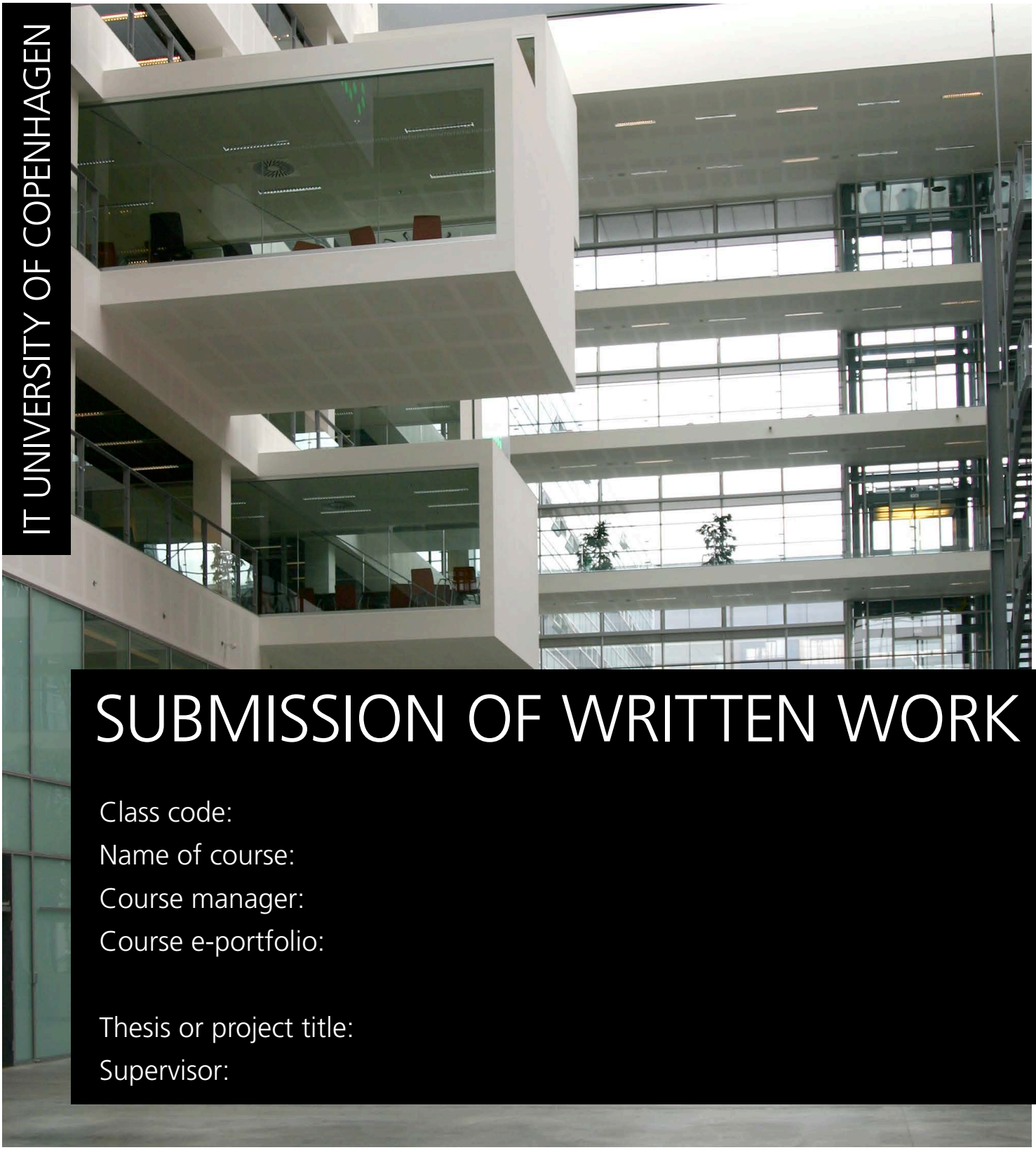
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title:

Supervisor:

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. _____	_____	_____@itu.dk
2. _____	_____	_____@itu.dk
3. _____	_____	_____@itu.dk
4. _____	_____	_____@itu.dk
5. _____	_____	_____@itu.dk
6. _____	_____	_____@itu.dk
7. _____	_____	_____@itu.dk

IT UNIVERSITY OF COPENHAGEN

Master Thesis

Combining Deep Learning and Neuroevolution for Visual Perception and Shooting in a First Person Shooter Game

Authors:
Andreas Precht Poulsen
Mark Thorhauge
Mikkel Hvilshøj Funch

Supervisor:
Sebastian Risi, Ph.D.

IT University of Copenhagen
Copenhagen, Denmark

A thesis submitted in fulfilment of the requirements for the degree of Master of Science.

December 2016

Abstract

This project investigates the potential of combining deep learning and neuroevolution in the context of a simple first person shooter (FPS) game to produce artificial intellegence capable of aiming and shooting based on raw pixel input. Deep learning is used for visual recognition, translating raw pixels to compact feature representations, while neuroevolution uses the feature representation to infer actions. Two types of feature representations are evaluated according to how precise they are approximated by deep learning, as well as how they support neuroevolution and the combination of the two. The results show limited success with combining the two techniques, as none of the feature representations perform well with both deep learning and neuroevolution. One of the feature representations is approximated very well using deep learning, which correctly predicts the position of targets that are hardly noticeable. The other feature representation is based of gradient descent with linear regression and while it shows promising results, the learning process and topology of the convolutional neural network is not optimised enough to reduce the error to a level that can adequately support the network evolved with neuroevolution. While neuroevolution produces agents capable of aiming and shooting well, their ability to generalise on other FPS games are heavily penalised by the simplicity of the FPS game in which they are trained. Overall, the results show that combining deep learning and neuroevolution is possible with the proposed approach, and it is likely that both feature representations can produce promising agents.

Foreword

We would like to thank our supervisor Sebastian Risi[1] for guidance throughout the project.

A special thanks has to be given to Muhsin Kaymak for his priceless help with getting the Unity framework to suit our needs.

The Deep Learning for Java[2] community has given much appreciated assistance with the DL4J framework - especially raver119[3], AlexDBlack[4] and agibsonccc[5].

The code for the project can be found at `https://github.com/Prechtig/FPSAgent` and is published under the GNU General Public License v. 3.0[6].

Four demonstration videos are available, showcasing:

- AR with ground truths at `youtu.be/-7dXk2JVJ_4`
- AR with VRC at `youtu.be/3qO9vd3SZqM`
- VPR with ground truths at `youtu.be/6PqhAdITsZo`
- VPR with VRC and a heatmap overlay, depicting the VRC's classification at `youtu.be/UNZIhHow4iw`

Section 1 explains the abbreviations AR, VPR and VRC.

[1] `http://sebastianrisi.com/`
[2] `https://deeplearning4j.org/`
[3] `https://github.com/raver119`
[4] `https://github.com/AlexDBlack`
[5] `https://github.com/agibsonccc`
[6] `https://www.gnu.org/licenses/gpl-3.0.en.html`

Contents

List of Figures and Plots

List of Tables

1 Naming conventions

Name	Definition
Visual recognition component (VRC)	The component responsible for creating a feature representation of an image. It is implemented as a convolutional neural network in this project.
Action inferring component (AIC)	The component responsible for inferring actions from the feature representation of the visual recognition component. This is implemented as a neural network evolved with neuroevolution in this project.
Pipeline	The combination of the VRC and the AIC component, translating the visual state to action.
Translates per second (TPS)	The number of times per second the state of the game is passed through the AIC or the pipeline to infer an action.
Angular representation (AR)	A representation of the visual state based on relative angles between the agent and the target, described in section 4.3.1 on page 24.
Visual partitioning representation (VPR)	A representation of the visual state, based on partition classes, described in section 4.3.2 on page 26.
Agent	The game entity that the developed AI controls.
Target	The game entity that the agent is rewarded for shooting.
Tap-fire	Shooting more accurate with a short delay between shots in order to reduce the penalty that weapon recoil has on accuracy.

2 Introduction

The field of artificial intelligence for games has seen significant advancements in the last few years, using new combinations of existing algorithms. Deep neural networks has been a central component in these advancements, and has proven to be a powerful logic representation, capable of solving a wide range of problems. Google DeepMind's AlphaGo [1] used deep neural networks and Monte Carlo tree search to achieve remarkable results, beating a professional Go player for the first time in history.

While the fundamental algorithms behind these solutions have existed for decades, the driving factor has primarily been combining and adjusting them to fit the problem domain, as well as having large amounts of data and computational resources available. This naturally raises the question of what the potential of these algorithms is, and which problems they can solve. Inspired by the work of both Koutník et al. [2] and Chenyi Chen et al. [3], who are experimenting with feature detection in convolutional networks for autonomous driving, and neuroevolution in visual decision making in a car racing simulation game, this project experiments with creating a visual first person shooter (FPS) agent using both neuroevolution and convolutional neural networks (CNN) as illustrated in figure 1.

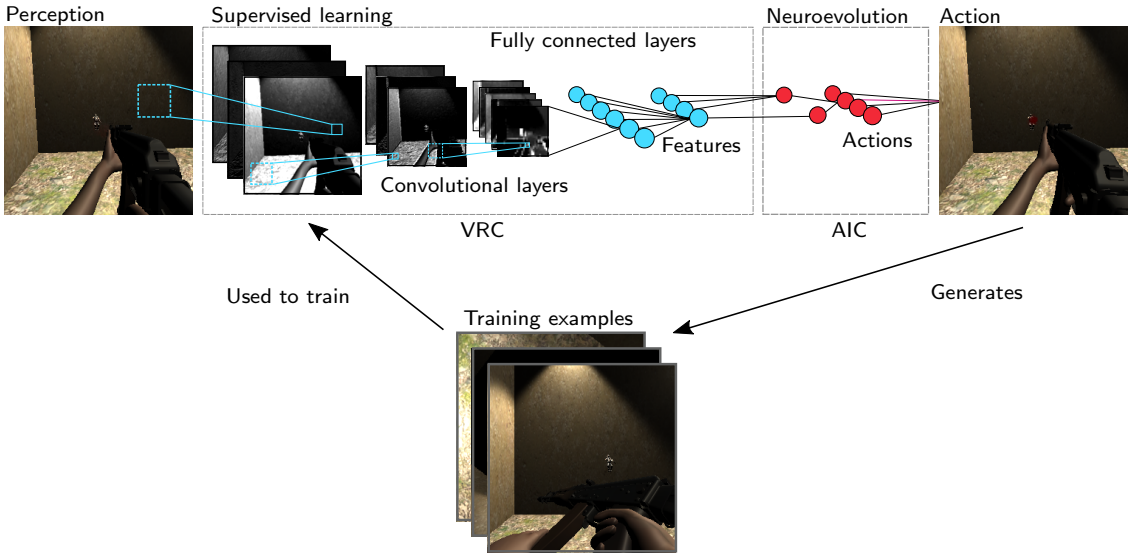


Figure 1: The combination of supervised learning and neuroevolution translates the visual state to actions.

The combination of the two techniques are an unexplored area, which could have advantages compared to both of the car controlling approaches. Using neuroevolution to decide the actions of the agent has the advantage, that the problem is solved without having to explicitly program the rules, thereby potentially consuming fewer development resources and allowing evolution of novel behaviour. The visual recognition component (VRC) solves the problem of dimensionality, as neuroevolution often proves too time consuming for high dimensional input, as seen in figure 52, were neuroevolution is tested with a 28x28 greyscale image as input. Aiming and shooting are interesting and realistically solvable problems, as they require precise visual recognition, and does not require long term decision making, which can prove problematic for neuroevolution.

The approach of Koutník et al. [2] has comparable advantages, as it uses neuroevolution to decide actions from the visual input, solving the entire problem of playing TORCS[7] using only neuroevolution, as clarified in section 3.5. However, the training time is a concern, as it takes almost 40 hours on an 8-core machine to evolve a controller that is capable of driving smoothly. Playing TORCS can be viewed as an easy problem to solve for neuroevolution, as it provides rapid feedback in the form of lap time and distance traveled, making it easy for the fitness evaluation to distinguish between fit and unfit networks. Therefore, the training time is a central concern in neuroevolution, and it might be addressed, by leaving out the CNN when training, which is possible using our architecture. A CNN trained with supervised learning could also provide a better foundation for the neuroevolution in terms of evolution speed, as the feature representation of the visual state is explicitly designed for neuroevolution.

The FPS game Doom has seen several implementations of deep reinforcement learning, as explained in section 3.5. The AIs developed by this method learns how to play deathmatchs, including navigation, aiming, shooting and identification of objects of interest. This is significantly broader scope than the scope of this project, but as it infers actions from a visual state, it is somehow similar. Deep reinforcement learning has yielded impressive results in Doom, and it is likely that our approach requires significantly more development to achieve similar result. However, our approach could be advantageous compared to deep reinforcement learning, as the visual recognition component(VRC) and the action inferring component(AIC) is trained separately. This could lead to more application opportunities in the field of real-world robotics, as reinforcement learning directly from visual perception to action is challenging due to the difficulty of simulating the real world with realistic visual inputs.

[7]The Open Racing Car Simulator

2.1 Artificial intelligence in first person shooter games

In 2015 the FPS game genre was the most popular game genre by sales in the United States [4] constituting 24,5% of the total video game sales. In popular FPS games, AI is frequently used to control enemies, and the replay value of a game reliant on an intelligent agent (IA) as a game obstacle, is dependent on interesting IA behaviour. Interesting IAs have greater resemblance of a human player and acts both unpredictably and adaptively. IAs are often implemented using finite state machines, which models repetitive behaviour. This leaves potential for improving the game genre, by making the IA behaviour more believable. Another technicality that separates these game obstacles from human player obstacles, are the way the IA interprets the game state. IAs of FPS games mostly uses the underlying game state as input, such as the coordinates and relative angles and distances of entities of interest. It is far simpler to determine an appropriate action from the game state, than reading the visual output of the screen. This design choice places a gap between the IA and the players understanding of the game state. An enemy in a dark corner or covered in smoke, might be hard to notice for a human player, but easily discoverable for an agent knowing the coordinates of enemies within sight. An IA accessing only the information accessible by a human player could possibly be more believable and more entertaining to play against.

2.2 Research question

The goal of this thesis is to clarify, how deep learning and neuroevolution can be combined to create AI for a FPS agent capable of aiming and shooting, using only the visual output of the game as input to the agent. The work of this thesis aims to contribute to future AI implementations, both in the world of AI for games and in the world of AI for real robotics. As a single focused research question, it is formulated as:

- How can supervised deep learning and neuroevolution be combined to create a visual FPS agent capable of aiming and shooting?

As a part of answering the question above, we answer the following question:

- Which feature representation of a visual partially-observable state makes the combination of neuroevolution and supervised deep learning perform well?

The research questions are answered by implementing two VRCs using deep learning with different feature representations, as well as two different AICs using neuroevolution and experimenting with them combined and individually. We measure how

well the VRCs estimate, and how well the AICs perform with the feature representations to fully analyse the advantages and disadvantages of using either representation and assert how well deep learning and neuroevolution are suited to solve the problem of shooting and aiming in a FPS. Furthermore, we train the VRCs on smaller samples of data to estimate the minimal data volume required to get the VRCs to generalise, as to assert how much labelled data a real-world application of the VRC requires. To measure how well the VRC generalises to other FPS games and real world scenarios with differing graphical settings, we vary the light settings of the game.

2.3 Content

The background explains the theory behind the results and gives an insight to the reader who is unfamiliar with neural networks, neuroevolution or gradient descent. Furthermore it briefly outlines related research, and the FPS game that the project uses as test bed. The analysis describes the rationale behind and the requirements for the proposed solutions, as well as how they compare. The approach section describes the details of the implemented solutions and how they were trained. The experiments section describes the experiments and explains the rationale behind. The results presents the results of the experiments and briefly comments on the most important observations. The discussion reflects on how our results can be interpreted, how well the experiments answer our research question, and how well the implemented solutions would fare under different circumstances and in more realistic virtual or real world environments. The future work section suggests additional work that could be relevant if the ideas of this project should be taken further. Finally, the conclusion answers the research questions to the extend that the results allow.

3 Background

This section explains the theoretical framework of the project, related studies and the first person shooter game that the AI is applied to.

3.1 Artificial neural networks

An artificial neural network (ANN) is a computational model, capable of approximating any continuous function to any precision [5]. Inspired by biological neural networks, it is composed of mathematical functions known as neurons, computing a non-linear function over the weighted sum of its input. To form a network, the neurons are connected with each other as a directed graph, such that the output of one neuron is fed into another neuron. As the arrangement, connectivity and number of neurons can be varied, ANNs can have very different sizes, shapes and capabilities.

3.1.1 Artificial neurons

The output of an artificial neuron is dependant on the input signals, the activation function, and its weights and bias. Each input signal is weighted individually. Let x be the vectorised input signals of the neuron, w be the vectorised weights, and b be the bias. Then the weighted input for the activation function, z, is calculated as $z = x \cdot w + b$.

The flexibility of the neuron model comes from the free parameters of the network - the weights and the biases. The relation between neurons are modelled by the weights, and the bias models the activation threshold of a neuron.

Depending on the purpose of the network, a neuron can use a wide variety of activation functions. The choice of activation functions can significantly influences the time required for training, as well the output range of the network. A frequently used activation function for training deep ANNs is the rectified linear function(ReLU), $f(z) = \text{Max}(0, z)$. The softmax activation function is used by the neurons in the highest layer of networks trained for classification tasks, where the output of the network is a probability distribution summing to one. Let z_j be the weighted input of the j'th neuron of the output layer, then the activation of the neuron is:

$$a_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

This activation function ensures that the output of the last layer sums to one, regardless of the weighted input signals, and is used in conjunction with the negative log likelihood cost function described in section 3.4.1.

The identity activation function($f(x) = x$) is used for networks trained for regression tasks. This allows the network to output in the range of $]-\infty, \infty[$ and avoids the dying ReLU problem in the output layer, as explained in section 3.4.4.

The ANNs evolved with neuroevolution uses the sigmoid activation function, defined as $a(z) = \frac{1}{1+e^{-z}}$.

3.1.2 Single-layered networks

In a fully connected feed-forward ANN with a single hidden layer, as shown in figure 2, the topology is based of the number of neurons in the hidden layer, as well as the input and output dimensions of the approximated function $f : \mathbb{R}^n \to \mathbb{R}^m$. The first layer is an input layer with n neurons, each connected to each neuron in the hidden layer. The neurons in the hidden layer computes their activation based on the output of the input layer, and feeds it forward to the output layer with m neurons.

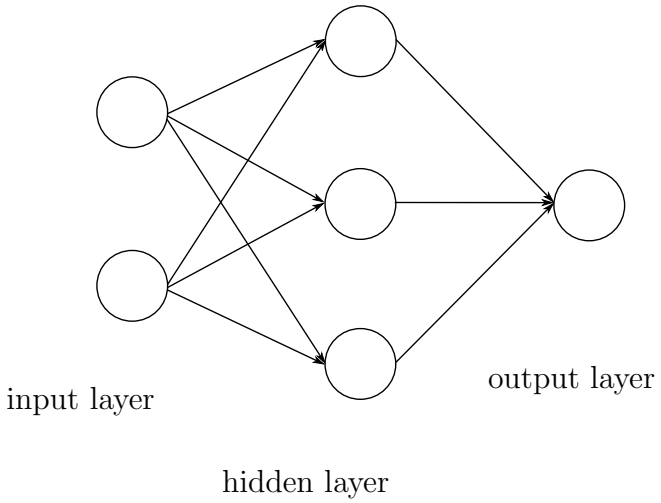


Figure 2: An example of a simple feed-forward topology for a network approximating a function of the form $f : \mathbb{R}^2 \to \mathbb{R}^1$

The neurons in the output layer computes the final output of the network based on the output of the hidden layer. This topology is complex enough to theoretically approximate any continuous function, with the error of the approximation being smaller with more hidden neurons.

3.1.3 Deep neural networks

Networks with multiple hidden layers, also known as deep neural networks, have the advantage over single layered networks, that they can approximate functions using fewer computational units than single layered networks [6]. In a binary circuit, the d-bit parity problem[8] can be solved using $O(d)$ logic gates with a network of depth $O(\log(d))$, requiring an exponential amount of logic gates [7] in a single-layered network. The topology of the image classification network in [1] demonstrates the expressive potential of deep feed forward networks. The effect of depth in convolutional neural networks have been widely studied, and research has repeatably shown, that increasing depth of the network can benefit accuracy of predictions [8].

However, deep networks also suffers from topology-specific problems. The vanishing gradient problem is a common problem in deep networks, that is specific to the gradient descent algorithm, described in section 3.4.4. Depending on the initialisation of the network and the activation function, the vanishing gradient problem can slow down learning in the lower layers of the network, making the entire network learn slowly. The number of free parameters in multi-layered networks are usually higher than their simpler single-layered counterparts, and a high number of free parameters introduces a greater potential of overfitting as described in section 3.4.3.

Recurrent connections and non-layered topologies are common in ANNs developed by TWEANNs[9], as exemplified in figure 3. In the previous layered example, the neurons were activated three times to get the output of the network. In a non-layered recurrent topology, the neurons can be activated any number of times, known as discrete time steps, as the output is cycled through the network. This models a short-term memory, as the network can keep cycling previous inputs.

3.2 Convolutional neural networks

CNNs are deep feed-forward networks inspired by the neural connectivity in the visual cortex of the brain, and operates on input data with a spatial relation between the inputs, such as images or text. A CNN usually consists of convolutional layers, pooling layers and fully connected layers, and varies greatly in topology.

The number of free parameters of a convolutional layer is not dependent on the number of inputs, which is an important advantage over fully connected layers. While convolutional layers have much fewer inputs, and therefore trains faster, they cannot

[8]The d-bit parity problem is solved by determining whether a sequence of d bits have an even or odd number of bits with a value of 1

[9]Topology & Weight Evolving Artificial Neural Network algorithms

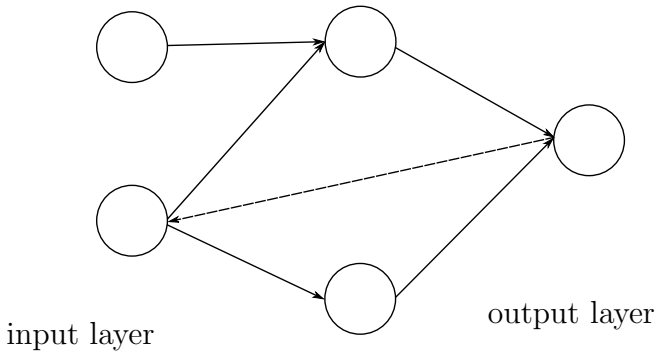


Figure 3: An example of an ANN with a recurrent connection (the dotted line) in a less constrained topology.

approximate the same functions as fully connected layers. A successfully trained convolutional layer detects a number of features, such as lines or blobs of colour and can detect such features independent of their position on the image. Convolutional layers and fully connected layers are therefore combined, to both be able to detect image features efficiently and learn abstract patterns from the features.

While CNNs can be applied to data of any dimensionality, the following explanation is in the context of three dimensional image data as input, with the colour channels being the third dimension.

3.2.1 Convolutional layer

A convolutional layer is defined by its convolutional operator, receptive field, stride, zero padding and number of filters. It produces a number of two dimensional filters, stacked on top of each other to produce a three dimensional output volume.

3.2.2 Filter

A filter performs a convolution operation multiple times on the input volume. The receptive field defines the data points used as input to the operation. The receptive field in figure 4 has a shape of 5x5 and fully extends through the depth of the volume. In an image with three channels, the operator would take 5x5x3 data points as input to produce one term of the two dimensional output of the filter. The stride defines the amount that the receptive field is shifted to select the input used to calculate the next term of the output. Zero padding is the number of zeros added to the edges of

the output filter. This is to shape the output and ensure that the edge data points are input to multiple convolution operations.

The convolution operation is defined by a set of weights, a bias and an activation function. In the same manner as a neuron in a fully connected network has weighted connections to the neurons in the previous layer, a convolution operation has weighted connections to the data points in its receptive field. The difference is, that the weights and the bias of the convolution are the same for every operation within the same filter. This is known as parameter sharing, and drastically reduces the number of free parameters within a layer.

