Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Intermediate Representations in Deep Multimodal Neural Networks

Masterarbeit (M.Sc. Thesis)
im Arbeitsbereich Knowledge Technology, WTM
Prof. Dr. Stefan Wermter

Department Informatik
MIN-Fakultät
Universität Hamburg

vorgelegt von
Fares Abawi
am
03.04.2019

Gutachter: Prof. Dr. Stefan Wermter
Dr. Manfred Eppe

Fares Abawi
Vogt-Kölln-Strasse 30
22527 Hamburg

Abstract

Artificial neural networks designed to learn multiple tasks were shown to outperform those with a single objective. We explore the effectiveness of introducing multiple auxiliary tasks to improve the performance of a multimodal neural network. We create a synthetic dataset explicitly designed for a robotic grasping task, with the goal of grasping objects and relocating them based on natural language commands. We design a neural model combining multi-sensory input through intermediate fusion. The multimodal integration process combines individual deep neural models specializing in specific tasks, composing a single network for grasping objects given an image and a natural language command. The vision and natural language modalities perform domain-specific tasks with independent output heads branching from both neural models. We refer to these output heads as intermediate representations. We use a symbolic Robot Command Language (RCL) as an intermediate representation between the language network and the fusion network. The vision network has two intermediate representations for localizing objects in images and for classifying them.

We ablate the intermediate representations forming all possible combinations. Our experiments show that certain intermediate representations result in an overwhelming loss contribution to the entire model, distorting the main task's objective. However, other losses contribute positively to the model's overall performance and act as regularizers. Our results also indicate that choosing RCL as an intermediate representation outperforms natural language as an intermediate representation.

Zusammenfassung

Es wurde gezeigt, dass künstliche neuronale Netzwerke, die zum Erlernen mehrerer Aufgaben entwickelt wurden, jene neuronale Netzwerke übertreffen, denen lediglich eine einzige Aufgabe zugeteilt wurde. Wir untersuchen die Wirksamkeit der Einführung mehrerer Hilfsaufgaben, um die Leistung eines multimodalen neuronalen Netzes zu verbessern. Dazu erstellen wir einen synthetischen Datensatz, der explizit für eine Roboter-Greifaufgabe konzipiert wurde, mit dem Ziel, Objekte zu erfassen und sie zu versetzen, dieses auf der Grundlage von Befehlen in natürlicher Sprache. Wir entwerfen ein neuronales Modell, das multisensorischen Input durch Zwischenfusion kombiniert. Der multimodale Integrationsprozess kombiniert einzelne Deep Neural-Modelle, die sich auf spezifische Aufgaben spezialisieren, und bildet ein Netzwerk um Objekte zu greifen auf Grundlage von einem Bild und einem Befehl in natürlicher Sprache als einzige Informationsquellen. Die Sicht- und Sprach-Modalitäten führen domänenspezifische Aufgaben mit unabhngigen Ausgaben, die aus den beiden neuronalen Modellen auszweigen. Wir bezeichnen diese Ausgaben als Mitteldarstellungen und verwenden eine symbolische Robot Command Language (RCL) als Darstellung zwischen dem Sprachnetzwerk und dem Fusionsnetzwerk. Das Netzwerk, welches sich mit den Bildern befasst, verfügt über zwei Zwischendarstellungen um Objekte in Bildern zu lokalisieren und sie zu klassifizieren.

Wir ablatieren die Zwischendarstellungen um so alle möglichen Kombinationen zu bilden. Unsere Experimente zeigen, dass bestimmte Zwischendarstellungen zu einem überragenden Verlustbeitrag zum gesamten Modell führen und das Ziel der Hauptaufgabe verzerren. Andere Verluste wirken sich jedoch positiv auf die Gesamtleistung des Modells aus und wirken als regularisierend. Unsere Ergebnisse zeigen auch, dass die Wahl der RCL als intermediäre Darstellung die natürliche Sprache als intermediäre Darstellung übertrifft.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robots have become an integral part of the industry, performing tasks which are considered too repetitive, precise or dangerous for humans. Many of these tasks require robots to interact with people, providing them with assistance. One such robot is NASA's Robonaut which helps astronauts during space missions [2]. As more tasks require engagement with humans, the need for communication with robots becomes vital. In the field of human-robot interaction, researchers aim to facilitate interaction with robots and provide a seamless means for conversing with them through verbal communication (oral and written). The goal is to allow robots to perform actions requested by people easily and naturally.

Robots would need to interact with their environment in response to requests by people. Grasping objects, which we as humans would consider a trivial task, remains a challenge for robots and machines to perform. Initial research focused on analytical methods for predicting grasp coordinates [77, 69]. Nowadays neural networks are commonly used for visuomotor tasks concerning object grasping and localization [21, 50]. A trend towards end-to-end learning developed with the advancement of deep neural networks and the availability of huge data sources [45]. End-to-end architectures are often criticized for their sizeable computational overhead and big data requirements, however, multi-task learning [13, 90] has been shown to improve performance with fewer data [84, 46].

Some of the mentioned studies rely on multiple sensory inputs, fused within the networks. Fusion of multi-sensory input could either be performed by removing correlations across sensory input sources or by reducing them to a common dimension [74]. Fusing units within neural networks is known as intermediate fusion [86]. Examples on intermediate fusion include [80, 59]. Late fusion in the context of deep neural architectures is employed through voting or ensemble networks, combining classification outcomes of multiple networks [12]. However, since late fusion relies on inferred values and not on the input data, it is not robust to subtle changes in the sensory input, especially when operating on inputs from different modalities.

In this Thesis, an intermediate representation refers to a structured, symbolic or feature output by a subnetwork. Intermediate representations learn auxiliary tasks, thus simplify the learning target at multiple stages within the network. Having an intermediate output branching from a subnetwork is a form of multi-task

learning with enforced hierarchy [34]. In other words, the fusion network processes the learned information without attempting to deduce an encoding that unifies all the modalities. An intermediate representation for natural language could be expressed in the form of a logical representation. Zettlemoyer and Collins propose an algorithm for lambda-calculus parsing of semantic representations from natural language sentences [107]. Berant et al. [8] propose an improvement upon Zettlemoyer's and Collins' approach by mapping natural language phrases to logical predicates through the alignment of a text corpus to Freebase [10]. For the purpose of this Thesis, we choose to use a more simplified language called the Robot Command Language (RCL) [19] directed towards the robotic command spatial domain.

The combination of multiple task oriented subnetworks results in the complete multimodal architecture. Multi-task learning approaches introduce multiple heads as final outputs for the entire model; however, it is uncommon to address intermediate representations at different stages within the network. Having intermediate outputs for each subnetwork encourages each network to specialize in its task while providing context for other subnetworks.

We focus on integrating multi-sensory input to realize a sensorimotor encoding, which allows a robot to grasp an object and place it at another location. This research will be centered around creating a network architecture which enables the combination of transcribed natural language input and visual input. Given natural language input, together with visual data, a robot will not only have to understand the visual scene, but it would also require a semantic understanding of the context of the utterance. The context is inferred by the combination of all the two specialized networks dedicated to understanding visual and textual data independently.

Having an intermediate representation may provide more profound insights into the functionality of the network but may require a hand-crafted labeled dataset. We instead resort to creating a synthetic dataset described in chapter 3. An end-to-end network without intermediate representations may perform better than its counterparts with auxiliary tasks; however, the latent representation may prove human-unreadable, restricting the network to the task for which it was built. We aim to explore whether a neural model can learn to grasp and perform actions yet maintain a coherent encoding at different stages with the employment of intermediate representations. This approach would facilitate the transferability of partial weights (weights belonging to a specific modality in the network) to other models as well.

We construct the modalities of the neural network using multiple deep neural network architectures, set up to address each problem individually. The modalities are then combined into a single network through intermediate fusion, which shall be referred to as FusionNet from this point onwards. The FusionNet decodes the inputs and produces joint coordinates, which trigger a moving arm to act. To develop each module, We chose networks which have been demonstrated to perform well at their task. We employ two networks, one targeted at detecting objects in images, and another for translating natural language commands to RCL

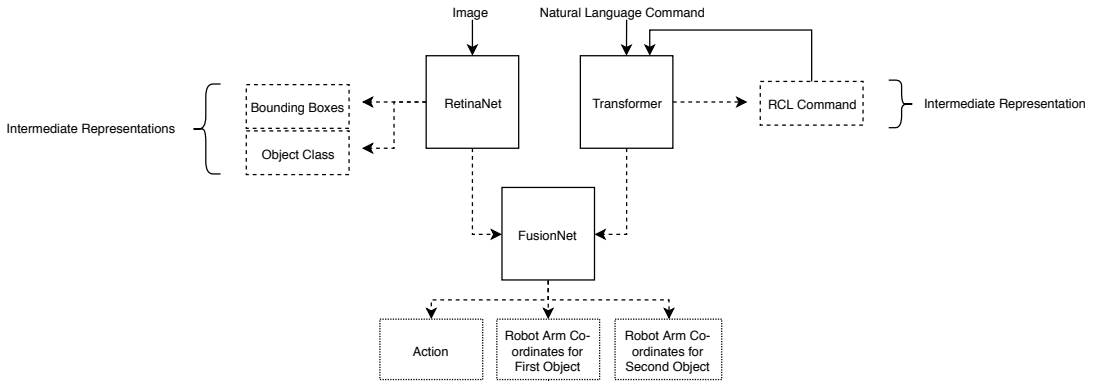


Figure 1.1: An overview of the complete FusionNet architecture

trees.

To translate natural language sentences to RCL, We use the Transformer [103], an attention [5] based feed-forward network. The Transformer outperforms a Convolutional Neural Network (CNN) [57] such as ByteNet [47] and CovS2S [29]. Although RNMT+ [14] outperforms the Transformer, it employs recurrence. Recurrent neural networks have a maximum path length complexity of $O(\log n)$ in comparison with the Transformer having a complexity of $O(\log n)$. The Transformer is, therefore, a more efficient alternative. State-of-the-art results in detecting boundaries and classifying objects in images were achieved by RetinaNet [62] which we use to localize and detect objects.

The ultimate goal is to realize an end-to-end system capable of performing motor actions given multi-sensory input. We employ several intermediate representations branching from the different modalities. The intermediate representation for the machine translation network is the symbolic RCL. The intermediate representation for the vision network is in the form of bounding boxes surrounding the targeted objects as well as their classes. The general structure of our FusionNet model is shown in figure 1.1.

To construct a robust FusionNet, we would need to optimize the modalities. We perform experiments on each modality independently, starting with the Transformer where we optimize the network to our dataset's requirements in chapter 5. The RetinaNet specific experiments are conducted in chapter 6. Eventually, we perform experiments on our FusionNet model and examine the necessity of intermediate representations in chapter 7. In chapter 7, we also test the optimized FusionNet on a simulated model of the NICO [49] robot. In chapter 2, we provide the background information necessary for constructing the FusionNet which is described in chapter 4.

In this Thesis, We aim to answer the following question:

Do intermediate representations increase performance in a multimodal neural network?

- *Is an intermediate symbolic representation required between the language translation network and our fusion network?*
- *Is an intermediate representation required between the vision network and our fusion network?*

Chapter 2

Background Theory

In this chapter, we discuss the different components and building blocks used in the Thesis. We mainly focus on artificial neural networks and detail the mechanism by which they operate. This chapter offers an overview of the tools and methods used, without relating them directly to our approach. We aim to introduce the fundamentals for understanding the scope of the Thesis, without diving deep into every concept and theory. The readers are advised to refer to the cited resources for comprehensive details on certain topics.

2.1 Neural Networks

Artificial neural networks are machine learning models for approximating functions, inspired by the behavior of neurons in brains. This class of techniques relies on observing multiple samples of data, adjusting their parameters (weights) to predict a value which closely resembles the ground truth. Such networks are composed of multiple layers, each learning different features at varying granularity. A neural network composed of multiple layers is known as a **multi-layer perceptron** (MLP) [91]. The layers are composed of units, analogous to their biological parallel known as neurons. The number of units per layer is arbitrary and can be adjusted according to a networks architecture and needs. In its simplest forms, an MLP is composed of three layers: an input layer which takes features as an input, an intermediate (hidden) layer which combines those features, and the output layer which presents the resulting features or classes. The weights mentioned earlier are the parameters of neural networks. These weights are adjusted as the network learns, having values which, when multiplied with their input, will reduce the error of the prediction. As the error reduces, the prediction begins to resemble the ground truth closely. The output of a layer is defined as:

$$y = \sigma(x \cdot w_x + w_b) \tag{2.1}$$

where x is the input vector to the layer y and w is the weight vector for a given unit's connections to units from the preceding layer. A bias unit is introduced, denoted by b, having an independent weight vector w_b. Considering that xw_x results in a

linear function with a gradient influenced by the values of the two vectors, we can influence the steepness of the line, yet shifting the function vertically would not be possible without the added bias w_b. Even though the inner term $x \cdot w_x + w_b$ is suitable to estimate a linear function, many of the problems that would require the usage of neural networks are non-linear by nature. Applying non-linearity to such functions resolves this setback, allowing neural networks to predict them approximately. In equation 2.1, σ denotes the non-linear function applied to the linear computation. A formerly common function known as **sigmoid** was used to introduce such non-linearity. The sigmoid is defined as:

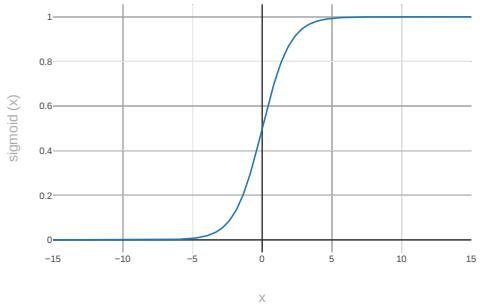


Figure 2.1: The sigmoid function

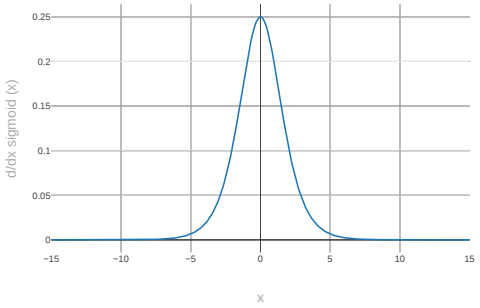


Figure 2.2: The sigmoid function derived with respect to x

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad (2.2) \qquad\qquad \sigma(x)' = \sigma(x) \cdot (1 - \sigma(x)) \qquad (2.3)$$

The sigmoid introduces a non-linearity resembling a smooth step function as shown in figure 2.1. The sigmoid output is limited to a range between $\in [0, 1]$. Such a function is especially useful when the output predicted is expected to be a probability of an event occurring. Suppose an output unit produces a binary output, i.e., $True(1)$ or $False(0)$ for a given category, the network would predict a probability for each of the two class. Another important aspect which should be sought after when choosing an activation function (a non-linear function) is its differentiability. As shown in figure 2.2, differentiating the sigmoid using equation 2.3 results in yet another function. An activation function must be differentiable in order to update the weights of the network when using techniques such as back-propagation for learning (more on back-propagation in section 2.1.2).

2.1.1 Forward pass

Each iteration of the neural network's learning process is composed of two stages: a forward pass where the output is computed, and a backward pass through which the weights are updated. The forward pass is identical to the process followed during inference. Suppose we try to perform the binary addition of two bits, i.e.:

$\{1\} + \{0\}$ equals $\{0, 1\}$. In figure 2.3, we observe the weight placeholders for each connection. We compute the projection of each input vector on to the weight vector and apply the activation function for each layer iteratively. Assuming the weights have already been acquired as shown in table 2.1 and table 2.2, the first step is to compute the outputs of each intermediate layer. To compute the output

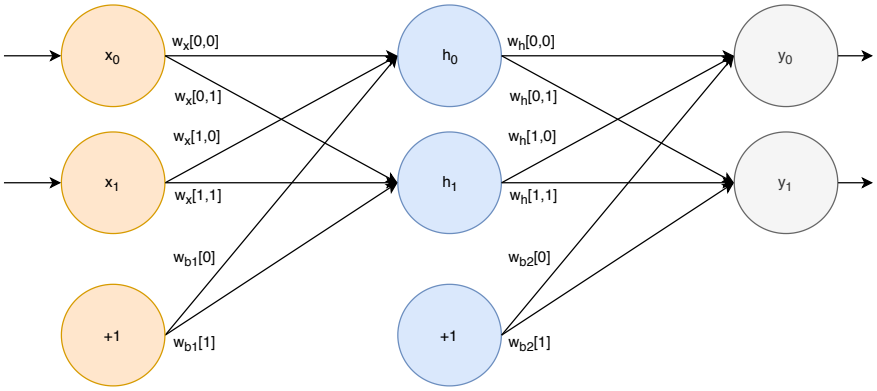


Figure 2.3: A simple feed forward fully-connected neural network with two layers. The input units are denoted by x, the intermediate units by h and the output units by y. Bias units are displayed as $+1$

Table 2.1: Connection weights

$w_x^{[0,0]}$	6.04	$w_h^{[0,0]}$	10.01
$w_x^{[0,1]}$	−6.05	$w_h^{[0,1]}$	−10.03
$w_x^{[1,0]}$	6.04	$w_h^{[1,0]}$	−2.62
$w_x^{[1,1]}$	−6.05	$w_h^{[1,1]}$	−10.29

Table 2.2: Bias weights

$w_{b1}^{[0]}$	−9.22	$w_{b2}^{[0]}$	−4.98
$w_{b1}^{[1]}$	2.48	$w_{b2}^{[1]}$	5.03

of the intermediate unit h_0:

$$\hat{h_0} = \sigma(w_x^{[:,0]} \cdot x + w_{b1}^{[0]}) \tag{2.4}$$

Having inputs of $x_0 = 1$ and $x_1 = 0$, h_0 would be computed as follows:

$$\hat{h_0} = \sigma((6.04 \cdot 1) + (6.04 \cdot 0) + (-9.22 \cdot 1)) \approx 0.02 \tag{2.5}$$

Replacing $w_x^{[:,0]}$ with $w_x^{[:,1]}$ and $w_{b1}^{[0]}$ with $w_{b1}^{[1]}$, we can acquire the the value for $\hat{h_1}$, having $\hat{h_1} \approx 0.03$. After acquiring the outputs of the intermediate layer, we proceed to compute the final output given the values of $\hat{h_0}$ and $\hat{h_1}$ as the outputs of the preceding layer. We would as a result get a value of $\hat{y_0} \approx 0.008$ and $\hat{y_1} \approx 0.990$. In table 2.3, the final outputs for all input combinations can be observed. The expected output column indicates the ground truth values, whereas the actual output column represents the resulting values computed by applying the forward pass. We observe that the actual outputs closely approximate the expected outputs.

Table 2.3: Forward pass computation for the binary addition neural network.

Input		**Expected output**		**Actual output**	
x_0	x_1	y_0	y_1	$\hat{y_0}$	$\hat{y_1}$
0	0	0	0	0.001	0.011
0	1	0	1	0.009	0.987
1	0	0	1	0.009	0.987
1	1	1	0	0.989	0.012

2.1.2 Backward pass

In the backward pass, we update the weights for each connection given an error value. The objective is to minimize the error as much as possible between the actual and the expected output. In the context of neural networks, such an error measurement is referred to as the loss.

Loss functions

Depending on the problem at hand, one would choose a loss accordingly. It is common to use the Mean Squared Error (MSE) for regression problems. The MSE measures the average of the squared errors between the the prediction and the ground truth. It is defined as:

$$MSE = \frac{1}{N}\sum_{i=1}^{i=N}(y^{(i)} - \hat{y}^{(i)})^2 \tag{2.6}$$

where N represents the total number of training samples, i is the counter iterating over the training samples, $\hat{y}^{(i)}$ is the prediction for sample i, and $y^{(i)}$ is the ground truth for sample i. Many loss functions could be found in literature, however, MSE remains a favourable function for regression problems. The MSE can be reformulated in the form:

$$MSE(\theta) = Variance_\theta(\hat{\theta}) + Bias_\theta(\hat{\theta}, \theta)^2 \tag{2.7}$$

The $Variance$, besides its mathematical definition signifies how closely the estimate resembles the training data. A low $Variance$ suggests minuscule changes to the target function's estimate in accordance with the training data. On the other hand, a high $Variance$ suggests large changes to the target function's estimate. $Bias$, however, represents the trade-offs the estimator makes in order to simplify the learning procedure, usually by ignoring or mitigating features at the cost of generalization. An illustration of a high $Variance$ and high $Bias$ is shown in figure 2.5. Since the goal of a neural network is to minimize the loss, this would also mean a minimization of the $Variance$ and $Bias$, which is the outcome that is sought after.

For classification problems, it is common to use the Cross-Entropy (CE) or a similar loss function to compute the error between the prediction and the ground truth. The cross-entropy is defined as:

$$CE = -\frac{1}{N}\sum_{i=1}^{i=N}(y^{(i)} \cdot \log(\hat{y}^{(i)})) + ((1 - y^{(i)}) \cdot \log(1 - \hat{y}^{(i)}))) \tag{2.8}$$

The cross entropy measures the performance of a classification model assuming the prediction is a probability, limited to the range of 0 and 1. Assuming that both the prediction and the ground truth represent two separate probability distributions, our goal is to measure the entropy (as defined in information theory) [101, p.4] between them.

Activation functions

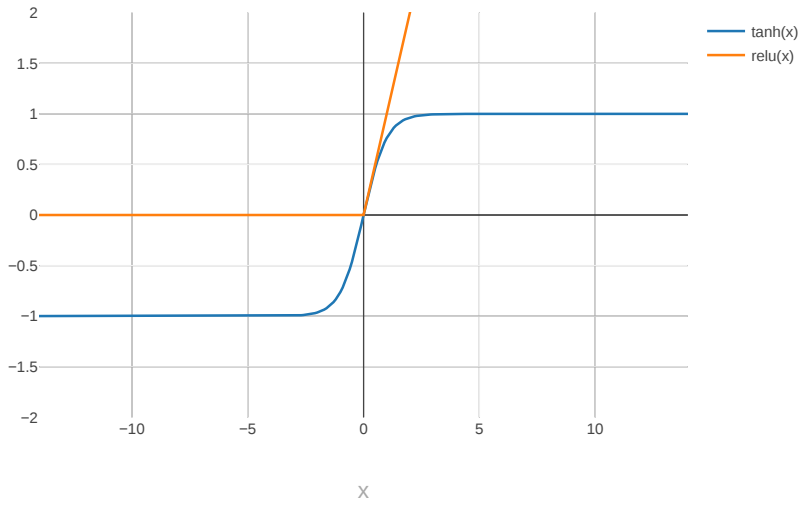


Figure 2.4: The ReLU and tanh activation functions

For the binary addition example described in section 2.1.1, we used the sigmoid as our activation function. Although it served the intended purpose of introducing non-linearity to the model, the sigmoid function suffers from major setbacks as the number of hidden layers increases. We observe that as the prediction tends towards higher or lower values, the gradient slope decreases exponentially, until it eventually settles (reaches a value of 0). At that point, the weights are not modified and no learning takes place. Another issue observed with the sigmoid is the strength of its gradients [58, s.4.4]. We can mitigate this issue by using a hyperbolic tangent function (tanh) instead. The **tanh** function is described as:

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.9}$$

As shown in figure 2.4, we notice that the range of the tanh function spans between -1 and 1, providing a stronger gradient which causes learning to become more efficient.

We notice that although tanh is preferred over sigmoid, it does not result in an output resembling a probability distribution i.e., the output is not limited between the range of 0 and 1. For classification problems with CE loss, tanh cannot be used as an activation function for the final layer. For binary classification, the sigmoid function generates values within the expected range. Extending the objective to multi-class classification, we replace the sigmoid with a **softmax** function. Softmax is defined as:

$$softmax(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \tag{2.10}$$

Where j symbolizes the index of the unit for which the probability is computed and K defines the total number of units in the softmax layer. Unlike the case with sigmoid where we assume binary classes, we calculate the probability of all possible classes independently and normalize each prediction with the total predictions across the entire range of classes.

Another essential activation function known as the **rectified linear unit** (ReLU) [73] is shown in figure 2.4. The ReLU mitigates all input values below 0 and simply return the input value when it exceeds 0. In recent years, ReLU became the preferred activation function when adding more layers to neural networks due to their low computational cost for one. We justify this preferential shift in section 2.1.6.

Backpropagation

In neural networks, the weights need to be adjusted to minimize the loss. In other words, the weights are set to a value which drives the function's output towards local minima. As noted in section 2.1, the activation function needs to be differentiable as a prerequisite. The loss function must be differentiable as well. The differentiability requirement is enforced for all neural networks using gradient-based optimization methods.

We update the weights using the **gradient descent** algorithm. Gradient descent is an iterative algorithm which progresses towards the direction of the steepest descent. Since it is an iterative algorithm, a step size should be defined for each iteration. The step size is known as the **learning rate**. Choosing a learning rate that is too high would result in an overshoot, causing the descent algorithm to miss a potentially good minimal point. On the contrary, setting the learning rate to a minuscule value would result in a prohibitively slow learning process.

The gradient is calculated upon the loss function with respect to the weights of the neural network. Suppose we design a neural network for performing linear regression. We would choose the MSE (described in section 2.1.2) as our loss function. For simplicity, we assume a linear activation function instead of a sigmoid, replacing $\hat{y}$ in the MSE loss function with $x \cdot w_x + w_b$. To compute the gradient of

MSE with respect to the weights:

$$w' = \frac{\partial E}{\partial w} = -\frac{2}{N}\sum_{i=1}^{i=N} x^{(i)} \cdot (y^{(i)} - (\hat{x^{(i)}} \cdot w + b)) \tag{2.11}$$

Where E is the MSE loss function, w represent the weights, b represents the bias, x represents the input, and i denotes the sample for all N samples. Similarly, we derive the MSE loss with respect to the bias:

$$b' = \frac{\partial E}{\partial b} = -\frac{2}{N}\sum_{i=1}^{i=N} (y^{(i)} - (\hat{x^{(i)}} \cdot w + b)) \tag{2.12}$$

Once we have acquired the derivative of the weights and the bias, we can update their values by:

$$w \mapsto w - (w' \cdot \eta) \tag{2.13}$$

$$b \mapsto b - (b' \cdot \eta) \tag{2.14}$$

where η denotes the learning rate. It becomes clear that such an approach is plausible assuming the network has a single layer. However, it is common to have multiple layers to estimate complex functions, requiring an expansion on the common gradient descent approach. To acquire the updated values for shallower weights, the chain rule is applied on the weights in reverse order, starting from the deepest layer (output layer) to the shallowest layer (first intermediate layer). We update derivatives for each layer separately and multiply the derived functions up until the layer for which we require the weights. Applying the chain rule in reverse order for updating the weights is known as **backpropagation**.

Training and initialization

Gradient descent is an algorithm used for minimizing the loss by driving the parameters towards optimal values. We demonstrated how gradient descent optimizes the parameters, however, we notice that the weights are only updated after all samples have been observed. Iterating through all the training samples is described as an **epoch** of time. In other words, an epoch is the total number of iterations required to traverse once through the entire training dataset.

Updating the weights after an entire epoch is known as **batch gradient descent**. The gradient descent is relatively smooth and should not fluctuate very often. However, such a method does not provide us with feedback on how well the network is performing until the loss for all samples has been computed. Another issue arises from the large memory consumption associated with such an approach. In theory, we can duplicate neural networks for the total number of samples in our dataset, perform the forward pass and compute the loss. Since the samples are assumed to be independent, no sequence should be obeyed, hence all computations can be done in parallel. This is considered time efficient but requires a large memory size. A group of samples processed in parallel are described as a **batch**.

Defining a batch size that is significantly smaller than the total number of training samples allows for a compromise between memory and time.

Updating the weights after a single batch is known as **mini-batch gradient descent**. Averaging more samples at once should reduce the noise and stabilize the gradient descent, however, when many samples are averaged at once (as the case with batch gradient descent), the gradient descent becomes very stable, risking a potentially suboptimal local minima [58, s.4.1]. On the other end of the spectrum, another method described as **stochastic gradient descent** (SGD) updates the weights after every sample. SGD is much faster than batch gradient descent depending on the size of the training dataset. Unless we are training on a large dataset (speed is an essential factor as well), SGD would be considered too noisy, since each sample influences the gradient's direction.

Mini-batch gradient descent offers greater flexibility to train the network based on our preferences and the resources available in comparison with batch and stochastic gradient descent. However, choosing a suitable batch size is rather challenging. It is important to shuffle the data in our training dataset to avoid any bias, especially when the dataset is ordered based on some criteria. This implies that unrelated samples are batched together, indicating that the features observed differ significantly. Frequently occurring features are therefore more likely to have a greater influence on the gradient, causing samples with less frequent features to have a smaller impact on the gradient. Increasing the number of samples in a single batch should mitigate this issue.

The learning rate has a significant impact on the convergence of the model. Introducing learning rate schedules [89] allows the learning rate to change during the learning phase. Although such dynamic approaches offer a remedy for falling into sub-optimal local minima, they are limited by the properties which have to be predefined, presenting yet another optimization challenge. As argued by Dauphin et al. [16], difficulty in convergence arrives mainly from **saddle points** instead of local minima. Saddle points are plateaus surrounded by a similar error in both directions, making it impossible for SGD to escape a sub-optimal point without modifications to the optimizer algorithm. Gradient based methods cannot distinguish between a local minimum and a saddle point. Ge et al. [28] introduced noise at each descent step, showing that gradient descent can escape saddle points in polynomial time.

