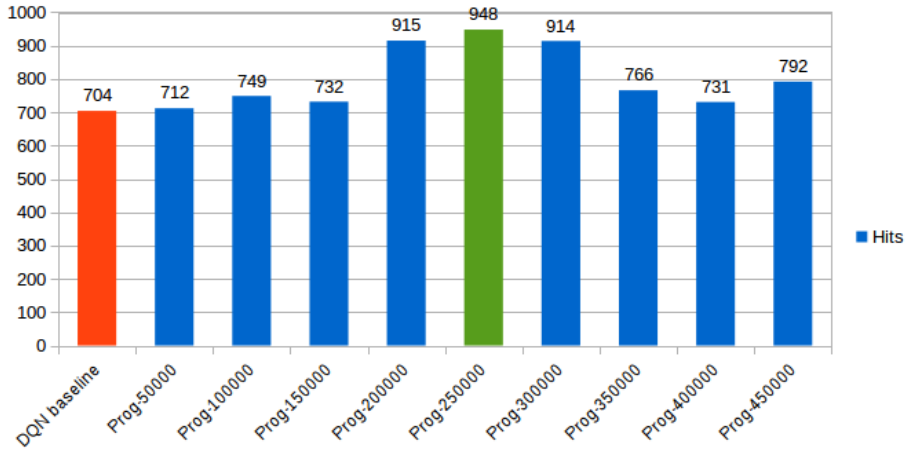


Figure 5.13: Noisy environment. Oscillations likely due to the noise.

5.2.2 Domain adaptation networks

The baseline supervised learning model lags behind the baseline DQN. It is not surprising that it also does not perform as well when transferred. What is noteworthy is that it performs worse in two out of the three modified environments when compared to a model trained only on a few samples from target domain. Of even bigger significance is that Domain adaptation improves performance when learning in a noisy environment by a not insignificant 35%.

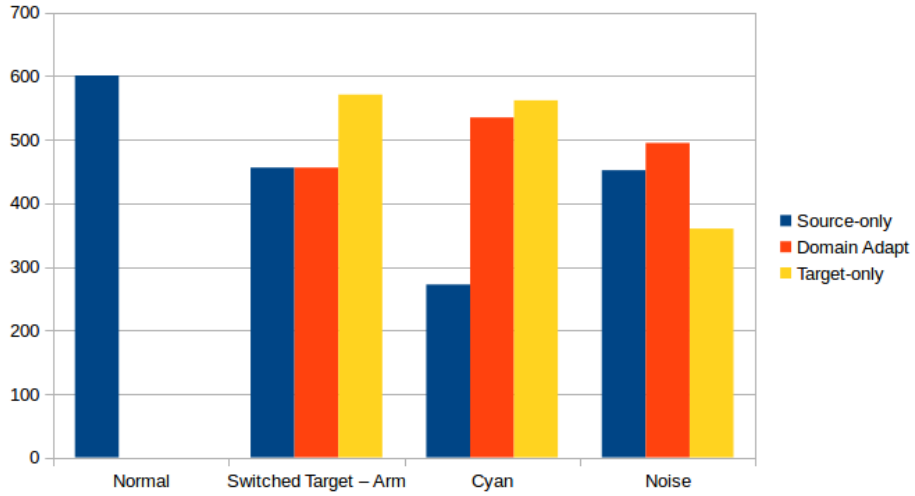


Figure 5.14: Domain adaptation achieves target accuracy improvement over training just on the target domain only in the case of a noisy environment

5.2.3 Domain alignment networks

As with Domain adaptation, Domain alignment improves only in the case of a noisy environment. The gain in accuracy is also higher than Domain Adaptation at 48%. In the other two environments, the model again loses ground against a network trained just on the target domain.

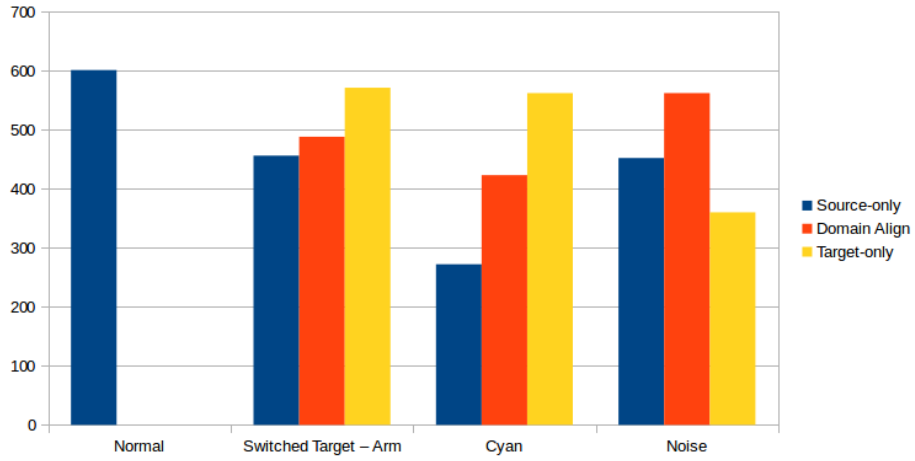


Figure 5.15: Domain alignment achieves bigger improvement over training in a noisy environment but still falls short in the other two domains.