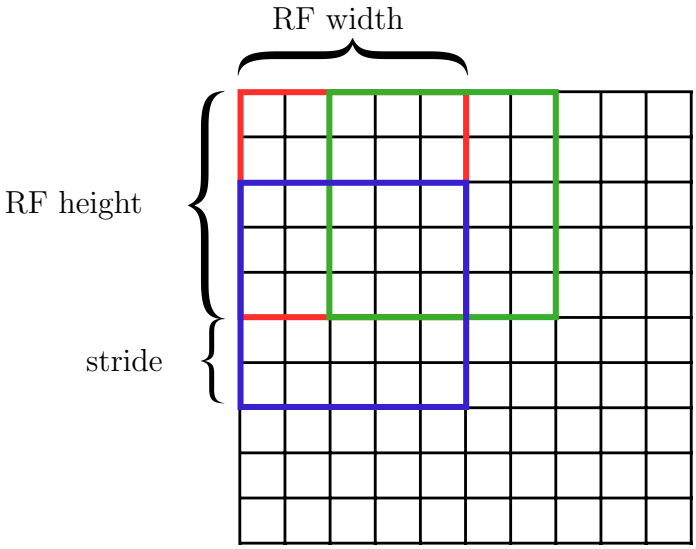


Figure 4: The input to the convolution is spatial in connectivity. The red square marks the pixels used as input to the convolution, to calculate one term of the resulting output volume. The blue and green squares mark the pixels used as input to calculate the neighbouring terms. RF is an abbreviation of receptive field.

3.2.3 Pooling

Pooling layers are used to reduce the size of the output volume. It functions like a convolutional layer, with the convolutional operation replaced with a max or average function and not extending through the full depth of the input volume. Consequently, the pooling layer has no free parameters, and is only defined by its receptive field and stride. As an example, a pooling layer with receptive field and stride of 2x2, receiving an input volume of 16x16x40, outputs a volume of 8x8x40.

3.2.4 Topologies

Convolutional neural networks have been successfully trained and applied using many distinct topologies, but does not resist some generalisation. The topologies are always layered and the convolutional and pooling layers are always lower than the fully connected layers. A common topology of a CNN alternates convolutional and pooling layers, and ends with a number of fully connected layers and an output layer. The CNN trained in [3] does not use pooling, but reduces the size of the input data by applying a stride greater than 1 in the convolutional layers.

The ability of the CNN to recognise visual features depends on its effective receptive field. A CNN with a single layer using a receptive field of 3x3 is unable to produce feature maps detecting visual features spanning more than a 3x3 area. However, multiple convolutional layers form an effective receptive field wider than any of the individual receptive fields, as illustrated in figure 5.

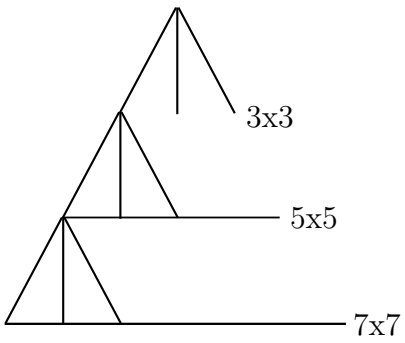


Figure 5: A cross-section of three convolutional layers with receptive fields of 3x3 and stride of 1. The output of the lowest layer is used as input to the next layer, and the receptive field of the entire network is widened. Together they form an effective receptive field of 7x7.

Consequently, the full topology of the CNN determines its ability to recognise visual features. It is worth noting, that the CNN with a single convolutional layer with a receptive field of 3x3 still would be able to recognise wider features, but the learning would be slower, as the topology is unable to take advantage of the full spatial relationship between inputs.

Google's LeNet as documented in [1] uses inception modules, which consists of several parallel convolutional layers. The results of the parallel layers are concate-

nated like filters of a convolutional layer to produce the output of the inception module. Inception modules can be viewed as an expansion of the filter idea, as having multiple concatenated parallel filters is essentially the same, except that the parallel layers can have different values of zero padding, stride and receptive field, and can be piped to produce depth within the module.

3.3 NEAT

NEAT [9][10] is an evolutionary algorithm for evolving both the topology and weights of artificial neural networks while using speciation to preserve the best and possibly new innovative networks.

NEATs ability to modify the topology of networks and make common evolutionary practices, such as crossover, work on genomes with different topologies, is what distinguishes it from many other evolutionary methods for evolving artificial neural networks. The purpose of NEAT, and similar methods, is evolving the most optimal network, with the best performance possible and a topology consisting only of the input and output nodes and connections between them.

NEAT works by continually mutating weights while steadily using complexification in order to allow more advanced behaviour to develop. This steady development of the topologies, along with speciation, allow NEAT to only complexify, if it benefits the solution. Other similar methods use pre-defined topologies based on the developers best intuition, empirical data or even random topologies to develop a satisfying network, by only mutating the weights. These methods rarely achieve an optimal solution with minimal dimensionality, as the task of pre-defining even a simple network is a very difficult task, at least if the complexity of the solution is a concern.

3.3.1 NEAT Encoding

NEAT uses a direct encoding scheme which consists of a list of node genes and connection genes. Node genes simply depict whether it is an input, output or hidden node. A connection gene describe a source and target node, weight, whether or not it is enabled and a unique innovation number. These innovation numbers are paramount for NEAT in order to do crossovers and comparisons as part of its evolutionary algorithm and to apply speciation techniques. The innovation number is a unique historical marking each connection has. It is used to quickly identify the shared ancestral traits in different networks in order to give a means of comparing

[10]Neuroevolution in augmenting Topologies

and classifying structures. When a new, previously unknown, connection is introduced in a structure, as a result of a structural mutation, is the new connection gene given the next innovation number as its identifier. Crossover in NEAT work by first identifying shared, disjoint and excess connection genes in the two parent genomes. Genes are shared if they have the same innovation number, and disjoint or excess if not. A new structure is then built from randomly inheriting shared genes, while excess and disjoint genes are inherited from the fittest of the parents. This method for working with different topologies also solves the competing conventions problem and allow the development of recurrent networks to solve non-markovian problems such as the pole balancing proble or parts of the experiments of this project.

The NEAT framework used in the paper, handles crossover / sexual reproduction slightly different. A newly produced genome will either enable or disable all disabled connections it received, whereas original NEAT, makes this decision on a connection by connection basis.

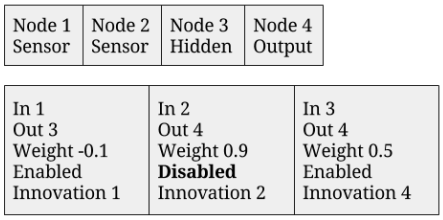


Figure 6: An example of a gene encoding of an ANN in NEAT

3.3.2 Speciation

Speciation in NEAT allows networks with very different topologies to compete, breed and evolve among other networks with largely the same topology.

A new topological innovation in a TWEANN, occurs when a structural change happens, like a new node or a new connection. Such a change is however likely to make the network perform considerably worse than before, compared to the networks in the population, which did not get such a significant change. In order to preserve new innovations, the entire population split into separate species, depending on their topology and they compete, primarily, within their own specie. This way, if a new innovation is placed in a new species, it will be allowed to evolve and tune its new structure to something fitting, and not just be discarded as a bad evolutionary step before it has even been explored.

The following formula is used to calculate the distance between genes for assigning species:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + \bar{W} c_3$$

The formula uses the number of excess genes, E, disjoint genes, D, and the average weight of the matching genes, W. The importance of these three parameters can be adjusted using the three variables $c1$, $c2$ and $c3$, while N is the total amount of genes in the larger of the two networks. There are no definitive optimal settings for these parameters, they have to be experimented with on a case by case basis. The goal of the parameters are to protect innovative changes as they adjust their weights to discover the potential of the change while also discarding structures which have proved not to work.

At each evolution step, a random genome in a species is selected to be that specie's representative, and all other genomes get their distance calculated with regards to that genome, to determine if it should be part of that species. If a genome is too far from any species, then it will be placed in a new species where it, itself will become the representative genome.

All genomes have their fitness changed, depending on how big the species it is part of is, and how different the genomes is from the other genomes in the species:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))}$$

$f'i$, is the modified fitness of the network, $sh(\delta(i,j))$ is a function returning 1 if $\delta(i,j)$, the distance between the networks, is below the threshold δ_t, otherwise it returns 0. This results in the modified fitness being lower the larger the species the genome is part of is meaning that larger species will have to perform better in order to survive as a large species.

3.3.3 K-Means speciation

The NEAT framework used for the AIC experiments in this paper does not use the aforementioned speciation strategy. It uses K-Means Clustering[11] for speciation which is an algorithm for distributing data points into K different clusters, or in this case, species. The algorithm works by first distributing all genomes into K a random clusters. Then it calculates the centre point for each cluster and continues to

[11] A description of the speciation strategy is available at: `http://sharpneat.sourceforge.net/research/speciation-kmeans.html`

distribute all the genomes into the nearest cluster. If a genome has changed cluster during the re-distribution, the algorithm starts again from calculating the centre point of the clusters as they are then and distribute the genomes again. This is done until all clusters are stable or for some fixed amount of iterations.

The centre point of a cluster is calculated as the componentwise mean of all the points in a cluster.

K-Means implementation uses a vector of a genes' connection's innovation ids and their weights to represent them in the coordinate system. Manhattan distance is used as the distance metric, which is not much different from the distance metric in the original speciation idea. Manhattan distance in the original distance metric is equal to c_1E and c_2E set to 0 and c_3 to 1.

3.4 Gradient descent

Gradient descent is an optimisation method that fits a model to a number of training examples, to learn the underlying data pattern, by minimising a cost function. In the context of image recognition, a training example is an image and associated features, such as the class that the image belongs to. The associated features are referred to as ground truths. The model subject to optimisation is an ANN.

In contrast to NEAT, gradient descent does not adjust the topology of the network. Hence, it is given a predefined topology and thus a set of parameters, known as θ, which is the set of biases of all neurons and weights of all connections in the network.

Gradient descent iterates through the set of training examples, and calculates cost through backpropagation based on the difference between the predicted features, $a(x)$, and the ground truths, $y(x)$. Euclidean loss is a common cost function.

$$C(x) = \|y(x) - a(x)\|^2$$

The parameters of the network are updated based on the gradient of the cost function ∇C with respect to θ. By knowing the gradient with respect to every parameter of the network, the parameters can be updated, such that the cost is reduced. Consider a change in cost, as described by the change in a parameter:

$$\Delta C \approx \nabla C \cdot \Delta v$$

If the change in the parameter, Δv, is chosen, such that $\Delta v = -\eta \nabla C$, where η is the learning rate, then it can be shown by substitution that C is reduced.

$$\Delta C \approx -\eta \nabla C^2$$

The learning rate is a small positive number, usually in the range of 10^{-6} to 1. If the learning rate is sufficiently small, then the cost is reduced at every parameter update, but if it set too high, then $\Delta C \approx \nabla C \cdot \Delta v$ is an inaccurate approximation of the change in cost, and the parameter update might result in a rise in cost.

The Euclidean loss cost function presented earlier applied to a single training example. The objective of gradient descent can be described as minimising the average cost for all training examples.

$$C_{Euclidean} = \frac{1}{n} \sum_{x} \|y(x) - a(x)\|^2$$

The true gradient of C is approximated by averaging the gradients calculated from one or more training examples. If parameters are updated from a gradient approximated from one example, the optimisation method is known as stochastic gradient descent. If it is approximated from a number of training examples less than the total number of examples, it is known as mini-batch gradient descent. Mini-batch gradient descent is usually preferred in practice, as it is a good compromise between computational efficiency and accuracy.

3.4.1 Cost functions

For classification tasks, where the output is a probability distribution summing to one, the negative log likelihood is used as cost function. Training examples of classification tasks are represented as a binary vectors, with 1 in the class that the input belongs to. Let y_{ij} be the j'th term of the feature vector of the i'th training example, and p_{ij} be the corresponding prediction from the network subject to training, then the log likelihood function can be described as:

$$C_{loglikelihood} = -\frac{1}{n} \sum_{i} \sum_{j} y_{ij} \ln(p_{ij})$$

Where n is the number of training examples. Explained in more simple terms, the cost of a probability distribution is exactly equal to the negative natural logarithm of the predicted probability of the correct class. Negative log likelihood is preferred over Euclidean loss for classification tasks, as it makes gradient descent converge to the optimal solution faster [10].

The Euclidean loss function described in the last section is used for regression tasks, where any output sum and any output range is allowed.

3.4.2 Momentum

Gradient descent, as described above, can be thought of as a ball rolling down a hill until it cannot go any further. However, the speed of the ball is only proportional to the steepness of the curve, and does not accelerate like its physical analogy. While adhering to the laws of physics should not be a motivation in itself, the idea of acceleration in gradient descent has two advantages. It reduces the chance that the ball gets stuck in a local minimum of the cost function, as the momentum of the ball allows it to roll over small hills of the cost function. Secondly, it reduces the number of time steps required to roll down to the bottom of a deep valley. The momentum rules used by Nesterov's Accelerated Gradient (NAG) are slightly different from classical momentum, as it modifies the velocity based on a future position of the ball, rather than the current position. NAG performs an update to the parameters based on the velocity:

$$\theta_{t+1} = \theta_t + v_{t+1}$$

In the non-modified version of gradient descent, the velocity was only based on the gradient. NAG modifies the velocity based on the previous velocity:

$$v_{t+1} = \mu v_t - \eta \Delta C(\theta_t + \mu v_t)$$

Where μ is a hyperparameter that determines the amount of momentum. It can be shown that NAG converges faster than gradient descent for differentiable and convex cost functions [11].

3.4.3 Regularisation

Regularisation in gradient descent helps preventing the model from overfitting to the training examples. Overfitting occurs, when the optimisation method reduces the models generalisation capabilities, but increases its capabilities of correctly predicting training examples. It occurs as minimising the cost function for the training examples is only an approximate representation of the goal of gradient descent, the true goal is to make the model generalise. When gradient descent reaches a certain point in training, the only way to get a lower cost is to overfit to the training examples. Early stopping is a strategy to reduce overfitting, by measuring the models generalisation strength on a data set not used during training.

A greater number of free parameters increases the overfitting potential of the model, while a greater number of training examples has the opposite effect. High weights are indicators of overfitting, as an overfitted network detects few unique features of a single example and lets them determine the entire output, to form

a memory from the parameters. Consequently, both l1- and l2-regularisation limits overfitting by reducing the magnitude of the weights. They do so, by adding a second term to the cost function, such that the change in parameters are influenced by the magnitude of the parameters. Let C_0 be the original cost function, then the cost function with l2 regularisation is,

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

where λ is a hyper parameter that dictates the strength of the regularisation. l1 regularisation is very similar, the only difference being that the weights are not squared.

Overfitting can be prevented in other ways. The more training examples gradient descent uses, the less likely overfitting becomes. In the problem domain of this project, training data is readily available, and optimising with additional training data is enough to eliminate overfitting, as seen in the performance results in section 7.

3.4.4 The vanishing gradient problem

The vanishing gradient problem is a common problem in deep learning. To understand the solution to the problem, it is necessary to understand some details of the way backpropagation calculates the gradient of the cost function. Backpropagation works by calculating the error of every neuron, and recursively using the error of the previous neuron to estimate the error of the next error, going backwards. The error of a neuron in the output layer is calculated as: $\delta_a = C_a \nabla \cdot f'(z)$. The error of the neurons in the next layer is based of a product of the derivative of its activation function. The point is, that the further the error travels backwards in the network, the more derivatives gets included in the product. If the derivative is small, the error gets small as well, which leads to small updates to the parameters, which finally leads to slow learning. The sigmoid activation function has an output range of $[0, 1]$ and the derivative is calculated as $s'(z) = s(z) \cdot (1 - s(z))$. It is apparent that the largest value of $s'(z)$ is $\frac{1}{4}$, and that the derivative gets small in the edges of the range. For examples, in 12 layer network, the error would in the best case scenario be multiplied by $\frac{1}{4}^{12} \approx 5.96 \cdot 10^{-8}$. Consequently, this activation function would result in very slow learning in the lower layers. The rectifier activation function partially solves this problem by having a derivative of:

$$f'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & z \leq 0 \end{cases}$$

This allows the error to persist through the layers. However, the error is canceled entirely if z is below zero, and the vanishing gradient problem still persists to some degree. If the initialisation of the network results in a ReLU function outputting exclusively zero for all the training examples, it dies and is useless. Consequently, avoiding dying units in the output layer might require multiple initialisations with different seeds. Google's LeNet [1] additionally addresses the vanishing gradient problem by having multiple output layers calculating outputs from only a subset of the network, thereby reducing the number of steps that the error travels to reach the lowest layers of the network.

3.4.5 Xavier initialisation

The initial values of the weights of the network is important when training deep neural networks. Too low weights causes the error signal of backpropagation to stagnate, and too high weights causes the weighted input sum, z, to become too large or too small. It is desirable to let be z within a low range, as the non-linearity of the activation function is centered at zero. Drawing the weights from a normal distribution with a mean of zero and a constant variance does not necessarily keep the weights within the desired range, as the number of inputs varies from neuron to neuron. A neuron with a thousand inputs have a larger variance on its weighted input sum than a neuron with ten inputs, if they both use the same variance. Xavier et al. [12] suggests an initialisation, where the variance is based on the number of input and output signals, respectively denoted n_{in} and n_{out}.

$$\mathrm{Var}(W) = \frac{2}{n_{in} + n_{out}}$$

This helps keep the weights within a range that makes gradient descent converge faster than with a constant variance.

3.4.6 Distribution imbalance in classification training data

According to recent research [13] having imbalanced representations of classes used for training with gradient descent can lead to poor performing networks. According to the study, the worst cases of networks trained on imbalanced data were only able to consistently classify images belonging to the most represented class, i.e. one out of a total of ten classes. The imbalance of data were at most a factor of two, i.e the most represented class having twice as many samples as the least represented class.

The complications introduced by imbalanced class distributions can be addressed by evening the distribution of classes. Overcoming the imbalances can be achieved

through different approaches.

One method is to delete data from over-represented classes until every class has an equal amount of samples. This can severely affect the amount of training data as data is deleted until every class has samples equal to the lowest denominator.

A second method is proposed in [13], showing promising results. The idea is to randomly duplicate samples from lesser represented classes, until classes are more evenly distributed. The most extreme form of this method duplicates samples until an even distribution is obtained.

3.5 Related work

In 2014, a visual agent for playing TORCS was developed using reinforcement learning to evolve a network with a CNN component [2]. The fitness functions used to train the top layer network was different than the fitness function used to train the CNN component. The top layer network adapted the output of the CNN component and successfully used it, despite the output of the CNN component being humanly incomprehensible, and based solely on the image. The TORCS AI demonstrates the strong input interpretation potential of NEAT.

Chenyi Chen et al. [3] trained a deep convolutional neural network to detect features of road images in TORCS, retextured to simulate real driving. The features included angle of the road tangent and distance to land markings. The features are similar to the AR we propose, as it estimates angles and distance. Chenyi Chen et al. trained their network with 484,815 training examples for 140,000 iterations, which is quite resource intensive. The features were translated to driving actions by a handwritten controller, which is the main difference between their and our approach. The advantage of their approach is that the desired behaviour of the agent can be formulated in detail, which is necessary when programming critical systems, like an autonomous car.

The real time shooter Doom has seen several successful AI implementations using convolutional neural networks with reinforcement learning in the Visual Doom AI competition, as for example by Michal Kempka et al. [14] and Lampre et al. [15]. The domain are similar to ours, but the training methodology differs. Many of the top performing entries for the competition used deep reinforcement learning to estimate the Q-value of states. The networks were trained with stochastic gradient descent with data labeled by reinforcement learning. This way of controlling the agent is quite different from out approach, as it is based on a state representation of the game and does not utilize neuroevolution.

Deep reinforcement learning has been applied to several different games, using

high dimensional sensory output, such as images. Volodymyr Mnih et al. [16] uses deep reinforcement learning to play 9 different Atari 2600 games, achieving state of the art results in 6 of them. One of the advantages of this approach is that it is independent of the feature representation, which can be a challenge to design manually. However, deep reinforcement learning does not perform well on games requiring long term strategic planning.

Numerous examples of applying supervised learning or evolutionary algorithms to FPS games exist [17] [18] and neuroevolution is frequently applied to games of other genre, as outlined by Sebastian Risi et al. [19]. Some of the examples uses HyperNEAT to deal with relatively high dimensional input[12], as HyperNEAT can evolve repeating connection and topological patterns. This approach is relevant for this project, and we elaborate on its application in section 9. Few of the examples, uses the visual partially-observable state. This difference is not only significant in terms of the methods required to solve the problem, but also in terms of the expected behaviour of the agent.

3.6 The FPS setting

The FPS environment used as a test bed for the pipeline is described in this section. The agent starts with a single full automatic weapon, which has 30 bullets in a single magazine. It has 6 extra magazines at its disposal. The weapon does not automatically reload when the magazine is empty, in order to force the agent to learn detecting when a magazine is empty. Since the agent does not get any input regarding bullets and magazines left, it should, optimally, learn to count how many bullets and magazines it has used. This is important in order to not waste time attempting to shoot or reload when there are no more bullets or the weapon is fully loaded, it is however possible to get very reasonable behaviour from the agent without perfect reload times.

The target spawns with 100 health points, and a single bullet does 10 damage to it, no matter where it hits. This means that 10 hits are required for the agent to kill the target. If the agent only reloads when it is out of bullets, it will have 210 bullets, meaning it could theoretically eliminate 21 targets. The agent is however not given enough time to eliminate such a high number of targets, but it is given plentiful extra bullets, to allow for mistakes, in order to increase the learning rate.

The weapon has a violent recoil which the agent has to learn how to control. The recoil is set to miss the enemy/bot already on the third shot in far the most cases and emptying an entire magazine will on average hit the target 3-4 times. As this

[12] 16x21 for Atari and 7x7 for Go, compared to 256x256 images used in this project.

will result in a very low score, the agent will have to learn how to control the recoil. The recoil is random, so the agent has to learn how to tap-fire in order to maximise accuracy. The weapon can shoot 10 times per second and the recoil is turned off for some of the experiments.

The arena the agent learns in is quadratic, with the agent spawning on one side and the targets spawning on the other, as seen in figure 7. Both the agent and the target spawn in a random x and y, while the z coordinate is constant. The first target in the arena always spawn within sight of the agent in order to improve the learning rate. The arena is outfitted with 3 lights to create different visual representations of the target. One light is in the middle of the agent side in order to create differing lightning images of how the agent sees himself. The other two lights are placed on the side of the target, in a way which provide very varied light settings on the target, to test the VRC under different conditions.

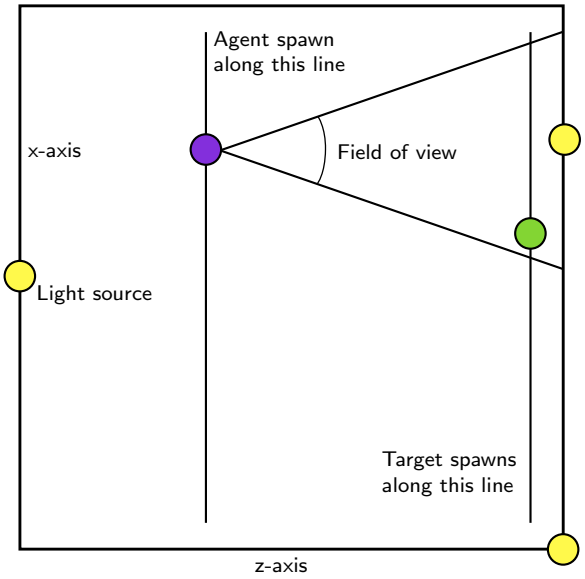


Figure 7: The arena as seen from above. This illustration omits the y-axis. The lines represent planes, as the agent and the target spawns randomly along both the y-axis and the x-axis.

4 Analysis

Creating a fully functional visual FPS agent requires solutions to a range of different problems. To narrow the scope of the solution, the functionality of the proposed solutions is limited to only a subset of the required functionality to play a full-fledged FPS game, as Counter Strike or Quake. Developing a FPS agent capable of playing like a human player requires lots of different functionality as for example map navigation, adaptation of team role and identification of sound. The focus of the solution is aiming and shooting, and the proposed solutions only attempts to solve this problem. The problem includes dealing with weapon recoil, reloading properly and aiming and shooting at the target.