The weights of the neural network need to have an initial value. The choice of the weights is critical and can have a major influence on the learning procedure. The bias weights are commonly set to 0 initially. The weights for hidden and input units, however, cannot be set to zero, since any projection on the weight vector would result in a 0 value and therefore the gradient would never change. Setting the weights to high values initially would slow down the learning process, increasing the learning time by folds. Another critical issue would arise from setting all the weights to an identical value initially. Considering that all the weights are the same, the gradient for those weights would be identical, making most units redundant. A simple alternative is to initialize the weights from a standard normal distribution.

The choice of weight initialization is heavily dependant on the activation func-

tion as well. For layers with a ReLU activation, He et al. [40] proposed a scaling factor $\frac{2}{\sqrt{fan_{in}}}$ multiplied with the randomly initialized values. fan_{in} refers to the number of units in the layer preceding the one for which the weights are initialized. The random variables could be sampled either from a normal or uniform distribution. A similar initialization factor $\frac{1}{\sqrt{fan_{in}}}$ was proposed by Glorot et al. [31] for initializing either sigmoid or tanh activated layer weights.

Optimizers

Ravines [99] present a non-trivial challenge for optimizing a neural network, especially when using SGD. Ravines are defined as areas where the search space curves are steeper in one direction than the other. This causes SGD to oscillate frequently between the two directions, slowing down the learning process. Momentum [85] introduces a damping factor which minimizes the oscillation. The damping factor relies on previous timesteps, introducing a fraction from past weights:

$$w \mapsto (\gamma \cdot w) - (w' \cdot \eta) \tag{2.15}$$

where γ represents the momentum term (fraction from the previous timestep). As the weight update gains greater momentum, it progresses faster towards the steepest slope. This approach does not integrate past knowledge into its progress, namely, it continues to accelerate towards steeper points even when a sudden change in the gradient occurs. **Nesterov accelerated gradient** [76] is a method which takes past steps into consideration. Nesterov subtracts the momentum from the gradient term of the previous timestep, allowing it to update the weights based on an approximation of the future timesteps instead of only considering the current timestep.

Both the momentum as well as Nesterov momentum can be used alongside the SGD optimizer. Other optimizers consider more elegant approaches than pure momentum. One such algorithm is known as the **Adaptive Moment Estimation** (Adam) [52] optimizer. Adam introduces an adaptive **decay** to the learning rate. Decay refers to a factor multiplied by the learning rate at each timestep. Adam computes the mean (the first moment) as well as the variance (the second moment) of previous predictions, multiplying them with decay factors β_1 and β_2 respectively. The two metrics are computed as such:

$$m \mapsto \frac{(\beta_1 \cdot m) + (1 - \beta_1) \cdot w'}{1 - \beta_1} \tag{2.16}$$

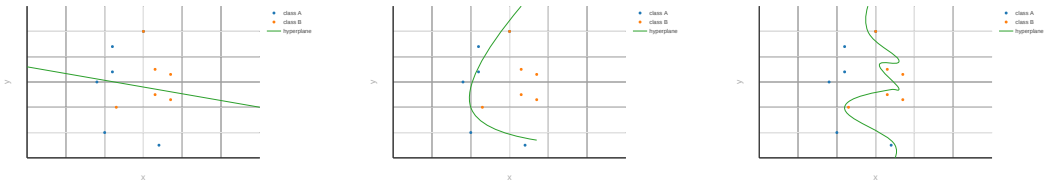
$$v \mapsto \frac{(\beta_2 \cdot v) + (1 - \beta_2) \cdot (w')^2}{1 - \beta_2} \tag{2.17}$$

where m represents the mean and v represents the variance. The constants β_1 and β_2 adjust the decay of the mean and the variance. Since the mean and variance are initially set to 0, they are biased towards zero. To counteract the bias, a divisor of $1 - \beta$ is introduced to each term. The weights are finally updated by:

$$w \mapsto w - \frac{\eta}{\sqrt{v} + \epsilon} + m \tag{2.18}$$

where ϵ is a smoothing term to avoid division-by-zero errors. Although Adam generally outperforms SGD, it fails to generalize as well when the number of training iterations increases [51]. The authors address the problem by proposing an algorithm which switches between Adam and SGD based on the training steps. Choosing the best optimizer is task dependent, and to our knowledge, it still appears to be an open research problem, since different optimizers have shown success in various applications.

2.1.3 Regularization



(a) high bias: underfitting (b) regularized: good fit (c) high variance: overfitting

Figure 2.5: Three function fitness modes

As the number of trainable parameters increases and the size of input data decreases, the network is more prone to **overfitting**. Overfitting refers to the model approximating a function that closely resembles the training data as shown in figure 2.5(c), causing a large variance in the decision boundaries. This may cause the data which was not observed during training to be misclassified.

Many techniques have been introduced for regularizing neural networks. One common method is known as **data augmentation**. Data augmentation refers to adding more samples to the training dataset (described in section 2.1.4) by introducing random noise to the data. Examples of noise applied on images, involve the introduction of random translations, rotations, scales, skewness etc. Such approaches create a larger training dataset and allow the model to overcome noise. Increasing the noise to an extreme results in what is known as underfitting as shown in figure 2.5(a), hence it is necessary to adjust the parameters of different noising techniques depending on the problem at hand.

Dropout [97] is another effective technique commonly used in neural architectures to prevent overfitting. Dropout randomly drops connections (sets their weights to 0) during each training iteration, based on a dropout probability defined beforehand. During inference, however, dropout no longer drops connections. As a result, inference cannot be performed without a simple modification. Since the model uses the full spectrum of connections during inference, the scales of the outputs are larger by a factor equivalent to the dropout probability in comparison with the training output. Hence, when applying dropout during training, the outputs of the inference model should be adjusted accordingly.

Dropout can operate well alongside other regularization techniques in the same network. Layer Normalization [4] is one such technique. Layer normalization is a method which normalizes the input features in a batch based on two learned parameters integrated as part of a hidden layer. The parameters are the mean and the variance, which iteratively scale the layer features within any given batch.

Other approaches for regularizing neural networks include external approaches, which do not modify the data, nor do they influence the neural model. A common technique known as early stopping interrupts the learning process based on the validation loss. When the training loss decreases and the validation loss begins to rise, this indicates that the model is on the verge of overfitting. Early stopping is parametrized by what is known as patience. If the trend of inverse proportionality between the two losses continues for a preset number of iteration (patience), the neural network is halted. This prevents networks from resuming learning as they overfit, and stop at a point where the network generalizes best.

2.1.4 Training, testing and validation

For training a neural network model on data, the examples are usually separated into three sets. The main data set which is usually the largest, known as the training dataset is used for the training the model. The parameters of the model are fitted against the training dataset; however, overfitting as described in section 2.1.3 becomes more likely to occur, if we were to measure the performance of the neural network based on the training examples. To remedy this issue, the overall dataset is split between training examples and validation examples. The validation examples are fewer than the training examples, and their loss is measured against the parameters which were trained with the training set. After training for a single epoch, the validation examples output is predicted. The validation dataset serves two purposes: (1) It can be used as a regularization enforcement technique, and (2) Provides an unbiased approach for evaluating the hyperparameters where the network properties could be modified as desired while ensuring the validation dataset remains the same throughout the evaluation. Another dataset which is usually split from the overall dataset is known as the test dataset. The test dataset is independent of the training and validation dataset but has a similar distribution to both. The test set is usually as large as the validation set and serves the purpose of ensuring the generalization of the model. After training and validating the hyperparameters, the model with the trained weights is used for inference on the test dataset. Based on the evaluation metric (section 2.1.5), we can decide if the model was able to generalize correctly according to how well it recognizes unseen examples from the test dataset.

There are several techniques for validating the performance of a network given preset hyperparameters [53]. Selecting the hyperparameters in the first place requires an informed decision. We can manually choose a set of values for each of these hyperparameters and alternate these values until all combinations of hyperparameters are covered. This approach is known as **grid search**. As the number of hyperparameters grows along with their potential values, the plausible combi-

nations increase exponentially making grid search prohibitive. We could instead randomly generate hyperparameter values, limiting them to a certain range. Random approaches have a clear advantage over grid search when the search space grows significantly; however, they do not incorporate prior knowledge in selecting possible values. **Bayesian optimization** [72, p.1-3] approaches for such as Parzen tree estimators [9] have been proposed for hyperparameter optimization. Such Probabilistic approaches build a probability model of the observations (previous experiments) and update the posterior function defining the hyperparameters based upon the prior function, with the objective of minimizing the loss of the network.

2.1.5 Evaluation metrics

To evaluate the performance of a neural network, we would require a metric against which we could compare the outcomes. The evaluation should be performed on the testing dataset. Measuring the same metrics on the training and validation datasets informs us about the generalization of a model. In other words, if the training metric value improves while the validation metric value degrades, this would indicate that the neural network is **overfitting**. For evaluating the performance on realistic examples, we would need to compare the results based on a metric which fits the task at hand. The simplest metric would be the accuracy of the prediction with respect to the ground truth. The accuracy represents how often the prediction and the ground truth match for all the testing examples. This approach, however, fails to represent the power of a test, where biases in the dataset could skew our perception of the outcome. To avoid this problem, we take the **precision** and **recall** into consideration. The precision measures the proportion of relevant examples given the prediction while the recall measures the ability to match relevant instances in the dataset. Precision is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{2.19}$$

where TP represents the number of **True positive** examples, meaning the examples that were predicted to belong to a class and the ground truth matched that prediction. FP represents the number of **False positive** examples, those which were predicted to belong to that class but the ground truth did not match the prediction. The recall is defined as:

$$Recall = \frac{TP}{TP + FN} \tag{2.20}$$

where FN represents the number of **False negative** examples, indicating the examples which were predicted to belong to another class, however, the prediction did not match the ground truth. The **F1 score** measures the harmonic mean of the precision and the recall and is defined as:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.21}$$

The F1 score offers a good compromise between the two measures, yet either of them might be of greater significance to the task at hand.

Visual object detection

Object detection in neural computer vision architectures addresses the classification of objects and localizing them in an image. The localization takes place by creating quadrilateral shapes enveloping the objects of interest. The quadrilateral shapes are represented as four-point coordinates and are called **bounding boxes**. Evaluating object detection requires a measure that encompasses the fitness of both the bounding box regression and object classification. One such metric known as the **mean Average Precission** (mAP) [23, p.313] addresses this problem specifically.

The mAP is the mean of average precisions at various recalls. The mAP employs the Jaccard index [37] which is also known as the **Intersection over Union** (IoU) to measure the overlap between bounding boxes. The IoU computes the area of intersecting bounding boxes over the total area covered by them. The mAP assigns the counts of the true positive, false positive, and false negative occurrences necessary for computing the precision and the recall. This assignment is based on the IoU between the prediction and the ground truth, considering that the IoU exceeds a predefined threshold. A precision versus recall construct for all the examples (objects detected) is created, and the precision values are adjusted to match the highest precision value to follow. The adjustment is performed by ordering the precisions according to their corresponding recalls. The mean of the adjusted precisions is computed over all examples resulting in the mAP. It is common to observe the losses of the classification (object class) and regression (bounding box location) outputs as an indicator of the model's improvement. However, the mAP remains to be a more robust measure which combines the evaluation of the two tasks at hand, while enhancing the interpretability of the model's performance.

Language translation

Measuring the validity of a translation is based upon the types of languages used and the objective of the evaluation. For models having cross-entropy as a loss, using that error to asses the translation would give a good indicator of how well performed. When translating natural languages to other natural languages, it is common to use metrics such as the BLEU score [79], since a correct translation could have different alternatives which are equally as correct. For translating between natural languages and non-natural languages, there are no standard metrics defined since the evaluation should be completely dependant on the task at hand. The cross-entropy, however, is more general by definition, since it quantifies the difference between the probability distributions of the prediction and the ground-truth, therefore, can be used for evaluating such translations.

2.1.6 Deep neural networks

Modern neural networks with several layers are described as **deep neural networks**. Although multi-layer perceptrons have existed since the year 1988 [91], common problems such as low computational resources, the curse of dimensionality [33, p.155], vanishing gradients [42] and exploding gradients [81], and the degradation problem [40] have hindered the development and research in MLP. With the resurrection of deep learning, the aforementioned issues were gradually mitigated.

With gradient-based learning approaches, such as backpropagation, the chaining of multiple numerical functions ought to diverge or converge at some point. Since the product of two or more small numbers (less than 1) would result in an even smaller number (less than at least one of the two values), we would expect the gradients to reach a value approaching 0. As we increase the number of layers and iterate through the various examples presented to the neural networks, we begin to observe the vanishing of those gradients. Careful initialization of the weights (refer to section 2.1.2) and correctly normalizing the data [98] as well as the weights and activation outputs of the network [4, 44] should reduce the prominence of the problem. Vanishing gradients are still significant, depending on the activation function. The sigmoid activation function is limited at the two extremes, making them more prone to gradients vanishing. On the other hand, a ReLU activation function saturates in a single direction [32], making it more robust to vanishing gradients. Using ReLU comes with its disadvantages such as the dying ReLU [102] problem and exploding gradients, yet they appear to be most suited for deep neural networks. To combat exploding gradients, we can also introduce a gradient clipping [33, p.409-411] threshold to the activation function.

The degradation problem refers to the neural network's inability to learn complex features as more layers are added. This observation was rather counter-intuitive since more layers implied an increase in the number of parameters, which indicates that the network has more degrees of freedom to fit varying non-linear functions. It was discovered that the addition of **residual connections** between the layers reduced the significance of the degradation problem. Residual connections are skip connections between two layers with a small number (relative to the depth of the network) of linear and non-linear layers in between. The output of the shallower layer is added to the deeper layer, making the learning objective easier for the network: learn the identity function, diverting the goal from learning $f(x) = x$ to $f(x) = 0$. This simple yet highly useful approach inspired the common practice of introducing residual connections in deep neural networks.

2.1.7 Convolutional neural networks

Computer vision neural networks require a design which differs from the conventional densely connected (fully-connected) architecture. A medium sized image, e.g., 640×480 pixels in width and height with 3 color channels, would result in about one million weight connects per single hidden unit. This requires immense

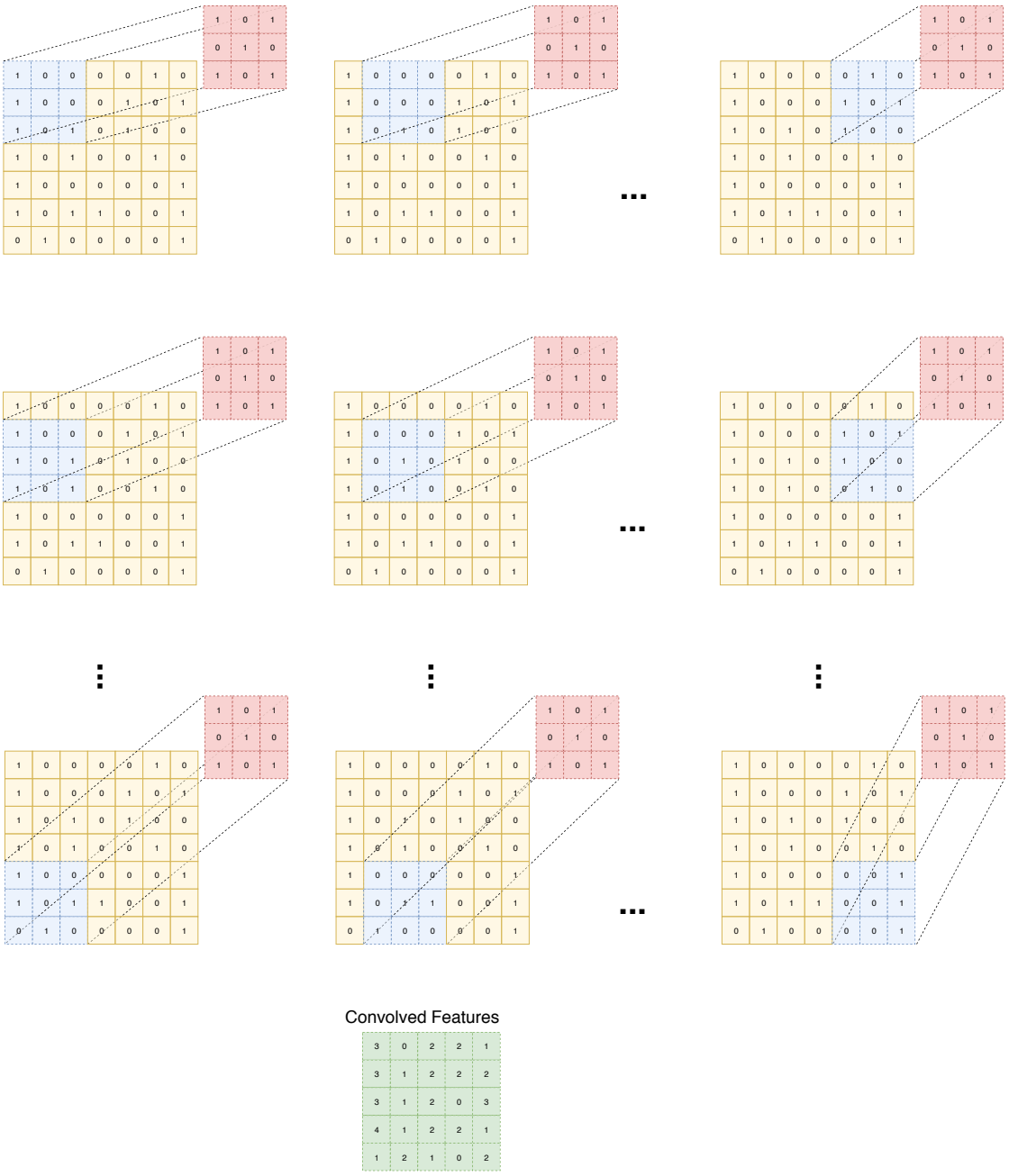


Figure 2.6: Convolution of a predefined kernel of size 3×3 with a 7×7 pixels monochrome image and a stride of 1×1 without padding. This results in an output of size 5×5

computational power and resources, with diminishing abilities to learn meaningful features in images. Due to the sparsity of significant features that would lead to correct classification or regression (e.g., Not all objects and features in an image need to be detected), dense feed-forward networks soon become infeasible. A common alternative is known as convolutional neural networks. These networks contain layers which do not traverse through all the units from a previous layer; instead, they have a predetermined size which governs their coverage.

Fully-connected deep neural networks are usually not well suited for images, due to their inability to take **translation invariance** [56] into account. Translation invariance refers to recognizing features within the image regardless of the location, e.g., an object of interest located in the corner of an image should be detected even when the training dataset only contained images with that same object located in the center of the image. Convolutional neural networks are known to be translation invariant mainly due to their incorporation of **pooling**. Pooling refers to the minimization of the features by disregarding or merging convolved features in a proximate locality (convolved features close to each other). Some common pooling methods include maximum pooling [111] where only the maximum of multiple convolved features is taken into account and average pooling where the average value of multiple convolved features is propagated to the following layer.

Convolutional layers are composed of multiple weighted layers called **kernels**, which are also known as filters. Kernels are limited in size, and depending on the dimension of the convolutional layer, we can specify the receptive field of those kernels overall dimensions. Multiple kernels could be introduced per layer creating several kernels. Each kernel is expected to learn a different set of features. Convolution as a mathematical operator expects a mechanism for sliding two signals against each other. We define a discrete step size which controls the shift in the sliding progression. In the context of convolutional neural networks, the step is known as a **stride**. As shown in figure 2.6, we can observe a simplified example of how a single kernel is convolved with an image. It is clear that the corners of the image are contained in fewer convolution operations. By creating a **padding** around the image, we can allow the convolution to occur for all units for an equal number of times. **Zero padding** is commonly used, whereby all units extending beyond the receptive field are set to 0.

To calculate the size of the convolved features output for a single kernel, we use the following formula:

$$d_{output} = \frac{d_{input} - d_{kernel} + 2 \cdot d_{padding}}{d_{stride}} + 1 \tag{2.22}$$

where d_{output} is the size of the output, d_{input} is the size of the input layer, d_{kernel} refers to the kernel size, $d_{padding}$ referes to the padding size, and d_{stride} refers to the number of strides in each dimension. Referring to the two-dimensional convolution example in figure 2.6, we calculate the height of the output by:

$$d_{output} = \frac{7 - 3 + 2 \cdot 0}{1} + 1 = 5 \tag{2.23}$$

resulting in 5 which matches the number of feature identifiers for the convolved features. The width of the output is identical to its height.

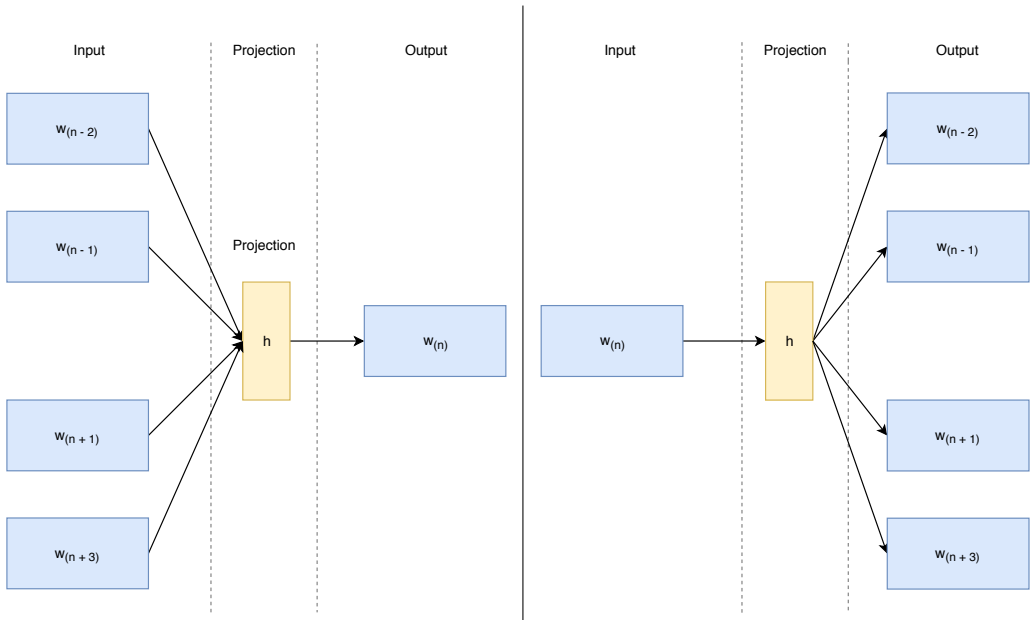
2.1.8 Attention

Until recently, recurrent and feed-forward variants of encoder-decoder neural networks [71, 103] relied on the entire spectrum of data within a training sample to learn approximating a function. The massive influx of information is rather distracting and could lead to sub-par ability to identify new information correctly. Instead, we tend to focus on certain aspects that appear to be critical or of greater importance in-order to optimally perform a task. An analogous mechanism resembling the human understanding of attention arises in the context of artificial neural networks.

Neural attention equips networks with the ability to attend to specific features, giving more weight to parts or units which contribute more to the achievement of the desired output. The simplest form in which attention could be achieved is the element-wise multiplication of a mask with the input vector. The mask is generally learned as part of the network parameters. Attention can be categorized into **hard attention** [3] and **soft attention** [5]. Hard attention constraints the values of the mask to a value of either 0 or 1. Soft attention, on the other hand, offers a more flexible limitation where the mask is constructed with values in the range of 0 and 1. Soft attention, therefore, assumes a mask which represents a probability distribution. A common approach for generating soft attention vectors is through applying a softmax to the attention mask.

Soft attention has gained more prominence in recent years due to the fact that their masks are differentiable, avoiding complex approaches such as variance reduction [71] as the case with hard attention. Luong et al. [65] propose an approach which combines global attention (soft attention) and local attention (hard attention) while maintaining the differentiability of the layers. Global attention is a term used to describe approaches which attend to the entire hidden layer. Local attention refers to attention constrained to a discrete position. By centering a subset of the global attention units around the local attention position, better performance was observed for a machine translation task [65]. Self-attention is another soft attention approach, shown to improve results for language modeling tasks [15] and visual attention tasks [106]. The idea of self-attention was developed for language modeling tasks, where different positions in a sequence are considered for computing a representation relating parts of the sequence to itself.

Different attention mechanisms were developed for visual tasks and language tasks including scaled dot-product attention [65], additive attention [5], and content-based attention [35] to name a few. Such approaches have given rise to the integration of attention with various neural architectures due to their continued success.



(a) Word2vec CBOW (b) Word2vec skip-gram

Figure 2.7: The two variants of the Word2vec model

2.1.9 Encoding

To present data samples to a neural network, they should be represented numerically. Every sample point should be indicated by a floating point or a real-valued number. For classification problems, the class of each example should be indicated by a unique scalar or vector which symbolizes a reference to the actual class.

Sparse encoding refers to a vector which is mostly represented by zeros, except for a few components. One of the simplest and most common sparse representations is known as **one-hot** encoding. A One-hot vector has a number of components equivalent to the total number of classes. All components of the vector are set to 0, except for the class represented by the example, which has a 1 valued component at the index indicated by the class. Suppose we have a dataset with only three classes: `red, blue, green`; The one-hot vector would have three components, where $\{1, 0, 0\}$ represents the `red` class, $\{0, 1, 0\}$ represents the `blue` class, and $\{0, 0, 1\}$ represents the `green` class.

A few categorical classes can be represented as one-hot encoded vectors; however, language models with vocabularies exceeding thousands of words would suffer greatly from such a representation. Consider as well that natural language related task rarely deal with single words or phrases, indicating that multiple words are represented to a neural network as thousands of units, causing the network to scale intractably. **Dense encoding** presents a more feasible alternative. Such encoding does not limit each vector component to a binary value of 0 or 1, nor does it restrict the vector to be sparse (a single active component). Assigning a unique vector for each word can be done by any random initialization technique. We specify the number of components in the vector and generate a unique vector for each word

type in the vocabulary.

Randomly sampling vectors to represent words is usually suitable for language-related tasks; however, the vectors do not represent any relation to the semantics of a given word. Mikolov et al. [66] introduced an approach for creating word vectors based on the likelihood of their co-occurrence in a given corpus. The approach was titled **Word2vec**. Words of similar meaning tend to frequently co-occur in similar contexts as per the distributional hypothesis [39]. The authors of Word2vec exploit the distributional hypothesis and the fact that neural language models outperform conventional their N-gram counterparts [95, 67].

Word2vec is constructed as a neural network with a single hidden layer. The hidden layer has a linear activation function (summation), with a softmax applied to the output layer. The authors introduce two variants of the Word2vec model as shown in figure 2.7. The **continuous bag of words** (CBOW) variant traverses through the entire corpus, considering the context window, which is described as N number of words preceding and following each word as input to the neural network, with the word at the current iteration as the ground truth class at the output. The **skip-gram** variant reverses the inputs and outputs, having the word at the current iteration as input and the preceding as well as the following words as output. The words are fed to the network as one-hot coded vectors, where the size of each vector is identical to the size of the vocabulary. All components are represented as 0 except for the indices where the preceding and following words occur in the vocabulary. The index positions in the vector representing the words in a given sample are replaced with 1.

The objective of the network is to maximize the probability of observing the output word or words conditioned on the context word or words. This can be achieved by measuring the cross-entropy (described in section 2.1.2) between the ground truth and the softmax output, applied as the loss function for the neural network. After training the network for several epochs, the weight matrix for the connections between the inputs and the hidden layer is stored in a dictionary. The indices represent the words in the vocabulary and the vectors for each index in the matrix is used to represent the word encodings. The vectors are referred to as **embeddings** which are later used for encoding words trained on different neural networks.

The Word2vec authors address various problems which could result from training the neural network on large corpora [68]. The authors introduce a technique for pairing consecutively co-occurring words to form phrases. They justify their approach by presenting examples where two or more words occurring consecutively could have a different meaning than each word observed independently. Infrequently occurring words below a certain threshold were eliminated from the training corpus as well. Greater improvement was observed by sub-sampling words, a technique by which most frequent words have a higher probability of being discarded from the training set. A technique described as **negative sampling** was also introduced as a modification to the training phase so as to reduce the training time significantly. During the backward pass, all weights of the network are usually updated. This proved slow and inefficient; hence the authors resorted to randomly

selecting a limited number of "negative" samples for which the optimization takes place. Negative samples refer to the words which did not contribute to the example of the current iteration.

Word2vec embeddings showed improvement over random encoding approaching [83]; however, Word2vec was recently disregarded for more complex latent approaches which considered the context of the word for representing its embedding [82, 17]. Although such approaches have shown significant improvement, Word2vec still provides a simple alternative, which requires minimal modifications to the targeted neural networks for integrating the words as learned embeddings.

2.1.10 RetinaNet: object detection and localization

RetinaNet [62] is a single stage (detector) deep neural network model for detecting and classifying objects in images. It has been shown to outperform other single stage [87, 64, 27] and multiple stage detectors [30, 88, 61]. The RetinaNet achieved an average precision of 37.8% on the challenging COCO [63] dataset.

