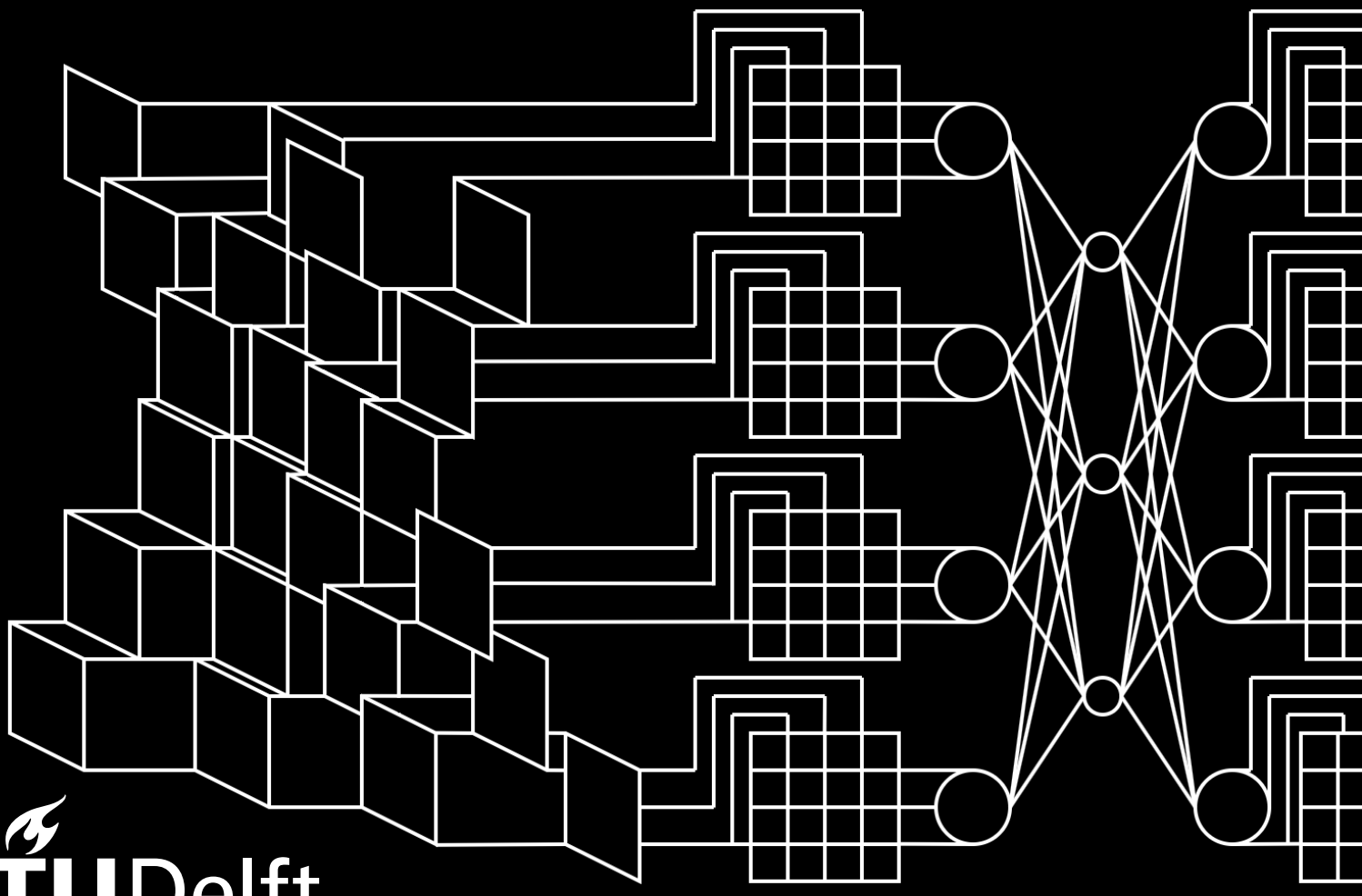
Towards Understanding RGB-Depth Pre-Training in ViT-based Models

An Exploration of a Novel Training Regime

Per Skullerud

Delft University of Technology



TU Delft

Towards Understanding RGB-Depth Pre-Training in ViT-based Models

An Exploration of a Novel Training Regime

by

Per Skullerud

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 2, 2026 at 14:00.