4.1 Actions

The agent should be able to aim, shoot and reload. Therefore, the output of the AIC is turn horizontally, turn vertically, shoot and reload. The turn actions are analog, such that the greater the output, the faster it turns. Shoot and reload are binary actions, that triggers when the output is above a specific threshold. Shooting takes precedence over reload, and turning can be done while shooting or reloading.

4.2 Granularity of control

The frequency of which the AI makes decisions is a parameter that shapes the potential and performance requirements of the solution. The finest level of control is achieved by parsing every single frame of the game to infer an action. This is performance intensive, as the process of parsing the visual output through the VRC requires significant amount of computational resources. Using a control frequency lesser than the number of frames reduces the potential reaction speed of the agent and the overall smoothness of its behaviour. The approach of Michal Kempka et al. [14] is also vision based, and repeats the chosen action for k frames. We chose the same approach, as the flexibility of skipping frames reduces the hardware requirements. However, as we are unable to guarantee the FPS of the game and as the game framework does not allow control on a frame-by-frame basis, we refer to granularity of control as translates per second. Translates per second is the number of times the pipeline or AIC translates the state to an action per second.

Some of the functionality requirements of an agent capable of playing a game like Counter Strike might require higher level of control, such as deciding the long term strategy plan or navigating larger maps. This is not in the scope of the proposed solutions.

4.3 Feature representations

Recall that the feature representation is the integration point of the VRC and the AIC, and that it is the output of the VRC and the input of the AIC. The feature representation of the visual state should allow the AIC to locate an enemy, aim and shoot it, while having as few dimensions as possible, as the evolution speed is a central concern. Additionally, the VRC has to be able to learn the features from training examples through supervised learning.

4.3.1 Angular representation

The angular feature representation unambiguously defines the position of the target on the screen. This representation has two angles, a distance and a binary output indicating whether the target is within sight. Figure 8 shows the vertical angle, given by the angle between the current facing of the agent and a projection of the vector from the agent to the target. The projection is onto a plane determined by the vector of the current facing and the up-vector.

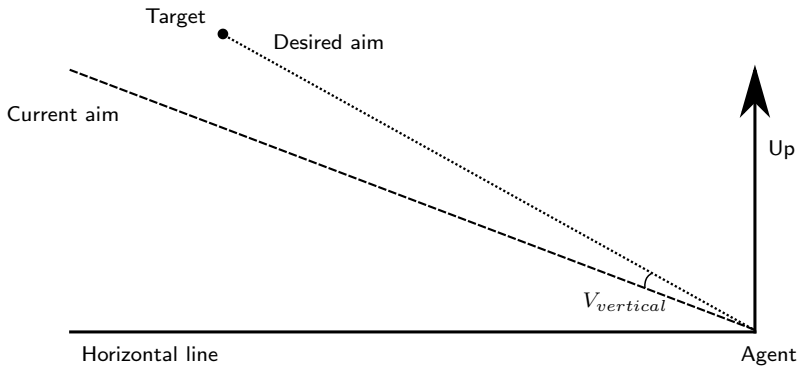


Figure 8: The relative vertical angle.

The horizontal angle is calculated in the same manner, except that the angle is calculated based on the projection of both vectors onto the horizontal plane. These two angles allows the AIC to infer the aiming directly. The distance is included to allow for changing shooting behaviour based on distance. For example, if the agent shoots a full automatic rifle at a long distance, it might be better to fire separate shots than using automatic fire. The binary indication of whether there is a target within sight, has the purpose of clarifying whether the angles reflects the position of a target, or just assumes default values. The scheme is visualised in figure 9.

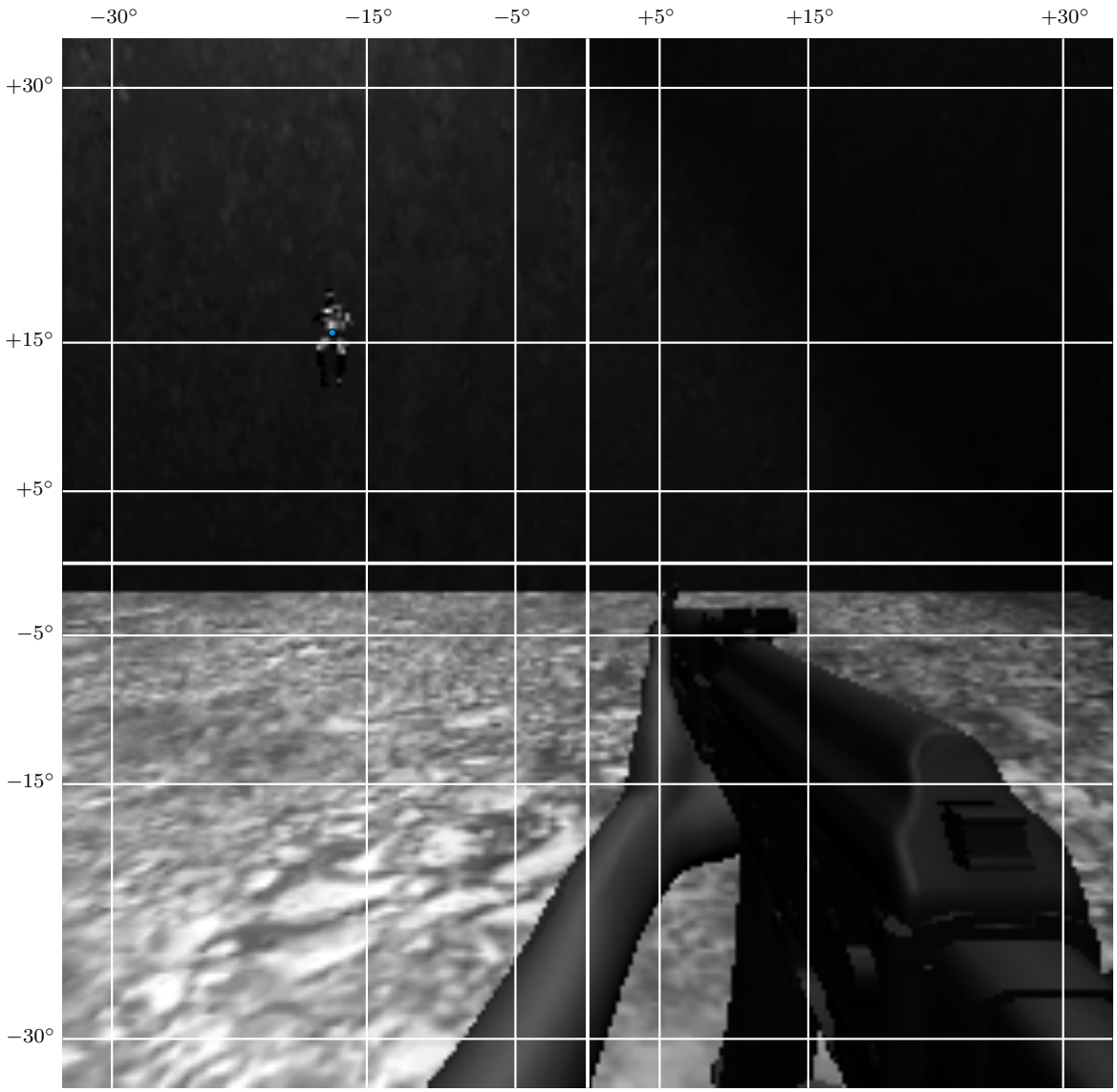


Figure 9: The vertical and horizontal angles define a position on the screen. The field of view is 65° and the target is at a horizontal angle of −17.17° and a vertical angle of 15.05° in this example.

4.3.2 Visual partitioning representation

The visual partitioning representation ambiguously indicates the position of the target on the screen. It defines the position of the target as a classification task, where each point on the screen belongs to a class bounded by a square as shown in figure 10.

Figure 10: The position of the target is interpreted as the square of which his centre of mass(marked by the blue dot) is located.

As the target can be visually present in multiple squares, the correct square is defined as the square that contains his centre of mass. The partitioning is finer in the centre of the screen to allow for fine adjustments of the aim. The notation of the partitioning scheme is defined as the number of partitions on the vertical and horizontal axis of the whole screen, followed by the number of partitions on the horizontal or vertical axis of the centre of the previous partitioning. The scheme is exemplified in figure 11.

The feature representation is a sparse vector with a term for each partition. The term of the partition that contains the target is one, while all others are zero. If no partition contains the target, an additional term indicates the absence of a target.

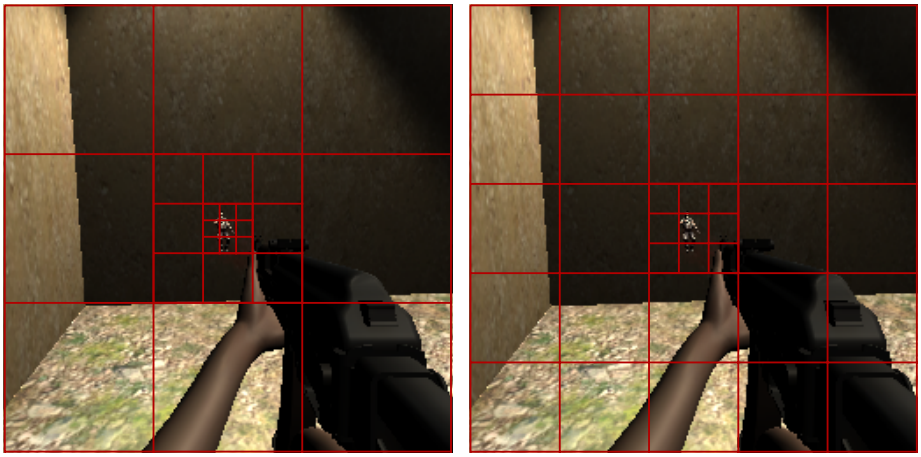


Figure 11: The leftmost partitioning scheme is notated 3, 3, 3 and the rightmost is notated 5, 3.

4.3.3 Optimal visual partitioning scheme

It is necessary that a part of the target is present in the exact centre of the image, if the position of the target is classified as the centre partition. In that situation the agent would always, assuming no recoil, hit the enemy if there is an enemy in the centre partition and the agent shoots. Without that guarantee the task of consistently shooting the enemy becomes more difficult. The agent receives no feedback from misses, consequently if the centre square is to large, it can become impossible for the agent to learn to hit. The size of the agent is depending on distance, and this requirement is therefore depending on the distance that the agent should be able to hit at.

Assuming the target is 11 pixels wide and has a box-shaped hitbox[13], having a centre square of 11x11 pixels would fulfil the aforementioned requirement. If 6/11 pixels is inside the centre square the enemy is considered to be inside that partition and since the sixth pixel is the centre pixel of the centre square, a part of the enemy is in the centre of the centre square - fulfilling the requirement.

A 5, 3 partitioning scheme would result in the centre partition being around 17 pixels wide, where it for a 3, 3, 3, partitioning scheme is approximately 9 pixels wide. There are 34 classes using a 5, 3 partitioning scheme, calculated as $(5 * 5 - 1) + (3 * 3) + 1 = 34$ and 26 classes using a 3, 3, 3, partitioning scheme, calculated as $(3 * 3 - 1) + (3 * 3 - 1) + (3 * 3) + 1 = 26$.

[13]The projection of the target onto the screen that registers as a hit when shot at

Since a 3, 3, 3 partitioning scheme results in fewer classes than a 5, 3 partitioning scheme, leading to faster training of the agent, and the resulting width of the centre partition is narrower, all experiments were done using a 3, 3, 3 partitioning scheme.

4.3.4 Comparing the representations

The feature representation of the VPR has a significantly larger number of dimensions than the AR, consequently slowing down evolution. The VPR is less detailed, as the precision of the AR is bound by the precision of the decimal used to represent the angle, while the precision of the VPR is bound by the width of the partitions. The AR allows the agent to choose shooting strategy based on distance and generally allows the agent to behave more smoothly. The large partitions of the VPR in the edges of the screen provide very ambiguous information about the position of the enemy, which makes it impossible for the agent to evolve smooth human-like aiming. If the agent was allowed to move or take strategic decisions, this lack of detail might reduce the potential of the agent furthermore. The AR allows the agent to aim precisely, which many FPS shooter games rewards by increasing the damage inflicted for hitting vulnerable areas on the target.

The only advantage of the VPR is that it is significantly easier to train a CNN to recognise this representation than the AR.

4.4 The topology of the convolutional neural network

The topology of the CNN has lots of different optimisation parameters, and as hyperparameter optimisation methods are based on trial and error, it takes time and computational resources to optimise. The project inevitably leaves room for better topologies that performs better, and the search for an efficient topology is not in the scope of this project. However, we impose some requirements on the topology. The stack of convolutional layers has to have a receptive field that can detect the target, as explained in section 3.2.4. The width and height of the target is depending on distance, but it is approximately 10 pixels wide and 25 pixels high. Therefore, the topology of the network should be able to recognise visual features spanning at least 25 pixels. The network should have some depth, as it is often correlated with better performance [1] [8], and use the rectifier activation function to reduce the vanishing gradient problem as described in section 3.4.4.

5 Approach

To answer the research question from section 2.2, we develop two VRCs as well as two AICs, combine them and run varying experiments. This section describes the details of how the networks subject to experimentation are trained and combined.

5.1 Training the convolutional neural network

This section describes the details of the training of the CNN, including the training examples, the network topology and the hyperparameters.

5.1.1 Topologies

We present four different topologies, two for classification and two for regression. Both types are tested with a 12 layered architecture and a 6 layered architecture, referred to as deep and shallow respectively. The difference between the two topologies is in the number of neurons in the fully connected layers and in the activation functions used for the output. All activation functions are ReLU(rectifier), except in the output layers, where softmax is used for classification and identity is used for regression. The regression network outputs the angular representation as described in section 4.3.1, scaled to fit a range of -1 to 1. The classification network outputs 26 probabilities summing to 1. The networks take inputs with a shape of 256x256x3, which is the 3 color channels of the image.

Deep neural network topologies The network topologies are illustrated on figure 12. Both have 12 layers, the first 8 being alternating convolutional and pooling layers, while the next 3 layers being fully connected layers followed by an output layer. The stride and zero padding of all the convolutional layers are 1, while the stride of all the pooling layers is 2.

Both networks have a total of 163,940 parameters(weights and biases) in the convolutional layers.

The classification network has 1,228,800 weights($16 \cdot 16 \cdot 120 \cdot 40$) from the last convolutional layer to the first fully connected layer and 4,386 parameters in the remaining layers.

The regression network has 7,680,000 weights($16 \cdot 16 \cdot 120 \cdot 250$) from the last convolutional layer to the first fully connected layer and 126,754 parameters in the remaining layers.

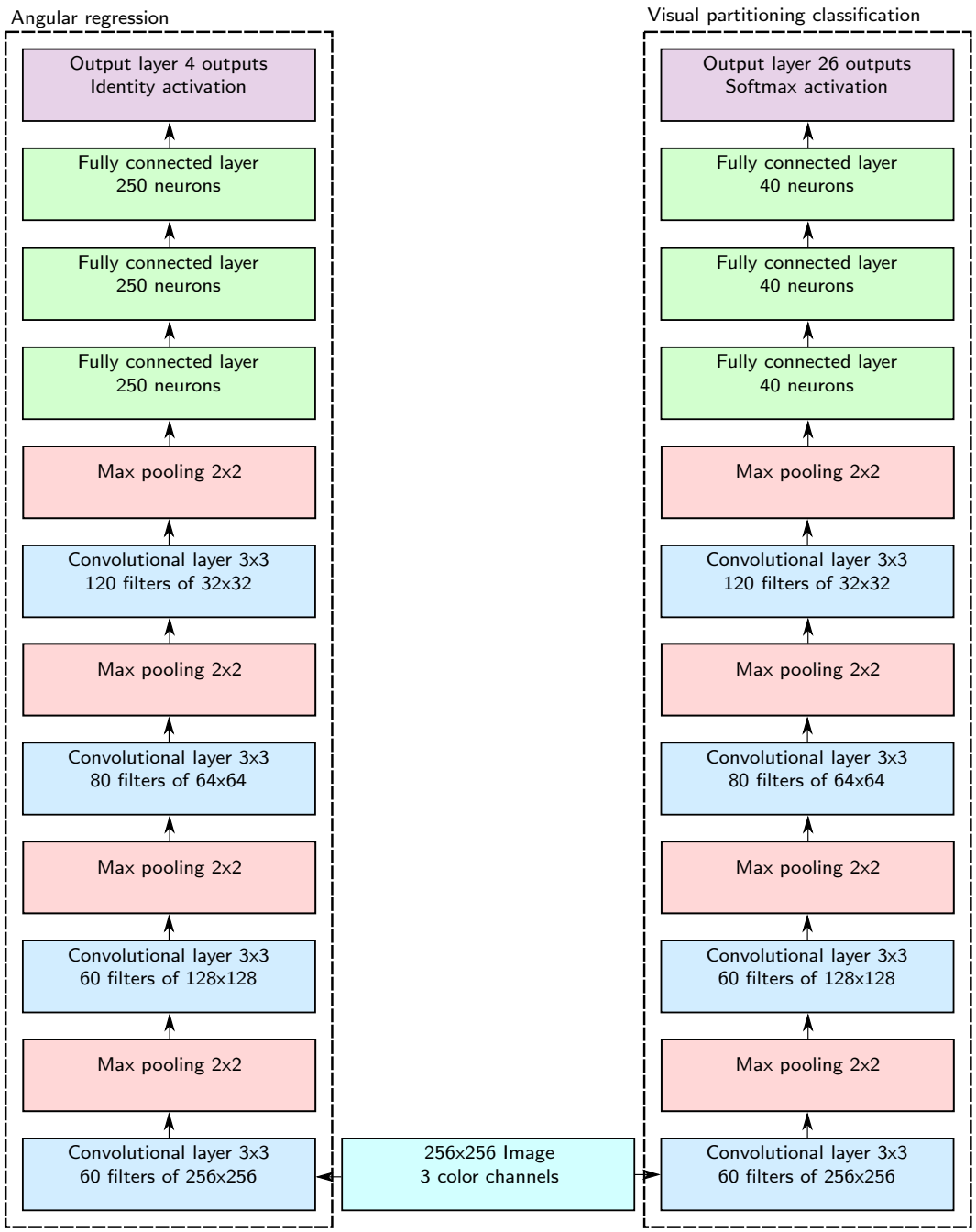


Figure 12: The full topology of the deep convolutional neural networks

Shallow neural network topologies The network topologies are illustrated on figure 13. Both have 6 layers, the first 4 being alternating convolutional and pooling layers, followed by a fully connected layer and an output layer. The stride and zero padding of all the convolutional layers are 1, while the stride of all the pooling layers is 4.

Both networks have a total of 246,200 parameters(weights and biases) in the convolutional layers.

The classification network has 3,686,400 weights($16 \cdot 16 \cdot 120 \cdot 120$) from the last convolutional layer to the first fully connected layer and 3,266 parameters in the remaining layers.

The regression network has 23,040,000 weights($16 \cdot 16 \cdot 120 \cdot 750$) from the last convolutional layer to the first fully connected layer and 3,754 parameters in the remaining layers.

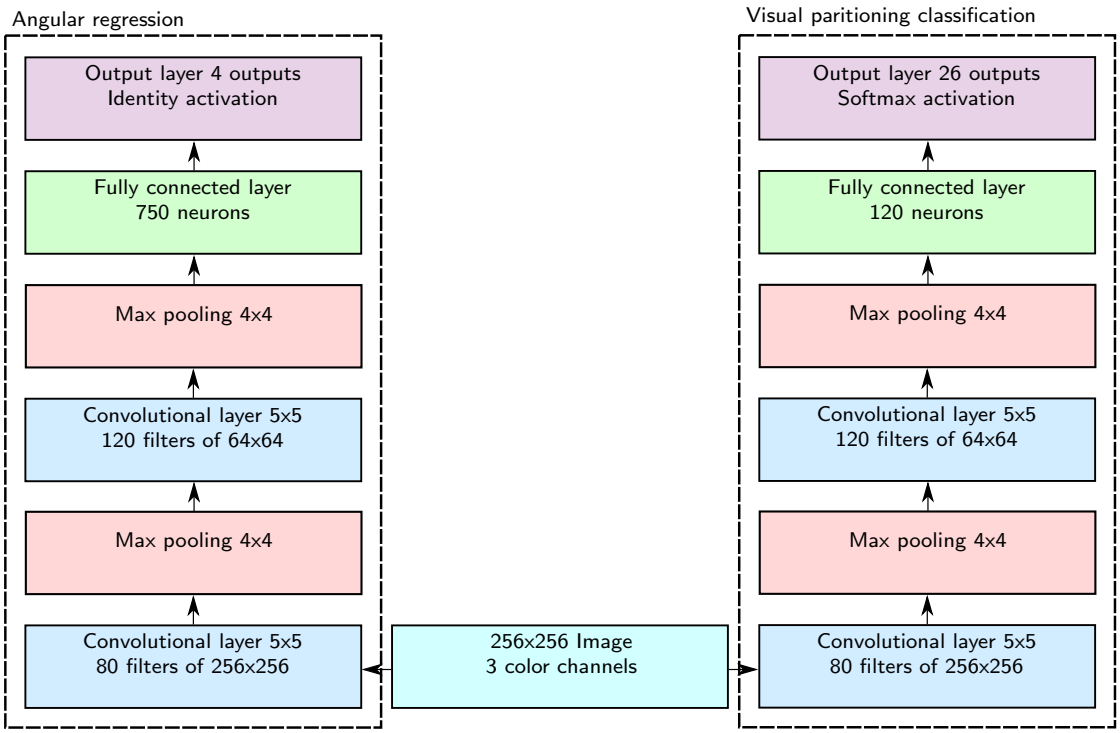


Figure 13: The full topology of the relatively shallow convolutional neural networks

5.1.2 Training with gradient descent

The optimisation process of gradient descent did not include continuous evaluation on a validation set for early stopping, as overfitting was of no concern. When the error of the network stopped improving, the process was stopped. The parameters were updated from a mini-batch of size 32, with a learning rate of 10^{-3}. The process did not include dropout, but included L2-regularisation with a coefficient of $5 \cdot 10^{-4}$ for some of the experiments. Nesterov's accelerated gradient was used with a momentum coefficient of 0.9. The networks were initialised with Xavier initialisation. The number of training examples used for training was 130,000 for all experiments. The framework used for supervised learning was DL4J, a deep learning framework for java, accessible at `https://deeplearning4j.org`.

5.1.3 Training data

The training examples consists of the raw image data and ground truths. The raw image data is, for each pixel, the byte value of the red, green and blue pixel, i.e. three numbers between 0-255, arranged as a 3 dimensional volume, as seen in figure 14.

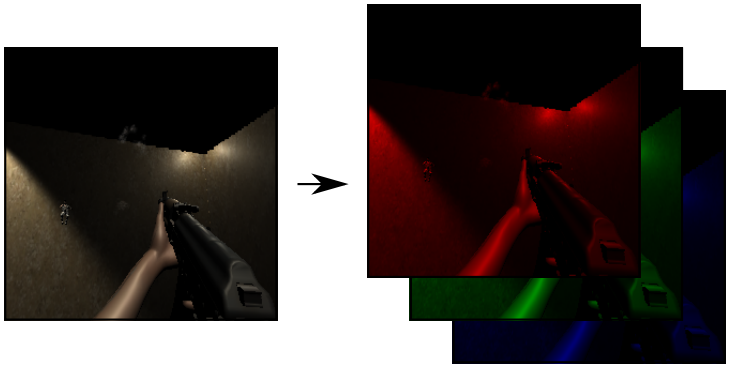


Figure 14: An image split into its three color channels

The ground truth is different for the two representations. For the VPR it is a vector with a 1 in the correct class, and for the AR it is a 4 dimensional vector with the correct representation. In section 5.2.4 it is explained how the training data for this project was gathered.

5.1.4 Binarisation of the visual partitioning representation

The output of the VRC approximating the visual partitioning representation is a probability distribution summing to 1, as described in section 4.3.2. When the VRC is used as input provider to the AIC, output of the VRC is transformed to a sparse vector with a 1 in the class with the highest probability. This makes the input similar to the input that the AIC is trained with.