The RetinaNet learns to predict bounds on objects. These bounds are in the form of quadrilateral shapes, which surround an object and are defined for each image from the training set. Learning different bounding boxes of all shapes and sizes would not be possible. Hence anchor boxes are employed instead. Anchor boxes are predefined shapes that cover a region of a given size. By limiting the number of possible outcomes, the learning becomes feasible.

The RetinaNet is composed of a deep residual network [41] called a ResNet, forming the backbone of the RetinaNet, which is connected at multiple stages to what the authors describe as Feature Pyramid Network (FPN) [103]. The FPN splits into two subnetworks, from which the detection output is extracted.

Feature pyramid network

A Region Proposal Network (RPN) [88] splits into two heads: specifically a regression head which predicts a shape for the anchor boxes in a given region, and the classification heads which predict the class for a given object. RPN form pyramids of anchor box shapes in the form of convolutional layers of different sizes on each region. These pyramids form a pyramid network. The RetinaNet adopts a similar approach to that employed by RPN with the exception that the outputs from a backbone network, such as the ResNet are discarded, and their hidden layers are combined.

The FPN uses the ResNet as its backbone model. In table 2.4, we summarize the architecture of three ResNet variants. Residual connections are found between all sublayers within each layer, e.g., The ResNet 50 has 3 residual connections for the conv2_x layer.

The Backbone architecture is segmented into five sections, each concerned with recognizing certain features in images. The first layers (shallow layers) in the backbone must represent low-level features such as edges and splines. Deeper into the layers, higher-level features are represented in the form of shapes and abstract

Table 2.4: The different ResNet variants constructed using convolutional layers, skip-connections and a final feed-forward layer

Layers	**ResNet 50**	**ResNet 101**	**ResNet 152**
conv1 112×112	$\begin{bmatrix} 7 \times 7, 64 \\ \text{stride } 2 \\ 7 \times 7, 64 \\ \text{pool}_{\text{max}}, \text{stride } 2 \end{bmatrix}$	$\begin{bmatrix} 7 \times 7, 64 \\ \text{stride } 2 \\ 7 \times 7, 64 \\ \text{pool}_{\text{max}}, \text{stride } 2 \end{bmatrix}$	$\begin{bmatrix} 7 \times 7, 64 \\ \text{stride } 2 \\ 7 \times 7, 64 \\ \text{pool}_{\text{max}}, \text{stride } 2 \end{bmatrix}$
conv2_x 56×56	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x 28×28	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x 14×14	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x 7×7	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
fc_final 1×1	$\begin{bmatrix} \text{pool}_{\text{avg}} \\ 1000 \\ \text{softmax} \end{bmatrix}$	$\begin{bmatrix} \text{pool}_{\text{avg}} \\ 1000 \\ \text{softmax} \end{bmatrix}$	$\begin{bmatrix} \text{pool}_{\text{avg}} \\ 1000 \\ \text{softmax} \end{bmatrix}$

visual constructs. Features represented in each layer are summed and combined hierarchically combined to form the FPN. The FPN is constructed with P_l layers, ranging from P_3 through P_7, where l indicates the pyramid level. The subscript l indicates the level and results in a resolution of 2^l less than the input image. An image with a resolution of 640×480 would therefore have a P_3 level with a resolution of 80×60.

Layers P_3 through P_5 are constructed using $conv3_x$ through $conv5_x$ from table 2.4 with top-down lateral connections between them. P_6 is a 3×3 convolutional layer with a stride of 2 connected to $conv5_x$. P_7 is computed by applying a ReLU activation, followed by a 3×3 convolutional layer with a stride of 2 connected to P_6.

The regression and classification heads extend from subnetworks connected to the FPN at multiple stages. The regression and classification subnetwork parameters are untied to one another but are very similar in shape. Each subnetwork is composed of four convolutional layers. The regression output regresses over the bounding box positions, whereas the classification head identifies the class of the object detected. The model could generate multiple bounding boxes, pointing to a single object. To avoid excessive bounding box generation, the overlapping boxes are suppressed using non-maximum suppression [24].

Focal loss

The RetinaNet introduces a novel loss known as the Focal Loss [62] which manipulates the cross-entropy loss in such a way that it emphasizes samples which were incorrectly classified. The cross-entropy loss is defined as:

$$CE(p_x) = -log(p_x) \tag{2.24}$$

here, p_x signifies the probability of the ground truth class. This definition does not address the imbalance between foreground-background classes, making classes which are difficult to classify even less likely to be predicted correctly. The authors introduced the Focal loss defined as:

$$FL(p_x) = -\alpha(1 - p_x)^{\gamma} log(p_x) \tag{2.25}$$

where γ is a tunable focusing parameter loss and $\alpha \in [0, 1]$ is a weighting factor, resulting in a balanced variant of the Focal loss.

2.1.11 Transformer: language modelling and machine translation

The Transformer [103] is a neural translation model which estimates the distribution of a target sequence given a source sequence. The Transformer focuses on the replacement of recurrent neural networks with an encoder-decoder architecture. Recurrent neural networks have a path complexity of t, where t signifies the number of timesteps. This hinders the parallelization of the computation due to the

sequential nature of such models. The core functionality of the Transformer relies on the idea of attention described in section 2.1.8, whereby the network learns to 'attend' to certain parts of the input, maintaining focus on regions of interest. The Transformer achieved BLEU [79] scores of 0.264 and 0.284 on the WMT NewsTest English-to-German translation dataset for the years 2013 [36] and 2014 [75] development sets respectively.

A machine translation model predicts the next words in a sequence conditioned on the probability of the previous words from the target language sequence as well as the source language sequence. The probability indicates the likelihood of predicting a target string given a source string.

$$P(t|s) = \prod_{i=0}^{C} P(t_i|t_{<i}, s) \tag{2.26}$$

where t is the target and s is the source. Index i indicates the token's position in the target sequence. C is the total number of tokens in a target sequence.

Multi-head attention

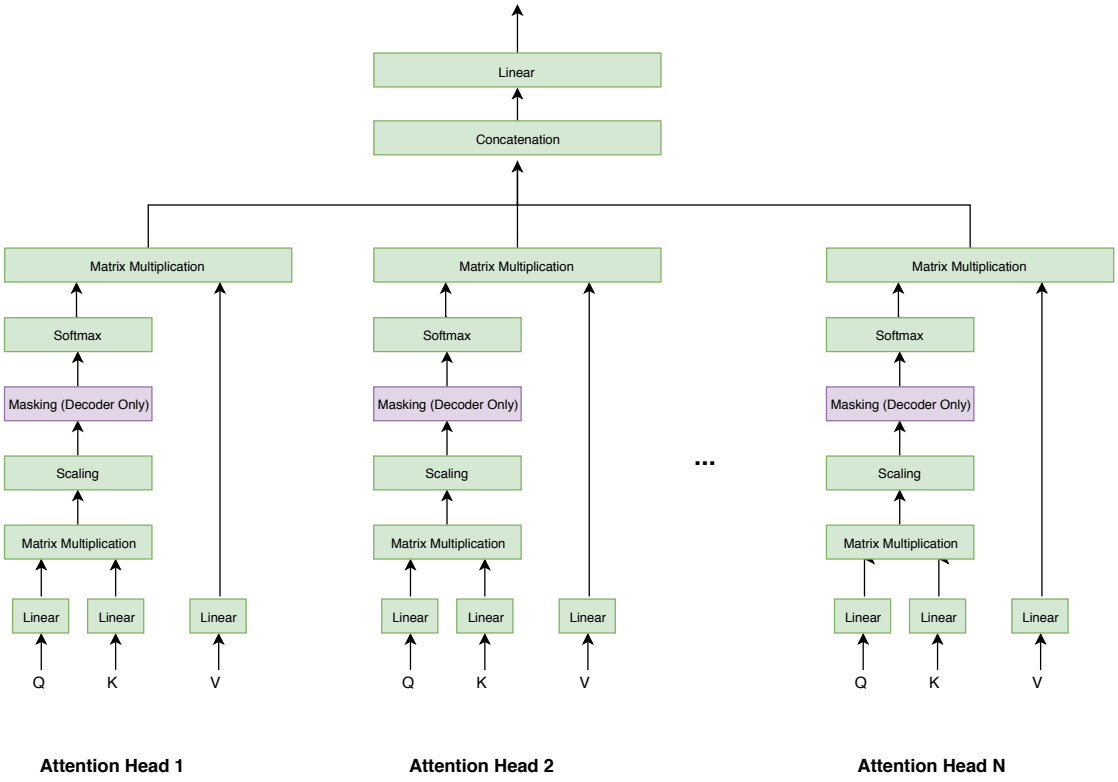


Figure 2.8: Multi-head attention of the Transformer network

The Transformer is constructed using a combination of multi-head self-attention and position-wise feed-forward networks [103], each followed by a layer normalization [4]. The Transformer applies what is known as self-attention [5] in the form

of dot product attention [65] instead of additive attention [5]. The authors modify the dot product attention by scaling it given a scaling factor of $\frac{1}{\sqrt{d_k}}$. d_k represents the dimension of the layer input. The resulting attention is formulated as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.27}$$

where Q is the query projection. K and V are the keys and value projections respectively. Within the multi-head attention block shown in figure 2.8, a linear transformation is applied to Q,K, and V. Scaled dot product attention is applied to the resulting matrices. Finally, the outputs are concatenated, and another linear transformation is applied to the concatenated output. A residual [41] connection extends from the layers before the multi-head attention block and its output. Layer normalization is then applied to the added residue.

Positional encoding

The normalized residual layer is connected to a feed-forward layer block which is composed of two ReLu activated layers. With the lack of recurrence, the model is oblivious to positional information. This lack is compensated by the introduction of positional encoding [103], which translates word positions to *sine* and *cosine* functions with different frequencies:

$$\begin{aligned} PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{model}}) \end{aligned} \tag{2.28}$$

where *pos* is the position of the word or character within the sequence, i is the dimension of the input, and d_{model} is the embedding size. The wavelength progresses linearly from 2π to 20000π allowing the model to learn a relationship between the word positions regardless of the sequence's length. The encoder and decoder are duplicated N times, and the decoder output is passed through a linearly activated layer, connected to a softmax layer, which computes the output probabilities.

2.1.12 Multi-task learning

In deep learning and machine learning in general, we aim to learn a specific task. We optimize our models to understand patterns and achieve an objective, minimizing the loss and improving the model's overall performance. By focusing on a specific task, the model might lack the ability to extract features which could otherwise be beneficial. Expanding the range of tasks for a model has shown to improve the model's ability to generalize more accurately upon the essential task at hand as well as the **auxiliary tasks** introduced [13]. Auxiliary tasks could be crucial to the final objective [88, 103] or integrated as a form of regularization [94, 1].

From a biologically motivated perspective, we learn to perform tasks by integrating prior knowledge acquired through performing other tasks. Not only do we achieve one objective, but we also tend to understand and analyze our sensory

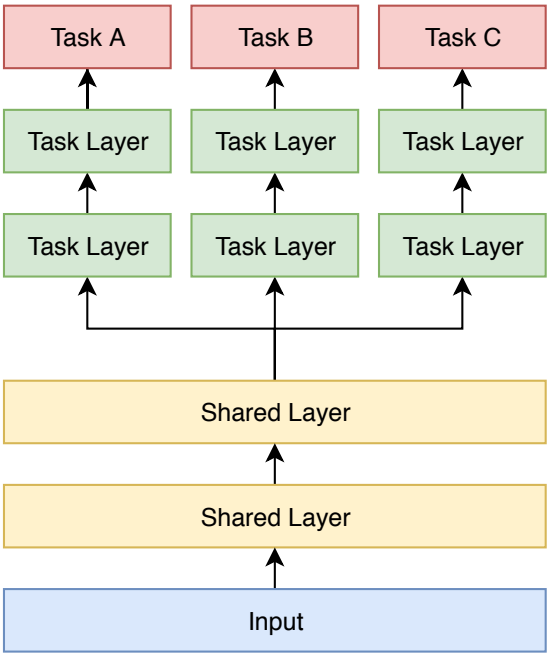


Figure 2.9: Multi-task learning architecture

input, as the case with vision: we observe our surrounding and accumulate knowledge pertaining to different domains like recognizing objects, their colors, their size and proximity to us. From a technical point-of-view, multi-task learning can be justified as a method which introduces biases to the model. Although bias is generally an undesired outcome of learning, in concert, these biases might as well assist our models in generalizing, e.g., an orange fruit is likely to also be orange in color

Regularization techniques introduce intrinsic and extrinsic knowledge to our learning models [104] and could be categorized as multi-task approaches. Multi-task learning approaches encompass multi-output neural architectures as well. A single task could potentially overfit to specific patterns; however, introducing more tasks could lead to an averaging of the noise across different tasks making the model more robust to irregularities and noise. In figure 2.9, we show the typical architecture of a multi-task neural network with hard parameter sharing. This indicates that in the lower layers, the parameters are shared across the task. Each task has a number of separate layers with dedicated parameters which are not shared across task. Eventually, the output of the neural network is represented by the tasks to be learned individually.

The tasks could have different objectives with different loss functions. The overall loss is a weighted summation of the losses across the tasks. The losses are weighted due to the different scales of each task output and the nature of their loss functions. Without weighing the contribution of each loss, a certain task might have an overwhelming effect on the overall loss rendering other auxiliary tasks redundant. Defining the loss weights could be done through manual tuning [54] or using more sophisticated approaches exploring homoscedastic un-

certainty [48] to adaptively adjust the weights according to the task-dependant uncertainty. Another non-trivial problem arises from the different losses introduced to a single neural network is known as **destructive interference** [110]. Destructive interference refers to tasks driving the gradients in opposing directions during backpropagation as the weights are adjusted for the layers shared across all the tasks. Zhao et al. [110] propose a modulation module which applies task-specific masks to layers within the network. The masks reduce the gradient angles across tasks internally since they are integrated as learnable parameters within the layer, mitigating destructive interference.

Best approaches in multi-task learning are still open to research. Although many approaches have been proposed to overcome common problems in multi-task learning, such approaches remain task-dependant. Eventually, tuning the tasks and integrating multiple models correctly should, in theory, improve the neural network's performance and enable learning features at a large scale [46].

2.1.13 Multimodal learning

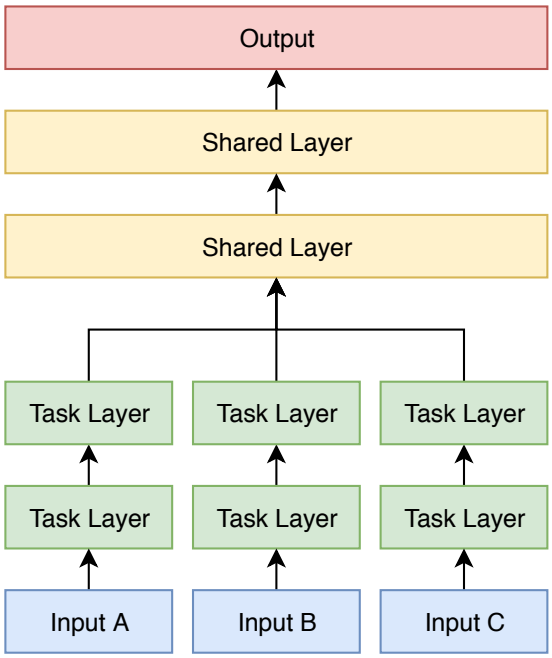


Figure 2.10: Intermediate fusion in multimodal learning architectures

Experiencing the physical world requires us to integrate a myriad of information, upon which we base our actions and decisions. Robots and computers have sensors which allow them to communicate with the outside world, enabling them to receive the sensory-input as digitized data, encoded to deliver meaningful information about the machine's surrounding. Images, audio, and text are examples of sensory-input data which can be fed into such machines. On receiving the input data, the system extracts features of interest and responds accordingly.

Responding to the data received from one input at a time might not be sufficient to perform a specific task. Combining sensory-input from multiple modalities is known as multimodal integration.

One of the key features for multimodal systems is their **complimentarity** [55]. Complementarity implies that each modality introduces information that is essential and cannot be deduced without the modality's integration within the multimodal system. In order to create a multimodal system, the data has to be combined. The process of combining the information from various modalities is described as multimodal fusion [55]. Fusion in deep neural architectures can be categorized into three broad groups: **Early fusion** [86] which is also described as data-level fusion refers to the preprocessing of signals or the manual adjustment of sensory data. For instance, either by decorrelating the multiple modalities, exploiting correlations in high-level representations [78], or representing the fused data in a common lower-dimension [105], the data could be combined and propagated to the neural network. **Late fusion** refers to voting [7] or ensemble [112] methods which weigh the contribution of each modality and decide accordingly. A majority of the neural approaches for multimodal integration implement fusion within the neural network, by combining the high-level representation of the data into a single shared layer [50]. Such methods are described as **intermediate fusion**.

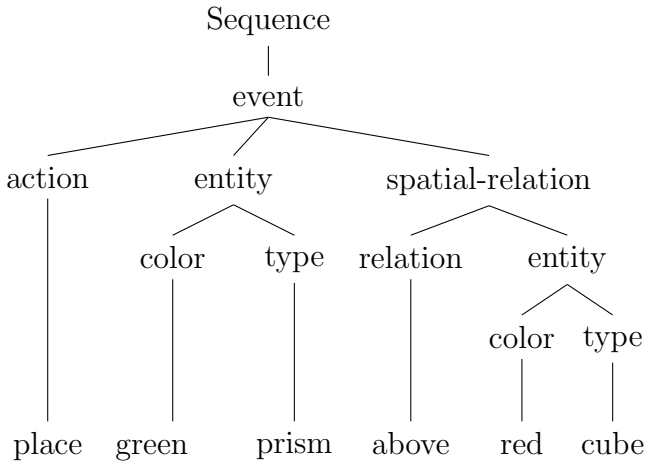
Figure 2.10 displays the structure of a multimodal network with intermediate fusion. Each sensory input is processed by a task-specific layer, extracting the lower level features associated with it. An intermediate layer merges (commonly through concatenation) data arriving from multiple modalities into a common representation, eventually learning a specific task which is influenced by the various sensory inputs.

2.2 Robot Command Language

The Robot Command Language (RCL) [18] is a tree-structured language used for instructing robots to perform tasks in the domain of object grasping. It is a simplied representation of English language commands, one which translates directly into spatial kinetic actions when processed by a robot with RCL support. The lexicon of the annotations slightly differs from that of the command sequences as shown in figure 2.11. RCL supports anaphora, cataphora, and indirect referencing.

The RCL structure was inspired by the semantic frame [25]. The leaf nodes of the RCL tree align to words in the natural language sequence as shown in figure 2.11. The alignment feature allows an association between the commands and the sentence, making it possible to relate the tokens in RCL to their corresponding natural language tokens.

RCL combines a tag set which is universal [19] i.e. can be used for robot commands in general, and domain-specific [19] i.e. designed specifically for the block grasping task. In figure 2.11, a preterminal node (e.g., action) along with its child (e.g., place) correspond to what Dukes [19] describes as a **feature-value pair**. The features shown are specific to the simulated robotic domain, e.g., the



RCL annotation (Target sequence):

```
(event:
 (action: move (token: 1))
 (entity:
  (color: green (token: 2)) (type: prism (token: 3)))
 (destination:
  (spatial-relation:
    (relation: above (token: 4 6))
    (entity:
     (color: red (token: 7)) (type: cube (token: 8))))))
```

English command (Source sequence):

```
place green pyramid on top of red brick
```

Figure 2.11: An RCL annotation and the equivalent English language command

action feature value represents moves for controlling the robotic arm, whereas the type and color feature values branching from the first entity describe the object to be grasped.

2.3 Augmented Reality

The process of superimposing three-dimensional computer-generated graphics on a real image is best described as **virtual augmentation**. In order to realistically place three-dimensional objects atop two-dimensional images, we need to know their orientation, position, and size. To transform between the two domains, one being the real view and the other being the virtual view, we need a reference that can be described in each domain. The reference is represented in what is known as the world frame, a common frame for positioning objects in both domains. To augment virtual objects on real images, we need to compute the intrinsic and extrinsic parameters of the camera which captures the real view. The intrinsic

parameters describe the optical center of the image and the focal length of the camera lens. The extrinsic parameters describe the position of the camera in the world frame. We can acquire the intrinsic parameters of a camera by either defining points on an image and their position in the real frame or calibrating it using a known visual pattern [109]. A simple pattern such as a checkerboard can be used to calibrate a camera. In order to successfully calibrate a camera, we would require at least three images of the same board from different angles and distances.We can assume that the camera is displaced in the real frame or that the checkerboard itself is allocated. Under both conditions, the computation of the parameters does not differ. The two assumptions which should be obeyed for this approach to work are that the points on the checkerboard must be on the same plane and the properties of the camera cannot be modified during the calibration. The camera captures multiple images of the checkerboard, after which, the edges of the board's pattern are identified [38].

The camera matrix is acquired as a prerequisite for performing what is known as **pose estimation**. Pose estimation deals with the projection of points from the real frame to the image frame, in what is known as the Perspective-n-Point (PnP) problem. Using methods such as random sample concensus [26], we can solve the PnP problem by projecting three-dimensional points on to the two-dimensional plane. By solving the PnP problem, we estimate the extrinsic parameters of the camera creating a virtual view resembling the real view. By knowing these parameters we can superimpose computer generated objects upon real images.

2.4 Simulation

Conducting experiments on a physical robot is a time-consuming procedure and rather inefficient for examining the effects of modifying the properties of a system. Simulating the physical properties of a system allows for immediate feedback, deterministic outcomes, safe experimentation, and lower financial overhead. The need for efficient and fast simulation gave rise to different approaches for simulating realistic environment through recursive algorithms [93]. Such algorithms aimed to achieve a smooth transition of multiple joints on a robotic system, yet they mostly ignored the need for simulating contact dynamics.

Erez et al. [22] argue that even though methods for contact modeling such as spring-damper methods [60] and impulse-based velocity-stepping methods [6, 70] are found in many modern simulators, such approaches seem more plausible for visualization and gaming tools rather than physically accurate simulation. A new generation of simulators focuses on the usage of efficient recursive algorithms for joint simulation and velocity-stepping methods for simulating contact dynamics.

One of the important features found in a majority of the robotic simulators, is the ability to translate positions in the form of Cartesian coordinates to actuator angles. Knowing an object's position, a robot must be able to reach that position if it's dynamic structure allows it. The positions of such objects are known relative to the robot's position, and the goal is to find a plausible joint configuration to reach

that point in space. This process is known as **inverse kinematics**. One method for performing inverse kinematics is known as **Newton's method**. Newton's method for inverse kinematics is composed of three main steps: (i) Finding the configuration of the joint angles, (ii) Calculating the changes in the joint rotations, and (iii) Computing the Jacobian matrix.

Initially, we need to transition from the robot's current configuration to the desired one. This can be represented as:

$$T = O + dO \tag{2.29}$$

where T is the pose vector representing the target configuration for all the joints. O represents the current joint configurations. Since the Newton method is a recursive process, we update the configuration until the change is minuscule. dO is the change in configuration and will be referred to as **delta**. To compute delta, we use the following equation:

$$V = J \times dO \tag{2.30}$$

here, V represents the difference in spatial location i.e. the difference between the target **end-effector** position and the current position, whereas J represents the Jacobian matrix. The end-effector describes the final moving point of a robotic arm, attached farthest from the base. To find dO, we invert the Jacobian:

$$dO = J^{-1} \times V \tag{2.31}$$

Since we are not guaranteed a solution after inverting the Jacobian matrix, we could instead acquire the transposed Jacobian matrix J^T. We can, therefore, guarantee that a solution will exist, even though it is a simplification of the inversion. This simplification may result in inaccurate transformations; however, it is significantly faster, and an outcome will be acquired.

The Jacobian matrix consists of the first-order partial derivatives of a function. For acquiring the partial derivatives of the axis positions of the end-effector with respect to the joints:

$$J = \begin{bmatrix} \frac{\partial p_x}{\partial \theta_1} & \frac{\partial p_x}{\partial \theta_2} & \frac{\partial p_x}{\partial \theta_N} \\ \frac{\partial p_y}{\partial \theta_1} & \frac{\partial p_y}{\partial \theta_2} & \frac{\partial p_y}{\partial \theta_N} \\ \frac{\partial p_z}{\partial \theta_1} & \frac{\partial p_z}{\partial \theta_2} & \frac{\partial p_z}{\partial \theta_N} \end{bmatrix} \tag{2.32}$$

We derive the positions $\partial p_x, \partial p_y, \partial p_z$ with respect to θ. θ describes the time steps for which the position will be computed, over the entire Cartesian plane. Having a large N number of steps implies greater granularity and more timesteps until the robot's end-effector reaches its final position.

Chapter 3

"Pick and Place" Dataset Generation

In this chapter, we describe the task addressed in the Thesis as well as the techniques used to generate the synthetic dataset adopted for learning the grasping task. To examine the influence of intermediate representations on the outcome of our network, we chose a three-dimensional block world scenario as the target task. In a three-dimensional block world, blocks with basic shapes should be placed in different locations based on natural language commands. We could request a block be moved from its initial position to a new location and a physical robot should be able to perform the action. Commencing the exploration of different datasets for achieving such a task, the Extended Train Robots [1] was the dataset of choice. The ETR dataset contains visual data; however, the images lack any kind of annotation. Our goal is to augment the dataset in a manner that allows for filling the missing data. To generate the missing information, we would, therefore, need to represent the data in different domains which will be described in the following sections. Figure 3.1 shows an example of an image provided by the ETR dataset, along with the images which we captured specifically for conducting our experiments, as well as the simulated environment which we designed to generate physical properties of a robot performing the pick and place task.

(a) ETR layout[1] (b) augmentation (c) simulation

Figure 3.1: The sixth layout in the ETR dataset shown in different domains

[1]The Extended Train Robots dataset: `http://archive.researchdata.leeds.ac.uk/37/`

(a) initial layout[1] (b) final layout[1]

Figure 3.2: The initial and final layouts for a command from the ETR dataset

3.1 The Extended Train Robots Dataset

The Extended Train Robots [1] is a synthetic dataset, explicitly designed for the task of grasping blocks using a robotic arm. The dataset was published by the University of Leeds and was collected through the Amazon Mechanical Turk [2] platform. Human annotators were shown two simulated images: the initial image displaying an $8 \times 8 \times 8$ grid with blocks laid on a surface at predefined positions on the grid, and a final image showing a configuration with a single block displaced. The participants had to annotate the scene with a natural language command that describes the difference between the two images in an imperative form, e.g., *place the purple pyramid on top of the blue and green stack of blocks*. The command describes the transition from the first image to the second image in figure 3.2. The dataset is segmented into five files, each containing information pertaining to a certain configuration. The files describe:

1. **Layouts**

   The layouts describe the visual grid world configuration, such as the block positions, their types and their colors as well as the robot arm's position along with its state. The positions of the blocks are represented as tuples with three elements, where the first element is an integer signifying the position of the block on the x-axis (0 implies the block is closest to the robot, whereas 7 implies it is farthest from the robot). The second element of the position tuple signifies the location of the block on the y-axis (0 implies that the block is position right-most of the grid with respect to the robot's view of the layout and 7 signifies the left-most location). The third element represents

[1]The Extended Train Robots dataset: `http://archive.researchdata.leeds.ac.uk/37/`
[2]Amazon Mechanical Turk: `https://www.mturk.com/`

the position of the block on the z-axis (0 implies the block is located on a flat surface and 7 implies the block is located at the top-most position on the grid). The types of blocks are described as *cubes* or *prisms*. The blocks could be any of the following colors: *yellow*, *white*, *gray*, *magenta*, *blue*, *cyan*, *red*, and *green*. The gripper position is represented as a tuple with three elements. The three elements of the robot arm's position tuple represent the location of the arm, following the same pattern as the block positions. The state of the robot's arm is also provided through a boolean flag called *open* (a value of $False$ implies the robot's fist is closed, whereas $True$ implies the opposite). Each layout has a unique identifier.

2. **Scenes**

   The scenes relate the initial layout and the final layout. The layouts are referenced through their identifiers. The scenes act as a lookup table for specifying the transition between the different layouts. Each scene has a unique identifier.

3. **Commands**

   The commands are the natural language sentences describing the transitions between layouts. The commands refer to the scenes and associate the English language sentences with their corresponding scenes. Each command has a unique identifier.

4. **Linguistically oriented semantic representation**

   The Linguistically Oriented Semantic Representation (LOSR) contains the tree-structured representation of the commands in RCL. Each LOSR annotation references the corresponding command through its identifier.

3.2 Visual Data: Augmented Reality

To generate the visual data used in this Thesis, we augment the three-dimensional blocks on a checkerboard pattern. We capture multiple images showing a checkerboard sheet lying on a table from different angles to facilitate the initial step of image calibration. After calibrating the images, three-dimensional computer-generated objects are superimposed on the images.

3.2.1 Environmental setup

An **environment** describes a collection of images which represent similar visual setups with slight differences. These images are captured from different angles and

distances. The individual images will be referred to as **environmental views**. Different variations are applied to the environment. The variations involve a change in:

1. **Lighting condition**

   The lighting condition refers to the illumination intensity in the room. We adjust the light intensity by switching on or off a portion of equally spaced lighting sources, each with an illumination intensity of around 450 lux. Three lighting modes are applied, with bright lighting having all six light sources switched on, dim lighting having three out of six lights sources switched on. A mixed lighting setup offers a random variation of lighting conditions per environmental setup.