Student number:	5585643	
Project duration:	November 2025 – July 2026	
Thesis Committee:	Dr. J. van Gemert	TU Delft, thesis advisor
	Dr. P. Kellnhofer	TU Delft, core member
	P. Reijalt	TU Delft, daily supervisor

TU Delft

Preface

This thesis represents the culmination of my studies, totaling five years at TU Delft. These have been five years of opportunity, discovery, and growth not only as a Computer Scientist but also as a thinker. As the final step in my academic journey, this thesis is a statement of my preparedness for what is to come next.

Within Computer Science, Computer Vision has always been that discipline that has always felt like magic when it finally works. After completing my BSc. thesis in Vision, I was more than happy to continue down the same path with this MSc. thesis in the Pattern Recognition & Bioinformatics Section at TU Delft. I cannot thank my daily supervisor, P. Reijalt, enough for never lowering the standards for me, always guiding me towards the highest possible level of achievement. Of course, I extend a massive “thank you” to Dr. J. van Gemert for accepting me under his supervision, which ultimately defined my whole understanding of what it means to do research. Finally, I must thank my friends and parents, who kept my mind fixed on the prize as I was slowly realizing the consequences of the in-hindsight simple reality that producing high quality research is, in fact, quite hard.

And to whoever reads this thesis – I hope you find it interesting and insightful, and if you just so happen to learn something new, I consider my mission accomplished.

Per Skullerud
Delft, June 2026

Contents

1

Introduction

Computer Vision is the ability for computers to perceive visual media, such as images and videos. The ability to “see” the world has many practical applications: medical imaging, quality control in manufacturing, precision farming, etc. Most research and application focuses on images recorded in visible light wavelengths, saved in computer memory as RGB – Red, Green, and Blue channels. Broadly speaking, RGB images depict color and aim to capture the world in the same way as humans see it. Since humans’ possess a remarkable ability to perceive visuals, RGB is an obvious data type to feed into Computer Vision systems.

Figure 1.1: Some RGB + synthetic depth pairs, generated using DepthAnythingV2 [37]. Image adapted from the DepthAnythingV2 [37] paper.

Despite the demonstrated potentials of Computer Vision systems built for RGB, it is possible to further improve them by introducing more information via additional modalities. For example, one may provide images recorded at infrared light wavelengths to allow a Computer Vision system to “see” at low-light conditions, such as at night. In this thesis, we focus on the modality of depth – the distance of scene elements from the camera; examples in Figure 1.1. Depth is often encoded as an image, where each pixel conveys a notion of how far the camera is from the point in the 3D world that corresponds to that pixel. Intuitively, complementing RGB with depth (RGB-D) should help models with scene comprehension, as depth communicates the existence of the 3D world and how it is projected onto a 2D image. In practice, it has been shown that across a broad range of settings, using depth together with RGB improves the accuracy of models compared to RGB-only baselines.

In this thesis, we focus on using RGB-D with models derived from the Vision Transformer (ViT [10]). Architectures adapting this landmark model have produced phenomenal results across a wide range of Computer Vision problems; integrating a depth input into ViTs was a natural next step in RGB-D research. Several approaches have been proposed, introducing sophisticated depth fusion modules motivated by how the unique properties of depth could be used to elevate the behavior of the underlying ViT-based backbone. However, all models using this backbone must overcome a shared challenge – ViTs only outperform other techniques if trained using extensive data, but RGB-D datasets tend to be

small. In part, the scarcity of RGB-D data is caused by the fact that producing it requires specialized camera sensors. By contrast, large RGB datasets can be assembled by simply scraping the internet. For this reason, using ViTs is most accessible with RGB-only datasets.

Luckily, it is possible to leverage RGB-only data in the process of training powerful RGB-D models. The idea is to pre-train on RGB data, and then to fine-tune on RGB-D data. In other words, we can use RGB data to teach our model an understanding of the world that generalizes across tasks, and then transfer this understanding to a scenario where depth is available. The obvious problem is that going from not having depth to having it represents a significant domain gap, which the model may or may not be able to overcome. In this thesis, we investigate a recent solution to this problem – pre-training with "fake" pseudo-depth, i.e. synthetic depth that has been estimated from an RGB image by a specialized model. In this way, depth can be made available to complement images from large, otherwise RGB-only datasets. Although real depth should intuitively be "better" than estimated pseudo-depth, several works have nevertheless shown that it is promising to use pseudo-depth in pre-training. However, the full set of implications of this approach remains little understood.