5.1.5 Scaling of the angular representation

The AR of the position of the target consists of four dimensions as described in section 4.3.1. The horizontal and vertical angle are scaled to $[-1, 1]$, the distance is divided by 20, and the bit indicating whether the target is within sight is either 1 or 0. When the AR is estimated by the VRC and used as input to the AIC, angles are cutoff if they are outside the range of $[-1, 1]$, and the indication of whether a target is present is rounded to 0 or 1. If the indication is rounded to 0, the other dimensions are set to 0 for consistency, regardless of the output of the VRC. When the AR is used as input to the AIC, each angle is split into two scalars. For both angles the scalars are defined as:

$$f_1(v) = \begin{cases} v, & \text{if } v > 0 \\ 0, & v \leq 0 \end{cases}$$

$$f_2(v) = \begin{cases} -v, & \text{if } v < 0 \\ 0, & v \geq 0 \end{cases}$$

This results in 6 non-negative real numbers making up the AR.

5.2 Training the agent

The base game[14] and textures used as test bed for our experiments from and heavily modified it to suit our needs. This includes removing the ability for the player/agent to move, bots to move and shoot and removing everything regarding multiplayer and server functionality.

The training of the agent was done in Unity 5[15]. It was trained with a, slightly, modified version of NEAT as described in section 3.3, using the UnityNEAT frame-

[14] `http://armedunity.com/files/file/107-multiplayer-fps-kit-raknet/`
[15] `https://unity3d.com/`

work[16], which is a port of the C# NEAT framework SharpNEAT[17], made to use with games developed in Unity.

Training the agent using the CNN as the ground truth provider is extremely computationally heavy using the setup of this project and therefore not feasible. Having access to more computational resources or more low level control with the hardware might overcome these challenges. Since neither was available at the time of writing, the CNN was not used as the ground truth provider. Instead of using the CNN to estimate the representation, it is calculated using the state of the game. This method is far less computationally heavy, making the training of the agent a lot faster, and allows the neuroevolution experiments to run at 10 TPS.

5.2.1 Evaluation

Network fitness was evaluated in the FPS game described in section 3.6. An evaluation lasts 15 seconds, and as the evaluation includes some randomness, each evaluation was repeated 20 times and the results averaged. The random spawn of the target, and agent, ensures that the agent does not learn a specific pattern, but learns to act in any situation. The agent was awarded for hitting the target and aiming closer to the target.

5.2.2 Fitness function

The fitness function for the evaluation can be formulated as:

$$f = \text{damage} + \text{aim\_bonus}$$

Where damage is the total damage dealt, and aim_bonus is calculated as:

$$\text{aim\_bonus} = \frac{1}{n} \sum_{n} \frac{c}{(1+v)^2}$$

Where c is a constant set to 75, n is the number of frames and v is the angle measured in radians between the forward pointing vector of the agent and the line from the agent to the target. Hence $v = 0$ if the agent aims directly at the target. This bonus makes the evolution faster, as dealing damage requires aiming, and evolving the ability to aim takes several generations. The aim_bonus is plotted in figure 15.

[16] `https://github.com/lordjesus/UnityNEAT`
[17] `http://sharpneat.sourceforge.net/`

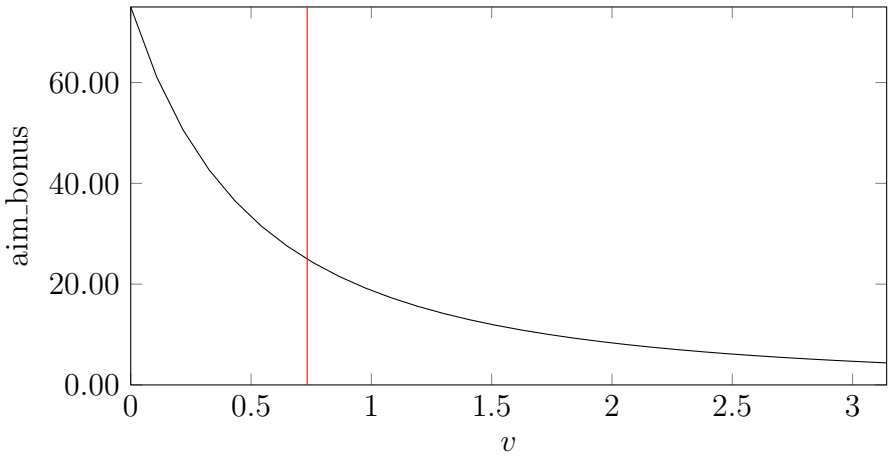


Figure 15: The fitness awarded as a function of the angle between the current and the desired pointing direction. The red line marks the value of v when the target is at the corner of the screen

5.2.3 NEAT setup

The hyperparameters of NEAT are shown in table 1. The activation function used is sigmoid, and biases are modelled by having a constant input, which when connected to, is the equivalent of a bias. The networks start fully connected and allow recurrent connections to appear.

5.2.4 Data generation

Training data representative of what the agent will encounter during actual play, will lead to better performing networks as they should not encounter anything they have not been trained for.

Taking pictures of the environment, in which the agent is during actual play, using random positioning and rotation of the camera should produce representative training data. In our case having the camera at a fixed position and random rotation should suffice as our agent is stationary.

An untrained NEAT agent, basing its actions on the angular game state, exhibits more or less random behaviour. During training the behaviour should become less random and more intelligent, which is desirable under normal circumstances, but not when the agent is used to generate training data. Disabling the fitness function, i.e. always giving a fitness of 0, disables learning, thereby persisting the random behaviour.

Hyperparameter	**Value**
Population size	100
Elitism	20%
Asexual offspring	80%
Sexual offspring	20%
Interspecies mating proportion	0.01%
Connection weight range	-5 to 5
Connection weight mutation probability	80%
Add node probability	3%
Add connection probability	5%
Delete connection probability	1%
Species	10
Speciation constant C1	1
Speciation constant C2	1
Speciation constant C3	0.4

Table 1: NEAT hyperparameters used to train the agent

We were able to almost fully automate the process of generating training data by disabling the fitness function and taking a screenshot every 5-10'th frame, thereby getting a set of images representative of the states the agent would be in, during actual play.

5.2.5 Data cleaning

Problems can occur when the representation of classes is imbalanced, as discussed in section 3.4.6. To overcome these problems some of the images not containing an enemy was deleted. The deletion reduced the number of images without an enemy from ~50% to ~15%.

The distribution of the rest of the classes depends heavily on the granularity of the partitioning scheme used. Using a 3, 3, 3 partitioning scheme, the most represented class, not counting images without an enemy, constitutes around 11% of the images. The least represented class contained a mere 0.0862% of the samples. In other words our representation of classes were majorly imbalanced as seen in table 2.

Even though we had these large differences in the number of pictures from each class, the convolutional neural network was able to learn what it should, as it can be seen in our results throughout section 7, contrary to the claims of [13].

<table>
<tr><th>Partition</th><th colspan="3">Percentage</th></tr>
<tr><td>0</td><td>8.1846 %</td><td rowspan="8">72.6507 %</td><td rowspan="16"></td></tr>
<tr><td>1</td><td>9.2277 %</td></tr>
<tr><td>2</td><td>7.8015 %</td></tr>
<tr><td>3</td><td>11.6000 %</td></tr>
<tr><td>4</td><td>11.3446 %</td></tr>
<tr><td>5</td><td>8.0100 %</td></tr>
<tr><td>6</td><td>8.6323 %</td></tr>
<tr><td>7</td><td>7.8500 %</td></tr>
<tr><td>8</td><td>1.4000 %</td><td rowspan="8">11.0278 %</td></tr>
<tr><td>9</td><td>1.3715 %</td></tr>
<tr><td>10</td><td>1.3662 %</td></tr>
<tr><td>11</td><td>1.4131 %</td></tr>
<tr><td>12</td><td>1.5477 %</td></tr>
<tr><td>13</td><td>1.1777 %</td></tr>
<tr><td>14</td><td>1.3762 %</td></tr>
<tr><td>15</td><td>1.3754 %</td></tr>
<tr><td>16</td><td>0.1646 %</td><td rowspan="8">1.3324 %</td><td rowspan="8">12.3602 %</td></tr>
<tr><td>17</td><td>0.2046 %</td></tr>
<tr><td>18</td><td>0.1677 %</td></tr>
<tr><td>19</td><td>0.0862 %</td></tr>
<tr><td>20</td><td>0.1100 %</td></tr>
<tr><td>21</td><td>0.0877 %</td></tr>
<tr><td>22</td><td>0.1762 %</td></tr>
<tr><td>23</td><td>0.1846 %</td></tr>
<tr><td>24</td><td>0.1508 %</td></tr>
<tr><td>No enemy</td><td colspan="3">14.9885 %</td></tr>
</table>

Table 2: Distribution of classes of the 130,000 training samples of the visual distorted setting

5.3 Pipeline

The AICs used to test the pipeline are trained with 5 TPS, compared to 10 TPS for the AICs used for neuroevolution, as the process of running the CNN in real time proved computationally expensive. Consequently, the networks evolved for this purpose does not have the prerequisites for performing as well as the ones used for neuroevolution experiments. The technical details of integrating the VRC and the AIC are described in section A.2 of the appendix.

6 Experiments

6.1 Convolutional neural network experiments

The CNNs described in section 5 are measured in several ways to find out how well they work individually. Both the visual partitioning and angular representation are measured under different circumstances and with different topologies.

The VPR are measured by the percentage of its prediction that are correct, referred to as accuracy. The class that the CNN assigns the highest probability to is regarded as its prediction. Consequently, whether the CNN predicts with a confidence of 51% or 99% does not change the accuracy, but it changes the cost.

The angular representation is measured in both accuracy percentage and absolute mean error. The horizontal angle, the vertical angle and the distance are measured in absolute mean error while the output indicating whether there is a target present on the image is measured in percentage accuracy. When the angular representation is used as input provider to the evolved ANN, the term indicating whether a target is present is converted to 0 or 1, whichever is closest. Hence the confidence of the prediction does not change its fitness as an input provider.

To measure whether the CNNs overfits the training data, they are measured against the training set and a test set, using 10,000 samples.

6.1.1 Visual distortion

To measure how the network is penalised by having a varying visual representation of the target, the networks are trained with two different data sets, from two different visual settings, as seen in figure 16. This is relevant, as both the real world and modern FPS games have various visual representations of targets. The textures are more detailed in the visually distorted version and the overlay of the player and the weapon is only present in this setting. The overlay fully or partially covers the target in some cases, making the recognition task harder, or even impossible.

Figure 16: The two different visual settings

6.1.2 Different topologies

The effect of depth of the CNN topologies described in section 5.1.1 are measured in combination with the different visual game settings to assert its effect on training speed, overfitting and accuracy.

6.1.3 Smaller training sets

In a real world application of visual recognition, the number of training examples is often limited, as the training examples are subject to manual labeling. Hence, we measure both the accuracy of the VPR and the angular representation trained on significantly smaller data sets from the visually distorted game setting. Furthermore, overfitting is measured separately on these networks trained with smaller data sets, as overfitting gets more likely with fewer training examples. The data sets used are subsets of the training sets with 130,000 examples, and has an even distribution of VPR classes. The training process uses L2-regularisation with $\lambda = 0.0005$.

6.2 Neuroevolution experiments

The fitness of the ANNs evolved with neuroevolution are measured with the game state as ground truth provider, to assert how well the networks work without being penalised by the error of the VRC. Fitness from aiming and damage dealing are

visualised separately to show how the aim fitness helps accelerate learning. We compare the ANNs based on VPR and AR on how fast they learn and how well they perform. Each AIC is evolved 3 times and its fitness averaged, as the evolution process is random.

6.2.1 Unnecessary reloads and misses

To find out how well neuroevolution solves the task of reloading and shooting without missing, we measure the number of unnecessary reloads and misses. An unnecessary reload is defined as a reload that does not increase the number of bullets in the magazine.

6.3 Pipeline experiments

We evaluate the fitness of the combination of the VRC and the AIC and compare it to the fitness of the same AIC with the ground truths as feature representation. To find out how much the pipeline fitness is penalised from using AICs trained with different providers, we evaluate the fitness of the combination with half the look sensitivity[18]. Lowering look sensitivity might reduce the impact of the VRC error, as the perception is translated more frequently per degree rotated.

[18]The factor that the output of the ANN trained with neuroevolution is multiplied with to calculate the resulting horizontal or vertical rotation speed.

7 Results

This section describes the results of the experiments explained in section 6.

7.1 Convolutional neural network experiments

The experiments performed are described in section 6.1. Note that the time per iteration measured in the following sections is dependent on the hardware used for training. The hardware details are described in section A.1 of the appendix.

7.1.1 Visual partitioning representation

Convergence Figure 17 and 18 show the convergence of the CNNs using VPR measured as the cost function on the mini-batch that the gradient is estimated from, over iterations. As the process of batch selection is random, the score is fluctuating. The cost function is negative log likelihood, described in section 3.4.1. The results presented are with and without visual distortion and with the network topologies described in section 3.2.4. The shallow topology converges in fewer iterations, but both networks manage to converge to a solution. The average time per iteration for the data with visual distortion is 4,782 milliseconds for the deep network and 2,451 milliseconds for the shallow network.

Performance The accuracy of the results in figure 19 is measured as the percentage of correct predictions. It is apparent from these results, that the models have not overfitted to the training data, as the difference in accuracy of the training set and the test set is insignificant. Furthermore, the topologies of the networks does not seem to have a significant impact on the accuracy. Examples of the training examples that the deep CNN with visual distortion fails to classify correctly can be seen in section B of the appendix. The incorrect predictions are due to the target being in between partitions or behind the weapon overlay.

7.1.2 Angular representation

Convergence Figure 20 and 21 show the convergence of the CNNs using AR shown as the cost function on the mini-batch that the gradient is estimated from, over iterations. The cost function is Euclidean loss, described in section 4.3.1. The results presented are both with and without visual distortion and with the network topologies described in section 3.2.4. The deep network reaches a lower cost, but requires additional iterations to converge to a solution. The deep network trained in the visually distorted setting has an average time per iteration of 3,734 milliseconds,

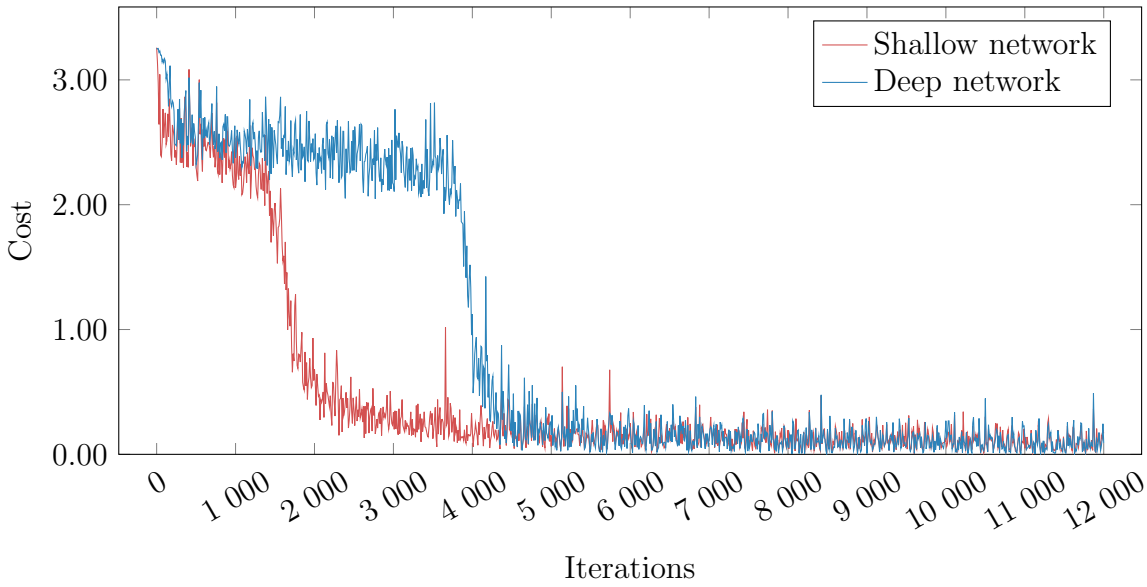


Figure 17: Negative log likelihood cost over mini-batch gradient descent iterations using VPR without visual distortion

while the shallow network has an average time per iteration of 2,572 milliseconds. Consequently, the deep network converges significantly slower.

In the setting without visual distortion the deep network also performs better than the shallow network, as seen in figure 21, but the difference in final cost is lesser.

Performance The performance is measured as mean absolute error on the angles and distance of the AR, and as percentage correct predictions of whether a target is present in the image(target detection).

Figure 22 shows that there is no significant difference between the accuracy on the test and the training set. This entails little to no overfitting on target detection.

It is apparent from the difference in accuracy on the test set and the training set that there is little to no overfitting on horizontal angle, vertical angle and distance, as seen in figure 23, 24, 25 and 26. The deep networks perform better than the shallow ones on both tasks, but the difference is especially significant without visual distortion. The error of the networks trained without visual distortion is visualised in section C of the appendix.

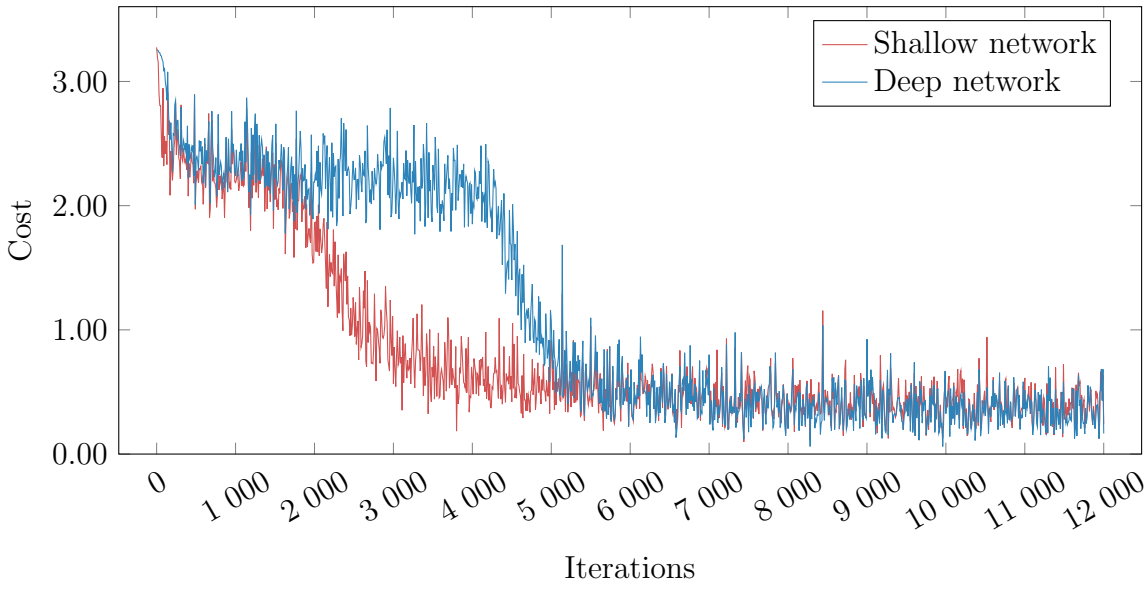


Figure 18: Negative log likelihood cost over mini-batch gradient descent iterations using VPR with visual distortion

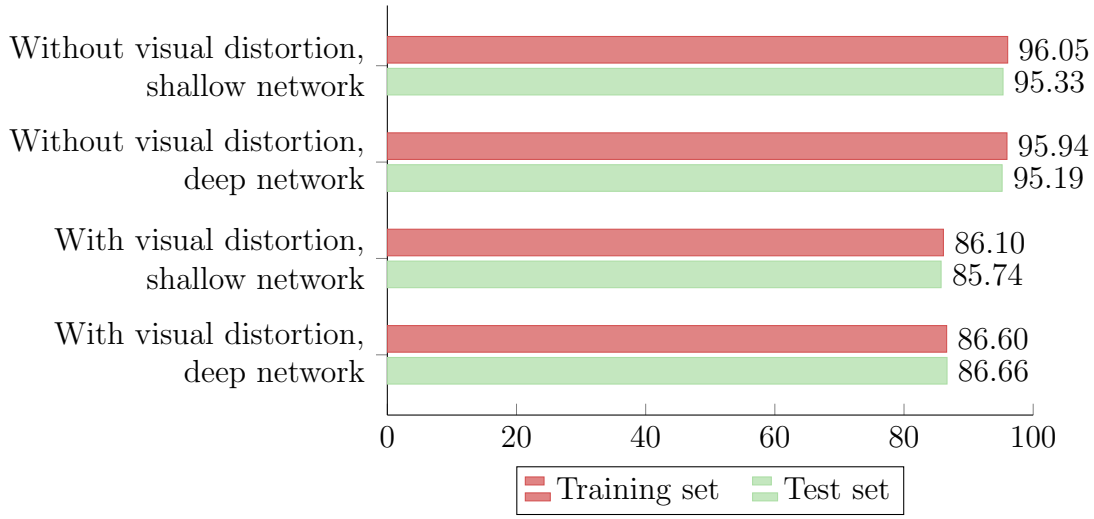


Figure 19: Accuracy of partitioning classification using VPR

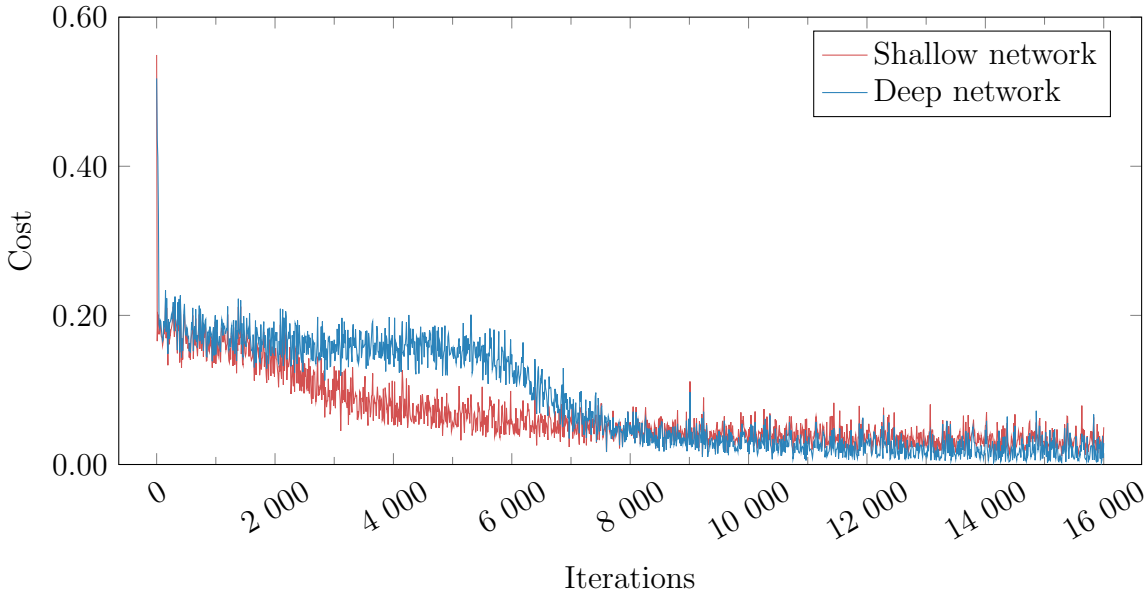


Figure 20: Mean squared error cost over mini-batch gradient descent iterations using AR without visual distortion