2. **Distractor objects**

   Randomly placed objects on the table or within the camera's field of view act as visual distractors. They serve the purpose of providing negative examples for visual object detection and assist in regularizing the dataset. These objects include toys of different shapes and sizes, resembling fruits, vegetables, and miscellaneous objects. An environmental setup could either include or exclude visual distractors.

3. **Table surface**

   Two different tables are used for augmenting blocks on top of them. One is a rectangular black table, while the other is a smaller round white table. The table is changed in an attempt to minimize background bias.

3.2.2 Calibration

We apply six variations to the environmental setup. The variants are:

1. **Bright lighting** showing a **round white table** having **no distractor objects** in view
2. **Dim lighting** showing a **round white table** having **no distractor objects** in view
3. **Mixed lighting** showing a **round white table** having **distractor objects** in view
4. **Bright lighting** showing a **rectangular black table** having **no distractor objects** in view

5. **Dim lighting** showing a **rectangular black table** having **no distractor objects** in view

6. **Mixed lighting** showing a **rectangular black table** having **distractor objects** in view

Around 60 images were collected per environmental setup with slight modifications to the camera's focus across environmental setups. The focus is fixed for each environmental setup, else camera calibration would not be possible. We set the aspect ration to 16 : 9, resulting in images with a width of 1920 pixels and a height of 1080 pixels. We use a 10×7 checkboard pattern for calibrating all images. The calibration is performed in two stages:

1. We use a fisheye lens camera for capturing the images. In the first stage, we apply fisheye image calibration using the OpenCV [11] python wrapper. The built-in function `fisheye.calibrate` [1] is used to generate the translation and rotation vectors along with the camera matrix. This function assumes that the checkerboard pattern is visible in all images for which the calibration takes places. The calibration function requires knowledge about the two-dimensional points in the image (image points) and three-dimensional points in the world frame (object points), signifying the position of the checkerboard square corners in both frames. The corners are acquired using the OpenCV function: `findChessboardCorners` [1] which takes the number of checkerboard squares in each dimension as input along with the image as arguments. The checkerboard corner finder detects the corners by converting the image color to monochrome and dilates the image to identify the edges. The corners are refined using the `cornerSubPix` [1] function with a window size of $(11, 11)$, running for a maximum of 10^{-6} iterations with an early stopping criteria of $\epsilon = 30$. After successfully acquiring the camera matrix for each environmental setup, we undistort the corresponding images through the `fisheye.initUndistortRectifyMap` [1] function, which generates two maps. We call the `remap` [1] function to linearly interpolate the image points projected onto the two maps, resulting in an undistorted image.

2. Assuming the images were successfully undistorted in the previous step, we proceed to recalibrate all the new images together in order to acquire a camera matrix that would facilitate performing pose estimation on images combined. We follow the same steps taken for the fisheye calibration, using the OpenCV function `calibrateCamera` [1] instead of `fisheye.calibrate` [1]. In this step, we are concerned with acquiring a unified camera matrix for all the images, therefore image undistortion is not required.

3.2.3 Pose estimation

After completing the calibration, we estimate the pose of the checkerboards visible in selected images. The estimation is acquired by applying Random Sample

[1]For more details refer to the OpenCV documentation: `https://docs.opencv.org/2.4/`.

(a) fisheye image (b) edge detection (c) pose estimation

Figure 3.3: The calibration and pose estimation procedures demonstrated using the sixth layout in the ETR dataset

Consensus (RANSAC) [26] to solve the Perspective-n-Point (PnP) problem. We turn to OpenCV for solving the pose estimation problem, calling the function `solvePnPRansac` [1]. We feed the camera matrix acquired in the second calibration step to the PnP solver. The PnP solver function returns a new set of translation and rotation vectors with outliers excluded. These vectors are used for the object augmentation. We then perform the pose estimation on each environmental view independently. Figure 3.3 displays the raw image followed by the image acquired after the first calibration step. After performing the pose estimation, we visualize a three-dimensional mesh signifying the grasping region upon which the the blocks are to be overlaid (figure 3.3(c)).

3.2.4 Augmentation and domain randomization

We virtually augment two three-dimensional objects, namely pyramids and cubes, with eight different colors per object. These objects and their properties are acquired from the Layouts file described in section 3.1. The ETR dataset contains prisms instead of pyramids; however, for technical reasons, we replace all occurrences of prisms in the dataset with pyramids. This simple workaround solves the problem of ambiguous shape recognition from different views, where a prism could be easily misclassified as a cube instead.

First, we extract the positions and descriptions of the objects from the dataset. We then apply color textures on either of the two shapes and project those blocks onto the environmental view until all blocks within a layout are generated.

We define a scaling factor for the blocks to match the positions of the blocks in both the camera view and the three-dimensional view. The scaling factor approximates the actual size of the blocks, which is around 2.63 cm in all directions. We use the OpenGL [96] library for creating the three-dimensional block shaped projections, after which we superimpose the projections onto the real images. Using the translation and rotation vectors as well as the camera matrix which was acquired earlier, we map the camera view to the model view matrix in OpenGL. During the augmentation process, we acquire the bounding boxes surrounding the blocks by setting all pixels around the generated object to zero. All non-zero pixels are detected, and the edges of these pixels are stored, forming the bounding boxes.

[1] For more details refer to the OpenCV documentation: `https://docs.opencv.org/2.4/`.

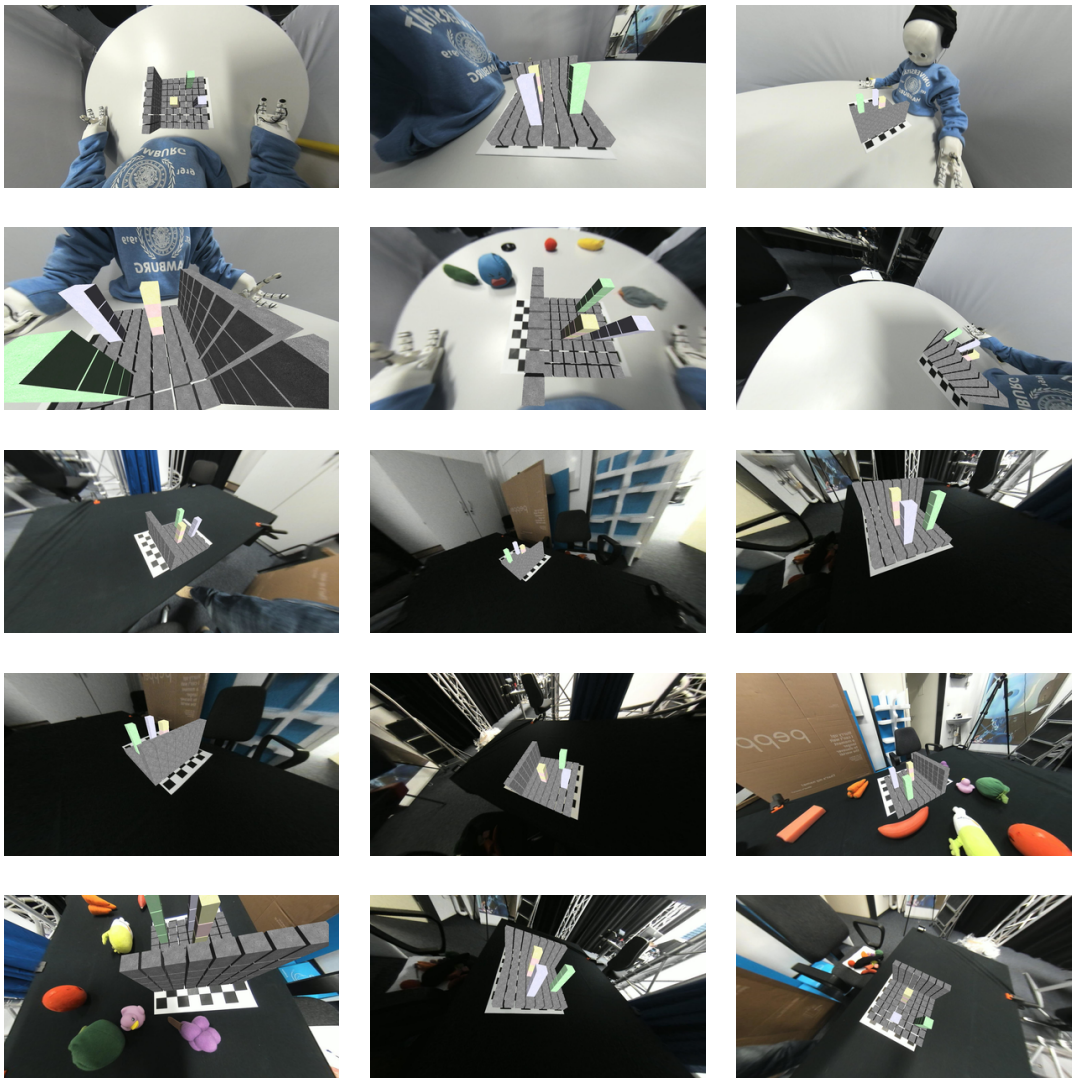
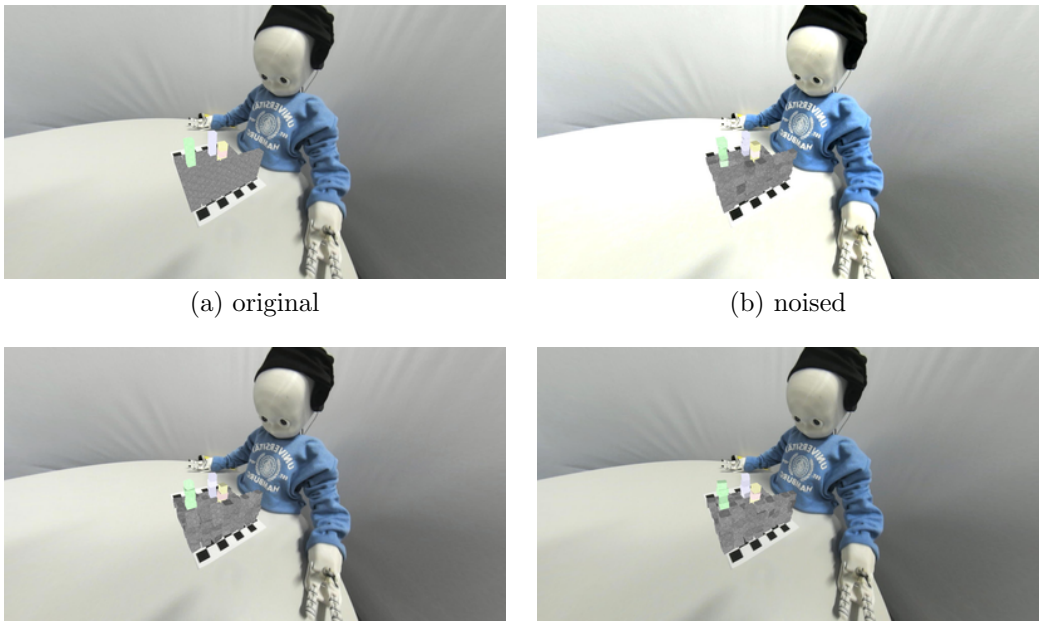


Figure 3.4: The sixth layout in the ETR dataset superimposed on 15 different environmental views

The 15 different environmental views chosen for augmenting objects are shown in figure 3.4. Note that the views contain the different variations including augmented objects, different poses of the view, and different tables with different lighting conditions.

3.2.5 Noise and Distractors

On choosing several environmental views, we also apply different noising schemes to regularize the data in preparation for training. We make sure that several environmental views contain distractor objects as well. The noising is done on two levels. The first noising level (domain randomization) involves the preprocessing of the dataset so as to create duplicates of the original environmental view for a given



(a) original (b) noised

(c) noised (d) noised

Figure 3.5: The sixth layout in the ETR dataset superimposed on one environmental view with random noise applied to the background, lighting and blocks (pose, size and color)

layout with the blocks slightly relocated, resized, and rotated. The light source is also modified, changing the contrast of the background image as well as the blocks. An example of this noising scheme is shown in figure 3.5. In the second level of augmentation, we randomly apply visual transformations to the rendered image after the blocks were augmented atop the background image. The second level of augmentation is applied during training and is essential for regularizing the data.

1. **Level 1: preprocessing noise (domain randomization)**

   Several transformations are applied to the blocks. Their scales are randomly varied in all directions within 10% of their original size. The blocks are randomly rotated within 6° on the x and y axes, and 10° on the z axis, relative to the block's centroid. Finally, the blocks are displaced by 1 cm in all directions. The contrast and lightness of the background, as well as the OpenGL virtual light source, are varied within a range of 0.8 and 1.2. On this scale, a value of 1.0 represents the original image, and a value of 0 represents a grey image. Similarly, the color intensity of each block is varied in the range of 0.8 and 1.2 following the same scale.

2. **Level 2: image transformation (data augmentation)**

   During the process of sampling images for training the neural network, we apply transformations to the image. The image is randomly rotated within 10° in all directions relative to the centroid of the image, randomly translated by 10% of the original image size in all directions relative to the centroid of the image. The image is also randomly sheared within 10° in all directions relative to the centroid of the image, and finally, the image is randomly scaled within the range of 0.8 and 1.2 times the scale or the original image.

3.3 Joint Coordinate Data: Simulation

The ETR dataset provides the **Cartesian coordinates** for the robot arm's end-effector in the three-dimensional block world space. For the purpose of this Thesis, we are interested in acquiring the coordinates of the robot's arm for reaching such a position. The process of acquiring the arm positions in the real world would be time-consuming. As an alternative, we resort to modeling the robot of interest in a simulated environment. The robot used is known as the NeuroInspired COmpanion (NICO) [49], a humanoid child-like robot developed in the Knowledge Technology group at the University of Hamburg. NICO was chosen for its child-sized figure and its dexterity which resembles that of a human. The robot's arm is composed of four Dynamixel [1] AX-12A speed controlled servo motors. At the end-effector, a Seed Robotics [2] SR-DH4D tendon-operated fist is attached. The gripper is composed of three fingers, suitable for grasping small objects. The robot, therefore, enjoys six degrees of freedom, making it suitable for the task at hand. The NICO robot is virtualized and described using the Unified Robot Description Format (URDF) [3]. The URDF contains the kinematic and visual specifications of the robot.

The Multi-Joint dynamics with Contact (MuJoCo) [100] simulator is used for acquiring the coordinates of the arm. MuJoCo is a model-based physics engine, which closely simulates real robots, designed for industrial and research purposes. MuJoCo has an internal converter capable of interpreting URDF files and representing the descriptions in a compatible format known as MJCF.

3.3.1 Environmental setup

In the context of simulation, we refer to the virtual setup of the robot's surrounding as an **environment**. We construct a single environment, constituting of the virtualized NICO robot, along with a **rectangular white table**, and **no distractor objects** in the robot's view. The visual properties are not of great importance for the purpose of acquiring the joint coordinates; however, we emphasize their precision for the purpose of testing the performance of the neural network based

[1] Dynamixel actuators by Robotis: `http://en.robotis.com/`
[2] Seed Robotics: `http://www.seedrobotics.com/`
[3] URDF: `http://www.ros.org/wiki/urdf/`

on images acquired from the simulator. The collision of boundaries match the visual structures, bringing the simulation closer to the physical environment's constraints. We set up the environment to have the default ambient light source, so as to maintain an illuminated view of the surrounding. Three cameras are placed in the simulated environment: two cameras are attached to the robot's eyes to get a view of the grasping region, similar to that of the physical robot. A single camera overlooks NICO and the table from a perspective view. A checkerboard patterned sheet is placed on top of the table, to resemble the physical view of the grasping region as closely as possible.

3.3.2 Simulation

After acquiring the layouts from the ETR dataset, we place three-dimensional blocks resembling the description's color and shape. As justified in section 3.2.4, we assume prisms to be pyramidal in shape. The blocks are placed approximately above the square spaces of the checkerboard pattern, where each square has a height and width equivalent to the blocks' widths and heights. The block dimensions are set to be 2.63 cm in all directions.

In order to reach the blocks using the robot's arm, we need to acquire joint angles that facilitate the movement of the gripper. The joints of interest are those which describe the robot arm's pose. We use NICO's left arm for grasping objects. We are interested in acquiring the arm's five joint angles as well as the gripper state (opened or closed). The $shoulder_{\psi}$ represents the yaw of the entire arm, whereas $shoulder_{\theta}$ represents the arm's pitch. Extending from the shoulder is an elbow which controls the pitch of the forearm and is denoted by $elbow_{\theta}$. The gripping fist connects to the forearm and is capable of rolling and changing its pitch, denoted by $fist_{\phi}$ and $fist_{\theta}$. Finally, we must also acquire the gripper state, where *grip* is a boolean, when set to $True$ signifies a closing of the gripper and $False$ signifies the release of the object. We set the simulator to operate in two modes. One mode accepts **Cartesian coordinates** in the form of x, y, and z discrete positions on the three-dimensional grid, whereas the other accepts **actuator angles** for each arm joint. The two modes of operation differ by the input to the simulator:

1. **Cartesian coordinate input**

   The Cartesian input variant is used to generate actuator angles given the location of the block. We create an $8 \times 8 \times 8$ grid for positioning the blocks in predetermined locations as shown in figure 3.6. We define these locations as **sites**. In the MuJoCo simulator, sites are geometric objects which are not influenced by the physical properties of the environment and are merely used for representing locations of interest relative to the **body** frames. The bodies represent elements encompassing geometric objects, sites, joints (moveable links connecting bodies), physical constraints and other bodies. The kinematic tree is constructed by nesting bodies in parent bodies. We aim for

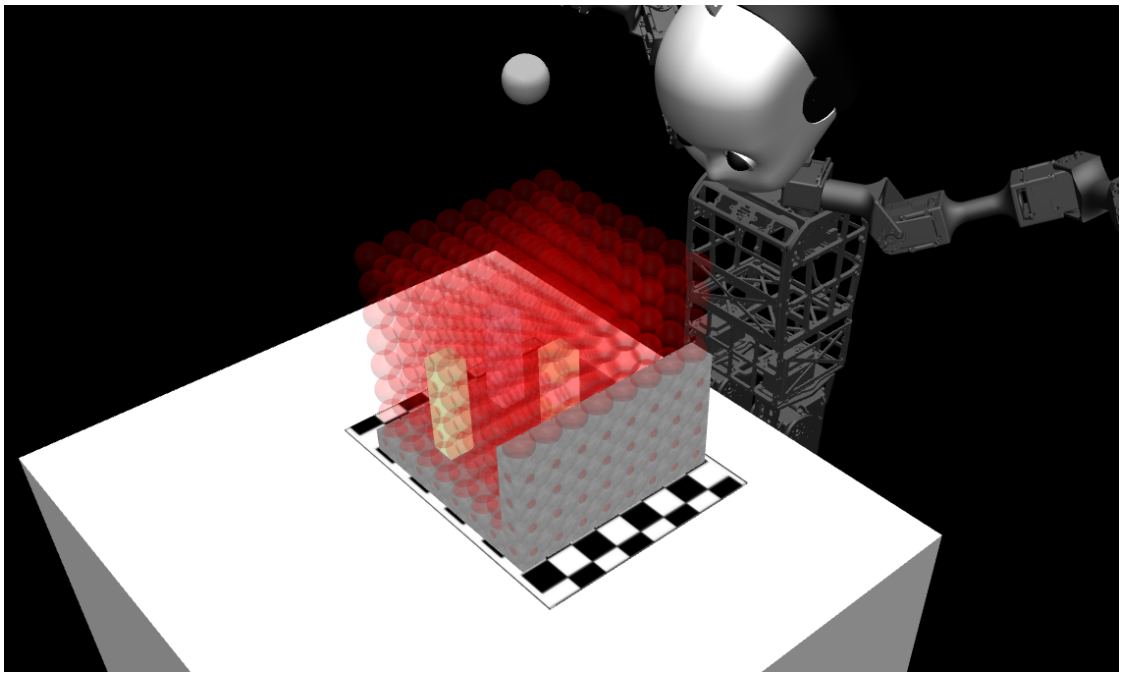
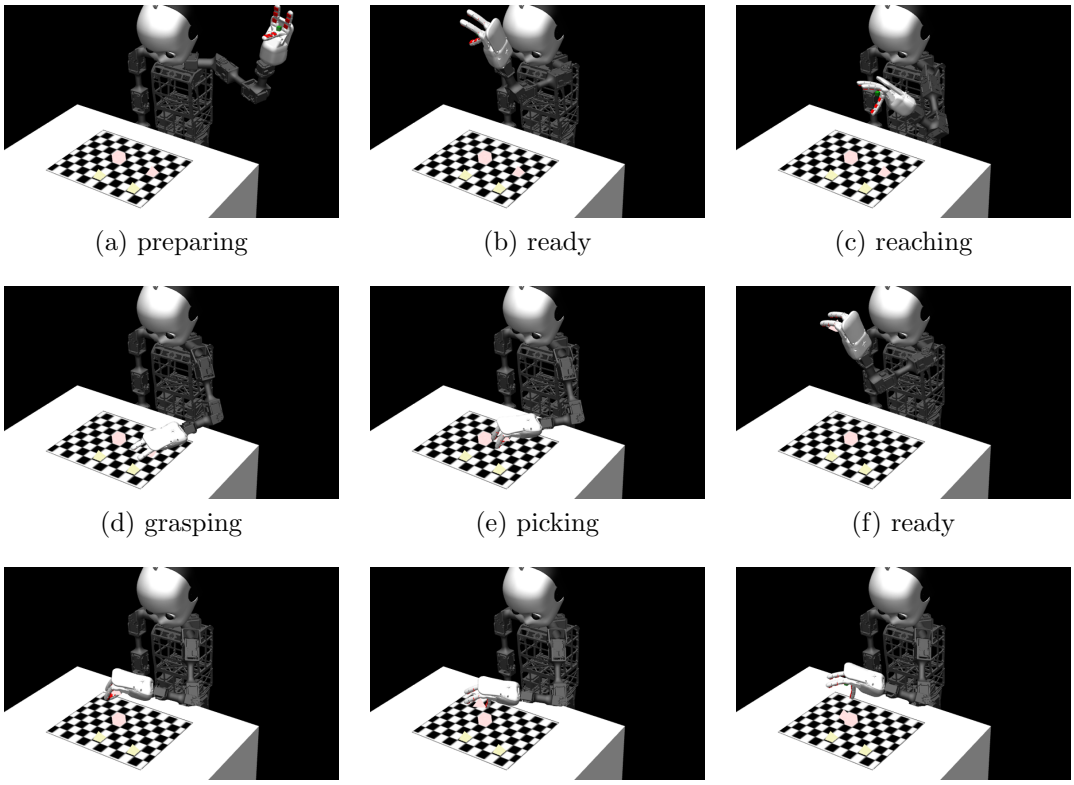


Figure 3.6: Grasping region highlighted in red showing the discrete positions for block placement in simulation

the bodies to reach the desired site by placing a motion capturing sensor (**mocap**) at the end-effector and moving towards the site. This allows the kinematic solver to perform inverse kinematics and produce the actuator angles for reaching the location of interest.

We use the DeepMind control library (dm_control) [1] for python, which interfaces directly with MuJoCo. To move the mocap, we set the `mocap_pos` to the `site_xpos`, and repeat the action until the end-effector reaches the block. Since the blocks can be reached through multiple poses, this would mean that the robot's arm might reach the desired location while obeying all the imposed soft constraints, yet the pose might appear irregular and physically implausible. To limit such consequences, we enforce a sequence of actions as shown in figure 3.7. The robot initially prepares for grasping to achieve the pose shown in figure 3.7(a). The robot is then ready to grasp the block. At this stage, the site position corresponding to the initial block location is extracted based on the Cartesian grid position. The arm approaches the block and closes its gripper. Once the robot reaches the block, the actuator angles are recorded, and the robot proceeds to place the block in the final location. On completion, the actuator angles are again recorded. Finally, we store the newly acquired actuator angles in the dataset with their corresponding scene identifiers.

[1] The DeepMind Control Suite and Package: `https://github.com/deepmind/dm_control/`



(a) preparing (b) ready (c) reaching

(d) grasping (e) picking (f) ready

(g) reaching (h) placing (i) complete

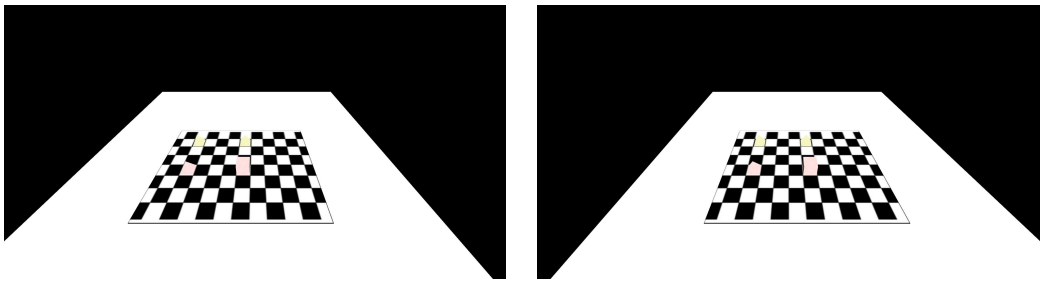
Figure 3.7: The grasping sequence performed by the robot in simulation when requested to place the red pyramid on top of the red cube

2. **Actuator angle input**

   It is necessary to map the actuator angles to actions for testing the outcome of our neural network. To perform the actions, we no longer use the motion capture sensor for moving the arm but instead feed the angles directly as input to the actuators. Since we do not have the complete sequence of actions stored, we have to define the interpolated movement steps during actuation as well. We follow a similar approach by moving the arm to the predefined poses, namely: preparing, ready and reaching as shown in figure 3.7. Instead of moving the mocap to pick and place the object, we feed the actuators with the angular input.

3.3.3 Visual Data

We capture images during simulation from the robot's view for testing the performance of the neural network on unseen data. The cameras are placed in positions approximating NICO's cameras. The buffered images are extracted before performing the simulation, and bounding boxes are generated following the method mentioned in section 3.2.4. Examples of the images are shown in figure 3.8.



(a) left eye (b) right eye

Figure 3.8: The views from the robot's camera showing the table and the blocks in the simulated environment

3.4 Linguistic Data

The commands and LOSR are acquired directly from the ETR dataset. By matching the commands with their corresponding RCL structured LOSR, we can understand the requested action in both the human-readable format and the robot oriented language. The single space delimiters separating the words are suitable for splitting the words into tokens; however, the LOSR commands require separators encompassing special characters (e.g., brackets and colons) to avoid chunking entire commands as single words. For consistency, we split both the commands and the LOSR by the spaces between the words and separate any special characters (e.g., punctuations, brackets, and symbols).

3.4.1 Noise

Augmenting natural language sources is an open research topic, and remains a challenge for researchers in the field of natural language processing. Adding noise to regularize the data is done with great caution as well, since shuffling words or rephrasing sentences might change their meaning. For our purpose, we are not interested in modeling language as accurately as possible, rather reducing overfitting without severely compromising the quality of the translation. The noise was applied exclusively to the natural language commands. The following noising schemes were conducted:

1. **Remove letter** erases a single character from a given word
2. **Add letter** inserts a single character to the word
3. **Repeat letter** inserts a single character to the word, following an existing character
4. **Flip letter** exchanges two characters in a word
5. **Remove punctuation** erases punctuations from the word

6. **Flip case** changes from lowercase characters to uppercase characters in a single word and vice-versa
7. **Relocate word** changes a word's position in a sentence
8. **Remove space** combines two words into a single word

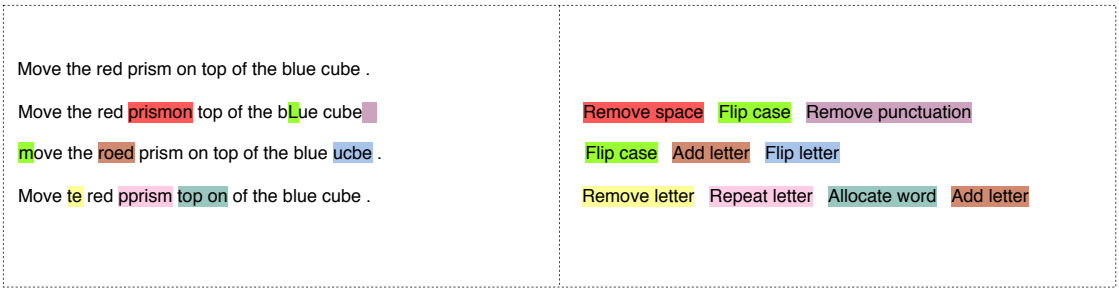


Figure 3.9: The first sentence represents the clean sequence. Perturbations are applied to the clean sequence as shown in the lines to follow. To the right, the types of perturbations applied to the sequence are shown

The noising schemes were intentionally kept simple and were chosen randomly for each command. Between one and four perturbation could be applied to a single sentence. Note that the noise is only applied on the original sequence and not transitively: we do not perturb already noisy commands. Applying the noise on a sentence results in perturbations as shown in figure 3.9.