Figure 1.2: Proposed toy problem, ultimately discarded to focus on pre-training. The RGB modality contains overlapping red or blue MNIST [18] digits, the depth modality explains the order of overlap. The problem is an XOR-like binary per-pixel classification task: class 1 is red even digits or blue odd digits; class 2 is red odd digits or blue even digits.

Originally, this thesis was not planned to be about pre-training. Instead, it was supposed to be about model architectures – the initial plan was to construct a toy dataset (controlled setting) that identifies some problem with an existing model and then to propose a fix. The toy problem/dataset we constructed required using depth to disentangle a stack of colored digits that confound each other in the RGB domain. One sample from this dataset is in Figure 1.2. The behavior of interest was whether a specific state-of-the-art model (DformerV2 [39]) could learn that different depth levels perfectly correspond to different objects; this toy setup isolates models' ability to correctly interpret occlusions in real-world settings. However, no matter how hard the toy dataset parameters were pushed, the same behavior was observed – when training from scratch, the model could not learn even the most basic patterns, but fine-tuning a pre-trained checkpoint consistently yielded accuracies so high that we could not replicate them even with models custom-built for the task. The conclusion appeared obvious – a large chunk of the success of ViT-based RGB-D models can be attributed to the pre-training process.

In the context of pseudo-depth pre-training, we are also interested in whether the complexity of architectures present in the literature is justified. In part, this interest arises from the fact that one current state-of-the-art model (DformerV2 [39]) uses depth in a superficial way, not encoding it into a latent space representation. The experiments presented in this thesis can be grouped into two parts – first, an investigation into how fake depth pre-training determines the representations learned by various models derived from the ViT, and whether the more complex models are justified versus the simpler ones. Second, we examine how well these pre-trained models, depending on their architecture, transition to fine-tuning on real depth. Overall, our goal is to present an overview of the consequences of pseudo-depth pre-training as an avenue towards better RGB-D models. We provide substantial insights that we hope will help future authors working in the field of RGB-D research.

Apart from this introduction, this thesis contains two more main parts. The first of these presents the subject-specific (not widely known) background knowledge required to understand the thesis. The second is the scientific paper, which is the core of this thesis. The scientific paper describes our research method, the results, and the conclusions of our work. This paper is a condensed version of the entire thesis, interpretable on its own by an expert in the field. The appendices include our model source codes, shared training code, and an AI usage statement.

2

Background

2.1. Depth

Depth is the notion of distance from the camera. When presented as a complementary modality to an RGB image, depth is a 1-channel image whose pixels denote the distance from the camera of the corresponding point in the imaged 3D scene. Intuitively, depth is an important supplement to the information provided in an RGB image, as depth communicates the existence of the 3D world and the rules by which it is projected onto a 2D image. We discuss the distinction between "real" depth, and estimated "fake" pseudo-depth in the context of their applicability to training RGB-Depth (RGB-D) Computer Vision models.

Figure 2.1: An RGB, real depth, and pseudo-depth image from the NYUDepthV2 [24] dataset. Real depth is more noisy, but has a more true-to-life value distribution. Semantic differences on the reflective surfaces are also visible; neither depth image is clearly superior.

Real depth loosely refers to reliable depth obtained at the time of the creation of the corresponding RGB image. For example, if creating synthetic RGB images using rendering engines such as Blender [5], the engine can additionally output depth. In this thesis, however, we do not work with synthetic RGB + depth pairs. Instead we only work with real depth recorded by specialized cameras, complimenting RGB photographs of the real world. Specifically, we use the NYUDepthV2 [24] dataset, which has RGB-D produced by the Xbox Kinect V1 [29] camera. This camera uses the structured-light [29] approach to sense distance. This works by projecting an infrared pattern onto the 3D world; depth is computed from distortions in the pattern caused by the scene's geometry, recorded by a camera at an alternate position. Depth images produced by the Kinect are noisy and otherwise imperfect, but have the advantage of denoting *absolute* depth, i.e. the distance in meters of each pixel from the camera (Figure 2.1). Intuitively, real depth should be more valuable than synthetic depth.

Synthetic depth refers to depth produced by a monocular estimator – a specialized model that estimates depth for each pixel of an input RGB image. Denoting distances in meters is beyond the scope of what synthetic depth could hope to achieve, since the parameters of the camera used to produce the input

RGB image are unknown. Instead, synthetic depth provides *relative* depth, i.e. for each pair of pixels which one of them is closer to the camera. In this thesis, we exclusively use the DepthAnythingV2 [37] monocular depth estimator. This model's success derives from a setup of initially training on a very large amount of synthetic RGB-D, then distilling the acquired knowledge to work with real RGB. Depth images produced by DepthAnythingV2 [37] feature low noise, well-defined edges and smooth gradients on surfaces as seen in Figure 2.1. However, the depth is relative as opposed to absolute, though the consequences of this difference are little understood.