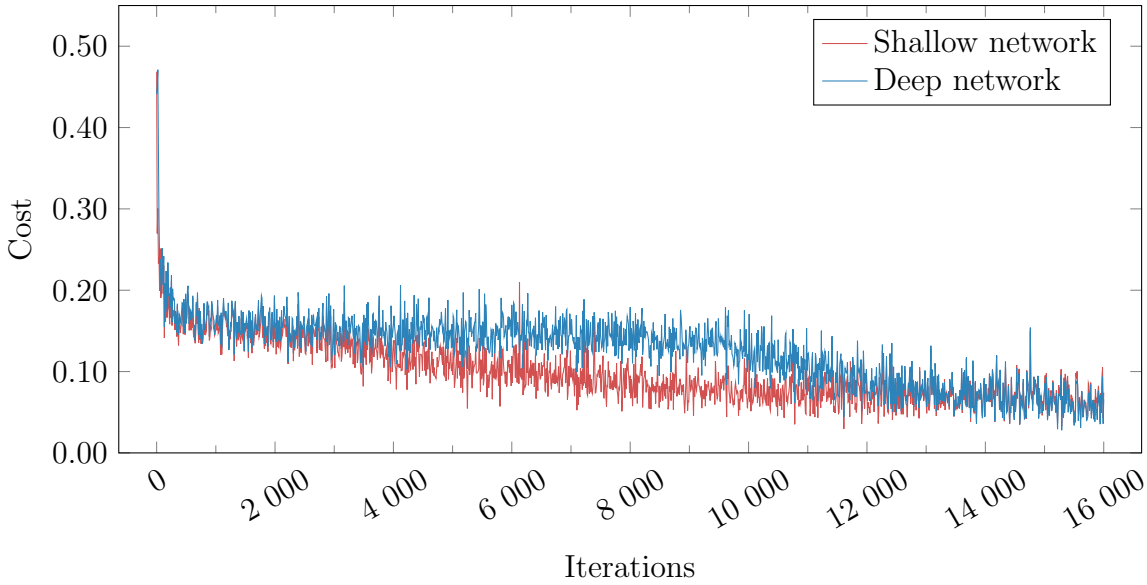


Figure 21: Mean squared error cost over mini-batch gradient descent iterations using AR with visual distortion

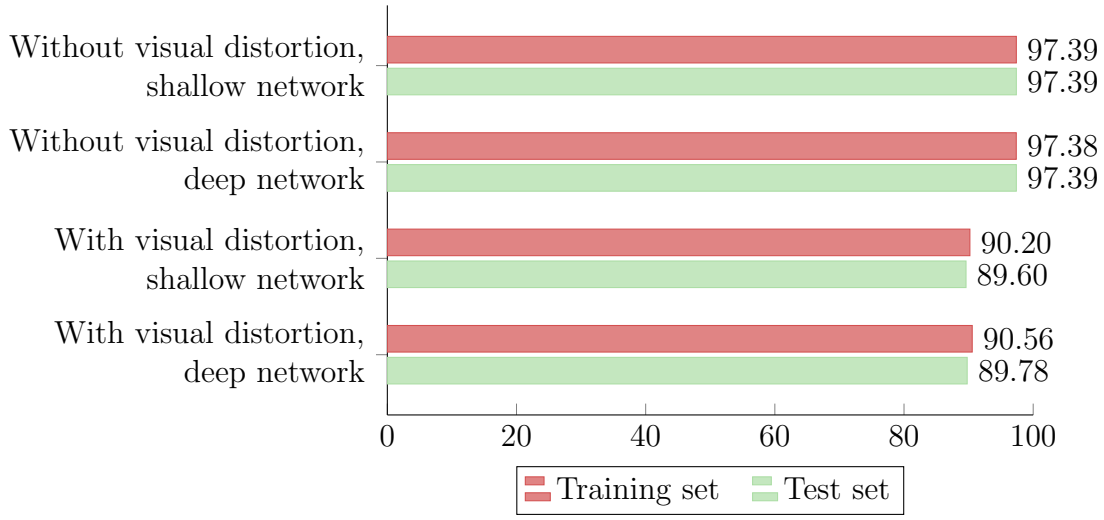


Figure 22: Accuracy of target detection using AR

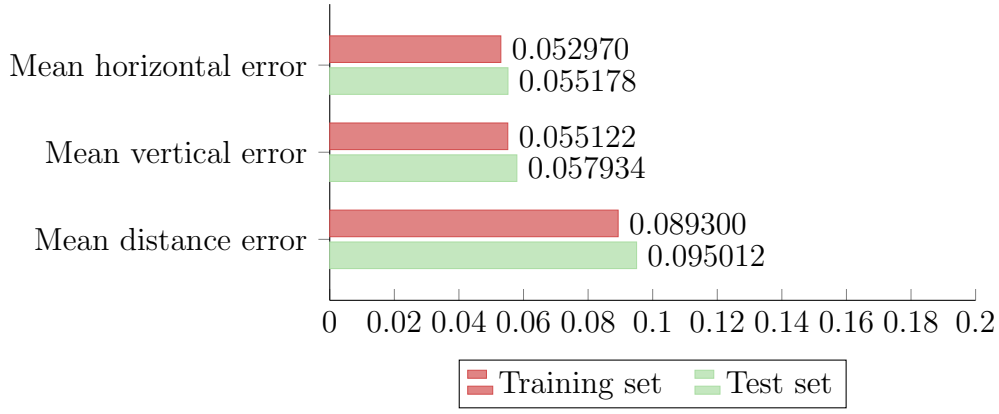


Figure 23: Mean error of the deep CNN without visual distortion

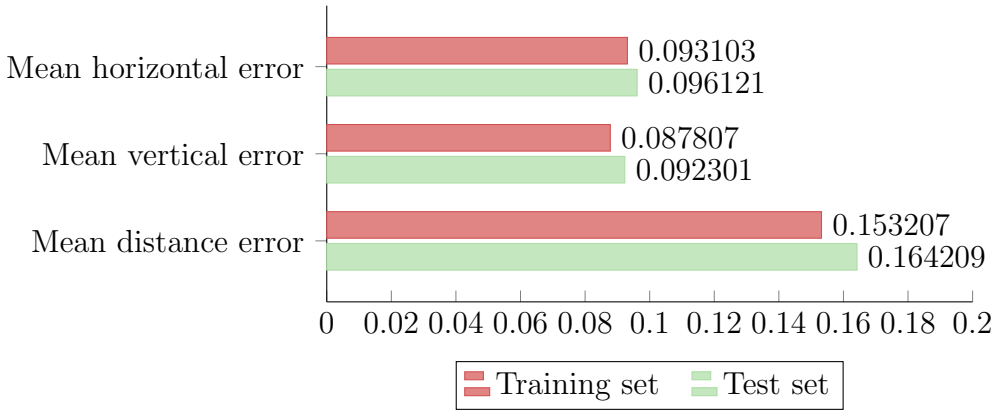


Figure 24: Mean error of the shallow CNN without visual distortion

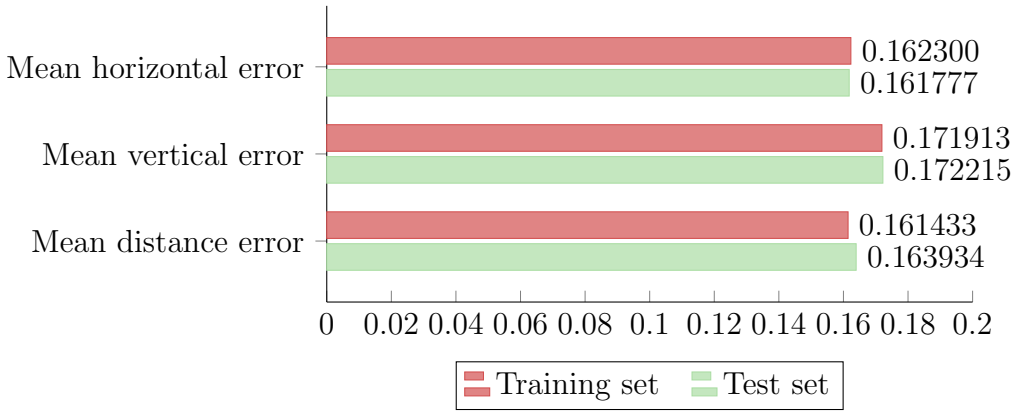


Figure 25: Mean error of the deep CNN with visual distortion

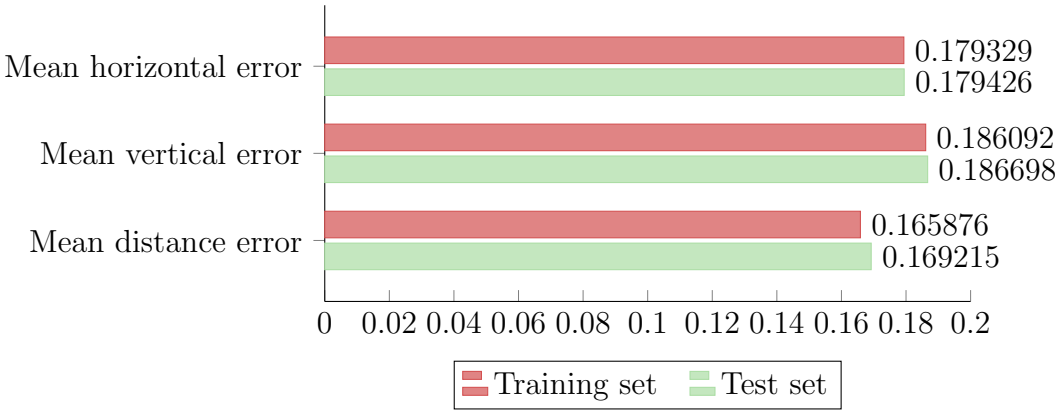


Figure 26: Mean error of the shallow CNN with visual distortion

7.1.3 Feature maps

The feature maps shown in figure 27 and 28 are visualised by scaling the output range $[0, 1]$ of every neuron in the convolutional layers linearly to the grey scale range $[0, 255]$. The leftmost column of feature maps are from the first convolutional layer, and the rightmost is the input to the fully connected layers. As a convolutional layer takes a depth slice of all the previous feature maps as input, there is no apparent connection between the visualised output of the max pooling layer and the following result of the convolutional layer. All of the feature maps are from the same input. Additional feature maps are included in section E of the appendix. The feature maps from the two different representations are different in both the magnitude and the variance of the output. The feature maps from the network estimating the AR has fewer useful feature maps in the last max pooling layer and has a tendency to highlight useless features, such as the weapon overlay, as much as the target.

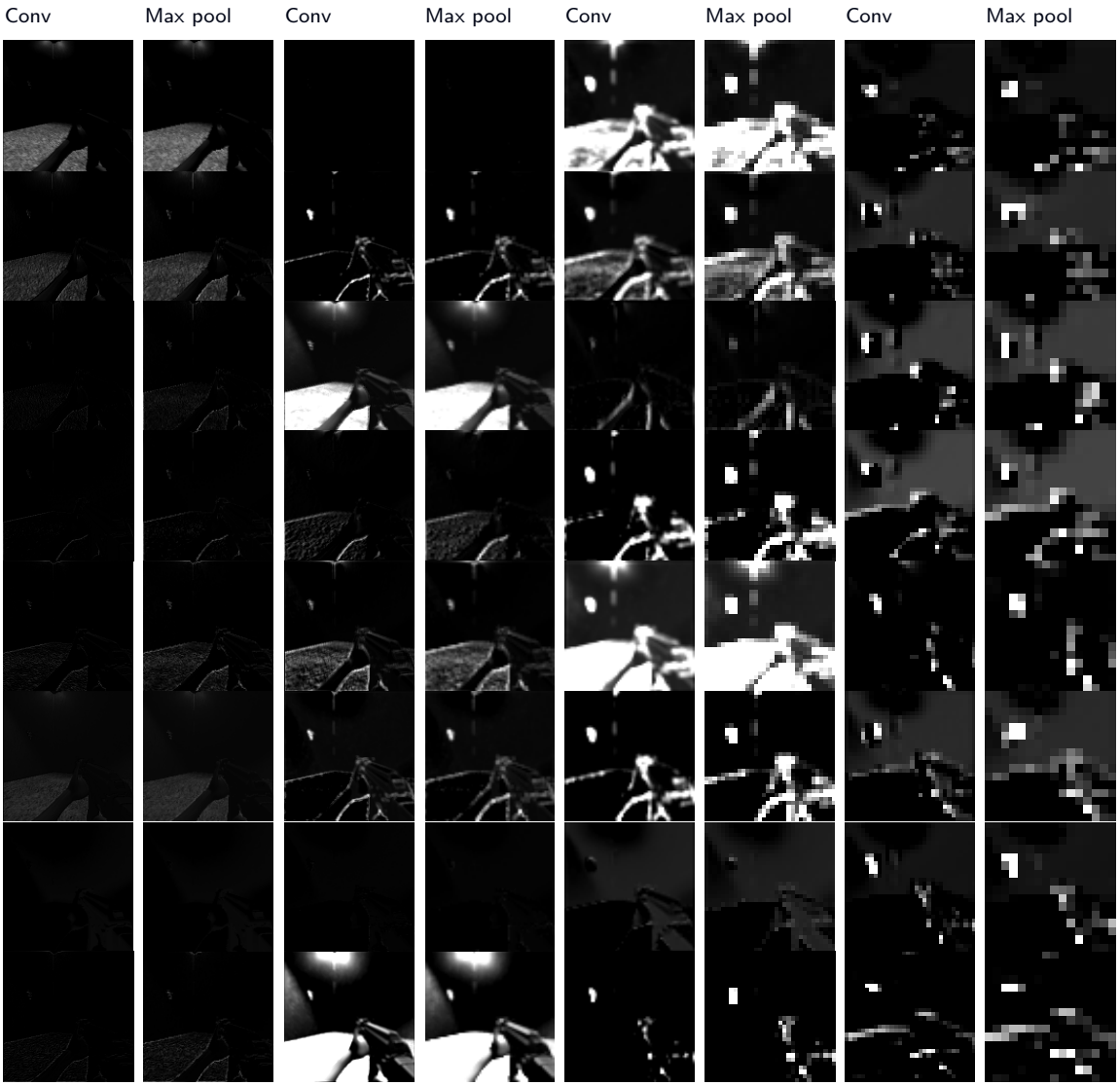


Figure 27: A small subset of the feature maps produced from running a training example from the lightened arena through the VPR deep convolutional neural network. The feature maps highlight the position of the target.

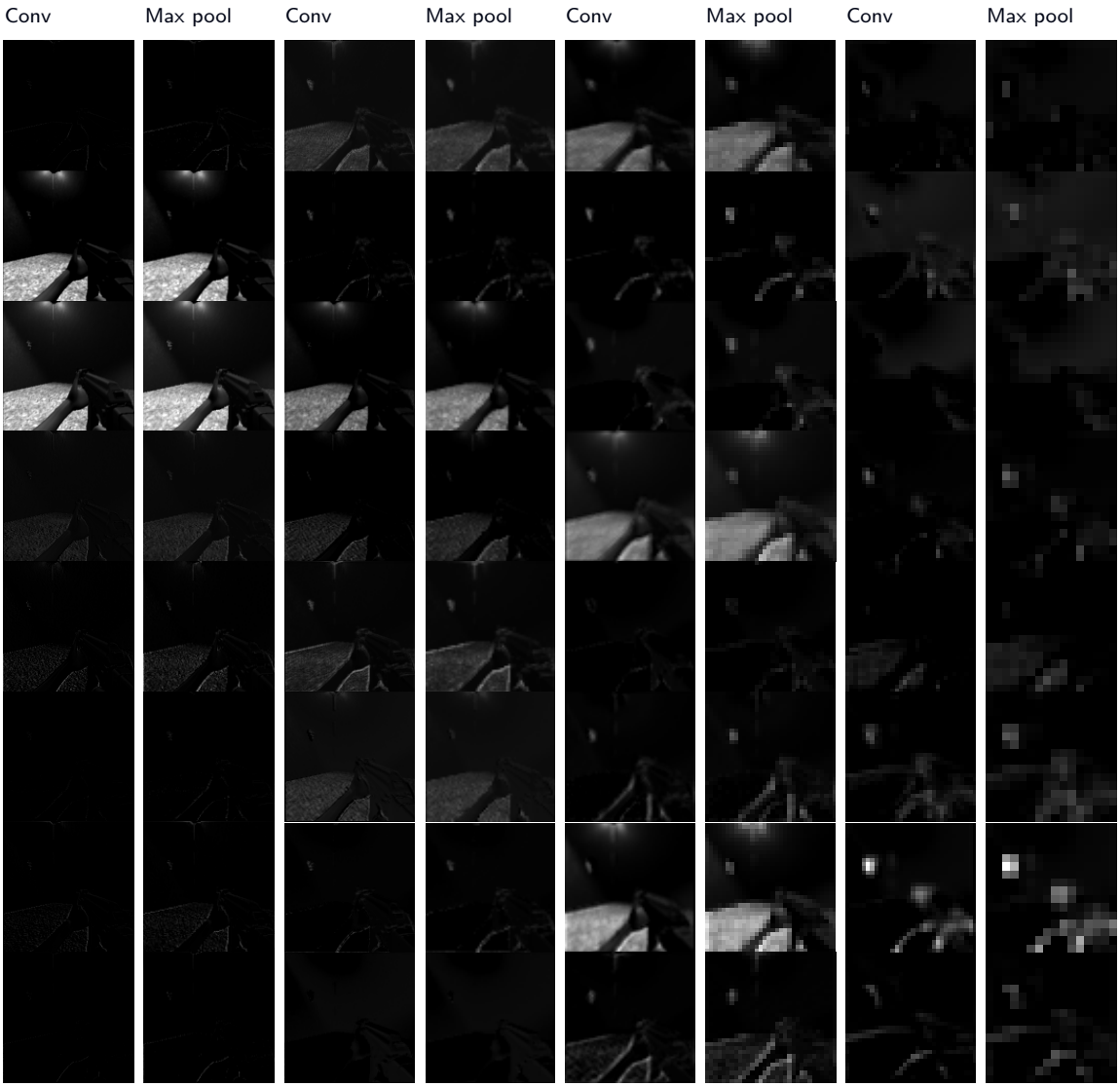


Figure 28: A small subset of the feature maps produced from running a training example from the lightened arena through the AR deep convolutional neural network. The feature maps highlight the position of the target.

7.1.4 Smaller datasets

The results shown in figure 29 shows that training with 5,015 training examples results in an accuracy of 71.30%, which is 15.36 percentage points less than the

network trained with 130,000 examples. As the accuracy is close to 100% on the training data, the network clearly overfitted. Whether the networks are trained for the optimal number of iterations is uncertain.

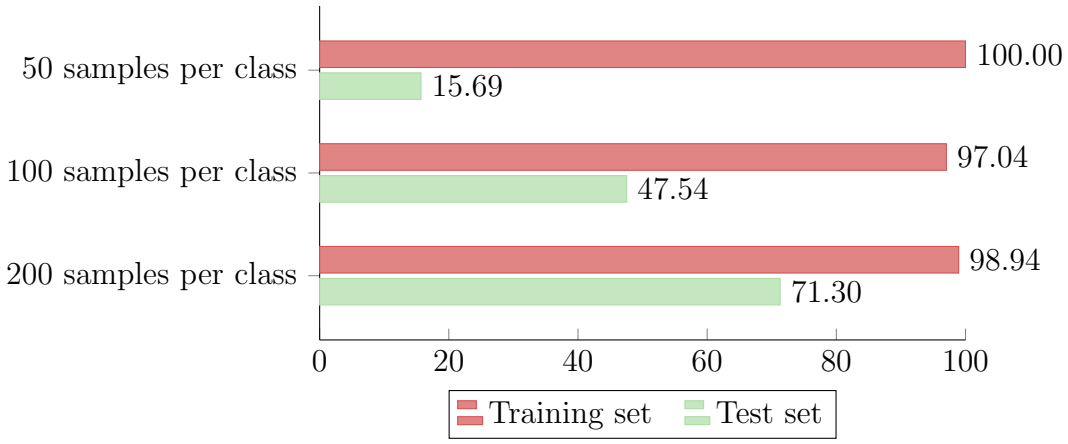


Figure 29: Accuracy of target detection using VPR on small datasets

7.2 Neuroevolution experiments

The graphs in figure 30 and 31 as well as the graphs included in section G of the appendix are based on evolution with the ground truths as feature representation. The experiments ran for approximately 4 hours each 100 generations. We observe an overall tendency for experiments with the AR to perform better than the experiments with the VPR, both learning faster and reaching a higher fitness. The AR without recoil misses significantly fewer shots than any other approach, as seen in figure 51. The VPR has fewer unnecessary reloads than the AR, which is the only parameter where it is superior. None of the approaches handles recoil well, as visualised in the graphs in figure 49 and 47. The AR with recoil eliminates approximately a single target every evaluation, and the VPR manages to hit the target 1 or 2 times. All approaches except VPR without recoil seems to reload with full magazine approximately every second evaluation, as seen in figure 50, which indicates that they do not learn proper reloading behaviour. The decreasing aiming fitness can be attributed to target elimination, as the next target is spawning in a random location.

From inspecting the evolved topologies, we observe different approaches to handling reloading and recoil. The ANNs based on the AR with recoil has a tendency to evolve tap-fire in two different ways. The first one is having a negative weighted recurrent connection from and to the shoot output neuron that allows it to alternate

between firing and holding fire. The other one is by slowly aiming away from the target while shooting, and then stopping when the aim is too far off. The AR tends to reload when the aim is far off, as this tends to happen when a target is eliminated and a new target appears. The VPR handles recoil by moving to a partition adjacent to the centre partition while shooting, and then moving back again to the centre partition, creating a short delay in between shots. It tends to associate some of the medium-sized partitions with reloading.

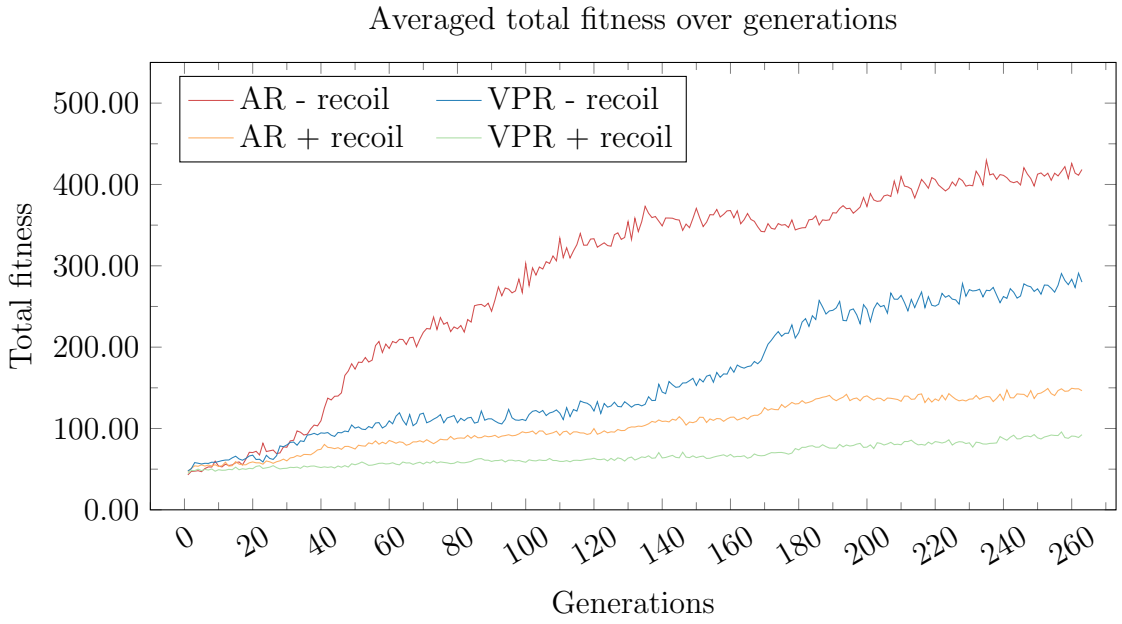


Figure 30: The total shooting and aiming fitness averaged. Each graph is an average of 3 runs.

7.3 The pipeline

The pipeline is the combination of the VRC and the AIC, and it is measured by evaluating fitness over 100 trials with the best performing VRC and the best performing AIC running at 5 TPS. The graph in figure 32 shows that both the pipeline based on the AR and the VPR are significantly penalised by using the VRC as feature representation provider, decreasing performance by 73.2% and 37.7% respectively. Reducing the look sensitivity by 50% increases the performance of the pipeline, but reduces the performance of the ground truth based AIC.