Chapter 4

Sensorimotor Intermediate Fusion

In this chapter, we propose a neural network architecture capable of learning motor coordinates given visual and textual data as input. We present the training procedure as well as the modifications applied to established neural networks which allow for a change of the task they were designed to address. We also describe the input variants and the preprocessing steps taken to construct the data variants as well as the purpose of each modification.

The sensorimotor neural network is designed to receive sensory input and generate robotic arm joint angles for performing an action. The joint coordinates are fed directly into the motors and drive the robot to move to a certain pose until it reaches that desired pose. We combine the outputs of the RetinaNet [62] object detection neural network described in section 2.1.10 with the outputs resulting from the Transformer [103] language translation network described in section 2.1.11. The combination is performed through a concatenation of the units branching from either network, after which the dimensions are reduced, and the final output of the network produces the targeted coordinates.

It is common to combine tasks in neural networks branching from a single input so as to improve the performance of a neural network. This is commonly known as multi-task learning and has proven useful for improving the performance of all the tasks at hand. Another approach where a network receives multiple inputs is described as multimodal learning. A combination of multimodal and multi-task learning is described as a multi-input and multi-output (MIMO) model. A MIMO architecture accurately describes our approach, where a number of auxiliary tasks are introduced to increase the neural network's performance. We refer to the auxiliary tasks as **intermediate representations**.

Both of the employed modules learn specific tasks independently. The RetinaNet learns object localization and classification in the form of bounding boxes surrounding those objects of interest. The Transformer transduces one language to another, encoding the source language to produce the targeted language as an output. The primary objective of the network is to successfully locate objects of interest based on a textually represented command, which also describes the target of displacement for that object. Not only would the proposed intermediate representations hypothetically enhance performance, but would generally facilitate

the explainability of the neural network's outcomes. By visualizing and examining the outputs of the independent modules, we could potentially understand how the network acquired the main objective's outcomes.

In the following sections, we detail the mechanism by which the modules are connected to the fusion network. A basic fusion network is also introduced, to realize a combination of the intermediate representations, eventually inferring the joint coordinates for actuating the robot's arm.

4.1 Vision Module

The vision module is concerned with the detection of objects in an image, as well as propagating the recognized features to the rest of the fusion network. Being the interface between the camera input and the fusion network, the vision module has to handle raw images captured before the robot initiates motion. We employ the RetinaNet for handling the object detection process.

4.1.1 RetinaNet architecture

The RetinaNet follows the architecture described by Lin et al. [62]. We extend the Keras [1] based implementation [2] of RetinaNet for making it compatible with our fusion architecture. The base model integrates ResNet [41] as the **backbone** model, with 50 layers by default. Extending from the backbone, a Feature Pyramid Network (FPN) connects to the ResNet creating lateral connections between the last three convolutional layers. The RetinaNet FPN is extended with 2 layers, resulting in a total number of usable pyramid layers equivalent to 5. Each pyramid layer has 256 output channels.

Branching from the FPN, the RetinaNet has two parallel subnetworks: The **classification subnet** which predicts the probabilities of objects existing at each spatial position, and the **regression subnet** for regressing over bounding boxes to match the ground-truth boxes. The parameters are not shared across the subnets. The subnets are designed as such:

1. The classification subnet is constructed using four 3×3 convolutional layers with 256 channels for each FPN layer, and parameters shared across all the FPN layers. Each convolutional layer is followed by a ReLU activation, with a sigmoid activation for the final layer. The outputs are of size KA where K represents the number of object classes to be detected, and A represents the number of anchors. Anchors describe the predefined sizes and locations for the potential objects found in an image. We use 9 anchors for all the experiments. The final layer results in an output representing whether a class belongs to an anchor or not, along with the class index.

[1] Keras: `https://keras.io/`

[2] Keras Retinanet: `https://github.com/fizyr/keras-retinanet/`

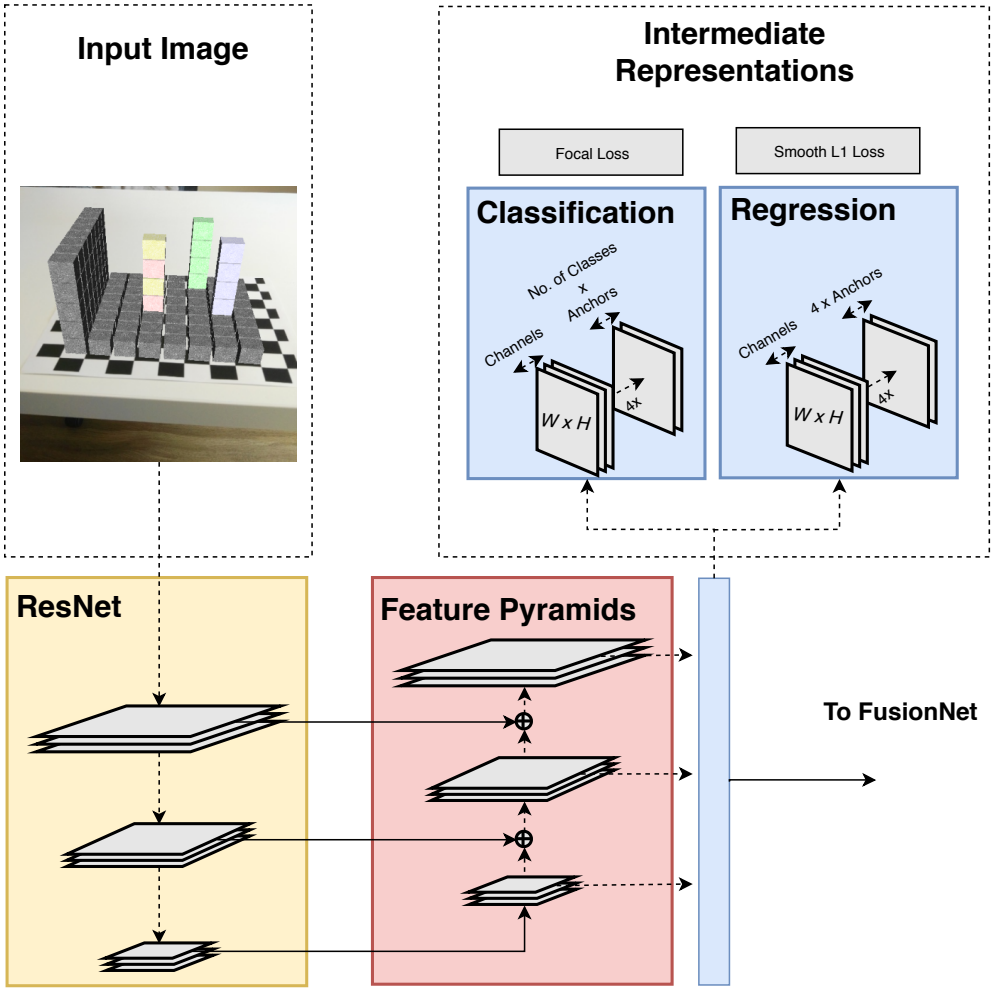


Figure 4.1: The RetinaNet architecture showing the node at which the network branches to the Fusion network

2. The regression subnet is constructed using four 3×3 convolutional layers with 256 channels for each FPN layer, and parameters shared across all the FPN layers. Each convolutional layer is followed by a ReLU activation, with a sigmoid activation for the final layer. The outputs are of size $4A$ where A represents the number of anchors. There are 4 linear outputs per anchor, each regressed to the nearest ground-truth box.

In addition to the two subnets described, we introduce a subnet to forward the FPN to the fusion network. We call it the **fusion subnet**. This subnetwork has a structure identical to the classification subnet, replacing K number of classes with an arbitrary number D defining the number of nodes. For a majority of the experiments, we set D to 4. The modified RetinaNet architecture, displaying the extra fusion subnet is illustrated in figure 4.1. Note that the fusion subnet does not have a loss function explicitly assigned to it since it has no output layers. The losses and their properties for the classification and regression subnets, along with

the training dataset and pipeline are described in the following section.

4.1.2 Training and validation

The regression subnet uses a smooth L_1 loss function [108, p.24], while the classification subnet uses a Focal loss function [62], with a γ mask of 2 and a class weight α of 0.5. The weights are randomly initialized, sampled from a Gaussian distribution with a standard deviation σ of 0.01. All layers excluding the subnets contain a bias units with a weight of 0. For the classification subnet, the bias units are initialized with a weight of -2. For experiments involving the RetinaNet exclusively, we use a stochastic gradient descent optimizer (described in section 2.1.2) with a weight decay of 0.0001 and momentum of 0.9. The initial learning rate is set to 0.01 initially and divided by 10 after $60k$ iterations and again after $80k$ iterations. The learning rate decay is ignored since all our experiments involving the RetinaNet independently are halted before reaching the $60k$ iteration mark.

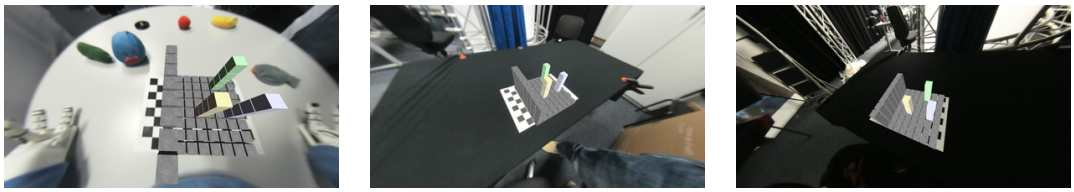


Figure 4.2: The sixth layout in the ETR dataset superimposed on 3 different environmental views used for validation

We train the RetinaNet on the augmented images created using computer-generated blocks as described in section 3.2.4. Note that we only train the network using the noised images. The image noising process is described in section 3.2.5. We apply the preprocessing noise (with three randomly noised images per layout and environmental view) and image transformation noise during the training phase. For validating the outcomes, we use the three environmental views shown in figure 4.2 without added noise. The remaining 12 environmental views are used for training. Figure 3.4 in section 3.2.4 displays all the environmental views used for generating the correctly annotated layouts in the ETR dataset (described in section 3.1).

We justify our training-validation split by pointing out that the number of layouts is limited; however, we are interested in understanding the visual scene from different angles and distances. For validation, we use the environmental views which closely resemble the scene as would be viewed through the robot's cameras. In this manner, we guarantee that there are no overlaps between the environmental views found in the training dataset and the validation dataset, indicating that any improvement in the validation set would imply an improvement in the model's generalization capabilities. The images extracted from the simulator as described in section 3.3.3 are used for testing the RetinaNet model.

The network regresses over the bounding boxes extracted using the method described in section 3.2.4. The bounding boxes are represented as four floating point values. To classify the blocks, we assign a class encoded as one-hot vectors for each of the block classes in the dataset. We have a total of 16 classes, where each class describes both the block's shape (*cube* and *prism*) as well as its color (*yellow*, *white*, *gray*, *magenta*, *blue*, *cyan*, *red*, and *green*). Examples of classes would therefore be `cube_red` or `prism_green`. Our dataset includes a total of 625 layouts and 15 environmental views per layout, with 3 noised versions (preprocessing noise) per image.

4.1.3 Inference

The inference phase entails forwarding the image through the RetinaNet. The objects classes are detected along with their bounding boxes. The detections are ordered by their detection scores, and the top $1k$ detections are maintained. A threshold of 0.05 is set for the FPN confidence, implying that any detections below that threshold are discarded. Greedy Non-maximum suppression (NMS) [43, p.20] is applied to all the detections to discard overlapping bounding boxes which infer the same class. A threshold of 0.5 is set for the NMS, indicating that any bounding boxes with an Intersection over Union (IoU) exceeding 50% are discarded in favor of the bounding boxes belonging to the most confident prediction.

4.2 Language Translation Module

The language translation module is concerned with the translation of English language commands to RCL, as well as propagating the modeled features to the rest of the fusion network. The language translation module has to handle text input describing the desired action before the robot initiates movement. We employ the Transformer for handling the machine translation process.

4.2.1 Transformer architecture

The Transformer follows the architecture described by Vaswani et al. [103]. We extend the Keras based implementation [1] of the Transformer for making it compatible with our fusion architecture. The model is composed of an encoder and a decoder. The encoder receives the English Language command in the form of a list of tokens as input, whereas the decoder receives the RCL command in the form of a list of tokens as input. The autoregressive nature of encoder-decoder model implies that the model should learn to predict the next translated word given the context of the **source** (English Language) sequence as well as the **target** (RCL representation) sequence. The target sequence is learned by enabling connections to the tokens in a sequential manner through a lower-triangle binary mask over all timesteps.

[1] Keras Transformer: `https://github.com/Lsdefine/attention-is-all-you-need-keras/`

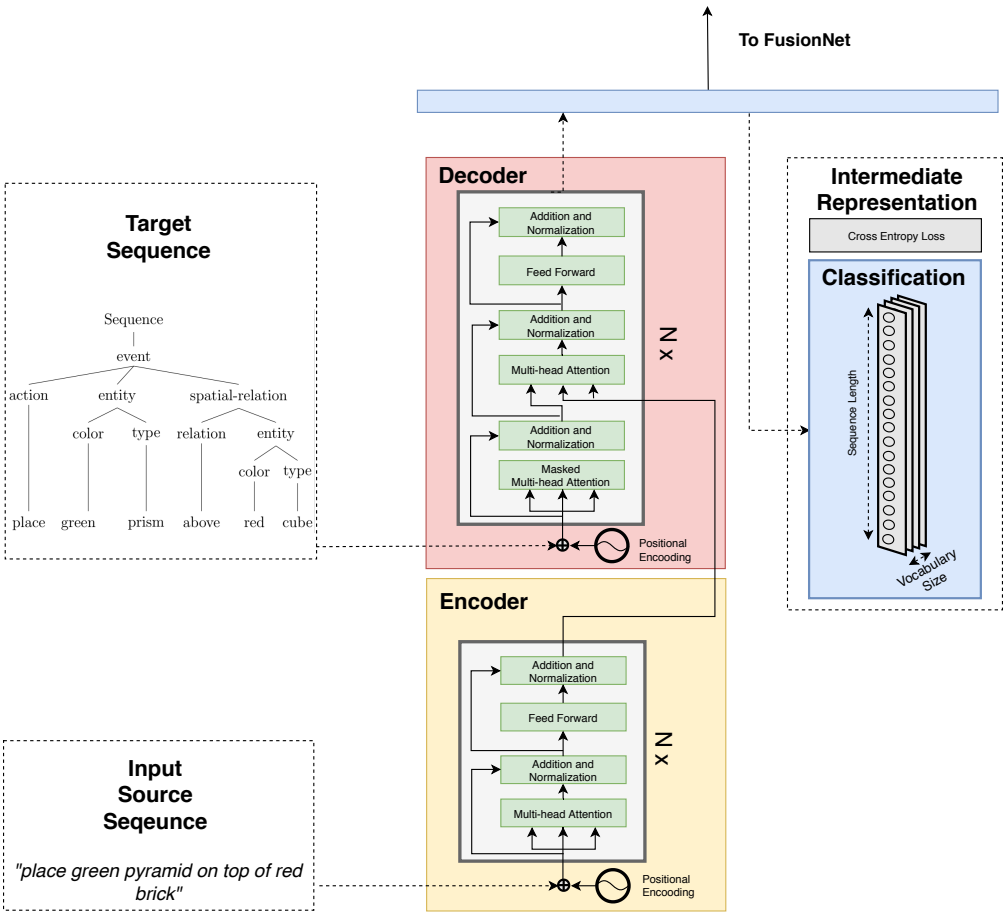


Figure 4.3: The Transformer architecture showing the node at which the network branches to the Fusion network

Since the Transformer is a feed-forward model, the ordering of the words in a sequence is lost. Therefore, to maintain knowledge of a word's position in a sequence, the encoding of a given word is summed with a positional encoding represented in the form of sine and cosine waves. In section 2.1.11, we describe the positional encoding method used in the Transformer. The motivation behind a periodic wave as a positional encoder revolves around enabling the model to generalize to longer sequences than observed during the training phase.

The encoder and decoder are stacked as blocks which are then repeated for N times. They are similar in structure; however, the decoder is masked to avoid leaking connections to words which are unseen yet. Another difference between the encoder and the decoder is observed in the input each sublayer receives. The encoder sublayers receive keys and values from preceding sublayers, whereas the decoder sublayers excluding the first, receive keys and values from the final encoder layer's output. The mechanism by which the encoder and decoder are connected is displayed in figure 4.3.

The intermediate representation of the model is a single fully-connected layer attached to the final decoder layer's output. The output layer has a dimension

of LV where L is the length of the output sequence, and V is the output vocabulary size. We introduce a **fusion sublayer** for propagating the features to the fusion network. The sublayer is identical to the output layer, connected directly to the final decoder's output, replacing V with an arbitrary number D defining the number of nodes. For a majority of the experiments, we set D to 4.

Most of the hyperparameters are dependent on the dataset; hence we refrain from reusing the properties of the baseline model proposed by Vaswani et al. [103] and instead experiment with various combinations which reap the best results. Note that the fusion sublayer does not have a loss function explicitly assigned to it since no output is attached to it. The loss and its properties for the output layer, along with the training dataset and pipeline are described in the following section.

4.2.2 Training and validation

The output layer uses a cross-entropy loss (described in section 2.1.2) applied on the logits of the layer after taking their softmax. The index of the maximum softmax value is set to 1, and all other word indices in the vocabulary are set to 0. This results in a one-hot vector equivalent to the size of the output vocabulary. The index of the ground-truth and the position of the maximum softmax prediction are then compared. When a match between the ground-truth and the prediction is observed, we assume the cross-entropy to be 0; otherwise, the cross-entropy is computed for the false prediction, and the average loss is returned. The network optimizes the loss accordingly, with the goal of reducing the misclassifications to a minimum (ideally 0) over the training examples.

The weights are initialized based on the Glorot et al. [31] initialization method with random uniform sampling. The layer normalization [4] (shown in figure 4.3) weights are initialized with a mean of 0 and a standard deviation of 1. For the layer normalization, an error constant υ is summed with the standard deviation weight for numerical stability. We set υ to 10^{-6}. For experiments involving the Transformer exclusively, we use the Adam optimizer [52] with a β_1 of 0.9, a β_2 of 0.98 and an ϵ of 10^{-9}. The learning rate is scheduled to decrease based on the formula:

$$\eta = d_{model}^{-0.5} \cdot min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \tag{4.1}$$

where η signifies the learning rate at a given training step $step\_num$. d_{model} represents the size of the multi-head attention outputs combined. An arbitrary number $warmup\_steps$ is set to 4000 as proposed by Vaswani et al. In the training phase, we propagate the English language commands as the source sequences. The words are split by spaces and preprocessed as described in section 3.4. The corresponding target sequences are the equivalent RCL commands. The source sequences are noised using the methods mentioned in section 3.4.1. Three noised sequences of a given clean sequence are used for the source sequence, whereas the target RCL sequences are untampered. We use the full range of undiscarded ETR (described in section 3.1) commands to train and validate the Transformer. 80% of

the commands from the ETR dataset are used for training, 10% for validation, and the remaining 10% are reserved for testing. Commands which contain incomplete sentences, incorrect correspondence between the English language commands and the RCL commands, spam, and missing references to the layouts of the visual scenes are discarded. We apply such a noising method (noise applied to the source only) to avoid learning incorrect transduction from natural language to RCL. This approach is considered a regularization technique by which we distort the input only, encouraging the model to generalize better by learning to denoise corrupted sequences.

Dropout regularization [97] is also applied to the Transformer. Dropout is applied to each sublayer before normalization. Dropout is also applied to the summation of the embedding and the positional encoding of the source and target sequences. For a majority of the experiments, we set the dropout probability to 0.1.

The network classifies the words represented to the network as vectors of a predefined size (depending on the embedding dimension). The output layer projects the decoder output to a size equivalent to the vocabulary size, where the layer's unit indices correspond to the word types (unique words) as ordered by their appearance in the dataset. Our dataset includes a total of 357 word types (without noise) and 4850 sequences (without noise), a total of 3882 source word types for the three noised versions of any clean sequence, and 106 target word types.

4.2.3 Inference

The entire source sequence is fed into the encoder. Since the encoder remains unchanged during the inference phase, the final encoder output units are acquired once for every sequence translation. The model predicts a single word at a time until an end-of-sequence token as observed, or the maximum sequence length has been exceeded. The output of the model is fed sequentially as the decoder input after every prediction, updating the mask to include the following word. Decoding the most probable sequence involves searching through all possible outcomes and determining the best matching translation based on its likelihood. Two common approaches for inferring word tokens in neural machine translation models are known as **greedy search** and **beam search**. We detail the two heuristic approaches as follows:

1. A simple approximation of the most likely sequence is known as greedy decoding. As the name implies, we are interested in sequential predictions which offer an immediate reward (minimum perplexity). This indicates that the word with the maximum likelihood is autoregressively fed into the decoder without considering alternatives. After the softmax of the output layer is computed, we find the unit with the maximum probability and look up the matching word in the vocabulary with an identical index.

2. In the greedy approach, we do not consider alternative sequences for possible translations. The alternatives might not immediately result in the best

translation; however, they may result in a lower perplexity once the entire sentence is completely transduced from the source language to the target language. Beam search [92, p.125-126] is a heuristic search which considers multiple possibilities for translating a sequence. The search technique employs a breadth-first algorithm for constructing its search tree. The beam width is a hyperparameter which defines the maximum number of possibilities at each pruning stage. W symbolizes the number of beams. Larger beam-width results in a more accurate translation at the cost of more resources (time or memory). At each pruning stage, the product of all probabilities is computed for all beams given the preceding sequence, and the words with the top W probabilities are selected for the next prediction. A typical value for W is set to 5 or 10.

4.3 Fusion Module

The fusion module is concerned with combining the features extracted from the vision module and the language translation module. On integrating the features, we aim to learn coordinates which enable the robot to initiate movement. Not only does the neural network learn the coordinates for reaching blocks, but also relocating them based on a natural language command. We design a simple feed-forward network for handling the multimodal integration. We call this network the **FusionNet**.

4.3.1 FusionNet architecture

The FusionNet combines the fusion subnet nodes branching from the RetinaNet (described in section 4.1.1) and fusion sublayer nodes branching from the Transformer(described in section 4.2.1). The two fusion layers must be of similar dimensions in order to merge them successfully. The number of nodes for both modules is set to four for a majority of the experiments. After concatenating the RetinaNet and Transformer nodes, we flatten them, forming a one-dimensional layer. The "join" subnetwork shown in figure 4.4 forms a sensorimotor mapping between the two sensory modalities and the output. The output layer has 14 units with a sigmoid activation, each describing a motor joint angle or a gripper state. The first seven units represent the initial grasping pose including the initial boolean state of the gripper, whereas the remaining 7 units represent the final grasping pose including the final boolean state of the gripper. Figure 4.4 displays the architecture of the FusionNet **base model** which we employ for conducting a majority of the experiments. The loss and its properties, along with the training dataset and pipeline are described in the following section.

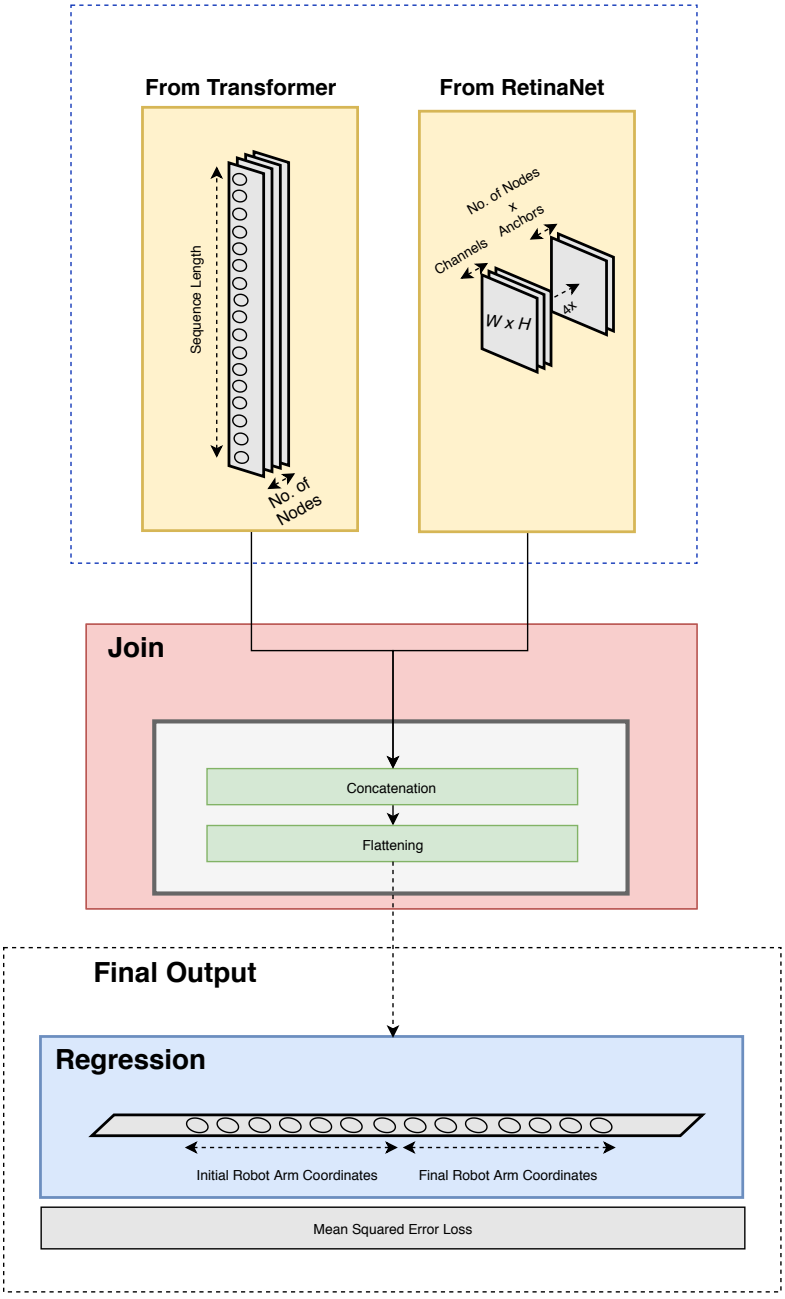


Figure 4.4: The FusionNet architecture showing the nodes arriving from the RetinaNet and the Transformer

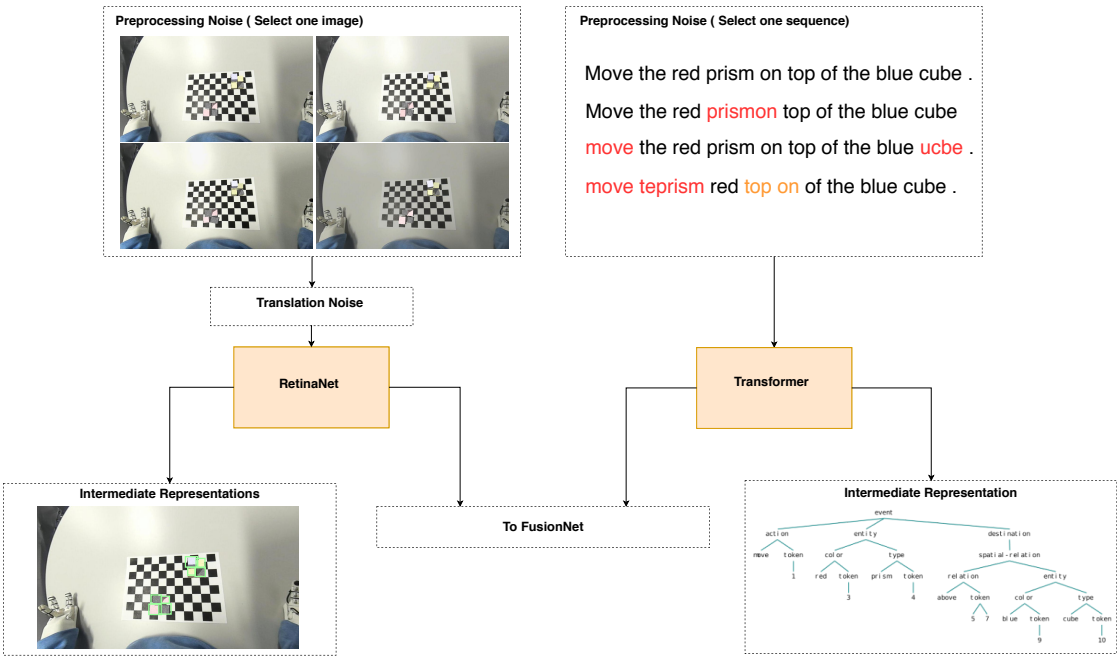


Figure 4.5: The training pipeline

4.3.2 Training and validation

The regression output layer uses a mean squared error loss function (described in section 2.1.2. The weights are initialized based on the Glorot et al. [31] initialization method with random uniform sampling. For a majority of the experiments involving the FusionNet along with the vision and language modules, we use the Adam optimizer [52] with a β_1 of 0.9, a β_2 of 0.999 and a learning rate of 10^{-5}.