If using real depth in a scenario where pseudo-depth is expected, the differences between the two domains should be minimized to ensure a fair comparison. For example, the raw output of DepthAnythingV2 [37] assigns higher values to nearby pixels, whereas the opposite is true for real depth in NYUDepthV2 [24] as its value denotes distance from the camera. It would be senseless to not account for this difference when evaluating the consequences of e.g. inputting real depth into a model trained using synthetic depth. A further common [38, 39] adjustment is to normalize both real and synthetic depth to have a mean of zero and a standard deviation of one. Naturally, these changes represent basic distribution alignment and do not account for most of the differences between the domains. Ultimately, under the assumption that real depth is more valuable, it is harmful to heavily modify either domain to achieve alignment as this would imply losing information in the real depth.

2.2. The Vision Transformer

In this section, we describe the building blocks of the Vision Transformer (ViT) [10] and the model itself. This model architecture has produced state-of-the-art results across a wide range of Computer Vision problems [15], and is therefore the focus of this thesis.

2.2.1. CNNs and RNNs

Both convolutional neural networks (CNNs) [19] and recurrent neural networks (RNNs) [27] should be familiar to the reader; we briefly summarize these techniques with an emphasis on how their limitations directed advancements finalizing in the ViT [10].

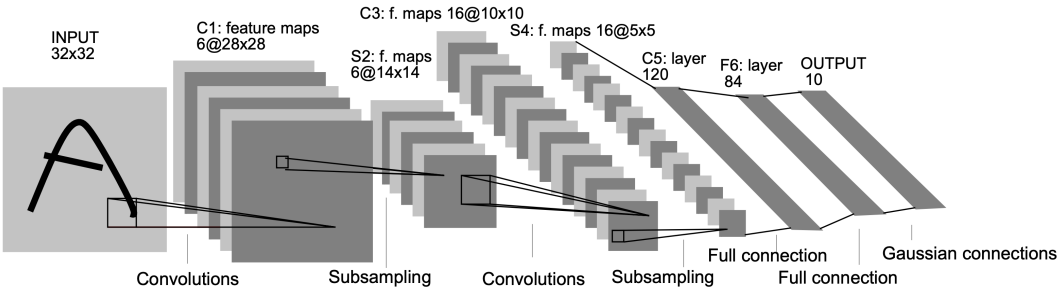


Figure 2.2: An example of a convolutional neural network (CNN): LeNet-5 [19], featuring interleaved convolutions and pooling layers, followed by an MLP. Illustration from [19].

A CNN [19] is a network that relies on the 2D convolution and max pooling operations to distill information from an input image. Convolution implies taking the weighted sum of each overlapping $k \times k$ patch in the image across several channels. Formally, for an odd kernel size k and ignoring border regions,

$$\text{Convolution2D}(\boldsymbol{X}, c_{\text{out}})_{i,j} = \sum_{c_{\text{in}} \in C(\boldsymbol{X})} \sum_{-\lfloor k/2 \rfloor \leq \Delta_i, \Delta_j \leq \lfloor k/2 \rfloor} \boldsymbol{W}^{c_{\text{in}}, c_{\text{out}}}_{\Delta_i, \Delta_j} \cdot \boldsymbol{X}_{i+\Delta_i, j+\Delta_j} \tag{2.1}$$

where $\boldsymbol{X}$ is the input image-like representation (feature-map), i, j are pixel indexes, $C(\boldsymbol{X})$ are the channels of $\boldsymbol{X}$, and $\boldsymbol{W}^{c_{\text{in}}, c_{\text{out}}}$ is a learnable weight parameter corresponding to the input and output channels c_{in} and c_{out}. A max pooling operation implies taking the maximum value from each non-overlapping $d \times d$ patch; there are no learnable parameters. At a high level, convolutions find patterns in images/feature maps, whereas max pooling shrinks the resolution of the feature maps by only keeping the most "relevant" feature. When interleaved multiple times as exemplified in Figure 2.2, these two operations extract information from local regions at various scales. This takes advantage of the broad

observation that natural images contain small-scale pieces that combine to define ever-larger objects [17]. The disadvantages of CNNs include an over-reliance on texture and an inability to perceive the entire scene at every step of the network [23].