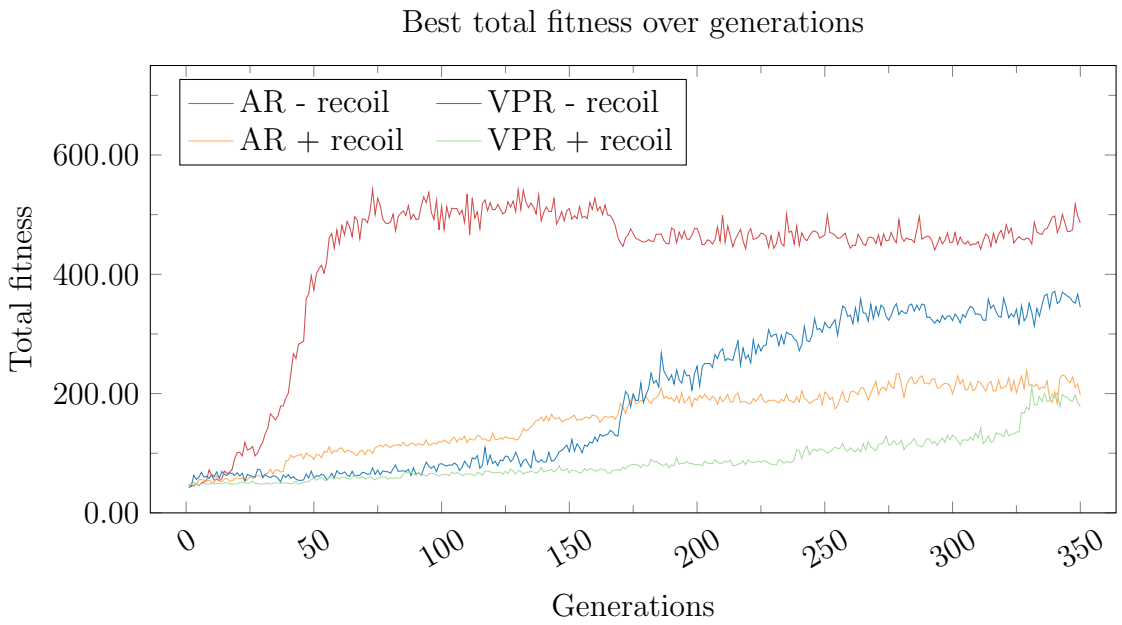


Figure 31: The best shooting and aiming fitness in a single run from each approach.

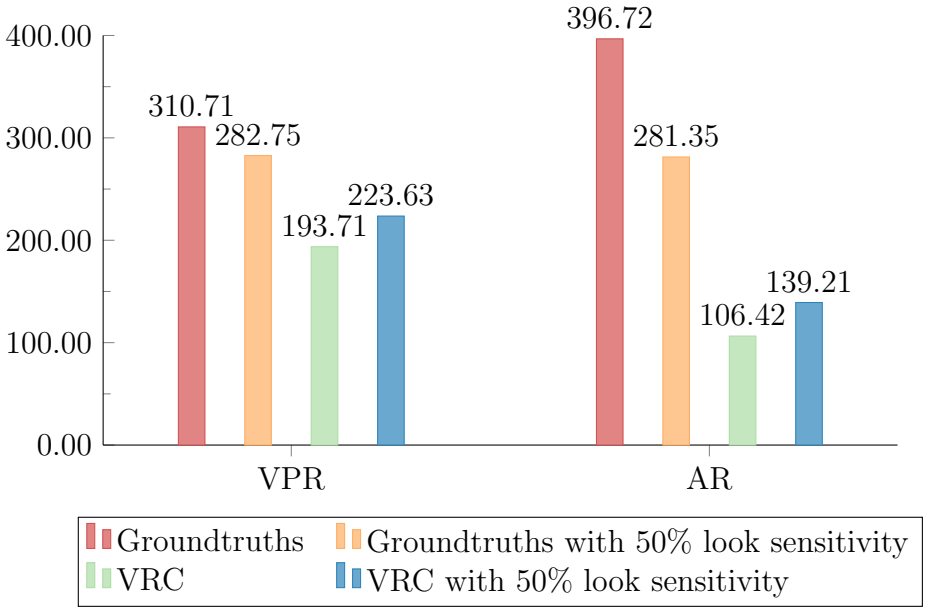


Figure 32: Fitness comparison of using the VRC and the groundtruths as features

8 Discussion

The discussion is divided into three parts, respectively discussing the VRC, the AIC and the combination.

8.1 The visual recognition component

From the results in figure 19, the CNNs estimating the VPR performed very well. The training examples that the networks failed to predict correctly were either when the target were in between partitions, or behind the weapon overlay, as seen in section B of the appendix. The networks were able to correctly classify images, where we found it difficult to locate the target, as seen in figure 33 and in section D of the appendix and the varying lighting did not seem to cause incorrect predictions.

Figure 33: The deep CNN estimating the VPR correctly predicts the class(green square) with a confidence of 55.7%, with only four pixels of the target being visible. This example was not used for training. The feature maps produced from this example are included in section E of the appendix.

The incorrect predictions of images where the target is present are generally cases where the target could be classified as being in either partition, depending on the exact location of the centre of mass of the target. Consequently, such an

incorrect prediction does not have a significant impact on an AIC using the incorrect classification to shoot and aim, as some of the target is present in the incorrect partition. These incorrections should therefore be viewed as a natural consequence of the vague classification definition, and not a failure of the optimisation of the model.

Optimising the model with gradient descent proved to be relatively trivial, performing adequately with two distinct network topologies, no regularisation and a very unbalanced distribution of classes. From these observations we conclude that the problem of estimating the VPR is well suited for deep learning. The network with 12 layers and the network with 6 layers performed equally well, but the deeper network did take approximately twice as long to train. However, we can not conclude which influence an increase in depth has on the performance of a network approximating the VPR. The observation should therefore not discourage the use of deeper networks to solve this type of problem. The distribution of classes of training examples is noticeably imbalanced, with more than 100 times as many classes in the most represented class, as in the least represented class, as seen in table 2. However, the imbalance does not seem to prevent the network from learning to recognise even the least represented classes. While it seems unlikely that the network can learn to recognise a class from less than 200 training examples, it is possible because the features recognised by the convolutional layer are similar for all the classes, as visualised in figure 27. Hence, the training of less represented classes uses the convolutional filters learned from the more represented classes to optimise the fully connected layers.

The experimentation with training with a smaller volume of data showed that the 5,015 examples were insufficient to estimate the VPR to a sufficient accuracy, but the test set accuracy tendency in the results suggests, that significantly less than 130,000 training examples could yield almost the same accuracy. This suggests that a real world application of the VPR would require an amount of labelled data greater than 5,015, as we assume that targets in the real world are more visually varied and harder to detect than targets in this game.

The results in the performance paragraph of section 7.1.2 shows varying success with estimating the AR. We observe that the topology of the network has an impact on the error of the horizontal angle, vertical angle and distance, especially when trained without visual distortion. The accuracy of target detection is not affected by the topology, which is not surprising, as it is a strictly easier problem than estimating the VPR. The fact that the topology has a significant influence on the results, leads us to believe, that the problem can be solved with a smaller error if the network topology is improved. Increasing the amount of training data might also yield smaller errors.

The error of the deep network estimating the AR with visual distortion is more

than twice as large as the counterpart without visual distortion. Recall that visual distortion both includes dynamic lighting, detailed textures and weapon overlay. This raises the question of how much each of these factors contribute to the error of the network. The visualised errors of figure 36, shows that the network has a large error on some examples, where the weapon overlay does not even partially cover the target. Consequently, the two other factors complicates the target detection when estimating the AR. However, it is expected that the weapon overlay adds error in both angles and distance, even if the network functioned optimally. A target in the lower right corner covered by the weapon overlay would be impossible to detect for the VRC, and would in the worst case scenario have an absolute error of 1 in both angles with a theoretically optimal VRC.

The results of the network estimating the VPR with visual distortion indicates, that the network estimating the AR theoretically should be able to approximate the position of the target regardless of visual distortion. Section H of the appendix shows experiments with even deeper networks, showing that the angular error can be further reduced by adding additional layers and neurons. Therefore it is assumed, that the inability of the network to estimate the AR with an adequate precision is due to insufficient tuning of the learning process, such as the volume or distribution of the training data, the topology of the network or hyper parameters of gradient descent. The network trained by Chenyi Chen et al. [3] is estimating features similar to the AR, and uses far more training data and trains for more iterations, which also indicates that the network presented by this project is far from optimal. The difference in quality of feature maps, visualised in section 7.1.3 also indicates, that the convolutional layers of the network estimating the AR is not as functional as the convolutional layers of the network estimating the VPR, which solidifies this assumption. From a theoretical point of view, the difference in the quality of feature maps is not surprising, as the negative log likelihood cost function optimises better than the euclidean loss cost function used for regression, as described and visualised by Xavier Glorot et al. [12]. Training the network estimating the AR with the convolutional layers of a trained network estimating the VPR could possibly decrease the AR error.

8.2 The action inferring component

The results of training the AIC with neuroevolution presented in section 7.2 show that NEAT is capable of producing networks that can aim and shoot in a FPS game, using both the AR and the VPR. However, the applicability of these networks to other FPS games is heavily challenged by their tendency to learn from the particular-

ities of the FPS game of the project. The distance to the target does not vary much, which the evolved networks exploit to implement reloading and shooting behaviour based on a fixed distance to the target, as the relationship between degrees rotated and the number of pixels that the target shifts on the screen is almost proportional. The network based on AR exploits the fact, that a new target spawns immediately after an elimination by reloading when the target is near the edge of the screen. This tends to result in a single reload as the time it takes to reload is greater than the time it takes to aim directly at the target. The AIC based on the VPR associates specific partitions with reloading. Consequently, none of the AICs learns generalisable reloading behaviour. Proper reloading behaviour requires either information about the number of shots left or long-term memory, none of which the AIC possess. The observations are therefore not surprising from a theoretical point of view.

The inability of the networks to develop human-like tap-fire is possibly because of the training duration. While we can conclude that 260 generations are not enough to develop this behaviour, we can not conclude that this approach is unable to develop tap-fire behaviour given enough time.

Neuroevolution evolves networks that are tailored to the task that they are evaluated on, and as our FPS game lacks variation, the networks does not learn much general applicable logic, except aiming and shooting without recoil. The best way to investigate the potential of NEAT in the domain of FPS games are to develop more varied evaluations that resembles modern FPS games. This is a significant shortcoming of the project, and optimisation of this area is recommended if the combination approach should be taken further.

8.3 The pipeline

The results in figure 32 show a significant difference between the reduction in performance from using a VRC with the AR and the VPR. The pipeline using a VRC estimating the AR clearly performs worse, and this observation corresponds with the performance evaluation of the individual VRCs in section 7.1.2. The estimation of the AR is far too inaccurate for the AIC to successfully infer an adequate action, and has its fitness reduced by 73.2% when using the VRC. The weapon overlay covering the target reduces performance for both approaches, but this factor does not explain the entire 37.7% fitness reduction from using a VRC with the VPR. From detailed examination of running the pipeline with a VRC estimating the VPR, we observed that the combination of low TPS and classification uncertainty when the target is in between partitions was a challenge. As the AIC is trained with the ground truths, assuming that the target is within a partition after a number of time steps given a

specific AIC output is relatively safe. However, when the estimation of the VPR is inaccurate when the target is in between partitions, this assumption does not hold and can lead to unwanted behaviour. The consequence of this behaviour is amplified by the fact, that the AIC using the VPR associates specific partitions with reloading, penalising inaccurate partitioning classification. That the reduction in fitness can be attributed to this problem is substantiated by the increase in pipeline performance from reducing the look sensitivity, as seen in figure 32.

The observations from the pipeline experiments harmonises well with the observations from the VRC experiments, and indicates that the VPR is easier estimated by deep learning than the AR.

Running the agent with the pipeline proved performance intensive, and the overall performance of the agent is definitely reduced by running with only 5 TPS. While the experiments failed to explore how well the combination could perform, they prove that it is possible for the combination to learn to aim and shoot with the VPR. We definitely believe that the approach can work for the AR, but the AR pipeline performance is not convincing enough to support any conclusions to this approach.

If this approach were to be implemented in a FPS game to control intelligent agents as game obstacles, the performance requirements would be a concern. Running multiple agents per human player along with the game would increase the performance requirement significantly. Using an AIC based on the AR with ground truths as input could be an alternative approach, which utilizes neuroevolution without deep learning and imposes less strict performance requirements on the game.

8.4 Applications in real world robotics

The modularity of the combined approach makes it applicable to real world robotics, such as military robots, drones and security camera and systems relying on identifying the position of targets based on visual input. The VRC can be combined with hand coded controllers, to create more reliable systems that are similar to the autonomous car driving agent developed by Chenyi Chen et al. [3]. Neuroevolution is challenging to apply to real world robotics, as the evaluation has to be faster than real time. Consequently, the robotics subject to neuroevolution has to be simulated virtually, which is further complicated by real world visual inputs. The abstraction provided by the feature representations makes this less complicated, as neuroevolution only has to interface with this representation. The VRC can theoretically be trained on labelled real world images, while the AIC can be trained without a visually realistic environment.

9 Future work

The integration of the VRC and AIC involved binarising the output of the VRC, such that it mimics the output that the AIC was trained with. However, this does not explore the potential of the AIC to use the analog output of the CNN, which might make the integration work better. Neuroevolution can possibly learn how to respond to the uncertainties and incorrect classifications of the VRC, and perform better than the combined agent developed in this project. The probability distribution of the features vector of the VPR reveals additional information about the position of the target, as an uncertainty usually means that the target is between two partitions, as exemplified in figure 34. Consequently, we believe that evolving the ANN with the analog output of the CNN could result in even better performance than using the ground truths directly.

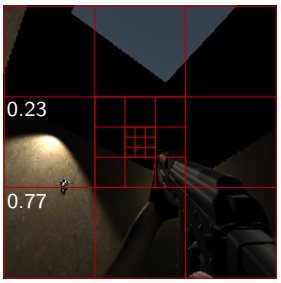


Figure 34: An example of a relatively low prediction confidence of the CNN, indicating that the target is in between partitions.

Using HyperNEAT as by Kenneth O. Stanley et al. [20], to evolve the AIC using the VPR, or even directly on feature maps, as seen in section 7.1.3, could potentially increase the evolution speed and the resulting performance. HyperNEAT effectively evolves large scale neural network on spatially related inputs, and the dimensions of both the feature maps and the VPR are spatially related.

10 Conclusion

This project investigated the potential of combining neuroevolution and deep learning in the context of a FPS game by implementing and evaluating two different integrations of the two AI algorithms. The goal of the experimentation was to answer the following research question:

- How can supervised deep learning and neuroevolution be combined to create a visual FPS agent capable of aiming and shooting?

The results of the experiments indicate, that both variations of the feature representations in conjunction with NEAT and deep learning can produce an agent capable of aiming and shooting in a limited representation of a FPS game. The results prove that NEAT is capable of developing networks that can aim and shoot based on both representations. However, the developed agents failed to learn proper reloading and tap-fire. The experiments makes a satisfactory estimation of the AR plausible, by showing that increasing depth decreases the error, and proves that estimating the VPR is possible for deep learning.

The lack of proper hardware, performance optimisation of the pipeline and optimisation of the supervised learning process clearly makes the agent perform worse, and these shortcomings should be addressed for this approach to be viable. The project also set out to answer the following research question:

- Which feature representation of a visual partially-observable state makes the combination of neuroevolution and supervised deep learning perform well?

We presented two different feature representations, the visual partitioning representation and the angular representation. Both of the representations proved useful, as the VPR was estimated successfully with deep learning with little optimisation, while the AR proved a better foundation for neuroevolution. Whether the correct choice of feature representation for future implementations of the combination is the AR or the VPR depends on the difficulty of the visual recognition task. If training examples are subject to labelling, or the target detection is challenging, it is more realistic to estimate the VPR. However, if large volumes of training examples and time for hyper parameter and topological optimisation are available, using the AR could yield faster neuroevolution and a more intelligent agent.

Appendices

A Technical Description

A.1 Hardware used for training

All training of the CNN was done on a server, at the IT University of Copenhagen, with the following specs:

CPU Intel® Core™ i7-5820K

GPU 2x GeForce® GTX TITAN X

RAM 32 GB HyperX Fury DDR4 2400 C15

Each training experiment was done using one of the GPUs and up to 12 GB of RAM allowing for two simultaneous runs.

A.2 Communication between the AIC and the VRC

The game used throughout the project was made using the game engine Unity, where it is possible to interface with the game using either C#, JavaScript, Boo or Unity Script. Since the game has to send the pixel data to a CNN to obtain the inputs for the AIC, the easiest option would be to implement the VRC in one of the languages supported by Unity.

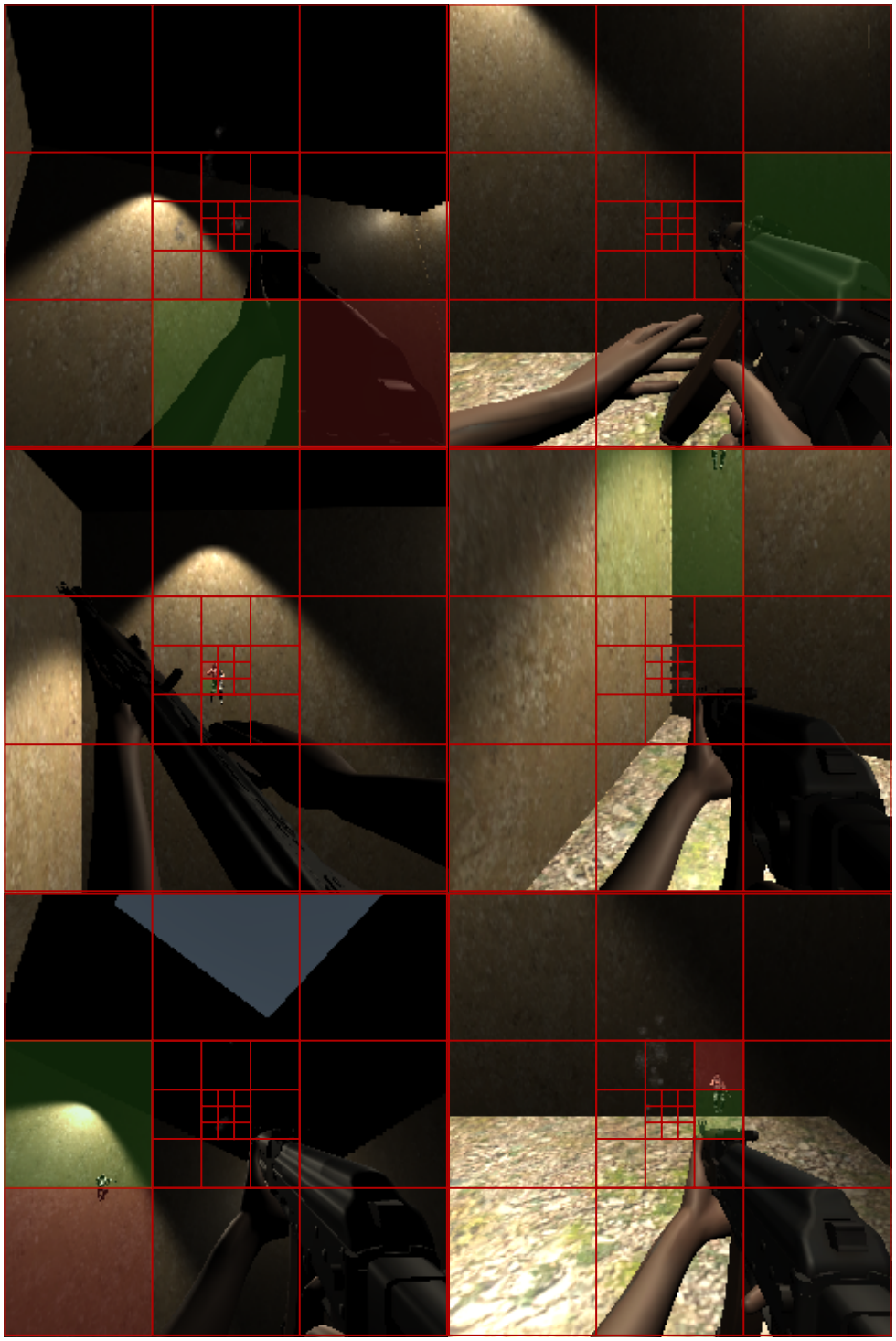
Unfortunately we were unable to find a framework suitable for our needs, why we decided to look for frameworks in other languages. Java is the language we are most comfortable in, why we decided to use a Java framework, called DeepLearning4J[19].

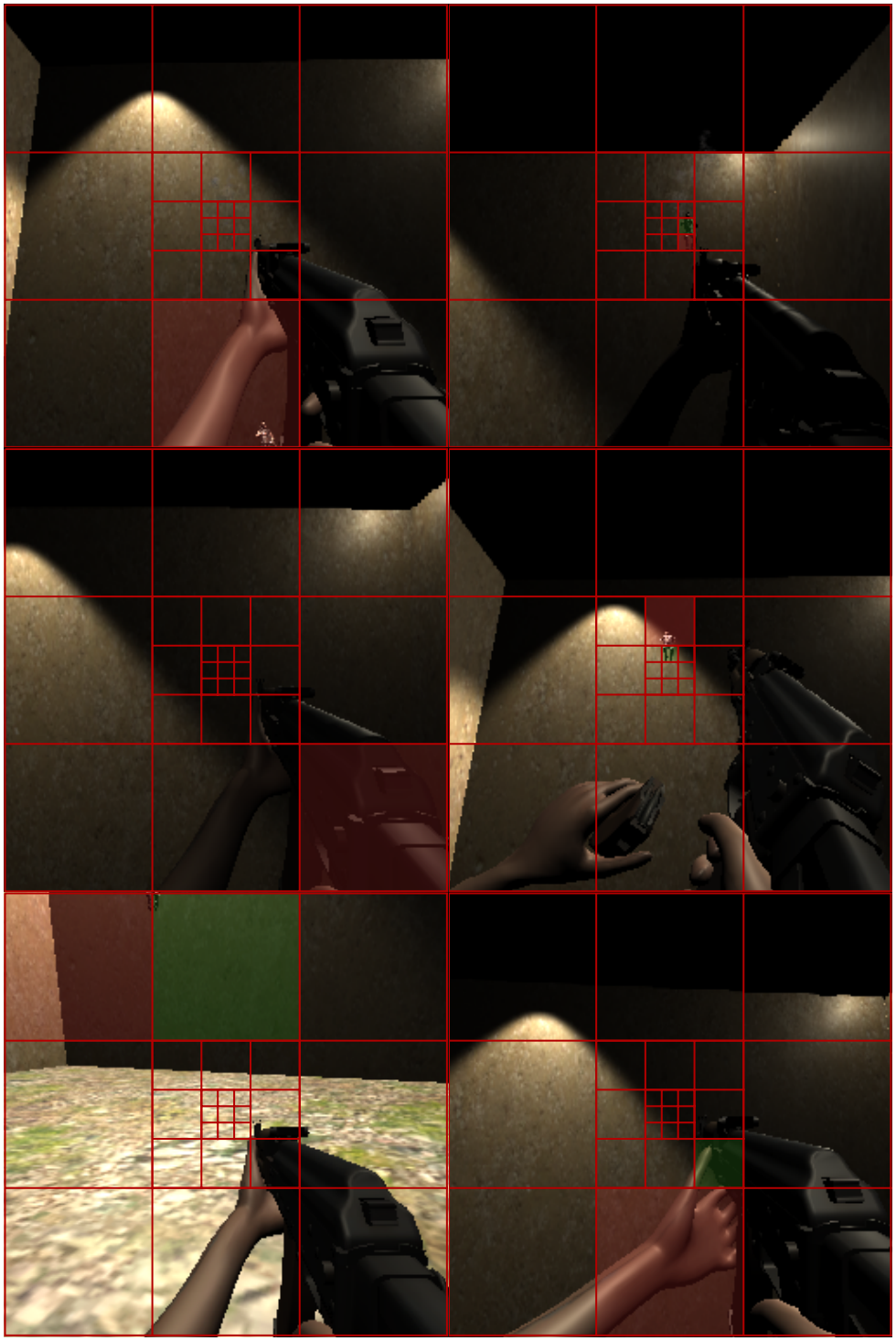
Neither of the languages used to interface with the game can natively communicate with Java. In order to allow communication between the VRC and the AIC a simple socket bridge was implemented. The overhead introduced by this bridge is minimal as it is possible to send the required data back and forth 10,000 times in approximately three seconds.