The training and validation sets for both the vision module and the language translation module remain unchanged. These act as inputs to the FusionNet. The FusionNet introduces the arm joint angles as an output sharing features from the RetinaNet and the Transformer. We extract the arm joint angles using the simulator as described in section 3.3.2. Each example fed into the network has an image layout (initial layout), a natural language command, an RCL command, and initial as well as a final arm joint angles associated with it. We mentioned earlier that the language translation and vision modules accept noisy versions of each example for training. Since we have a single set of initial and final joint angles, we would randomly select an example from the noised versions for each module. Figure 4.5 shows the input selection process during the training phase. The joint angles are represented in radians. We perform feature scaling upon the joint angles, such that they are limited to a range of 0 and 1. The approach used is known as min-max normalization applied by:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{4.2}$$

where x represents the joint angle in radians, and x_{min} and x_{max} represent the

minimum and maximum values respectively. We extract the minimum and maximum values by scanning the entire dataset (training and validation datasets) after the joint angles were generated using the simulator and store the extremes in a file. The normalization of values is a preprocessing step which allows for faster training. 80% of the scenes from the ETR dataset are used for training, with 10% used for validation, and the remaining 10% for testing.

4.3.3 Inference

Due to the autoregressive nature of the Transformer, inferring the initial and final joint angles must be done over two stages. We follow a similar inference pipeline as described in section 4.2.3 for translating English language commands to RCL. On completing the translation, we feed the natural language sequence into the encoder and the decoded RCL sequence into the decoder of the language translation module. The images are directly fed into the vision module, resulting in 14 coordinates at the FusionNet output.

Chapter 5

Experiment 1: Transformer

The Transformer described in section 2.1.11 acts as the language translation module for our fusion network. In this chapter, we conduct experiments related to the Transformer's functionality with respect to our dataset. The Transformer is composed of multiple units, each with its own set of hyperparameters. The values of these hyperparameters greatly influence the performance of the network. Hence we try to explore their effect on the Transformer. We proceed by defining these hyperparameters and their functions.

An essential hyperparameter is the sequence length. Within the context of the language translation Transformer, the sequence length refers to the number of words in a sentence. The length of the RCL sequences tends to be significantly larger than that of natural languages. Since the objective of using the Transformer is to translate from the English language to RCL, we have to account for the length of an RCL sequence when setting up the model. For simplicity, the sequence length for both the encoder and the decoder of the Transformer are set to the same value, with that being the minimum number of words for all individual RCL sequences in the ETR dataset. The sequence length is denoted by l_{seq}

The multi-head attention is defined by the number of attention heads. Setting a higher number of attention heads for the multi-headed attention sublayer allows the model to focus on different parts of a sequence. However, setting the number of attention heads to a value too large might add unnecessary complexity and redundancy in the network. The number of heads is denoted by N_h. The output of a single attention head must be unified across the network to facilitate residual projection and to enable the dot product attention.

The feed-forward layer facilitates point-wise projection after performing layer normalization on the multi-headed output. The feed-forward layer's dimension is denoted by d_{ff}.

The query and value projections of the Transformer can have a custom size. Since the weights of these projections are changing throughout the learning process, their size also has a major effect on the outcome, by indirectly influencing the granularity of the attention. The query and value projection dimensions should match. The size of the input embeddings can be predefined as well but is not restricted by the dimensions of the query and the value projections. The query

dimension is denoted by d_q, and the value dimension is denoted by d_v. Since the dimensions of both the query and the value are identical, we denote d_q and d_v by d_a. The embedding dimension is denoted by d_{emb}.

The decoder and encoder blocks can be repeated for any number of times, stacked on top of each other. Even though increasing the number of layers should improve performance since the Transformer implements residual connections, the number of model parameters would increase significantly. This increase would reflect on the training and inference time and would have a major impact on the memory allocated for the model. The number of encoder blocks is denoted by N_{enc}, whereas N_{dec} denotes the number of decoder blocks. Since the number of both blocks is identical, we denote N_{dec} and N_{enc} by N_m

A dropout with a certain probability is applied to the Transformer network during training. This should reduce overfitting and regularize the model. Each sublayer has a dropout associated with it. Dropout is also applied to the layer following the summation of the embedding and positional encoding layers for both the encoder and the decoder. The dropout probability is denoted by P_{drop}.

Other hyperparameters describing the training procedure include the number of training iterations, the batch size, and the optimization algorithm as well as its properties. The number of training iterations is denoted by N_{itr}, and the batch size is denoted by s_{batch}. The optimizer is denoted by *opt* with subscripts indicating its properties where necessary.

The base model properties are summarized in table 5.1.

Table 5.1: The base Transformer network hyperparameters

N_{itr}	s_{batch}	opt	opt_{β_1}	opt_{β_2}	opt_{ϵ}	P_{drop}	N_h	N_m	d_a	d_{emb}	d_{ff}	l_{seq}
$16k$	60	*adam*	0.9	0.98	10^{-9}	0.1	2	4	12	50	128	200

The authors of the Transformer proposed a technique known as label smoothing for computing the loss [103]. Label smoothing was shown to harm the perplexity at the cost of an improved BLEU score. However, the BLEU score in our task holds little meaning, since RCL is different from natural languages for which the score was intended to evaluate. Consequently, we did not use label smoothing in the Transformer experiments. For the experiments to follow, the hyperparameters are assumed to default to the base model's properties unless otherwise stated. For a detailed description of the network and the dataset properties, refer to section 4.2.

5.1 Embeddings

For this set of experiments, we explore the influence of various embeddings on the outcome of the Transformer. We hypothesize that using a word embedding other than a randomly initialized matrix could affect the network output. As discussed in section 2.1.9, Word2vec is implemented in two forms: a continuous bag of words

model where a word is predicted given its context and a skip-gram model where the context is predicted given a word. The model is denoted by $w2v_{mod}$, where a value of *sg* represents the skip-gram model and *cbow* represents the continuous bag of words model. Negative sampling is denoted by $w2v_{ns}$, where any value greater than 0 signifies the number of negative samples. The $w2v_{ns\_exp}$ identifier represents the exponent shaping the distribution of the negative sampler. The window size, which defines the distance between the input word or words and the predicted word or words, is denoted by $w2v_{window}$. We define a minimum frequency of word occurrence as $w2v_{min\_freq}$, where any word occurring less than the minimum frequency is ignored. The Word2vec initial learning rate is denoted by $w2v_{lr}$ and $w2v_{min\_lr}$ defines the minimum learning rate. The minimum learning rate changes linearly from its initial value towards the minimum learning rate value over a predefined number of iterations. The number of iterations is dependent on the size of the corpus itself. Therefore, we define the number of epochs instead, and denote it by $w2v_{epoch}$. The properties of the base Word2vec-CBOW model are shown in table 5.2. $w2v_{mean}$ is a flag for defining whether the sum or the mean of the context vector will be used for the *cbow* model. Enabling the flag uses the mean, and disabling it uses the sum. Note that $w2v_{mean}$ is enabled for the base Word2vec-CBOW model. The properties of the base Word2vec skip-gram are

Table 5.2: The base Word2vec CBOW hyperparameters

$w2v_{epoch}$	$w2v_{mod}$	$w2v_{ns}$	$w2v_{ns\_exp}$	$w2v_{window}$	$w2v_{min\_freq}$	$w2v_{lr}$	$w2v_{min\_lr}$
200	*cbow*	5	5	9	0.75	0.025	0.001

shown in table 5.3. Most of the properties are identical for both Word2vec models to avoid biases when using the produced embeddings. Note that the $w2v_{window}$

Table 5.3: The base Word2vec skip-gram hyperparameters

$w2v_{epoch}$	$w2v_{mod}$	$w2v_{ns}$	$w2v_{ns\_exp}$	$w2v_{window}$	$w2v_{min\_freq}$	$w2v_{lr}$	$w2v_{min\_lr}$
200	*sg*	5	5	9	0.75	0.025	0.001

size is defined as a function of the input data. Since we have four samples of each sequence for training (the original sentence and its three noised versions), we set $w2v_{window}$ to $1 + (4 \times 2)$. Other properties are set as per the hyperparameters of the base models introduced in [66]. Note that the results reported in the following experiments represent the validation losses unless specified otherwise.

5.1.1 Experimental setup

We use the properties defined for the base model. The datasets described in section ADD SECTION are used for training and evaluation. We experiment with both Word2vec base models (base Word2vec CBOW and base Word2vec skip-gram) and the default randomly initialized vectors as an alternative. We also vary the d_{emb}

within the range of $\{10, 50, 100, 200, 500\}$. Note that the lookup embedding table for both the English language source sequence and the RCL target sequence are pretrained on their respective training sequences separately. The experiment was repeated three times for each setup. We ran the experiments for three days on an Nvidia GTX 1050 Ti graphics processing unit.

5.1.2 Results and discussion

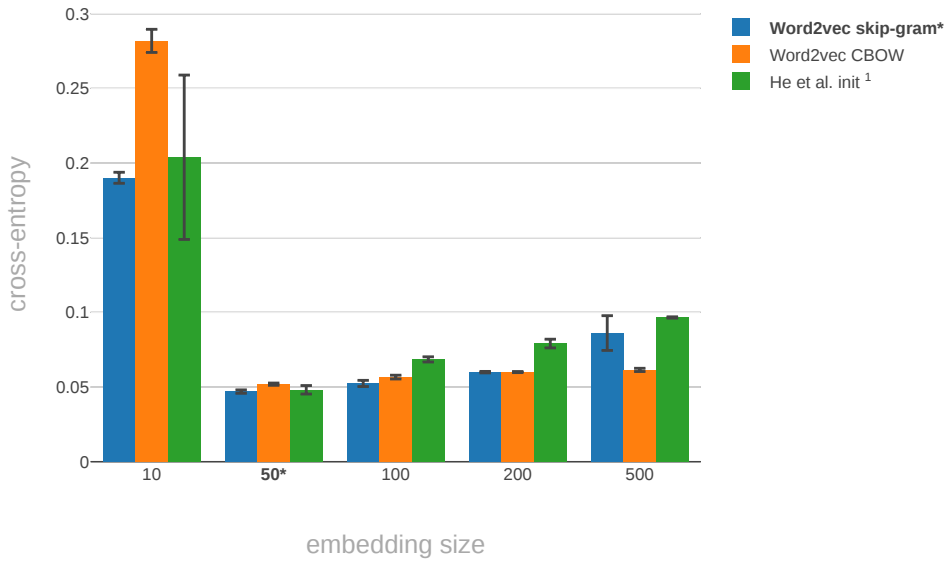


Figure 5.1: The average cross-entropy for three repetitions per embedding type and dimension
[1] The default embedding is randomly initialized and scaled using the method proposed by He et al. [40]

Figure 5.1 shows the average cross-entropy for all three repititions per embedding type and dimension. We observe that embeddings of size 10 result in the highest (worst) cross-entropy for all three embedding types. For the three repetitions, we acquired a mean and a standard deviation of 0.2818 ± 0.0077 for the Word2vec CBOW embeddings, 0.2038 ± 0.055 for the randomly initialized embeddings with He et al. [40] scaling, and 0.1901 ± 0.0036 for the Word2vec skip-gram embeddings. We also observe that embeddings of size 50 yield the lowest (best) cross-entropy, yet increase proportionally with the embedding size. This indicates that embeddings of size 50 are best suited for our dataset. Word2vec skip-gram embeddings were found to achieve the lowest cross-entropy amongst the other two embedding types. The best Word2vec skip-gram model with $d_{emb} = 50$ achieved a mean of 0.0469 and a standard deviation of 0.0011 for all three repetitions.

Higher embedding sizes add complexity to the learning procedure without added information; on the other hand, smaller sizes are unable to represent the semantics of both languages. We did not explore the entire space of possible combinations nor do we tie the embeddings of the encoder and the decoder even though their embedding sizes are set to be identical. Nevertheless, the consistency in the trend towards a rising cross-entropy with an increasing size indicates that an embedding size of 50 for the target and source languages is relatively proximate to the ideal size.

5.2 Layers

For this set of experiments, we explore the influence of varying the number of layers and their dimensions on the network performance. We hypothesize that the performance of the network would increase as we stack up more layers of encoders and decoders. However, the degradation problem might still be observable.

5.2.1 Experimental setup

We use the properties defined for the base model. We examine the influence of N_m by changing its value between $\{4, 6, 8, 10\}$, and varying N_h between $\{2, 4, 8\}$. We also vary d_a within the range of $\{12, 32, 128, 256\}$ and d_h within the range $\{128, 256, 1024, 2048\}$. For all setups, we use pretrained Word2vec skip-gram embeddings with $d_{emb} = 50$, since it appears to result in the least cross-entropy following from the discussion in section 5.1.2. We set s_{batch} to 5 sequences due to memory constraints being exceeded as we increase the size of the network. The experiment was repeated three times for each setup. We ran the experiments for 8 days on an Nvidia GTX 1050 Ti graphics processing unit.

5.2.2 Results and discussion

Increasing the number of layers in the network should in theory improve the performance since residual connections exist between the sublayers. However, the observation contradicts the expectation. As seen in figure 5.2, the performance degrades as we linearly increased the number of layers from 4 to 10. We acquired the lowest (best) cross-entropy with a mean of 0.076 and a standard deviation of 0.0015 for three repetitions of a setup with $N_h = 4$, $N_m = 4$, $d_a = 12$, and $d_h = 256$.

To verify the correctness of the implemented Transformer, we trained the network with the European Parliament Proceedings Parallel Corpus English-to-German translation for the years 1996-2011 using a set of hyperparameters identical to those described in the experimental setup except for embeddings, which we randomly initialized instead. We set $N_h = 8$, $d_a = 128$, $d_h = 256$ and ran the experiment for $50k$ iterations with 3 repetitions per layer size variation. The results were consistent with those achieved by [103]: increasing the number of layers

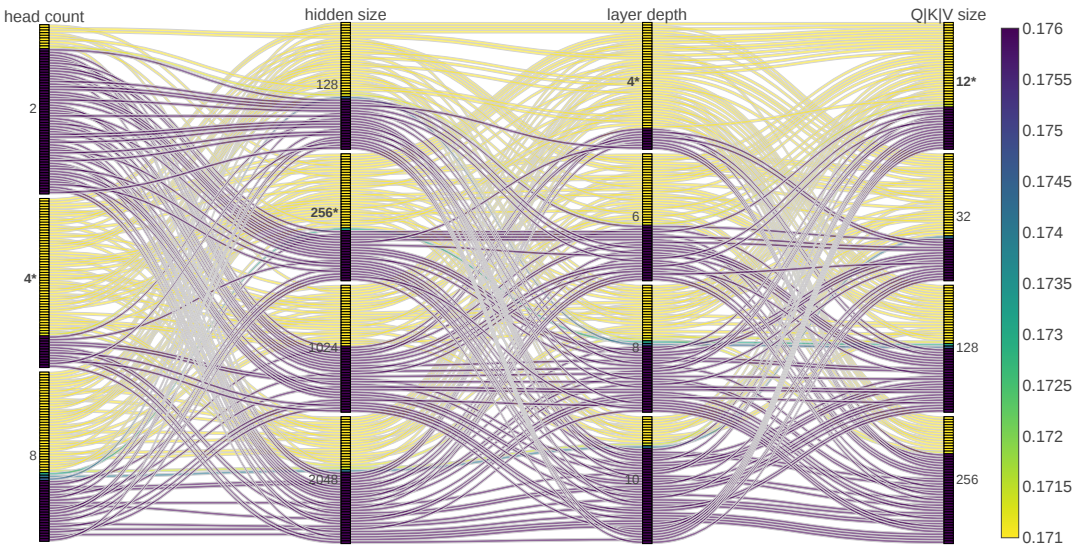


Figure 5.2: The average cross-entropy for three repetitions with different layer depths, hidden layer dimensions, number of heads, and query dimension.

improves the Transformer's performance.

In an attempt to solve the issue caused by increasing the number of layers using our dataset, we turned our attention to the embedding initialization. We restored the model to initialize the weights using the approach specified by He et al. [40], however, the performance still degraded as we increased the number of layers. Optimizing the learning rate and re-adjusting the decay schedule described by Vaswani et al. [103] would likely yield optimal results as well. However, we avoid adjusting global hyperparameters such as the learning rate, since the optimizer will be unified for all modules during the fusion phase. The optimizer, the learning rate, the learning rate scheduling along with the batch size will be disregarded in favor of the FusionNet's requirements.

We observe that increasing the hidden layer size as well as the number of attention heads benefits the network. Increasing the number of attention heads makes the network wider, which should result in a degradation in the performance beyond a certain point. The same applies to the point-wise feed forward (hidden) layer (d_h) and the projection matrices (d_a). As we increased the number of heads beyond 4, the hidden layer size beyond 256, and the projection layer size beyond 12, we observed a reduction in the model's accuracy.

5.3 Regularization

For this set of experiments, we explore the influence of dropout on the model's ability to generalize well to unseen examples during training. We also explore the influence of data augmentation on the evaluation outcome.

5.3.1 Experimental setup

We use the properties defined for the base model. The P_{drop} is chosen from values in the range $\{0, 0.1, 0.2, 0.3, 0.4\}$. After realizing the best P_{drop}, we use the dropout configuration with different modes of perturbation (noise) on the training dataset. The default training dataset is composed of source sequences with clean English language commands along with three perturbed commands of each sequence. We ablate the perturbed sequences linearly ranging from two to no perturbations per example. We also examine whether training the Transformer on clean sequences achieves better results. For all experiments, we use the Word2vec skip-gram embeddings, with $d_{emb} = 50$. We set $s_{batch} = 5$ to achieve results comparable to those acquired in section 5.2. The experiment was repeated three times for each setup. We ran the experiments for 27 hours on an Nvidia GTX 1050 Ti graphics processing unit.

5.3.2 Results and discussion

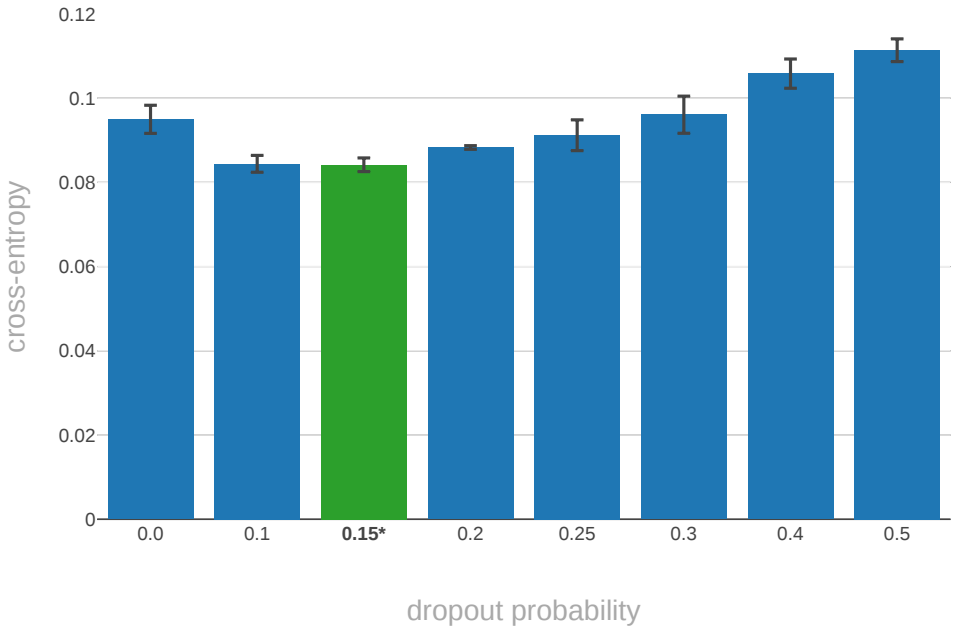


Figure 5.3: The average cross-entropy for repetitions trials with different dropout probabilities

The dropout probability controls the likelihood of discarding words from a sequence at each training step. We notice from figure 5.3 that the dropout probability influences the cross-entropy on the validation dataset. A probability of 0.0

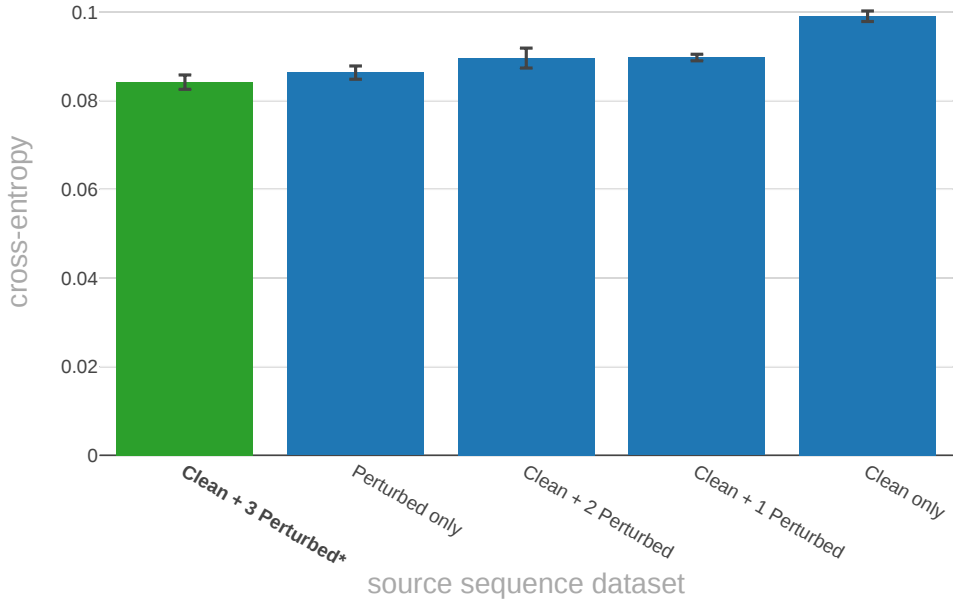
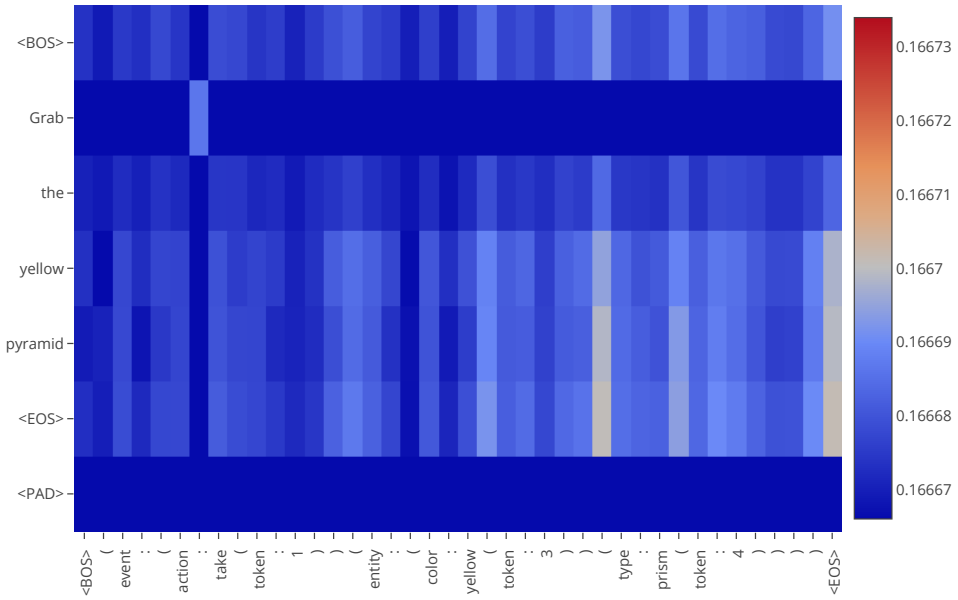


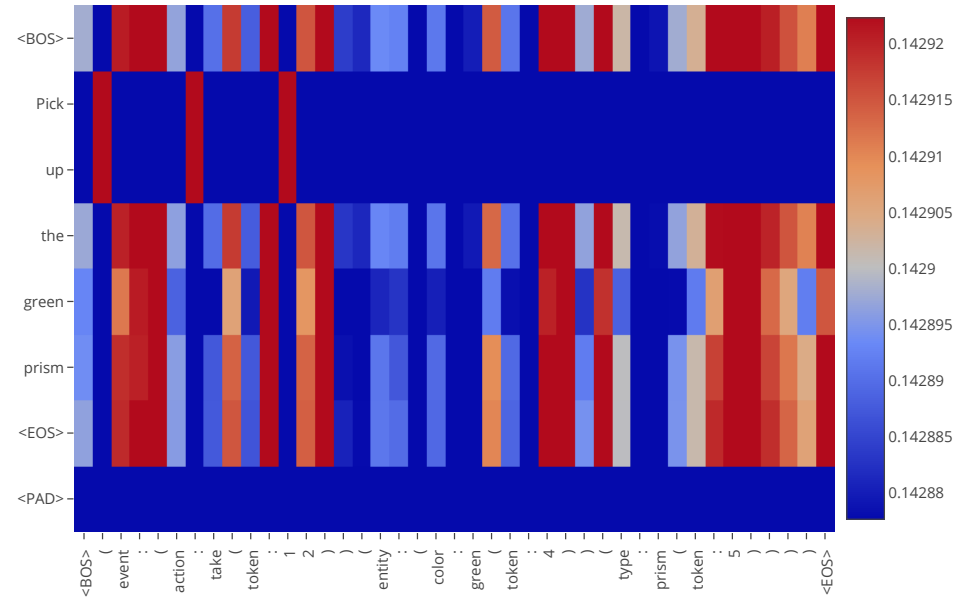
Figure 5.4: The average cross-entropy for three repetitions with different perturbations applied to the training dataset

indicates that dropout did not take place. With a probability of 0.15, we achieved the lowest (best) cross-entropy with a mean of 0.0841 and a standard deviation of 0.0016 for all three repetitions. As we increased the probability beyond 0.15, the results degraded as a result of under-learning since the sequences became too distorted. Without dropout, we also observe worsened results since it acts as a regularizer, hence reducing the network's capability to generalize.

With a dropout probability of 0.15, we moved on to examine whether the sequence perturbation method described in section 3.4.1 was beneficial. We observe in figure 5.4 that the default training dataset having clean sequences along with their three perturbed versions achieved the lowest (best) cross-entropy with a mean of 0.08417 and a standard deviation of 0.0016 for all three repetitions. Experiments conducted on clean sequences only achieved the highest (worst) cross-entropy with a mean of 0.099 and a standard deviation of 0.0011 for all three repetitions. This indicates that our mixed dataset with clean and perturbed sequences introduces sufficient regularization to the model without compromising its ability to learn meaningful patterns.



(a) Grab the yellow pyramid



(b) Pick up the green prism

Figure 5.5: The average transformer encoder-decoder attention projecting from 4 multi-head attention blocks in the last layer. The source sequences are displayed vertically, whereas the predicted target sequences are displayed horizontally. In both (a) and (b), the actions (e.g., Grab, Pick up) have a unique attention pattern with the decoded sequences compared to other tokens in the sequence.

5.4 Analysis

We trained the Transformer with hyperparameters matching those of the best models as concluded through the experiments described in this chapter. We set $l_{seq} = 200$, $N_m = 4$, $N_h = 4$, $d_a = 12$, $d_h = 256$. We used Word2vec skipgram embeddings with $d_emb = 50$ and trained the model for $32k$ iterations with a batch size of 30. All other hyperparameters matched the base model's hyperparameters.

We achieved a minimum cross-entropy (computed over $1k$ iterations) of 0.0987 for the training dataset with $15.1k$ commands, 0.0968 for the validation dataset with 485 commands, and 0.0973 for the testing dataset with 485 commands. This resulted in a word-level accuracy of 96.8% for the validation dataset and 95.2% for the testing dataset. The sentence-level accuracy assumes a full match between the ground truth command and the prediction to be considered a successful parse. We achieved a sentence-level accuracy of 16.7% on the testing dataset using a greedy decoder, and 17.5% using beam search with a beam width of 5.

On a word-level, the model appears to show significant improvement over [20]; however, most commands appear to be distorted on a sentence-level. Two major contributors to this failure were the token position number aligning the natural language tokens with the RCL tokens, as well as the block colors. In figure 5.5, we show the encoder-decoder attention matrix projected from the fourth layer in the Transformer model, averaged over all 4 heads. Although the Transformer has proven to form meaningful attentive relations between words of the same sequence (encoder-encoder or decoder-decoder) [103], we observe that it does not attend well to the encoded sequences with relation to the decoded sequences. We justify our model's failure to align words correctly by pointing out that the Transformer was examined on natural language translation where both sequences are mostly aligned. However, RCL reorders the sequences to form a parsable tree, and the fertility ratio between the two sequences is high.

In figure 5.5, we observe a unique attention mask in relation to the actions. The model segregates actions from other words in the source sequence, signifying that actions are distinguishable and therefore have a major influence on the decoded transduction.

Chapter 6

Experiment 2: RetinaNet

The RetinaNet described in section 2.1.10 acts as the vision module for our fusion network. In this chapter, we conduct experiments related to the RetinaNet's functionality with respect to our dataset. We proceed by defining the hyperparameters of the RetinaNet and their functions.

The size of the image greatly influences the performance of the RetinaNet. Images with a high resolution require a lot of computational resources, therefore, they must be resized before passing them as input to the network. The image input size is denoted by l_{res}

We denote the Focal loss [62] used for classification by $focal$ with subscripts indicating its properties where necessary.

All backbones used for the RetinaNet were pretrained on $1.2M$ images from ImageNet [1] dataset. For all experiments to follow, the backbone is assumed to be initialized with the ImageNet trained model weights.