Domain alignment is sensitive towards the initial choice of weak pairings since they are kept fixed throughout the training. This makes it unreliable when the two domains of data are initially very different and not easy to map to the same space.

5.2.4 Supervised versus reinforcement

Baseline

When comparing baselines scores, the reinforcement learning model has the upper hand both in performance in the original environment and ability to handle different environments without any modification.

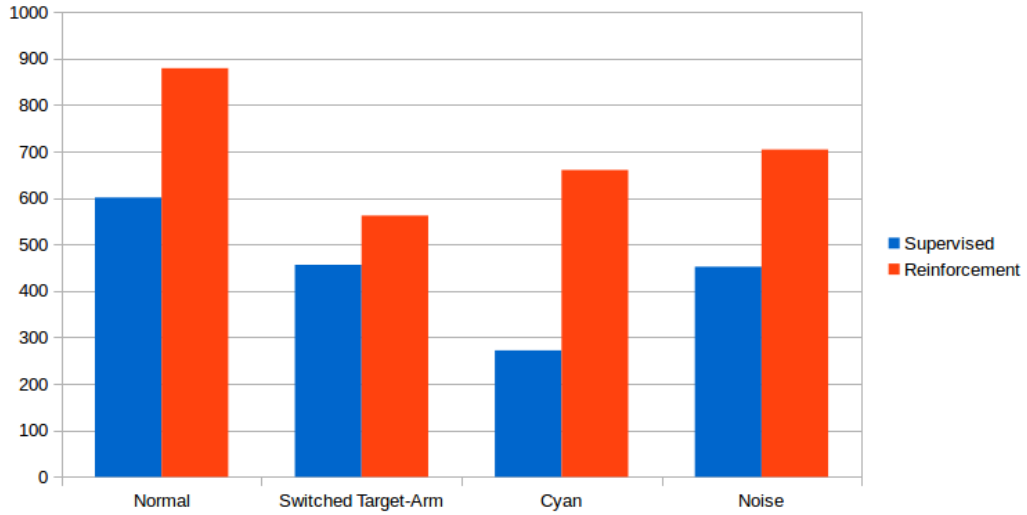


Figure 5.16: The base DQN model outperforms the supervised baseline in the number of targets hit under all conditions.

However, there are a number of factors which prevent the supervised networks from competing in terms of performance. Two of them are straightforward - increasing the data sample training size and expanding the network both vertically (more layers) and horizontally (more neurons and filters per layer). Two other factors are dependent on the environment. First is the complexity which arises when modelling inverse kinematics in an environment where an individual pixel is a deciding factor. If we were to scale up the resolution of the environments, such that individual pixels matter less, I suspect that supervised models will begin to bridge the gap. This is one reason why I expect performance to improve when training on a higher fidelity 3D simulation and transferring on the real world. Switching from a discrete output space to a continuous e.g. velocities of joints instead of discrete angle changes might also lead to a performance increase as there would be less opportunities for the agent to get stuck alternating between two states.

Transfer

Although the supervised network has lower baseline performance, a good transfer rate achieved by either Domain adaptation or Domain alignment could improve it.

This is not the case in two out of three scenarios. With the exception of noisy environments, where both techniques achieve mid-to-high improvement, transfer actually decreases performance compared to only training on the target domain. Progressive network perform better than the standard DQN in their respective environments while all supervised models perform worse than their baseline.