[19] `https://deeplearning4j.org/`

B Incorrect predictions

The following examples are incorrect predictions by the deep visual partitioning convolutional neural network. The green squares mark the ground truths, and the red squares mark the wrong predictions. Absence of either the red or the green square means that either the the prediction or the ground truth is that no target is present on the screen.





C Angular representation error

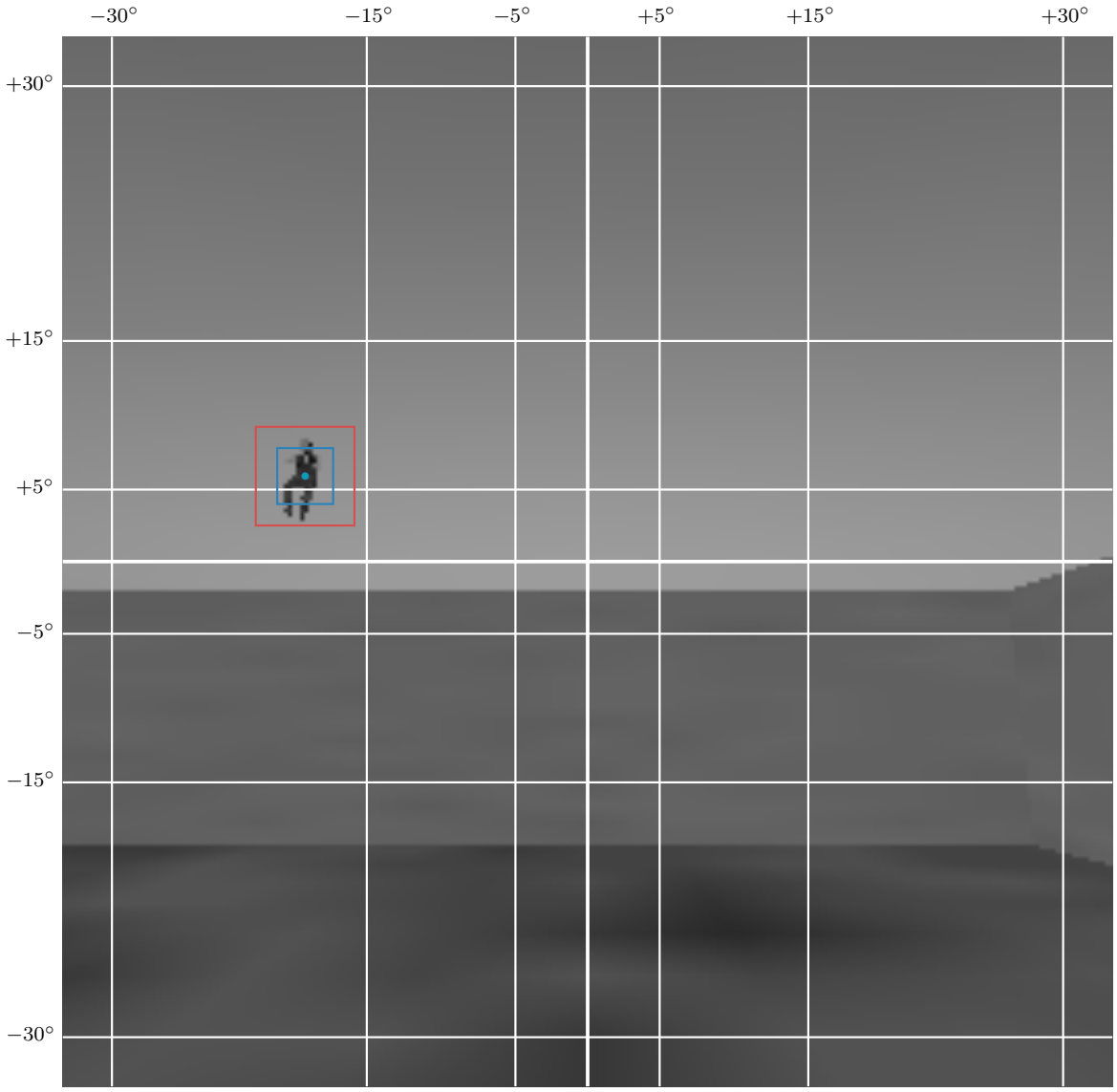


Figure 35: The blue box marks the average error margins for horizontal and vertical angles for the deep CNN without visual distortion, while the red box marks the same error for the shallow CNN.

Figure 36: The green dot marks the correct position and the red dot marks the estimated position using the deep network.

D Visually distorted examples

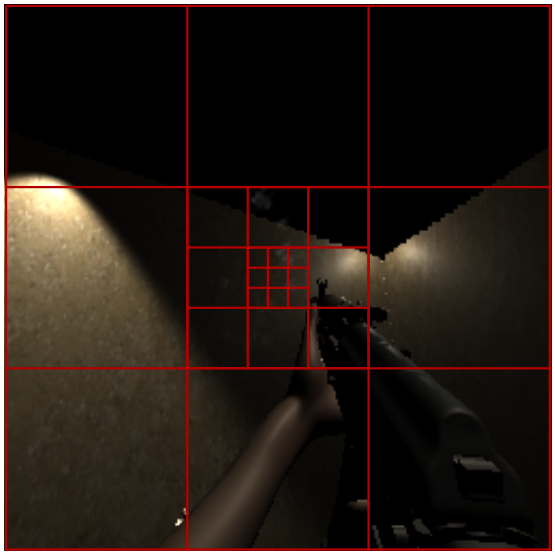


Figure 37: The deep CNN estimating the VPR correctly predicts the class(id 0) with a confidence of 71.9%.

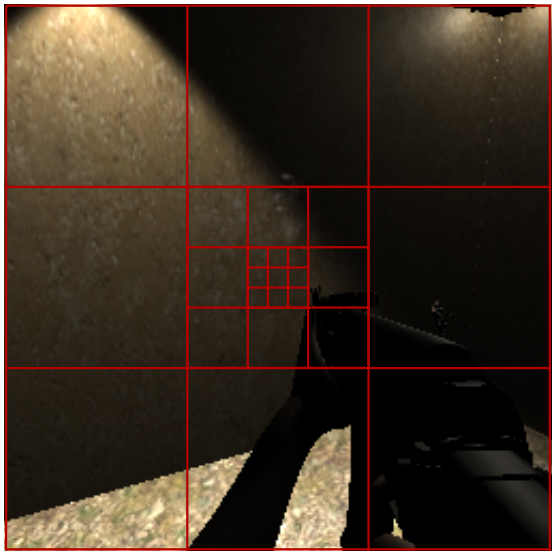


Figure 38: The deep CNN estimating the VPR correctly predicts the class(id 4) with a confidence of 78.0%.

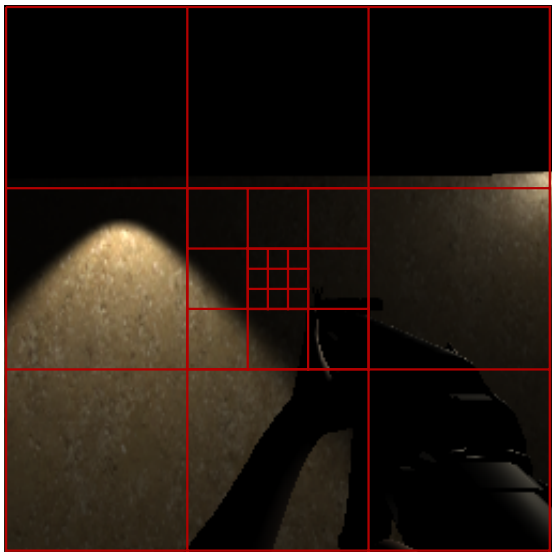


Figure 39: The deep CNN estimating the VPR incorrectly predicts the class(id 2) with a confidence of 58.9%. The correct class has id 4.

E Feature maps

The following feature maps are from the image in figure 40.

Figure 40: The green square marks the correct class.

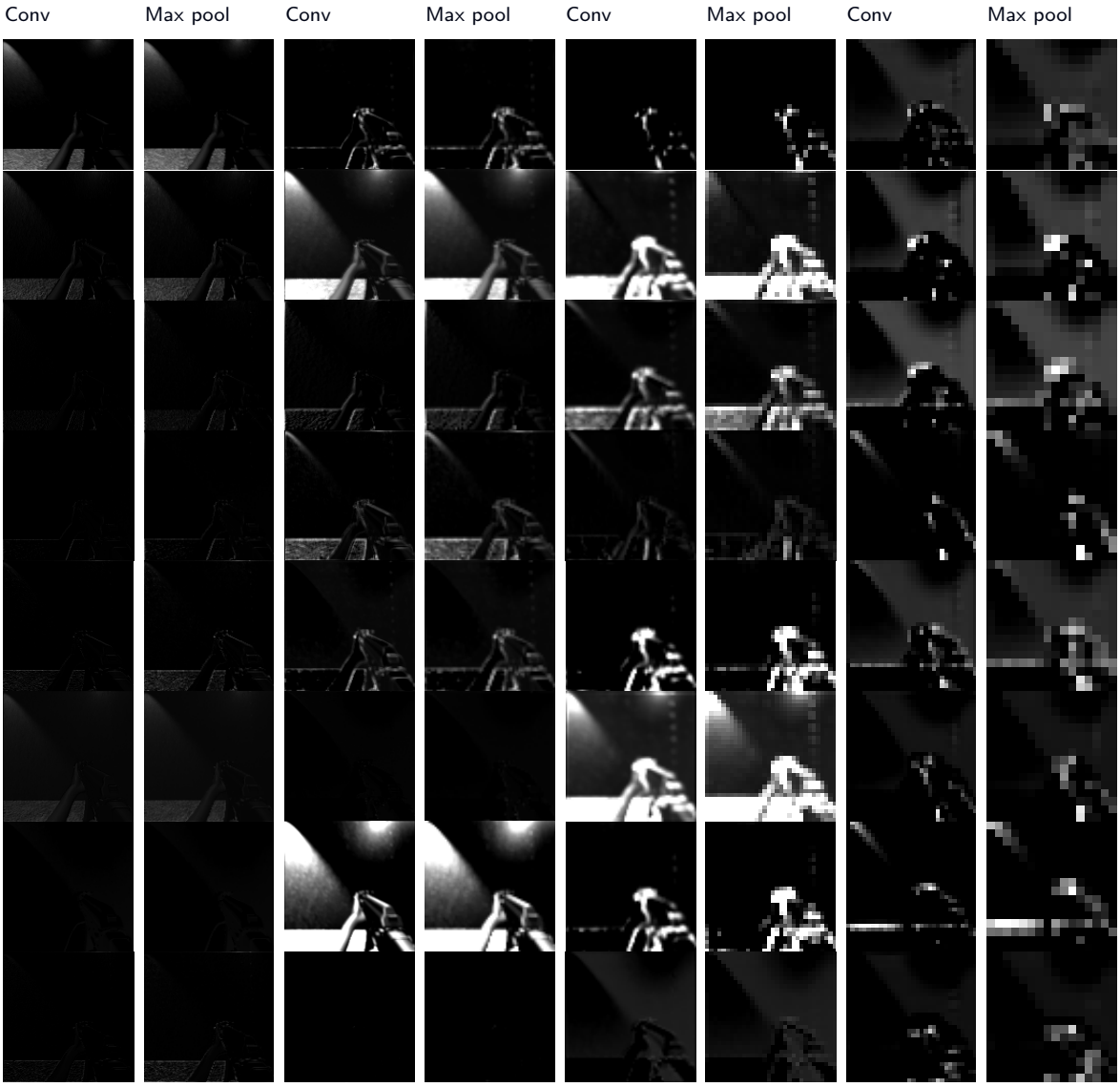


Figure 41: The activations of the convolutional layers in the deep CNN approximating the VPR. The feature maps highlights the position of the target behind the weapon overlay.

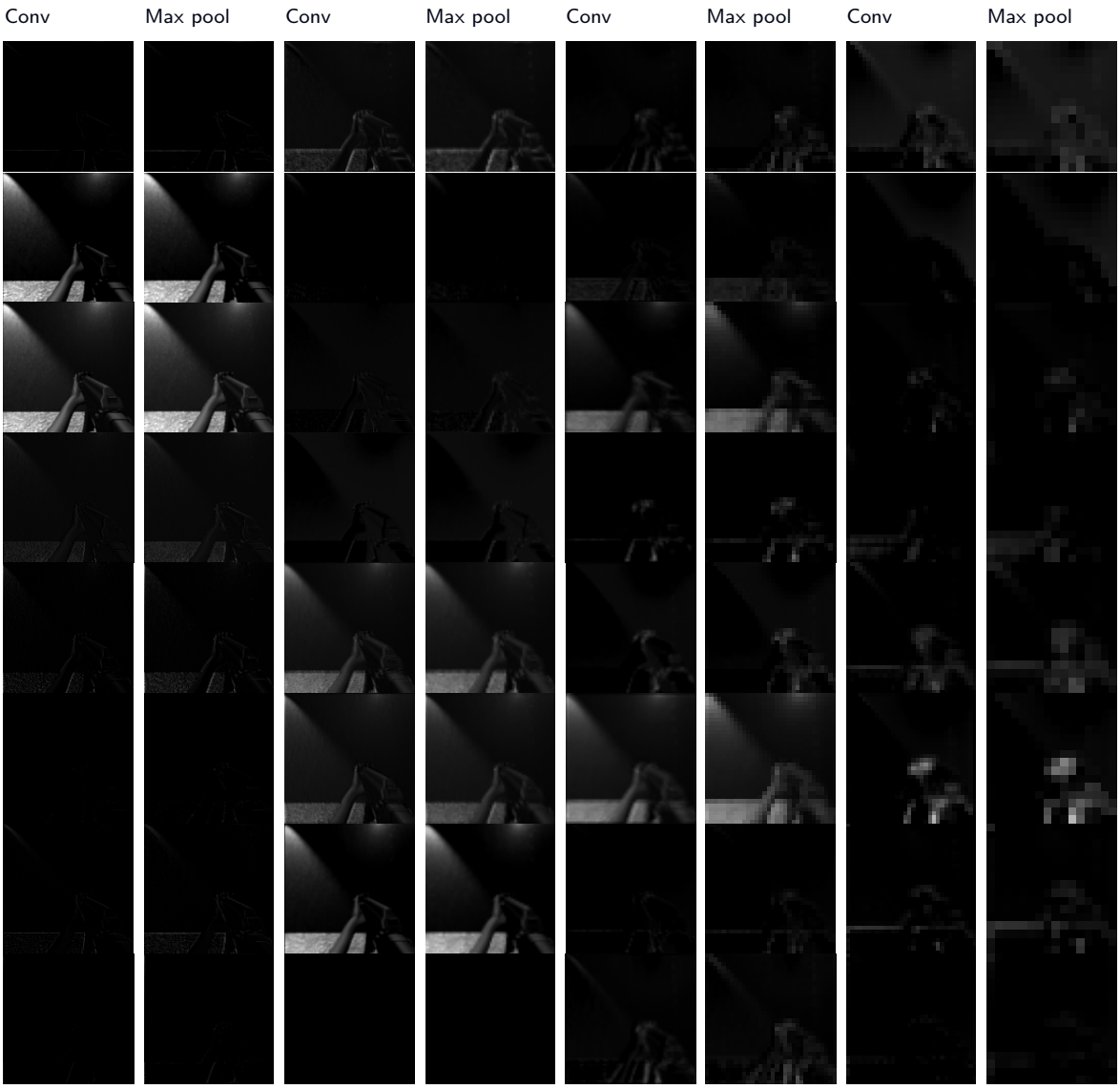


Figure 42: The activations of the convolutional layers in the deep CNN approximating the AR. The feature maps vaguely highlights the position of the target behind the weapon overlay.

F Partitioning scheme

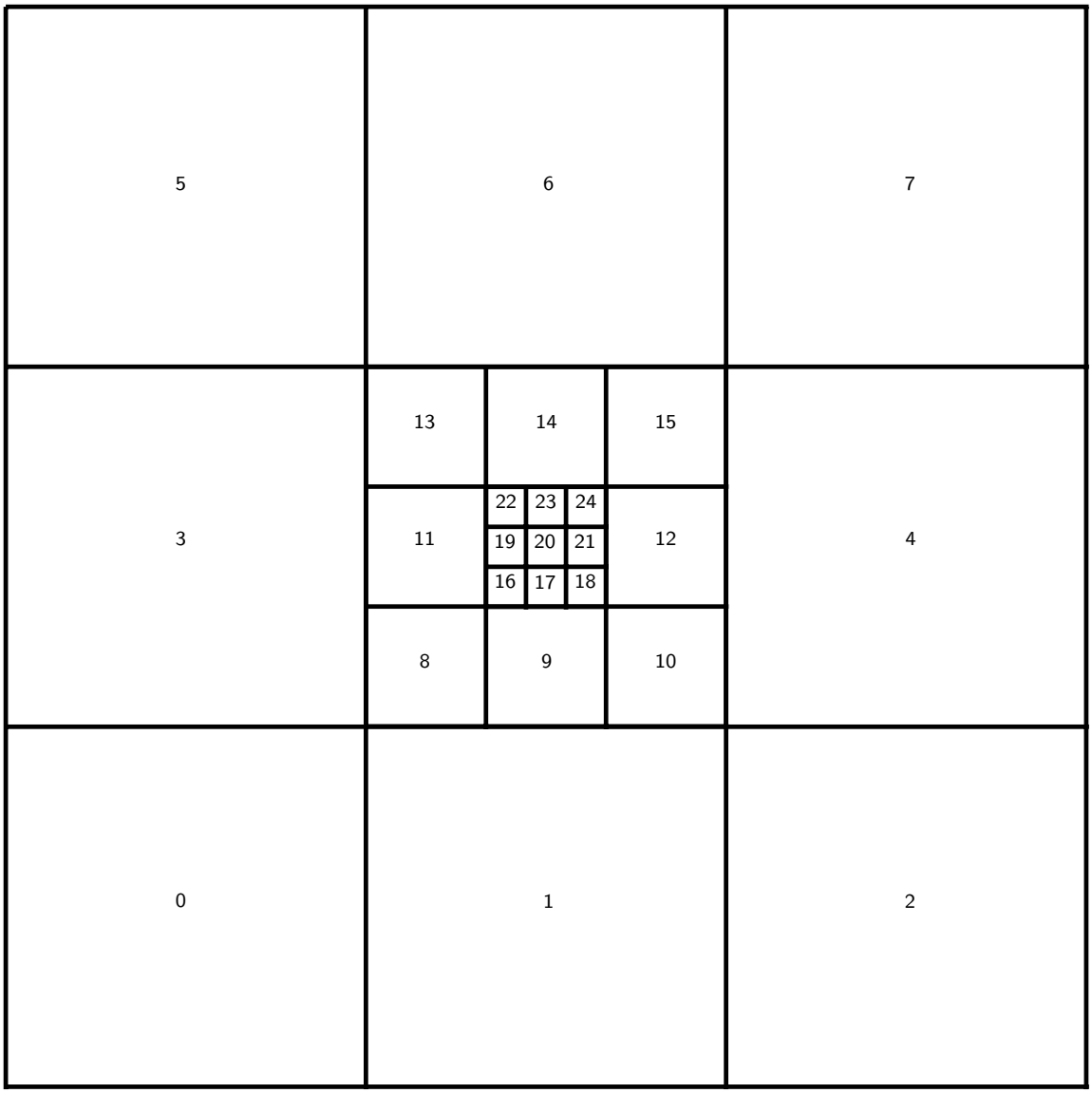


Figure 43: Ids of the 3, 3, 3 partitioning scheme

G Neuroevolution graphs

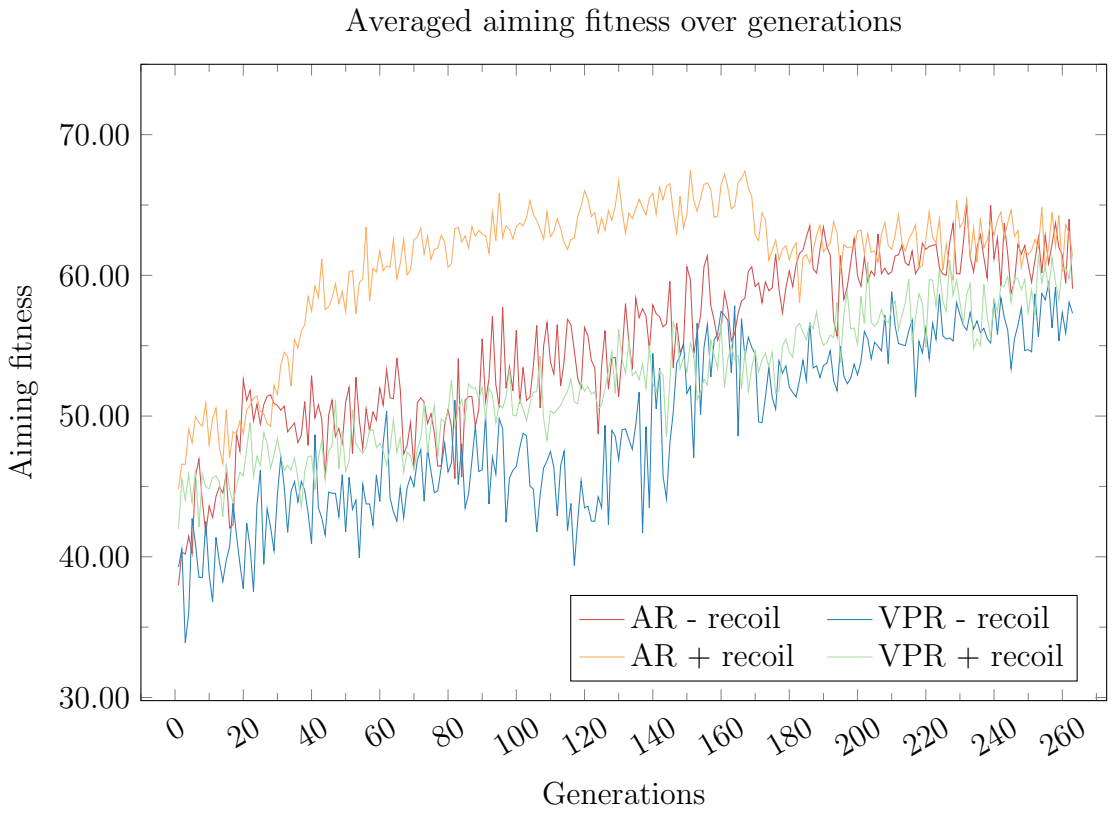


Figure 44: The aiming fitness averaged. Each graph is an average of 3 runs.

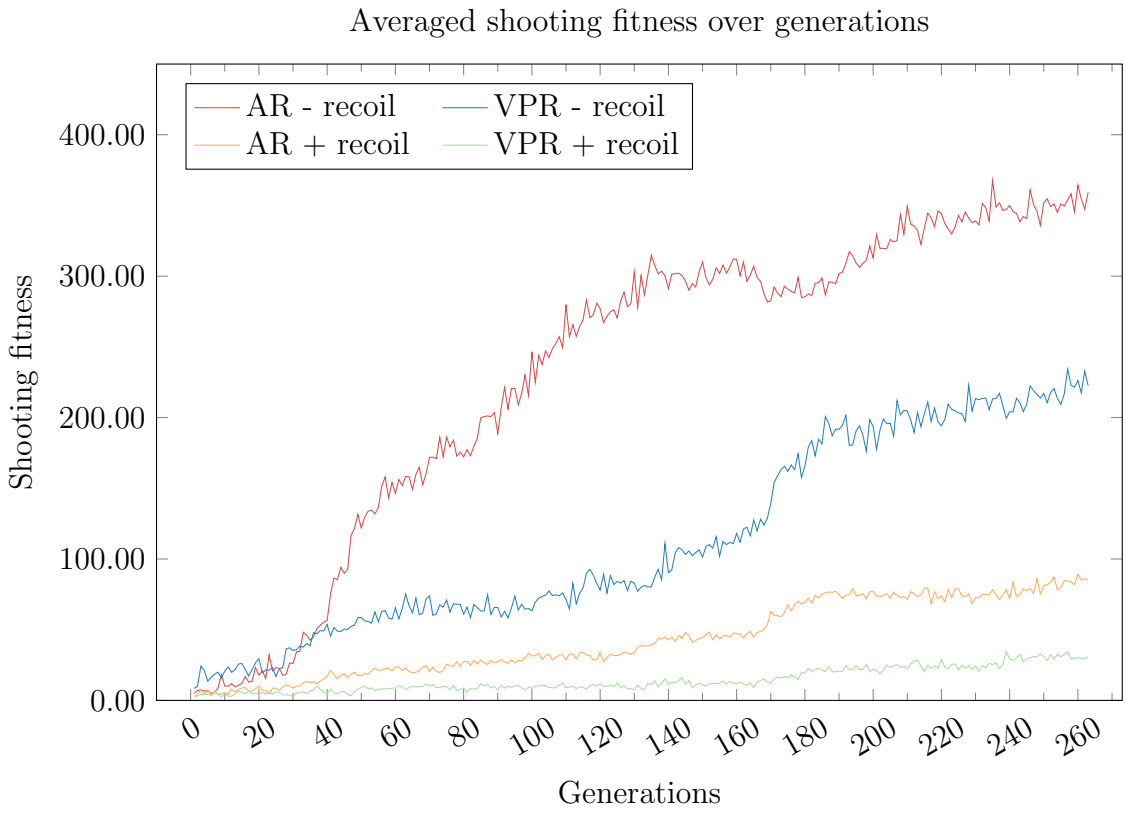


Figure 45: The total shooting fitness averaged. Each graph is an average of 3 runs.