Other hyperparameters describing the training procedure include the number of training iterations, the batch size, and the optimization algorithm as well as its properties. The number of training iterations is denoted by N_{itr}, and the batch size is denoted by s_{batch}. The optimizer is denoted by opt with subscripts indicating its properties where necessary.

We summarize the hyperparameters for the RetinaNet base model in table 6.1.

Table 6.1: The base RetinaNet hyperparameters

N_{itr}	s_{batch}	opt	opt_{η}	$opt_{momentum}$	opt_{weight_decay}	$focal_{\gamma}$	$focal_{\alpha}$	l_{res}
$50k$	1	sgd	0.01	0.9	10^{-4}	2	0.5	640×480

For the experiments to follow, the hyperparameters are assumed to default to the base model's properties unless otherwise stated. For a detailed description of the network and the dataset properties, refer to section 4.1.

[1] Tensorflow ResNet pretrained on the ImageNet: `https://github.com/tensorflow/models/tree/master/research/slim#pre-trained-models`

6.1 Anchors

For this set of experiments, we adjust the anchors of the RetinaNet. Adjusting the anchor sizes and shapes plays a major role in defining the precision of the training outcome. The anchors must be defined before the training takes place since the bounding box regression is based on computing the error between the anchor locations and the predictions of the box boundaries. Lin et al. [62] report results based on 9 anchors per feature pyramid level, covering the scale ranges from 32 pixels to 813 pixels with respect to the input image. We avoid diverging too far from those ranges, yet, we were still interested in assigning anchor sizes that match the block dimensions in our dataset.

Redmon et al. [87] proposed an anchor-based approach to replace the common grid-based approaches. The grid-based approaches employ a filter map and stride with a predefined window size over all regions of the filter to create anchors. Such approaches assume all targeted objects are of similar proportions and sizes. With the anchor-based approach, quadrilateral shapes are predefined for different objects making the approach more flexible to varying sizes. Redmon et al. define the anchor boxes using a k-means algorithm, by which annotated boxes in the training dataset are clustered into c clusters, equivalent to the number of anchors. Although the RetinaNet relies on the grid-based approach, we apply a similar method of clustering to get an impression of the ideal anchor sizes, after which we interpolate the sizes to match the RetinaNet anchor box configuration.

We apply a k-medians clustering algorithm [1] to group bounding boxes by the proximity of their Intersection over Union (IoU) [37] to the centroids of the clusters. Moreover, we apply the distance measure described by Redmon et al. [87]. The medians widths and heights of bounding boxes in each cluster represent the updated centroids of the clusters after each iteration. We eventually compute the IoU to evaluate the fitness of the clusters to the ground truth boxes. The difference between our approach and the approach proposed by Redmon et al. [87] is that they consider the means of the cluster elements to update the centroids, whereas we use the median. Through an empirical evaluation [1], the medians were found to generate better clusters.

By default, the RetinaNet applies 9 anchors to each pyramidal layer, where they cover a range of 32 pixels up to 813 pixels. Each layer anchors are sized by three factors $\{2^0, 2^{\frac{1}{3}}, 2^{\frac{2}{3}}\}$ and three aspect ratios $\{1:1, 1:2, 2:1\}$. We use a similar number of 9 anchors for clustering.

Although anchors play a significant role in vision-based object detection models, other factors come into play. Setting the anchor boxes to a minuscule size would not enable learning even when the anchors match the ground truth boxes. Small objects are a challenge for neural networks to detect and can hinder accurate learning of bounding boxes. To minimize the issue, we crop the images from our synthetic "Pick and Place" dataset to the size of the object placement region. Since we are able to extract bounding boxes using the method defined in section

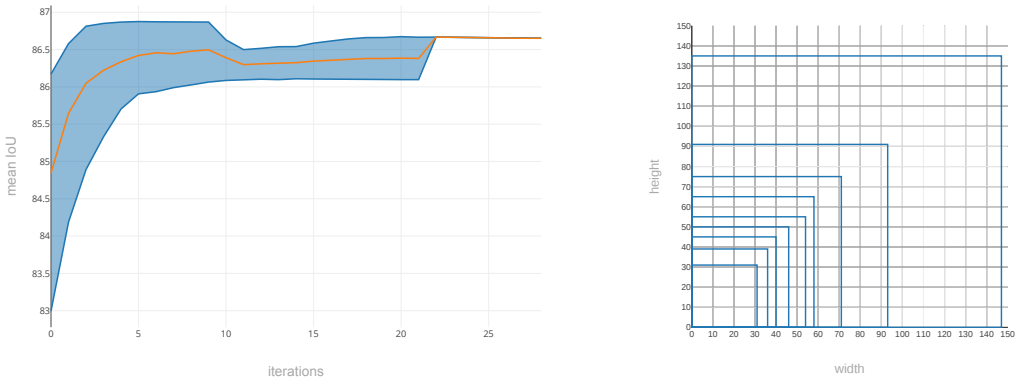
[1] k-means for generating anchors: `https://github.com/lars76/kmeans-anchor-boxes/`

3.2.4, we can infer the size of the visible region (where objects are overlaid on the image) by ensuring that none of the bounding boxes are cropped. We define a crop ratio identical to our images' aspect ratios, and multiply it by scale factors of $\{0.12, 0.25, 0.5, 0.75, 0.9, 1.0\}$. We crop the images with a crop region centered around the object placement region, starting with the minimum crop ratio up until all bounding boxes are visible or the maximum scale factor is reached. On cropping the images, we translate the anchor boxes to the left and to the top by the number of cropped pixels in either direction to restore their original position. We examine our k-median clustered anchors on the images before and after cropping.

6.1.1 Experimental setup

We assign $c = 9$ clusters to the k-medians algorithm. The k-medians clustering algorithm is trained on all $30k$ images in the training dataset. We set the maximum number of k-medians iterations to 100, with an early stopping criterion. The early stopping criterion is triggered when no change in clusters is observed since the last iteration. Nine random bounding box centroids are chosen as the initial centroids for the clusters. The k-medians algorithm is trained on both the cropped and uncropped images independently. Each k-medians experiment was repeated 10 times.

6.1.2 Results and discussion

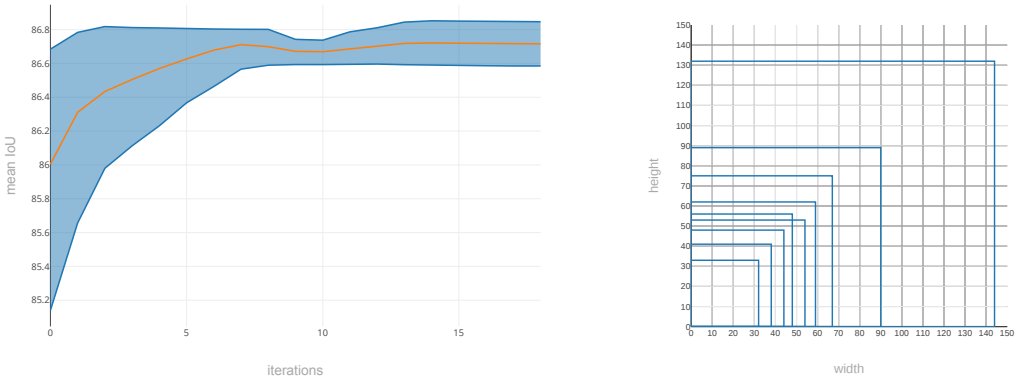


(a) k-median iterations

(b) anchors

Figure 6.1: (a) The mean IoU for 10 k-median runs with 9 clusters applied on the IoU for all training images (b) The 9 anchor boxes generated by the best achieving run

We ran the k-medians clustering algorithm on the training images achieving a mean IoU of 86.87% after 25 trials with 10 repetitions. We also ran the k-medians clustering algorithm on the cropped training images achieving a mean



(a) k-median iterations

(b) anchors

Figure 6.2: (a) The mean IoU for 10 k-median runs with 9 clusters applied on the IoU for all cropped training images (b) The 9 anchor boxes generated by the best achieving run

IoU of 86.82% after 20 trials over 10 repetitions. In figure 6.1 (b) we observe the different cluster shapes and sizes for the given the original images in our training dataset. The original images are too large for the resources at hand since a single image has a resolution of 1980×1080 pixels. As a result, the grid of generated anchors becomes prohibitively large, forcing us to resize the images. The RetinaNet was operational as we resized the images to a resolution of 990×540. However, the bounding boxes became significantly smaller, causing the anchors to no longer match the block shapes in our images. To work around this issue, we cropped the images to cover the grasping region.

We used the anchor shapes shown in figure 6.2 (b) to guide our decision in defining the optimal anchor shapes for the RetinaNet. We set the anchors for the five pyramidal layers to $\{30, 53, 90, 100, 150\}$ starting from the first to the last layer respectively. We modified the aspect ratios setting them to $\{17:20, 1:1, 11:10\}$ based on the minimum and maximum aspect ratios for the 9 inferred anchors. We did not modify the size factors, and maintained the same scales proposed by Vaswani et al. [103] $\{2^0, 2^{\frac{1}{3}}, 2^{\frac{2}{3}}\}$.

6.2 Backbones

For this set of experiments, we examine the influence of different backbones on the RetinaNet's outcome. The RetinaNet requires a ResNet backbone model for constructing the FPN. We provide a more more details on the FPN and its structure in section 2.1.10. Although the ResNet with 50 layers is sufficient for detecting objects, having deeper residual networks have shown improvement over their shallower counterparts. We varied the backbone to observe the influence of deeper

ResNet layers on the RetinaNet's performance. For more details on the employed ResNets and their layer structure, refer to table 2.4 in section 2.1.10.

6.2.1 Experimental setup

We use the properties defined for the base model. We use three different ResNets and compare their weighted mean Average Precision (mAP). The Jaccard index threshold for the mAP is set to 0.5 with non-maximum suppression applied to all evaluations. We evaluate the RetinaNet network with ResNet 50, 101, and 152 as a backbone. The experiment was repeated three times for each setup. We ran the experiments for three days on an Nvidia GTX 1050 Ti graphics processing unit.

6.2.2 Results and discussion

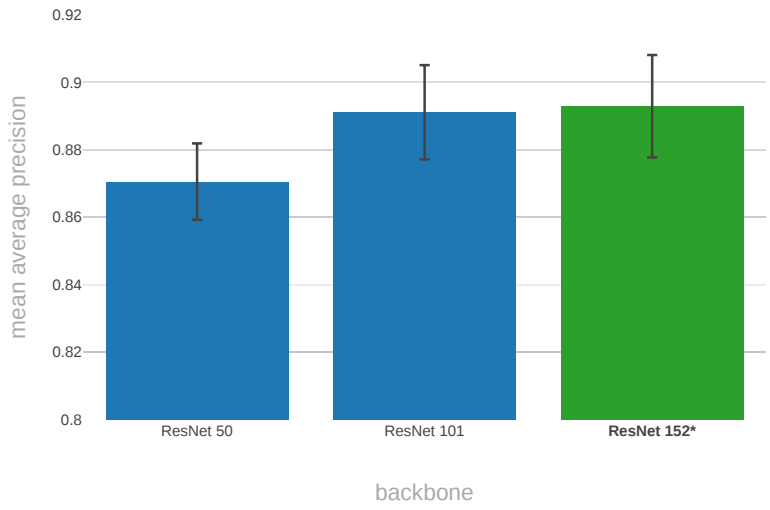


Figure 6.3: The average mAP for three repetitions using different ResNet backbones

In figure 6.3, we show that the ResNet with 152 layers achieved the highest mAP with a mean of 0.893 and a standard deviation of 0.0152 for all three repetitions. The increase in precision occurs due to the residual connections between the ResNet layers. Since each layer is responsible for learning the identity function, the degradation problem is mitigated. More layers offer the network a larger range of parameters to accurately estimate a mapping between the input and output.

6.3 Analysis

We trained the RetinaNet with the optimal anchor properties as deduced in section 6.1.2 using ResNet152 as a backbone. All other hyperparameters matched the base model's hyperparameters. In table 6.2 we show the resulting mAP (with a IoU threshold of 0.5 and non-maximum suppression) on the validation dataset for all the classes individually. The neural model successfully learns to identify the objects in our dataset, hence no further optimization was required for the RetinaNet.

Table 6.2: Individual classes with their mAP trained on the "Pick and Place" dataset

Instances	**Class**	**mAP**	**Instances**	**Class**	**mAP**
297	cube_magenta	0.8664	255	prism_magenta	0.9545
5355	cube_green	0.9143	342	prism_green	0.9547
2001	cube_white	0.8884	69	prism_white	0.5431
6132	cube_red	0.9091	534	prism_red	0.9627
4128	cube_yellow	0.9277	342	prism_yellow	0.9611
5718	cube_gray	0.8928	423	prism_gray	0.8782
1125	cube_cyan	0.9514	381	prism_cyan	0.9585
3783	cube_blue	0.9367	276	prism_blue	0.7556
mean average precision					0.9130

Chapter 7

Experiment 3: FusionNet

The FusionNet is the cornerstone of this Thesis. Its ability to learn a good representation based on the inputs branching from the RetinaNet and the Transformer has a great influence on the final output. We implement the FusionNet as a simple feed-forward network to combine the nodes branching from the preceding networks.

In section 4.3 we describe the FusionNet as a neural architecture relying solely on the subnetworks branching from both modules. The number of nodes branching from both modules is denoted by N_n. The final layer creates an aggregated representation of visual and linguistic features. The number of units for the output layer is denoted by l_{output}. The output layer has a sigmoid activation, producing the output joint angles and states of the robotic arm and gripper respectively. We do not introduce any layers between the output layer and the concatenated features from the modules to restrict the scope of our problem. Creating a deep network composed of several layers might result in better learning. However, we are mainly interested in observing the influence of intermediate representations on our neural model, rather than adjusting the network for optimal performance. On the other hand, we are also concerned with optimizing the network to a certain degree that enables the network to perform the targeted task: grasping objects and placing them elsewhere.

Based on the experiments conducted in chapter 5 and chapter 6, we acquired the optimized hyperparameters for the Transformer as well as the RetinaNet. We summarize the chosen hyperparameters for the Transformer in table 7.1 and the RetinaNet in table 7.2.

Table 7.1: The optimized Transformer network hyperparameters

P_{drop}	N_h	N_m	d_a	d_{emb}	d_{ff}	l_{seq}	$w2v_{mod}$
0.15	4	4	12	50	256	200	sg

For the FusionNet **base model**, we use the synthesized datasets described in chapter 3. We follow the training pipeline described in section 4.3.2, where one of four (clean and noised) images chosen at random is fed as input to the vision module, while one of four (clean and noised) sequences chosen at random is fed

Table 7.2: The optimized RetinaNet hyperparameters

anchors	l_{res}	*backbone*
number: 9 **sizes:** {30,53,90,100,150} **strides:** {8,16,32,64,128} **ratios:** {17:20, 1:1, 11:10} **scales:** $\{2^0, 2^{\frac{1}{3}}, 2^{\frac{2}{3}}\}$	990×540	ResNet152

as input to the language translation module. The output values are scaled using min-max normalization (described in section 4.3.2) limiting the joint angles and their states to a range of 0 and 1.

Other hyperparameters describing the training procedure include the number of training iterations, the batch size, and the optimization algorithm as well as its properties. The number of training iterations is denoted by N_{itr}, and the batch size is denoted by s_{batch}. The optimizer is denoted by *opt* with subscripts indicating its properties where necessary. We schedule the learning rate opt_η to reduce on the plateau of the FusionNet's Mean Squared Error (MSE) loss. The learning rate is reduced by a factor of 10 and checked every $N_{itr} = 1k$. If the loss has not imroved for two checks in a row, the learning rate is reduced.

We summarize the hyperparameters for the FusionNet base model in table 7.3.

Table 7.3: The base FusionNet hyperparameters

N_{itr}	s_{batch}	*opt*	opt_{β_1}	opt_{β_2}	opt_ϵ	opt_η	N_n	l_{output}
$32k$	1	*adam*	0.9	0.999	10^{-9}	10^{-5}	4	14

For the experiments to follow, the hyperparameters are assumed to default to the base model's properties unless otherwise stated. For a detailed description of the network and the dataset properties, refer to section 4.3.

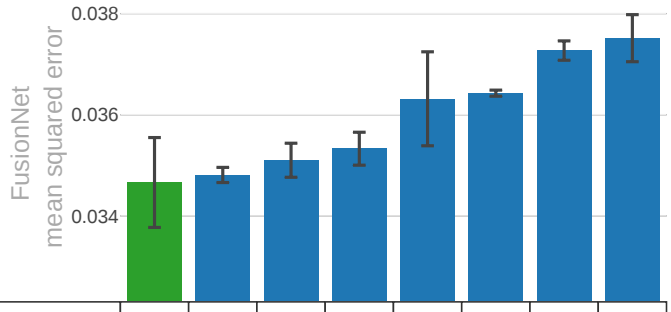
7.1 Ablation study

For this set of experiments, we perform an ablation study on the intermediate representations of the FusionNet. The introduction of different outputs as auxiliary tasks is hypothesized to improve the overall learning of all tasks as described in section 2.1.12. We remove the intermediate representations to explore whether they influenced the overall performance, specifically the precision of the joints angles predicted by the network.

7.1.1 Experimental setup

We apply the properties defined for the FusionNet base model. We ablate each output branching from the modules independently and in combination with other outputs. The FusionNet base model has four output layers, three of which are intermediate representations. The output layer producing the joint angles cannot be removed since it resolves the main objective of our task. The remaining output layers are the Transformer classifier for decoding the translated sequence words, the RetinaNet classifier for identifying the objects in the image, and finally the RetinaNet regressor for localizing the objects. We experiment with all combinations of ablated outputs resulting in a total of 8 combinations. Each combination of outputs was repeated three times. We ran the experiments for 3 days on an Nvidia TITAN X graphics processing unit.

7.1.2 Results and discussion



FusionNet regression (MSE)	**0.0347***	0.0348	0.0351	0.0353	0.0363	0.0364	0.0373	0.0375
Transformer classification (CE)	✖	✖	1.9175	✖	2.4352	✖	2.6328	2.6619
RetinaNet classification (FL)	0.3139	✖	✖	0.3666	0.2968	✖	0.3696	✖
RetinaNet regression (L1)	✖	✖	✖	0.6912	✖	0.5995	0.7015	0.6683

Figure 7.1: The average FusionNet mean squared error for three repetitions per ablated loss

The results of the ablation experiment indicate that learning all the weights in concert increase (worsen) the MSE. Although multi-task learning is known to improve the overall performance (refer to section 2.1.12), we observe the opposite. Generally, this is an expected outcome when dealing with different loss functions

since they operate on different scales. The overall loss of the neural network is the summation of all losses combined. A loss function producing an error which is significantly larger relative to other loss functions skews the gradients towards improving the most erroneous objective.

Since all active losses are scaled by a factor of 1 and all inactive losses have a loss weight of 0, we assume all losses to result in proportionate errors in this experiment. The active losses are indicated by the mean of their error as shown in figure 7.1. Inactive losses are indicated by an ✖ symbol.

The L1 loss for the RetinaNet regression appears to be the most contributing culprit to the worsening of the MSE loss used for performing the FusionNet regression. Discarding the L1 loss would be a viable option. However, we would also lose any meaningful representation deduced by the RetinaNet object classifier. The RetinaNet classification subnetwork would act as a regularizer without producing interpretable outcomes.

The Transformer classification loss appears to have a negative influence on the final objective. We cannot, however, discard the intermediate representation of the Transformer. Considering that the Transformer is an auto-regressive model, the decoded outputs are fed sequentially to the decoder as input during inference. The model would be rendered unusable during inference without the Transformer output layer given the nature of such an architecture. It is still possible, however, to discard the Transformer output under the condition that RCL annotations are parsed externally from the English language commands. Such non-trivial contributor to the neural model's functionality should not be preprocessed; else the introduction of RCL to the training dataset is considered redundant. We would have to ensure that the RCL parser generates accurate trees before feeding them into the network, causing our model to lose its end-to-end capability.

We observe from figure 7.1 that maintaining the RetinaNet classification subnetwork with all other intermediate representations removed results in the lowest (best) MSE with a mean of 0.0347 and a standard deviation of 0.0008 for all three repetitions of the same setup. The model excluding all intermediate representations achieves the second-lowest MSE with a mean of 0.0348 and a standard deviation of 0.0002 for all three repetitions of the same setup. The variance of the lowest scoring model is significantly higher than the model without intermediate representations, implying that given more repetition samples, the performance of the best model might change. The two best scoring models cannot be used for inference since they do not include the Transformer's intermediate representation. A model which can be used for inference, however, includes the Transformer's intermediate representation only and achieves the third-lowest MSE with a mean of 0.0351 and a standard deviation of 0.0004 for all three repetitions of the same setup.

7.2 Weighted Losses

For this set of experiments, we adjust the loss weights for all intermediate representations. Based on our discussion in section 7.1.2, we have hypothesized that the difference in loss scales caused a degradation in the results for the model with intermediate representations compared to the model without intermediate representations. To address this problem, we sought after altering the weights of each loss to acquire an optimal combination. The loss weights are constant factors by which each loss is multiplied before summing the various losses in a single model.

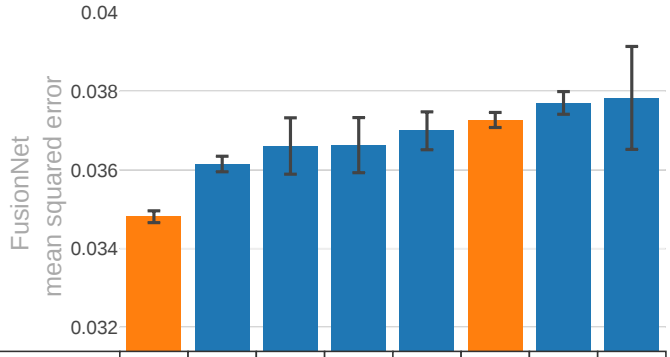
The choice of weights for the losses is challenging and setting them manually is unfeasible. We approach the loss weight adjustment as an optimization problem. Our goal is to minimize the MSE of the FusionNet given a set of weights. We would need to adjust the weights of all four loss functions before computing the final loss encompassing them. We use a Parzen tree estimator (PTE) [9] to optimize the four weights.

7.2.1 Experimental setup

We apply the properties defined for the FusionNet base model. The PTE for hyperparameter optimization receives the FusionNet MSE loss, the RetinaNet L1 loss, the RetinaNet Focal loss, and the Transformer CE loss as the priors. The PTE runs the experiment and receives the FusionNet MSE as the evidence, generating the next posteriors in the form of weights for each of the loss. We limit the range of all four weights to a minimum of 0 (the loss is excluded from the training) and a maximum of 10. The optimization process was repeated for 35 trials. Each combination of weights was repeated three times. We ran the experiments for 14 days on an Nvidia GTX 1050 Ti graphics processing unit.

7.2.2 Results and discussion

We observed that several weight combinations resulting from the PTE optimization outperform the model having all intermediate representations. As shown in figure 7.2, the best scoring model resulting from the weight adjustment process achieved a mean MSE of 0.0361 and a standard deviation of 0.0002 for all repetitions of the same setup. The best scoring model did not outperform the model without intermediate representations. Since the loss weights for the experiment conducted in section 7.1 were binary (0 when a loss was ablated and 1 otherwise), we can directly compare the weights from that experiment to the current. The best scoring model from section 7.1.2, has loss weight proportions similar to the best scoring model in this experiment. This indicates the PTE hyperparameter optimizer was able to find an ideal combination of loss weights. Discarding the intermediate representation losses entirely is not feasible for our model, mainly due to the Transformer's autoregressive design. Altering the weights allows for a more lenient compromise: we do not need to discard losses which have a negative influence. Instead, we can reduce their contribution to the overall loss.



FusionNet regression weight	**1.00***	6.82	5.86	6.45	7.27	1.00	1.88	5.20
Transformer classification weight	0.00	0.23	5.39	4.56	6.36	1.00	9.93	2.12
RetinaNet classification weight	0.00	5.71	8.18	2.92	5.80	1.00	1.80	4.68
RetinaNet regression weight	0.00	0.38	1.34	8.51	9.54	1.00	7.29	8.07

Figure 7.2: The average FusionNet mean squared error for three repetitions per weighted loss

Although the optimization of the weight losses does witness an overall improvement over the uniformly weighted model with all intermediate representations, it still does not achieve the desired outcome of outperforming the model without intermediate representations. Destructive interference, a term coined by Zhao et al. [110] describe a problem where different tasks drive the optimizer's gradients in opposing directions. We can deduce from the results shown in figure 7.1 and figure 7.2 that the Transformer classification loss and the RetinaNet regression loss are acting against each other. The destructive interference could be a result of the abstraction brought by the Transformer since it reduces the English language to RCL commands, as opposed to the RetinaNet which provides specific locations of all blocks in the image, including those which do not contribute to the overall task.

7.3 Model Simplification

For this set of experiments, we observe the influence of having different intermediate representations and modules. By reducing the vision module to a convolutional model (section 2.1.7) instead of the deep RetinaNet which is currently used, we examine whether a complex object detection architecture was necessary for the functionality of the FusionNet. We implement a simple Convolutional

Neural Network (CNN) composed of two-dimensional convolutional layers. The structure of the CNN used for our experiment follows a CNN designed by Kerzel and Wermter [50]. The first convolutional layer has 16 filters with a kernel size of 3×3 and a ReLU activation. A second convolutional layer is added with similar properties with the only difference being the kernel size, which is 4×4 instead. A maximum pooling layer follows each of the two convolutional layers. The weights are randomly initialized, sampled from a uniform distribution and scaled using the method proposed by Glorot et al. [31]. We eventually reshape the output to the size of the FusionNet nodes. Note that the intermediate representations branching from the vision module are no longer available. We can fairly compare the performance of this network with the base FusionNet model having the two intermediate representations belonging to the RetinaNet removed. For the language related task, we do not replace the Transformer with any other network. Instead, we set the target sequence to the English language, matching the source sequence. This approach turns the Transformer into an autoencoder. Translating to the same language is a simplified task as opposed to translating from English to RCL, and should provide us with an indicator as to whether RCL as an intermediate representation was necessary for our primary task.

7.3.1 Experimental setup

We apply the properties defined for the FusionNet base model. For one set of experiments, we replace the RetinaNet with the simple CNN and compare the FusionNet MSE resulting from the alteration. The RetinaNet regression and classification intermediate representations are discarded from the base FusionNet model to create a fair comparison. For another set of experiments, we compare the FusionNet performance having English language commands as target sequences with RCL as target sequences. Each setup was repeated three times. We ran the experiments for 2 days on an Nvidia GTX 1050 Ti graphics processing unit.

7.3.2 Results and discussion

In figure 7.3, we observed that the FusionNet with a CNN as a visual module achieves an MSE with a mean of 0.0356 and a standard deviation of 0.004. Compared to our previous results, the RetinaNet variant with intermediate representations excluded achieves a mean of 0.0351 and a standard deviation of 0.00136. We conclude that the RetinaNet outperforms the CNN variant. Although the FusionNet base model outperforms the CNN variant, we have to also consider the high computational overhead introduced by the RetinaNet. Optimizing the CNN might reap better results, but our experiments in this domain were not comprehensive since it goes beyond our research question.

In figure 7.4 we show a comparison between having English and RCL as a target sequence. For the full FusionNet with all losses uniformly weighted, we acquire an MSE of 0.0376 ± 0.00415 for the English language as a target and 0.0372 ± 0.00191

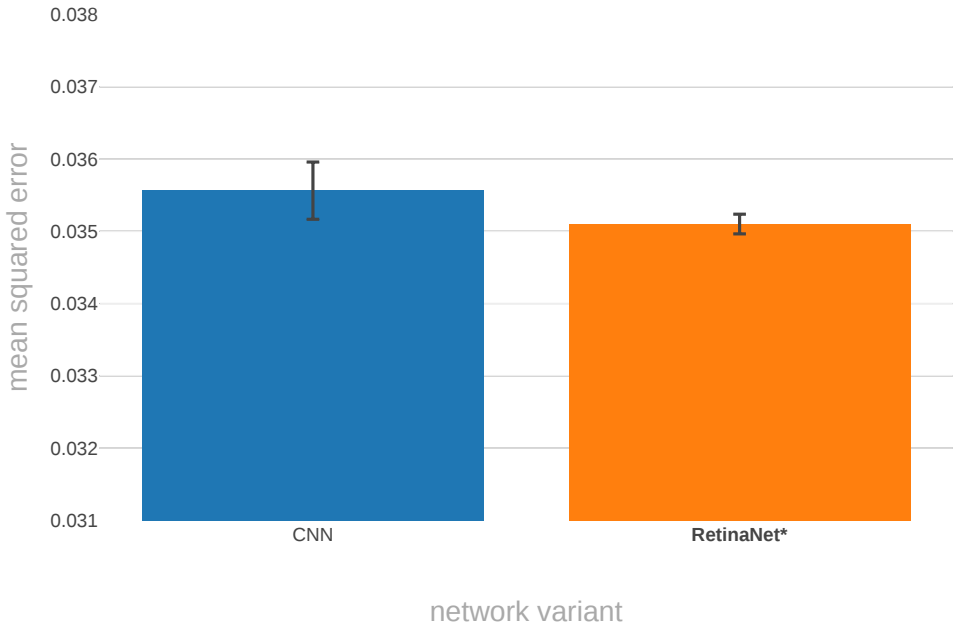


Figure 7.3: The average FusionNet mean squared error for three repetitions of two different visual modules

for RCL as a target over all repetitions. We experimented again with the Transformer loss excluded and acquired an MSE of 0.0374 ± 0.00163 for the English language as a target, and 0.035 ± 0.0007 for RCL as a target over all repetitions. We observe that RCL outperforms English as a target for the two variants. We hypothesize that RCL provides a suitable intermediate representation of language for our purpose, adding structure to an otherwise unstructured language. RCL sequences follow an organized, repetitive pattern, which we hypothesize to be the reason behind the improved performance since fewer features are needed to distinguish between sequences.