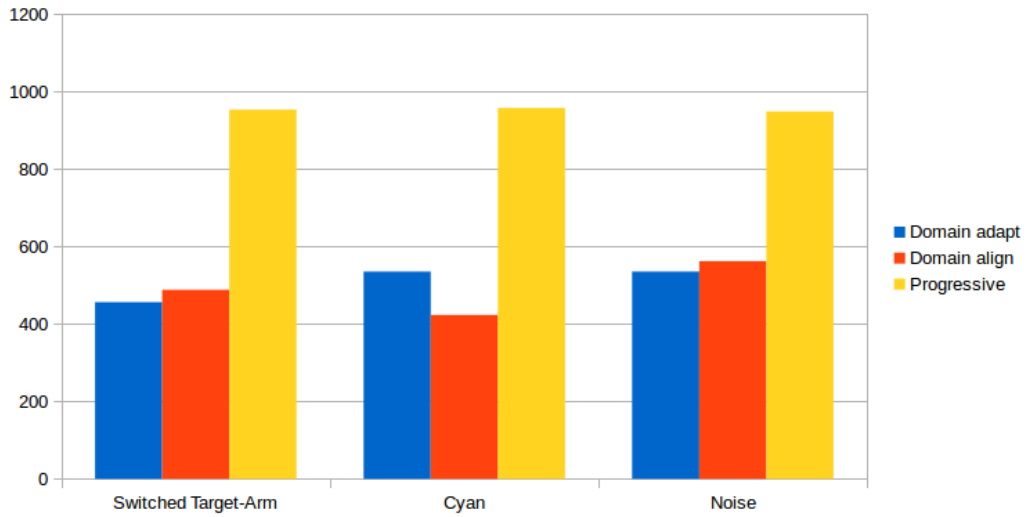


Figure 5.17: Progressive networks start learning from a higher baseline and retain their performance edge against the supervised approaches

It important to note that while the supervised training methods didn't produce desirable results in two of the modified environments, the modifications are somewhat stark and artificial. A more complex transfer-learning model such as one transferring between a 3D simulation and reality is unlikely to face such radical domain shifts between select features of the two domains. A much more likely domain shift scenario is one where every feature gets shifted by roughly the same amount. Adding random noise to the frame is a way to simulated this gradual shift. Under those circumstances, both Domain adaptation and alignment have demonstrated a sizeable boost to accuracy.

Chapter 6

Future Work and Conclusion

6.1 Future work

6.1.1 Issues stemming from low-dimensionality

Both supervised learning models studied in this project have shown great difficulty in successfully learning the policy of the inverse kinematics engine. I have identified two issues. One is that due to the low resolution of the learning environment a lot of situations arise where a single pixel is the only difference between two images but the optimal actions which inverse kinematics would provide are different. In order to handle such minute details the models would have to be extremely complex given the nature of the environment, be trained on a large dataset and take days of training on a GPU which is impractical for such a simple task. Second is the fact that it is possible that the policy of the inverse kinematic engine itself is too complex to be grasped without a deeper, and therefore more complex model which again brings us to the already mentioned issues.

DQN and by extension progressive networks do not suffer from this drawback. The layers in this implementation are both few and small. The trade off is that the reinforcement models are far from optimal, often performing more than twice as many actions as the optimal inverse kinematics solution.

6.1.2 Learning in three dimensions

An important next step is the application of transfer learning from three dimensional simulated domains to the real world. Developments in that direction have been made by DeepMind[23] using a slightly modified version of their progressive net model. There are a number of issues which need to be addressed before this application becomes viable beyond research use. First, a good set of simulation engines balancing performance and fidelity must be established. Fidelity is important as the farther the simulated domain is from real life, the greater the challenge posed for transfer learning methodologies. Performance is important when it comes to training in simulation. A life-like simulation isn't of much use if you are only able to obtain a small number of images to train on every second. For tasks which do not require

precise control such as grabbing non-fragile objects a game engine such as Unity [1] might be the correct choice as its optimised for performance and has better visual fidelity than most physics simulators[2].

Figure 6.1: Simulation or real life?[24]

Second - the complexity of the state space, even of 3D simulations, is vast. This limits the current application of reinforcement-trained models to simple tasks like locating or picking up an object. In order to handle more complex tasks researchers would have to tackle the challenge of setting up reward schemes that will not reward agents too infrequently and risk the agent not learning or too frequently and have it learn an ineffective policy.

6.1.3 Improving network performance

A common theme in the techniques explored in this project is that the underlying models, trained in simulation, are comparatively imprecise when put against traditional robot control systems. Transfer learning techniques are only as good as the models they are transferring from. Better reinforcement learning algorithms such as Deep Mind's Asynchronous Advantage Actor-critic[25] (A3C) provide better performance and faster convergence than DQN.

6.1.4 Adapting ideas to a reinforcement learning setting

Two of the models explored in this project - Domain adaptation and Domain alignment are not well suited for a reinforcement learning context since they rely on training in both domains simultaneously which is hard and carries risk when considering real world robotics. However, the ideas in those papers of anchoring weights together and bringing closer together representations of both domains are worthy of further exploration in a reinforcement context. For example, implementing a domain confusion loss between columns in a Progressive network might help them converge faster and generalise better at the expense of requiring images from the simulation when training a column which operates in the real world.