Figure 46: The total shooting and aiming fitness averaged without recoil for the AR. Each graph is an average of 3 runs.

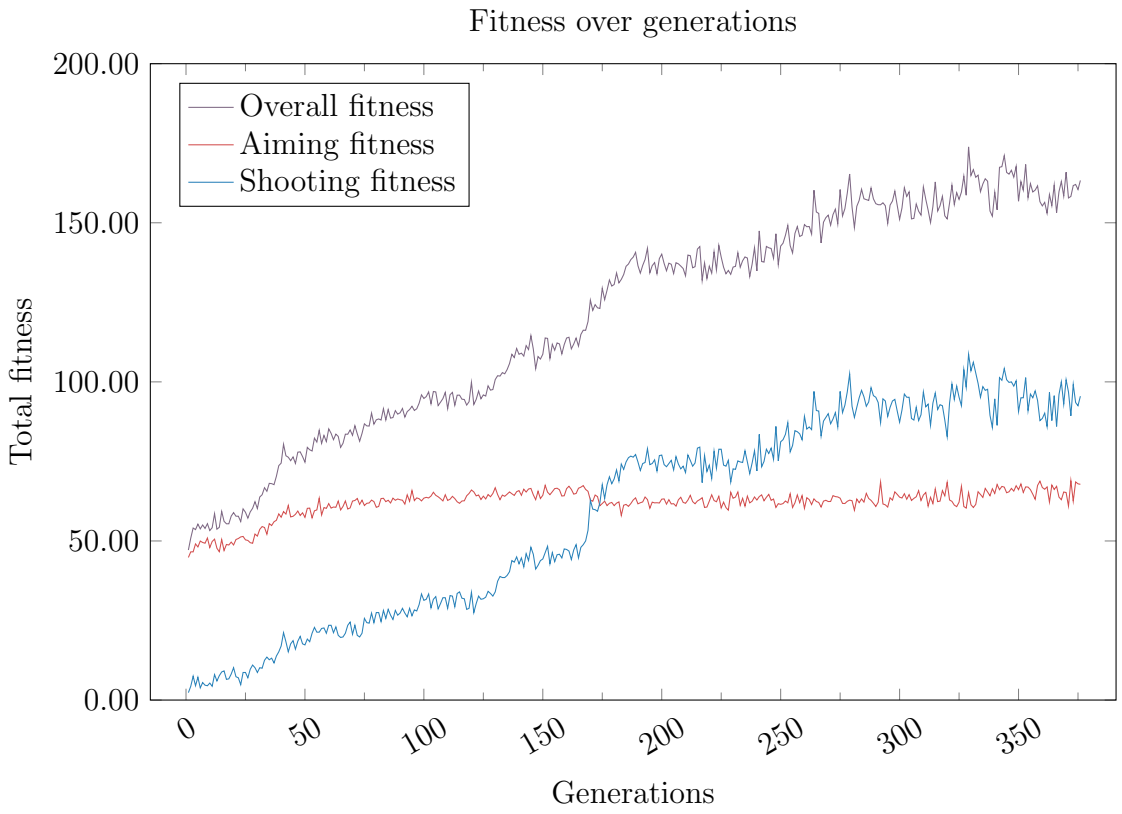


Figure 47: The total shooting and aiming fitness averaged with recoil for the AR. Each graph is an average of 3 runs.

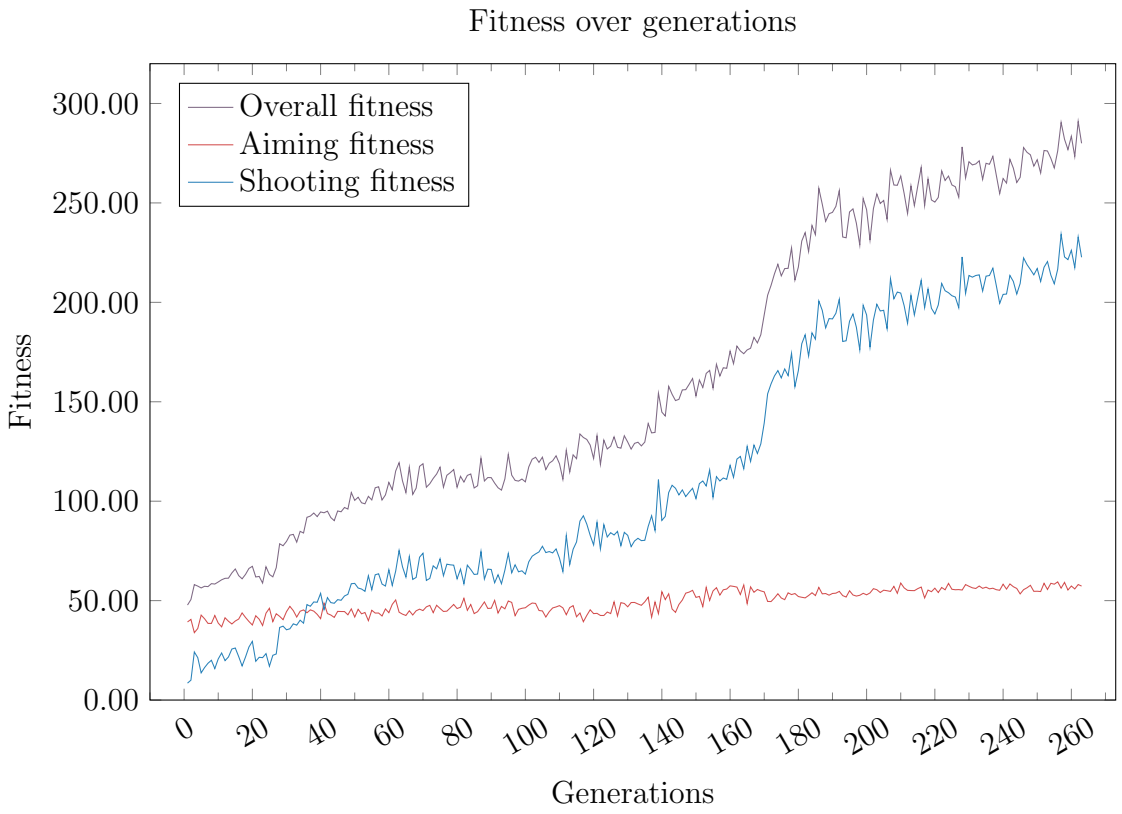


Figure 48: The total shooting and aiming fitness averaged without recoil for the VPR. Each graph is an average of 3 runs.

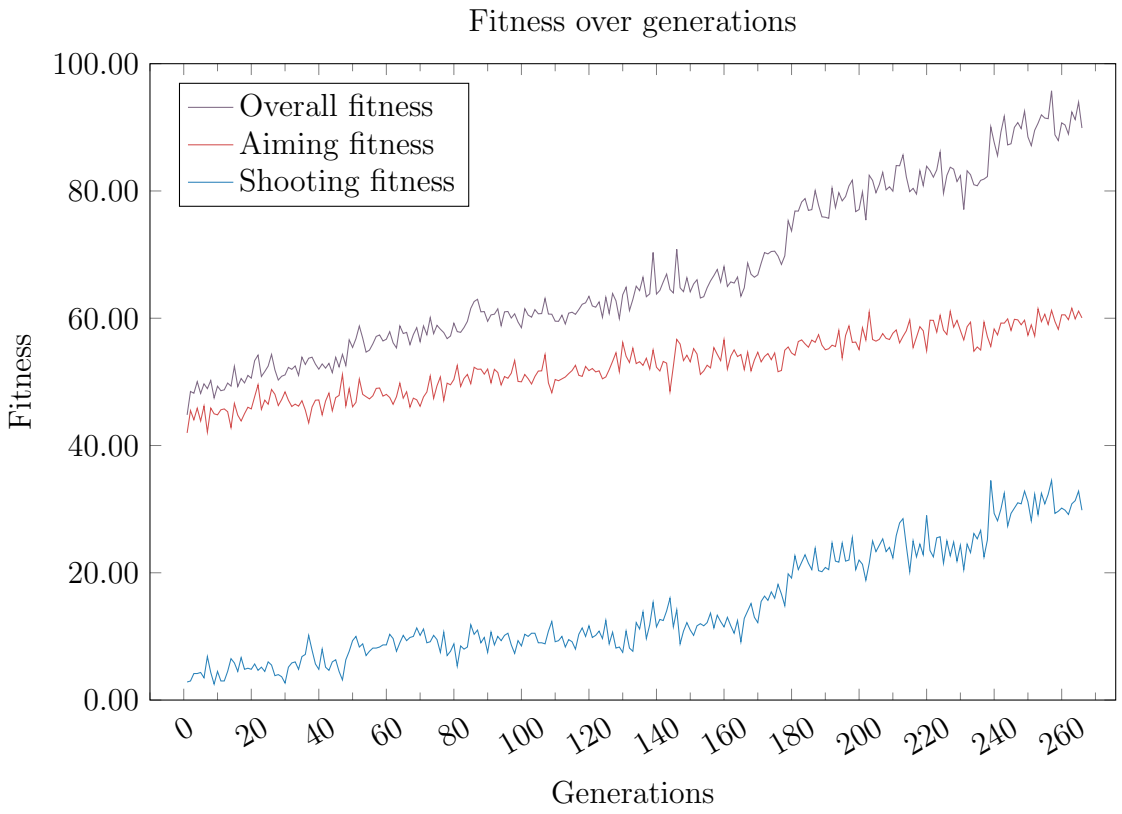


Figure 49: The total shooting and aiming fitness averaged with recoil for the VPR. Each graph is an average of 3 runs.

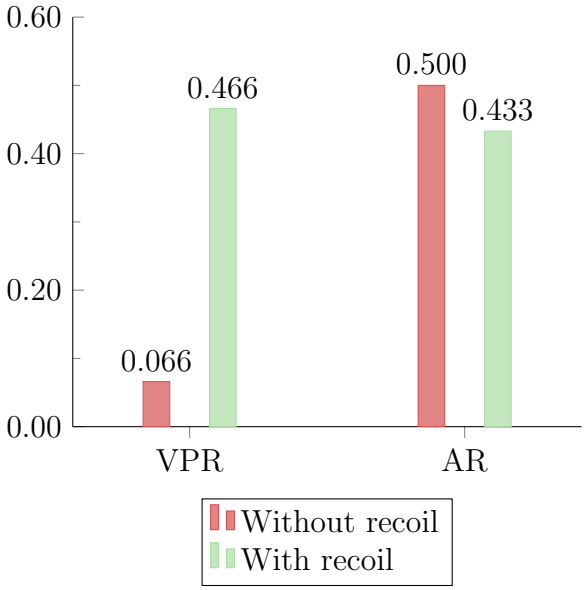


Figure 50: Average reloads with full magazine after 250 generations

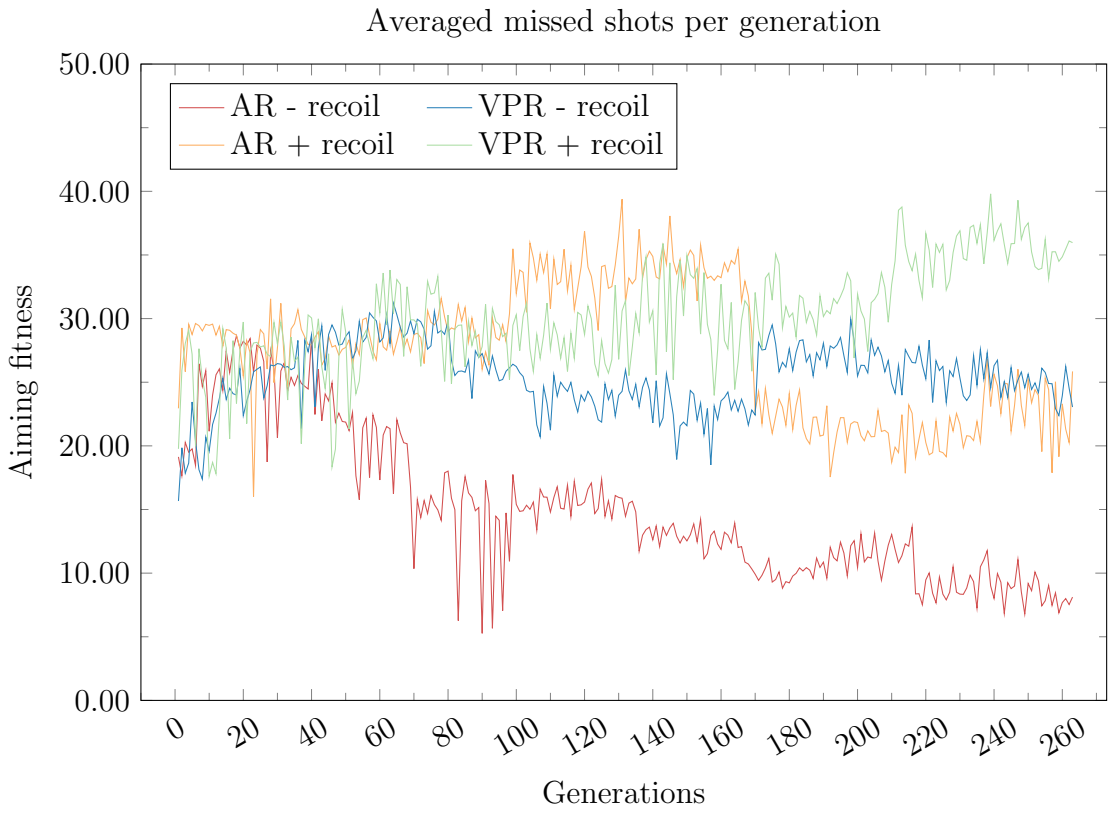


Figure 51: The number of missed shots are averaged from 3 runs.

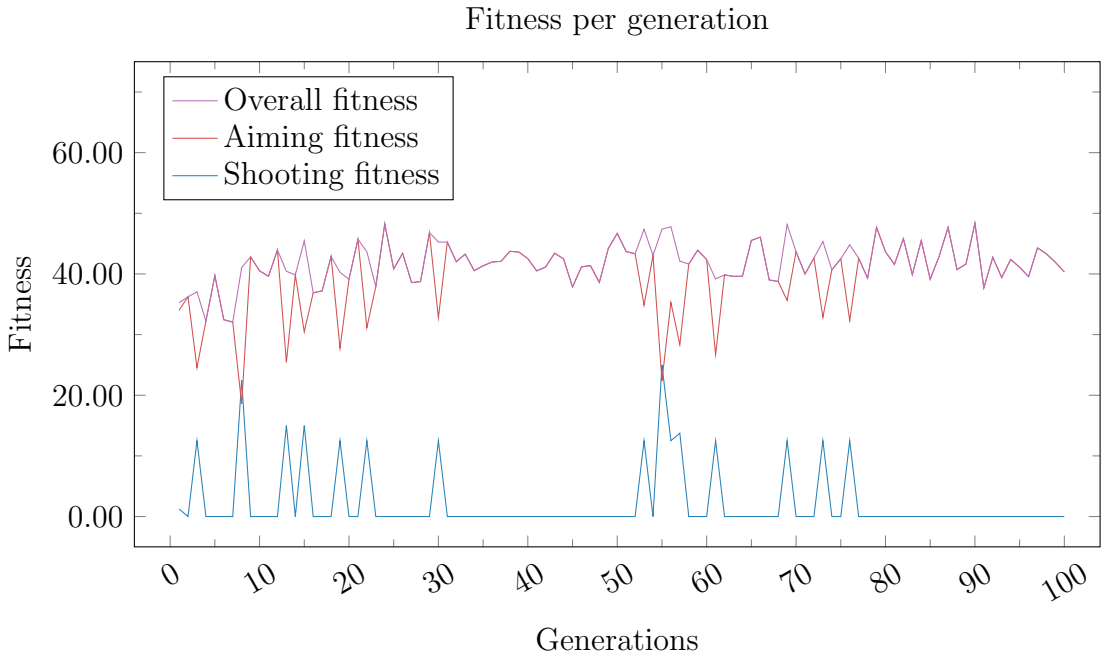


Figure 52: Fitness from training an agent with neuroevolution using direct visual input scaled to 28x28 and greyscaled using visual distortion without recoil. The approach is the same as the neuroevolution experiments in section 6, except that the input is raw pixels.

H Deeper convolutional neural networks

The deeper CNN estimating the AR was trained in the same way as the networks described in section 5, except that the batch size was set to 5 due to memory implications. The deeper network has an additional convolutional layer and max pooling layer as well as 5 fully connected layers with 1,000 neurons in each instead of 3 layers with 250. This sums up to 16 layers. The graphs show that the deeper network has approximately half the horizontal and vertical mean error of the deep network.

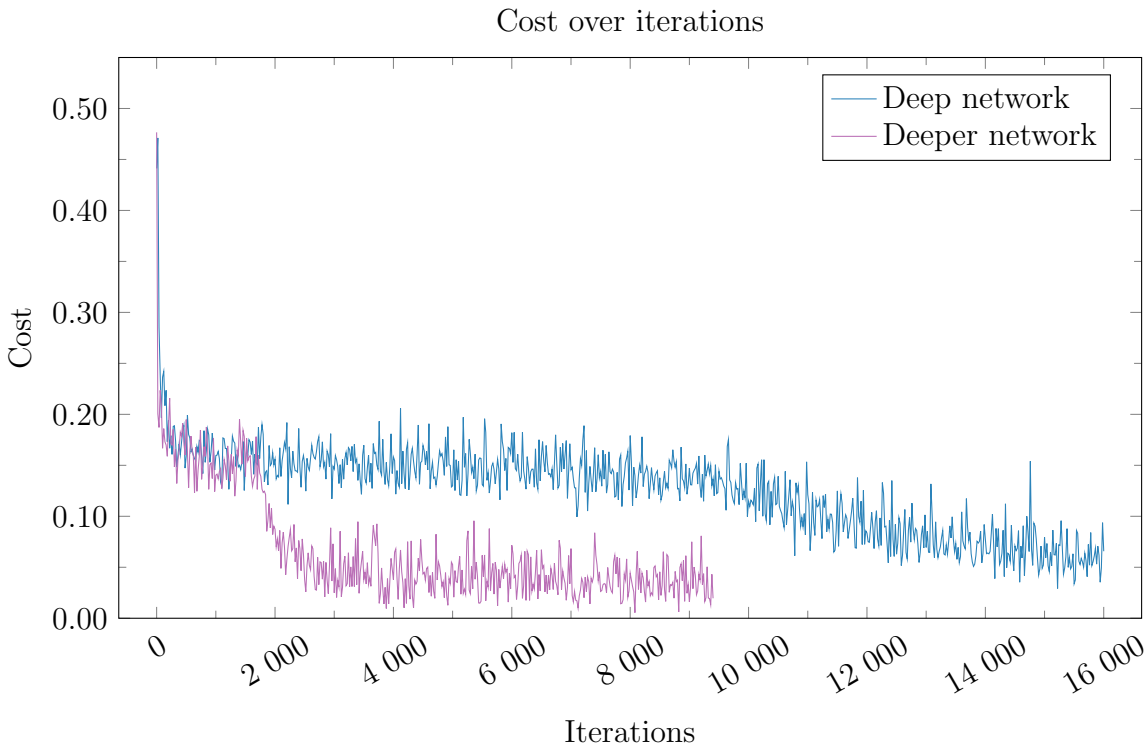


Figure 53: Mean squared error cost over mini-batch gradient descent iterations using AR with visual distortion

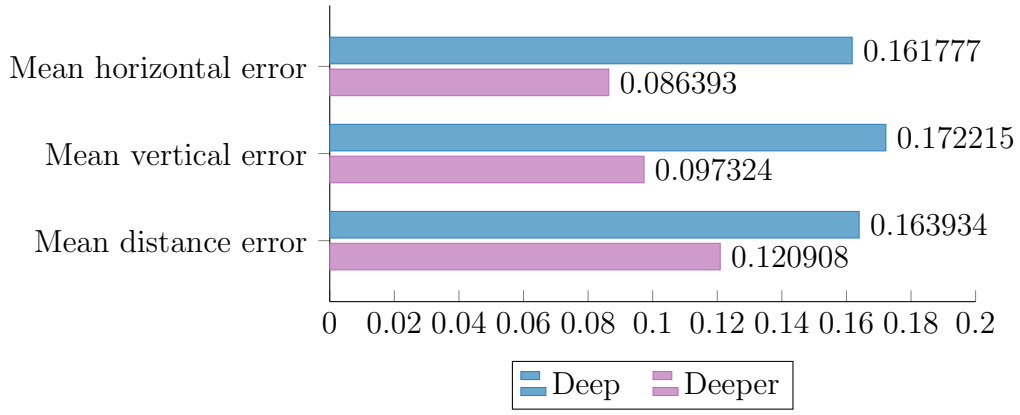


Figure 54: Mean absolute error of the deeper topology compared to the deep topology with visual distortion, evaluated on a test set.

References

[1] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[2] J. Koutník, J. Schmidhuber, and F. Gomez, "Evolving deep unsupervised convolutional networks for vision-based reinforcement learning," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, (New York, NY, USA), pp. 541–548, ACM, 2014.

[3] C. Chen, A. Seff, A. L. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," *CoRR*, vol. abs/1505.00256, 2015.

[4] "Genre breakdown of video game sales in the united states in 2015." https://www.statista.com/statistics/189592/breakdown-of-us-video-game-sales-2009-by-genre/, 2016. Accessed: 2016-15-10.

[5] K. Hornik, M. B. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[6] Y. Bengio and O. Delalleau, "On the expressive power of deep architectures," in *Discovery Science - 14th International Conference, DS 2011, Espoo, Finland, October 5-7, 2011. Proceedings*, p. 1, 2011.

[7] A. C.-C. Yao, "Separating the polynomial-time hierarchy by oracles," in *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 1985.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[9] K. O. Stanley and R. Miikkulainen, "Evolving neural network through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[10] S. A. Solla, E. Levin, and M. Fleisher, "Accelerated learning in layered neural networks," *Complex Systems*, vol. 2, 1988.

[11] Y. Nesterov, "A method of solving a convex programming problem with convergence rate o(1/k2)," *Soviet Mathematics Doklady*, 1983.

[12] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, pp. 249–256, 2010.

[13] D. Masko and P. Hensman, "The impact of imbalanced training data for convolutional neural networks," 2015. Accessed: 2016-12-01.

[14] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, "Vizdoom: A doom-based AI research platform for visual reinforcement learning," *CoRR*, vol. abs/1605.02097, 2016.

[15] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," *CoRR*, vol. abs/1609.05521, 2016.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[17] B. Geisler, "An empirical study of machine learning algorithms applied to modeling player behavior in a first person shooter video game," 2002.

[18] S. Bonacina, P. Luca Lanzi, and D. Loiacono, "Evolving dodging behavior for openarena using neuroevolution of augmenting topologies," 2008.

[19] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *CoRR*, vol. abs/1410.7326, 2014.

[20] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial Life*, vol. 15, no. 2, pp. 185–212, 2009.