7.4 Nodes

For this set of experiments, we explore the influence of varying the number of nodes on the network performance. By increasing the number of nodes, we do not add more layers of abstraction, instead, we widen the layers connecting the vision and language translation modules, expanding the size of the bottleneck between the FusionNet and its modalities.

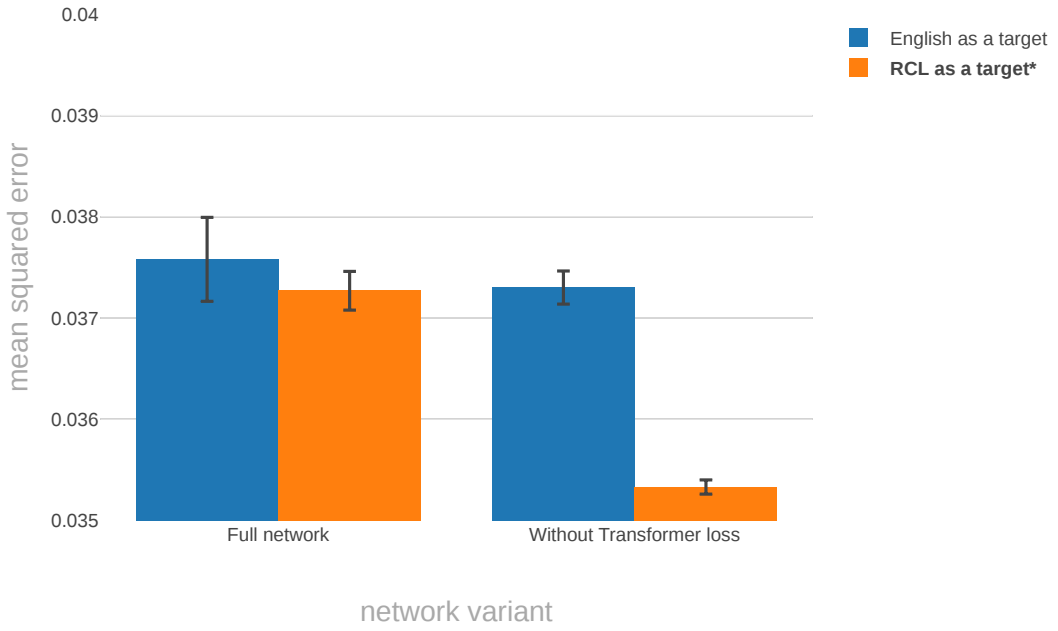


Figure 7.4: The average FusionNet mean squared error for three repetitions per translation target sequence

7.4.1 Experimental setup

We use the properties defined for the FusionNet base model. We examine the influence of N_m by changing its value between $\{4, 6, 8, 16, 20, 25, 32\}$. All intermediate representation losses were uniformly weighted. The experiment was repeated three times for each setup. We ran the experiments for 9 days on an Nvidia GTX 1050 Ti graphics processing unit.

7.4.2 Results and discussion

In figure 7.5, we show the different nodes and their resulting MSE. With 16 nodes connecting the modules to the FusionNet, the model achieves the lowest (best) MSE with a mean of 0.0363 and a standard deviation of 0.0006. The size of the network increases significantly since we connect more units to the FusionNet, and were unable to exceed the 32 nodes. We notice however that the MSE worsens as we increase the nodes beyond 16, hence increasing the number of nodes even further is unnecessary. Having a large number of nodes encourages the neural network to overfit, causing the validation loss to increase when too many units are available.

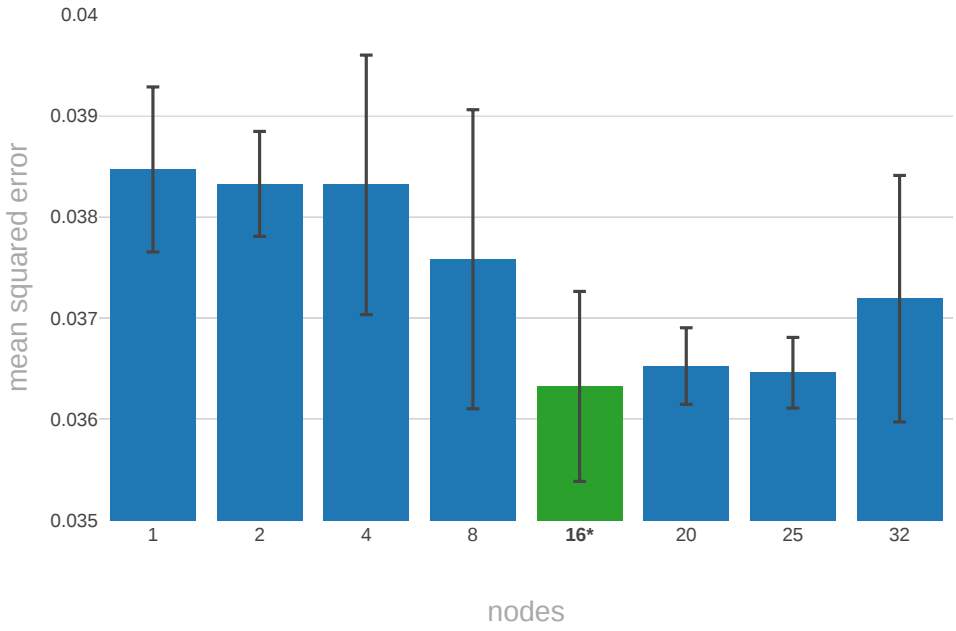


Figure 7.5: The average FusionNet mean squared error for three repetitions per node number

7.5 Analysis

We trained the FusionNet with hyperparameters matching those of the best models as concluded through the experiments described in this chapter. Based on our observations from section 7.2.1, we set the FusionNet regression weight to 6.82, the Transformer classification weight to 0.23, the RetinaNet classification weight to 5.71, and the RetinaNet regression weight to 0.38. According to the experiments performed in section 7.4.2, we set $N_m = 16$. Since the Transformer output is crucial for the functionality of the model, it would take a prohibitively long time to achieve any results during inference given the low weight assigned to the Transformer loss. We overcome this issue by transferring the weights from the model trained in section 5.4 to the Transformer model within the FusionNet. We trained the model for $60k$ iterations with a batch size of 1. All other hyperparameters matched the FusionNet base model's hyperparameters.

In table 7.4 we observe that the RetinaNet performs significantly worse than in section 6.3. This is mainly due to the low loss weight assigned to it, slowing down the learning. By running the experiment for more iterations, the RetinaNet is likely to achieve better results, however, we might risk overfitting on other tasks.

We tested the model on the simulated images extracted through the method described in section 3.3.3. We observe that the model has not yet converged,

Table 7.4: Results of the optimized FusionNet

Output	**Train**	**Valid.**	**Test**
FusionNet regression (MSE)	0.0286	0.0270	0.0276
RetinaNet regression (L1)	0.9291	0.6135	1.9374
RetinaNet classification (FL)	0.1961	0.3091	1.0649
Transformer classification (CE)	0.0589	0.0757	0.0789
RetinaNet (mAP)	-	0.3654	0.0759
Transformer accuracy (%)	0.9791	0.9741	0.9726

given that the training MSE loss is still higher in value than both the testing and validation. Due to the prohibitively long training time, we were unable to conduct the experiment until the model converged. We resumed to measure the accuracy of the model on the simulator. The robot's arm was about 10° away from the targeted joint angles. The initial joint angles were approximated by a 3.45° precision whereas the final joints were significantly more erroneous being 16.73° off target on all 485 scenes. We point out that the final joint angle prediction is a much harder problem to solve, since the robot has no visual input to rely on and has to predict the next action solely based on the command and the initial image.

Chapter 8

Conclusion and Future Work

In this Thesis, we presented a tailored synthetic task for examining the influence of intermediate representations on a multimodal neural network. The task involved commanding a robot to grasp objects and relocate them using natural language. The robot would receive two inputs, one being the image with the objects of interest displayed on a table, the other being the natural language command. We expanded the Extend Train Robots (ETR) dataset by creating augmented images with computer-generated blocks based on the properties provided by the ETR. We also designed a simulator for injecting missing information related to the mechanical movement of the robot.

To examine whether intermediate representations are necessary for performing the task, we designed a novel neural architecture combining the RetinaNet and the Transformer. The RetinaNet was used for detecting objects found in the images, whereas the Transformer was used to translate the natural language command to a tree-structured language know as the Robot Command Language (RCL). The outputs branching from these networks form what we refer to as intermediate representations. We combined the two networks through intermediate fusion by merging their representations to learn motor actions. We began our study by optimizing the Transformer and the RetinaNet in order to avoid errors accumulating due to suboptimal design decisions. We found both the RetinaNet and the Transformer to perform the tasks at hand with agreeable precision as per our quantitative and qualitative analysis. Next, we examined the necessity of the three intermediate representations, two of which are associated with the RetinaNet classification and regression branches for detecting blocks in images along with their location. The last intermediate representation is associated with the Transformer classification branch for decoding RCL commands given natural language.

On ablating all combinations of intermediate representations, we inferred that the neural network without intermediate representations outperformed the neural network with all intermediate representations employed. We hypothesized that the reduction in performance was due to the varying scales of the errors associated with the losses belonging to the different intermediate representations. To counteract this negative influence, we employed a Bayesian-based hyperparameter optimizer to weigh the different losses in the model for harvesting less erroneous

results when achieving our main motor-related task. We were able to achieve better results with weighted losses compared to the unweighted network with all intermediate representations. The model with the best combination of weighted losses, however, did not outperform the model without intermediate representations. Although the model without intermediate representations outperformed a majority of the intermediate representation combinations, we observed that the Focal loss used for the RetinaNet classification has a positive influence on the fusion network's performance. The Transformer's cross-entropy loss tends to reduce the model's precision as well. However, we cannot discard the Transformer's output. Considering that the Transformer is an autoregressive model, discarding the decoded RCL annotations would render our fusion model unusable for inference.

To examine whether our choice of RCL as an intermediate representation was justifiable, we set the Transformer to translate natural language sequences to themselves. This simple modification alters the objective of the Transformer, allowing it to act as an autoencoder. We observed that although decoding the sequences to themselves is a more straightforward task than translating to a different language, RCL as output improves the performance of our fusion network. This indicates that RCL as an intermediate representation was well suited for the task as opposed to merely modeling the natural language commands. We also examined whether a much simpler network instead of the RetinaNet would reap better outcomes. A simple convolutional neural network did not outperform the sophisticated RetinaNet model when performing our targeted task. We observed that the number of nodes connecting the RetinaNet and the Transformer to our fusion network plays a role in affecting the model's performance. Increasing the nodes implies a widening in the network, and consequently, an expansion to the error surface. Although increasing the number of nodes should allow more features to be detected, the possibility of reaching an optimal solution reduces significantly. We increased the nodes and observed a proportional improvement in the results up to a certain point until the results degraded as we exceeded a given number of nodes.

Based on the optimization steps followed, we trained the model using the best hyperparameters acquired. We tested the model in the simulated environment and trained on real images with augmented blocks atop a virtual surface. We observed that a majority of the failed grasps were resulting from the final motor targets. The initial motor targets were approximated more accurately during inference than the final targets. We hypothesized that the failure was due to the model being unaware of the changes between the initial and final steps. Since our model does not support any form of servoing, the robot is oblivious to the changes resulting from any environmental influences. Adding servoing capabilities, such as streaming images during grasping and updating the joint coordinates of a robotic arm could improve the precision of our model. Other approaches involving reinforcement learning might need to be integrated with our model for achieving such a goal. Another interesting aspect to address is the choice of intermediate representations. We have not explored whether intermediate representations for learning different tasks are more beneficial to our network. Although we have shown that RCL as an intermediate representation was more suited than natural language as an

intermediate representation for our task, we have not examined other structured languages. A simpler or more abstract description of the features of interest in the natural language commands might result in improved performance.

Expanding the fusion network to support more layers could result in a lower error, hence would be a viable option to explore. Attention between the features resulting from the vision and translation modalities could also improve the results significantly. The classification output resulting from the RetinaNet's intermediate representation describes the entities resulting from the Transformer's intermediate representation. By attending to the similarities between the two features, the model might infer patterns suited for focusing on the objects of interest only. We have observed that our approach in weighing the losses presented an improvement over uniformly weighted losses. This rather naïve approach relies heavily on the weights of the losses involved without considering the correlations between the tasks. A dynamic approach such that the homoscedastic uncertainty between the tasks might be well suited for our objective, and will be explored in future work.

Appendix A

Experiment 3: Additional Data

Table A.1: Weighted loss experiment results showing the mean and standard deviation of three trials per weight combination

RetinaNet regression weight	RetinaNet classification weight	Transformer classification weight	FusionNet regression weight	FusionNet train. loss (MSE)	FusionNet valid. loss (MSE)
1.0	1.0	1.0	1.0	0.0068 ± 0.0024	0.0373 ± 0.0002
8.86	9.23	6.63	8.59	0.0080 ± 0.0012	0.0370 ± 0.0005
9.54	5.8	6.36	7.27	0.0094 ± 0.0027	0.0370 ± 0.0005
1.34	8.18	5.39	5.86	0.0084 ± 0.0025	0.0366 ± 0.0007
9.11	0.18	2.4	9.41	0.0109 ± 0.0060	0.0376 ± 0.0002
7.61	6.95	6.55	8.49	0.0102 ± 0.0029	0.0378 ± 0.0005
6.54	7.72	6.09	9.81	0.0101 ± 0.0025	0.0374 ± 0.0006
4.15	2.11	3.49	5.54	0.0081 ± 0.0079	0.0374 ± 0.0007
9.35	8.2	5.9	6.93	0.0080 ± 0.0012	0.0375 ± 0.0002
9.61	5.95	4.39	2.37	0.0127 ± 0.0036	0.0377 ± 0.0008
7.81	7.14	1.38	5.32	0.0101 ± 0.0023	0.0371 ± 0.0004
7.29	1.8	9.93	1.88	0.0083 ± 0.0056	0.0377 ± 0.0003
8.54	6.27	3.09	7.8	0.0083 ± 0.0027	0.0379 ± 0.0006
9.89	7.76	8.42	8.02	0.0099 ± 0.0008	0.0378 ± 0.0005
5.4	8.34	6.57	6.37	0.0106 ± 0.0037	0.0376 ± 0.0012
0.65	8.88	5.02	7.25	0.0061 ± 0.0027	0.0365 ± 0.0003
5.75	4.22	8.5	7.27	0.0093 ± 0.0051	0.0375 ± 0.0009
6.26	6.84	4.1	8.06	0.0076 ± 0.0014	0.0378 ± 0.0010
8.07	4.68	2.12	5.2	0.0107 ± 0.0041	0.0378 ± 0.0013
0.0	0.0	0.0	1.0	0.0041 ± 0.0031	0.0348 ± 0.0002
9.98	4.9	9.04	4.49	0.0083 ± 0.0042	0.0377 ± 0.0002
2.21	4.55	3.71	6.09	0.0082 ± 0.0022	0.0364 ± 0.0007
7.1	3.64	9.21	8.57	0.0090 ± 0.0013	0.0375 ± 0.0004
6.07	5.41	7.29	4.14	0.0074 ± 0.0032	0.0379 ± 0.0005
8.51	2.92	4.56	6.45	0.0082 ± 0.0005	0.0366 ± 0.0007
4.75	9.8	7.93	3.18	0.0073 ± 0.0024	0.0380 ± 0.0006
5.59	5.36	7.69	10.0	0.0091 ± 0.0055	0.0366 ± 0.0002
8.14	2.66	8.33	3.91	0.0091 ± 0.0048	0.0372 ± 0.0013
9.18	0.85	9.72	0.89	0.0095 ± 0.0052	0.0375 ± 0.0005
10.0	3.81	3.93	4.7	0.0096 ± 0.0050	0.0380 ± 0.0007
0.38	5.71	0.23	6.82	0.0070 ± 0.0027	0.0361 ± 0.0002
8.6	7.58	4.64	8.88	0.0091 ± 0.0040	0.0375 ± 0.0009
7.02	6.28	5.31	7.57	0.0073 ± 0.0027	0.0367 ± 0.0006
3.74	6.45	6.97	6.66	0.0068 ± 0.0041	0.0371 ± 0.0003
6.06	8.45	3.82	2.63	0.0025 ± 0.0000	0.0367 ± 0.0000

Bibliography

[1] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. Character-level language modeling with deeper self-attention. *arXiv preprint arXiv:1808.04444*, 2018.

[2] Robert O Ambrose, Robert T Savely, S Michael Goza, Philip Strawser, Myron A Diftler, Ivan Spain, and Nicolaus Radford. Mobile manipulation using NASA's robonaut. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 2104–2109. IEEE, 2004.

[3] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques (SIGGRAPH)*, pages 23–34. ACM, 1994.

[7] Roberto Battiti and Anna Maria Colla. Democracy in neural nets: Voting schemes for classification. *Neural Networks*, 7(4):691–707, 1994.

[8] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1533–1544, 2013.

[9] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

[10] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1247–1250. ACM, 2008.

[11] G Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[12] Roberto Calandra, Andrew Owens, Dinesh Jayaraman, Justin Lin, Wenzhen Yuan, Jitendra Malik, Edward H Adelson, and Sergey Levine. More Than a Feeling: Learning to Grasp and Regrasp using Vision and Touch. *arXiv preprint arXiv:1805.11085*, 2018.

[13] Rich Caruana. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998.

[14] Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Niki Parmar, Mike Schuster, Zhifeng Chen, and Others. The Best of Both Worlds: Combining Recent Advances in Neural Machine Translation. *arXiv preprint arXiv:1804.09849*, 2018.

[15] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

[16] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2933–2941, 2014.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Kais Dukes. Contextual Semantic Parsing using Crowdsourced Spatial Descriptions. *arXiv preprint arXiv:1405.0145*, 2014.

[19] Kais Dukes. Semeval-2014 task 6: Supervised semantic parsing of robotic spatial commands. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, pages 45–53, 2014.

[20] Manfred Eppe, Tayfun Alpay, Fares Abawi, and Stefan Wermter. An Analysis of Subtask-Dependency in Robot Command Interpretation with Dilated CNNs. In *Proceedings of the 26th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*, pages 25–30, Bruges, Belgium, 2018.

[21] Manfred Eppe, Matthias Kerzel, Sascha Griffiths, Hwei Geok Ng, and Stefan Wermter. Combining deep learning for visuomotor coordination with object identification to realize a high-level interface for robot object-picking. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robotics (Humanoids)*, pages 612–617, 2017.

[22] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4397–4404. IEEE, 2015.

[23] Mark Everingham, Luc Van Gool, Christopher K I Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

[24] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[25] Charles J Fillmore and Collin F Baker. Frame Semantics for Text Understanding. In *Text*. Proceedings of WordNet and Other Lexical Resources Workshop, NAACL., 1993.

[26] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[27] Cheng-Yang Fu, Wei Liu, Ananth Ranga, Ambrish Tyagi, and Alexander C Berg. DSSD: Deconvolutional single shot detector. *arXiv preprint arXiv:1701.06659*, 2017.

[28] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle pointsonline stochastic gradient for tensor decomposition. In *Proceedings of the 28th Conference Conference on Learning Theory (COLT)*, pages 797–842. PMLR, 2015.

[29] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional Sequence to Sequence Learning. *arXiv preprint arXiv:1705.03122*, 2017.

[30] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448. IEEE, 2015.

[31] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256, 2010.

[32] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 315–323, 2011.

[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[34] Nico Görnitz, Christian Widmer, Georg Zeller, André Kahles, Gunnar Rätsch, and Sören Sonnenburg. Hierarchical multitask structured output learning for large-scale sequence segmentation. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2690–2698, 2011.

[35] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

[36] Spence Green, Daniel Cer, Kevin Reschke, Rob Voigt, John Bauer, Sida Wang, Natalia Silveira, Julia Neidert, and Christopher D Manning. Feature-rich phrase-based translation: Stanford Universitys submission to the WMT 2013 translation task. In *Proceedings of the Eighth Workshop on Statistical Machine Translation (WMT)*, pages 148–153. ACL, 2013.

[37] Lieve Hamers and Others. Similarity measures in scientometric research: The Jaccard index versus Salton's cosine formula. *Information Processing and Management*, 25(3):315–318, 1989.

[38] Christopher G Harris, Mike Stephens, and Others. A combined corner and edge detector. In *Alvay Vision Conference*. Citeseer, 1988.

[39] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, 2016.

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

[43] Jan Hosang. Analysis and improvement of the visual object detection pipeline. *Phd. Thesis. Saarland University, Germany*, 2017.

[44] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[45] Eric Jang, Sudheendra Vijaynarasimhan, Peter Pastor, Julian Ibarz, and Sergey Levine. End-to-End Learning of Semantic Grasping. *arXiv preprint arXiv:1707.01932*, 2017.

[46] Lukasz Kaiser, Aidan N Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit. One model to learn them all. *arXiv preprint arXiv:1706.05137*, 2017.

[47] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

[48] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7482–7491, 2018.

[49] Matthias Kerzel, Erik Strahl, Sven Magg, Nicolás Navarro-Guerrero, Stefan Heinrich, and Stefan Wermter. Neuro-inspired companion: A developmental humanoid robot platform for multimodal interaction. In *Proceedings of the IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 113–120, 2017.

[50] Matthias Kerzel and Stefan Wermter. Neural end-to-end self-learning of visuomotor skills by environment interaction. In *International Conference on Artificial Neural Networks (ICANN)*, pages 27–34. Springer, 2017.

[51] Nitish Shirish Keskar and Richard Socher. Improving generalization performance by switching from adam to sgd. *arXiv preprint arXiv:1712.07628*, 2017.

[52] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[53] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1137–1143, 1995.

[54] Iasonas Kokkinos. Ubernet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6129–6138, 2017.

[55] Dana Lahat, Tülay Adali, and Christian Jutten. Multimodal data fusion: an overview of methods, challenges, and prospects. *Proceedings of the IEEE*, 103(9):1449–1477, 2015.

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

[57] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[58] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–48. Springer, 2012.

[59] Chuan Li, René-Vinicio Sanchez, Grover Zurita, Mariela Cerrada, Diego Cabrera, and Rafael E Vásquez. Multimodal deep support vector classification with homologous features and its application to gearbox fault diagnosis. *Neurocomputing*, 168:119–127, 2015.

[60] Kuinian Li and Antony P Darby. Modelling a buffered impact damper system using a spring–damper model of impact. *Structural Control and Health Monitoring: The Official Journal of the International Association for Structural Control and Monitoring and of the European Association for the Control of Structures*, 16(3):287–302, 2009.

[61] Tsung-Yi Lin, Piotr Dollár, Ross B Girshick, Kaiming He, Bharath Hariharan, and Serge J Belongie. Feature Pyramid Networks for Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, 2017.

[62] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *arXiv preprint arXiv:1708.02002*, 2017.

[63] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 740–755. Springer, 2014.

[64] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 21–37. Springer, 2016.

[65] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[66] Tomáš Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[67] Tomáš Mikolov, Anoop Deoras, Stefan Kombrink, Lukáš Burget, and Jan Černocký. Empirical evaluation and combination of advanced language modeling techniques. In *Twelfth Annual Conference of the International Speech Communication Association (INTERSPEECH)*, 2011.

[68] Tomáš Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3111–3119, 2013.

[69] Andrew T Miller, Steffen Knoop, Henrik I Christensen, and Peter K Allen. Automatic grasp planning using shape primitives. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 1824–1829. IEEE, 2003.

[70] Brian Mirtich and John Canny. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 181—-ff. ACM, 1995.

[71] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Others. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2204–2212, 2014.

[72] Jonas Mockus. Bayesian Approach to Global Optimization and Application to Multiobjective and Constrained Problems. *J. Optim. Theory Appl.*, 70(1):157–172, 1991.

[73] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[74] Akio Namiki, Takashi Komuro, and Masatoshi Ishikawa. High-speed sensory–motor fusion for robotic grasping. *Measurement Science and Technology*, 13(11):1767, 2002.

[75] Julia Neidert, Sebastian Schuster, Spence Green, Kenneth Heafield, and Christopher Manning. Stanford Universitys submissions to the WMT 2014 translation task. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 150–156, 2014.

[76] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence O (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[77] Van-Duc Nguyen. Constructing force-closure grasps. *The International Journal of Robotics Research*, 7(3):3–16, 1988.

[78] Andrew Owens, Jiajun Wu, Josh H McDermott, William T Freeman, and Antonio Torralba. Ambient sound provides supervision for visual learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 801–816. Springer, 2016.

[79] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 311–318. ACM, 2002.

[80] German I Parisi, Pablo V A Barros, Di Fu, Sven Magg, Haiyan Wu, Xun Liu, and Stefan Wermter. A Neurorobotic Experiment for Crossmodal Conflict Resolution in Complex Environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018.

[81] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *arXiv preprint arXiv:1211.5063*, 2012.

[82] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[83] Thai-Hoang Pham and Phuong Le-Hong. End-to-end recurrent neural network models for vietnamese named entity recognition: Word-level vs. character-level. In *International Conference of the Pacific Association for Computational Linguistics (PACLING)*, pages 219–232. Springer, 2017.

[84] Lerrel Pinto and Abhinav Gupta. Learning to push by grasping: Using multiple tasks for effective learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2161–2168. IEEE, 2017.

[85] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999.

[86] Dhanesh Ramachandram and Graham W Taylor. Deep Multimodal Learning: A Survey on Recent Advances and Trends. *IEEE Signal Processing Magazine*, 34(6):96–108, 2017.

[87] Joseph Redmon and Ali Farhadi. YOLO9000: Better, Faster, Stronger. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.

[88] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 91–99, 2015.

[89] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[90] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

[91] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, and Others. Learning representations by back-propagating errors. *Cognitive Modeling*, 5(3):1, 1988.

[92] Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Malaysia; Pearson Education Limited,, 3rd edition, 2009.

[93] Subir Kumar Saha. Recursive dynamics algorithms for serial, parallel, and closed-chain multibody systems. In *Mechanics of Structures and Machines*, 2007.

[94] Ramon Sanabria and Florian Metze. Hierarchical Multi Task Learning With CTC. *arXiv preprint arXiv:1807.07104*, 3, 2018.

[95] Holger Schwenk. Continuous space language models. *Computer Speech & Language*, 21(3):492–518, 2007.

[96] Dave Shreiner. *OpenGL reference manual: The official reference document to OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

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

[98] Maria M Suarez-Alvarez, Duc-Truong Pham, Mikhail Y Prostov, and Yuriy I Prostov. Statistical approach to normalization of feature vectors and clustering of mixed datasets. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 468(2145):2630–2651, 2012.

[99] Richard Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pages 823–832, 1986.

[100] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5026–5033. IEEE, 2012.

[101] Roberto Togneri and J S Christopher. *Fundamentals of information theory and coding design*. CRC Press, 2003.

[102] Ludovic Trottier, Philippe Gigu, Brahim Chaib-draa, and Others. Parametric exponential linear unit for deep convolutional neural networks. In *Proceedings of the IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 207–214. IEEE, 2017.

[103] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008, 2017.

[104] Baiyang Wang and Diego Klabjan. Regularization for unsupervised deep neural nets. In *Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*, 2017.

[105] Daixin Wang, Peng Cui, Mingdong Ou, and Wenwu Zhu. Learning compact hash codes for multimodal representations using orthogonal deep structure. *IEEE Transactions on Multimedia*, 17(9):1404–1416, 2015.

[106] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 2048–2057, 2015.

[107] Luke Zettlemoyer and Michael Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007.

[108] Zhengyou Zhang. Parameter estimation techniques: A tutorial with application to conic fitting. *Image and Vision Computing*, 15(1):59–76, 1997.

[109] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22, 2000.

[110] Xiangyun Zhao, Haoxiang Li, Xiaohui Shen, Xiaodan Liang, and Ying Wu. A Modulation Module for Multi-task Learning with Applications in Image Retrieval. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 401–416, 2018.

[111] Yi-Tong Zhou and Rama Chellappa. Computation of optical flow using a neural network. In *Proceedings of the IEEE International Conference on Neural Networks (ICANN)*, volume 1998, pages 71–78. IEEE, 1988.

[112] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: many could be better than all. *Artificial Intelligence*, 137(1-2):239–263, 2002.

Erklärung der Urheberschaft

Ich versichere an Eides statt, dass ich die Masterarbeit (M.Sc. Thesis) im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum Unterschrift

Erklärung zur Veröffentlichung

Ich erkläre mein Einverständnis mit der Einstellung dieser Masterarbeit (M.Sc. Thesis) in den Bestand der Bibliothek.

Ort, Datum Unterschrift