[1] `https://unity3d.com/`
[2] Left is the real image

6.2 Conclusion

By not focusing on the optimal solutions the DQN reinforcement algorithm avoids the pitfalls of the supervised inverse kinematics models. Using a much smaller network, DQN achieves a respectable result even if its not the most precise. However, in robot control precision can be an important factor. In those cases, a very deep supervised model combined with a high-fidelity environment may be able to achieve precision beyond the scope of reinforcement learning.

When it comes to transfer ability, compared to Progressive networks, Domain adaptation and Domain alignment are more limited when operating in domains where the domain shift is very large yet affects only some objects. However, in the case of noise where the domain shift is smaller and more uniform across all objects in the frame both techniques showed significant ability to transfer. Domain adaptation and alignment show potential for a more narrow transfer between somewhat close domains, especially when combined with a more favourable environment and a powerful base model. Progressive networks in combination with DQN demonstrate an enviable ability to transfer and build upon previous knowledge, not only quickly reaching baseline performance but surpassing it.

Bibliography

[1] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," *CoRR*, vol. abs/1606.04671, 2016. [Online]. Available: http://arxiv.org/abs/1606.04671 pages 7, 21, 41

[2] E. Tzeng, C. Devin, J. Hoffman, C. Finn, P. Abbeel, S. Levine, K. Saenko, and T. Darrell, "Adapting deep visuomotor representations with weak pairwise constraints." pages 7, 23, 25

[3] A. Rozantsev, M. Salzmann, and P. Fua, "Beyond sharing weights for deep domain adaptation," *CoRR*, vol. abs/1603.06432, 2016. [Online]. Available: http://arxiv.org/abs/1603.06432 pages 7, 25

[4] Wikipedia, the free encyclopedia, "Artificial neural network," 2013. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Colored_neural_network.svg/300px-Colored_neural_network.svg.png pages 10

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980 pages 11

[6] G. Hinton. (2012) Neural networks for machine learning lecture 6a. [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf pages 11

[7] Tarin Ziyaee, "Deep neural network," 2016. [Online]. Available: https://stats.stackexchange.com/questions/234891/difference-between-convolution-neural-network-and-deep-learning/235265 pages 13

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf pages 13

[9] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436–444, 2015. pages 13

[10] S. Siriwardhana. Why convolutional neural networks. [Online]. Available: https://www.linkedin.com/pulse/why-convolutioanl-neural-networks-shamane-siriwardhana pages 14

[11] G. G. I. M. Program, "8.2. convolution matrix." [Online]. Available: https://docs.gimp.org/en/plug-in-convmatrix.html pages 14

[12] Wikipedia, the free encyclopedia, "Max pooling," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network/media/File:Max_pooling.png pages 15

[13] F. Shaikh, "Simple beginner's guide to reinforcement learning and its implementation." [Online]. Available: https://www.analyticsvidhya.com/blog/2017/01/introduction-to-reinforcement-learning-implementation/ pages 15

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. pages 17, 18

[15] L. Torrey and J. Shavlik, "Transfer learning." pages 18

[16] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" *CoRR*, vol. abs/1411.1792, 2014. [Online]. Available: http://arxiv.org/abs/1411.1792 pages 19

[17] Math Works, "Inverse kinematics." [Online]. Available: https://www.mathworks.com/help/examples/fuzzy_featured/win64/xxinvkine_angles.png pages 19

[18] M. LLC. (2000) pygame.org. [Online]. Available: http://pygame.org/docs pages 19

[19] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A kernel two-sample test," *Journal of Machine Learning Research*, vol. 13, no. Mar, pp. 723–773, 2012. pages 19, 26

[20] Tensorflow - an open-source software library for machine intelligence. [Online]. Available: https://www.tensorflow.org/ pages 27

[21] F. Chollet. Keras: The python deep learning library. [Online]. Available: https://keras.io/ pages 27

[22] R. Juckett. (2008) Analytic two-bone ik in 2d. [Online]. Available: http://www.ryanjuckett.com/programming/analytic-two-bone-ik-in-2d/ pages 28

[23] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-real robot learning from pixels with progressive nets," *CoRR*, vol. abs/1610.04286, 2016. [Online]. Available: http://arxiv.org/abs/1610.04286 pages 48

[24] Rockstar entertainment, Xilandro, “Rstarhancer photorealism mod.” [Online]. Available: https://www.gta5-mods.com/misc/r-hancer-graphics-mod pages 49

[25] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783 pages 49