RNNs are a class of neural networks designed to process sequences of arbitrary length, such as arbitrary texts. RNNs keep a hidden state that gets updated by processing the sequence elements one after another; the hidden state is the model's "memory" of past elements and the representation from which a task-specific output can be extracted. Abstractly, given sequence elements $X_1, X_2, ..., X_n$ and an initial hidden state H_0, the hidden state is updated according to some function f implementing $H_i = f(H_{i-1}, X_i)$. At each step, a task-specific output O can be obtained according to a function g implementing $O_i = g(H_i)$. RNNs struggle with gradients vanishing over long sequences, resulting in difficulties to prioritize relevant information in the hidden state [30]. Also, their recurrent component makes it difficult to distribute training across a large compute cluster [30]. The limitations of RNNs set the precedent for the *attention* [4] mechanism.

2.2.2. Self-Attention

The attention mechanism is a fundamental building block of (Vision) Transformers. Its purpose is to exchange information among a collection of latent space vectors, referred to as "tokens" [4]. We focus on the specific case of full self-attention, as other formulations are beyond the scope of what is relevant for this thesis. We describe full self-attention mathematically by starting with its input: a collection of tokens

$$\boldsymbol{X} \in \mathbb{R}^{N \times D} \tag{2.2}$$

where N is the token count and D is the hidden dimension size, i.e. the size of each latent space vector (token). $\boldsymbol{X}$ is then multiplied by query, key, and value matrices $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V \in \mathbb{R}^{D \times D}$ to yield

$$\boldsymbol{X}\boldsymbol{W}^Q = \boldsymbol{Q} \in \mathbb{R}^{N \times D}, \qquad \boldsymbol{X}\boldsymbol{W}^{\boldsymbol{K}} = \boldsymbol{K} \in \mathbb{R}^{N \times D}, \qquad \boldsymbol{X}\boldsymbol{W}^{\boldsymbol{V}} = \boldsymbol{V} \in \mathbb{R}^{N \times D} \tag{2.3}$$

. $\boldsymbol{K}$ is then multiplied with $\boldsymbol{Q}$ transposed and divided by $\sqrt{D}$ for numerical stability to form

$$\frac{\boldsymbol{K}\boldsymbol{Q}^\top}{\sqrt{D}} \in \mathbb{R}^{N \times N} \tag{2.4}$$

, an expression referred to as the *attention score*. In practice, a high entry in the $N \times N$ attention score matrix $\boldsymbol{S}_{i,j}$ means that token i strongly "wants" to receive information from token j. A softmax, expressed on a 1D vector x as

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\Sigma_j \exp(x_j)} \tag{2.5}$$

is then applied individually on each of the N vectors in the outer dimension of the attention score, resulting in the expression

$$\text{softmax}\left(\frac{\boldsymbol{K}\boldsymbol{Q}^\top}{\sqrt{D}}\right) \in \mathbb{R}^{N \times N} \tag{2.6}$$

. This expression is referred to as the *attention weight*, as each outer dimension vector sums to 1, effectively producing a distribution of how much each token j should influence the new value of token i. Finally, the attention weight is multiplied by V to produce the commonly used complete expression for self-attention, written by Vaswani et al. [35] as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{K}\boldsymbol{Q}^\top}{\sqrt{D}}\right)\boldsymbol{V} \in \mathbb{R}^{N \times D} \tag{2.7}$$

. Intuitively, $\boldsymbol{Q}$ "asks" how similar each token is to each other token, $\boldsymbol{K}$ answers that question, and $\boldsymbol{V}$ is the value (information) of each token that gets used in a weighted sum according to the attention weight to produce the new value of each token. Note that the output in Equation 2.7 has the same shape as the input in Equation 2.2, effectively maintaining the same number of tokens of the same dimension, but with their information exchanged among themselves.

In practice, it is advantageous to improve the formulation in Equation 2.7 by using multiple attention *heads*, a technique called multi-head attention [35]. This implies splitting the $N \times D$ input into H heads of shape $N \times (D/H)$, repeating steps 2.3 - 2.7 separately on each head, and finally concatenating the heads' outputs back into the original input shape. Using multiple heads allows attention to examine the input from multiple perspectives, each head considering a distinct interpretation of each token when determining how they should attend with each other.

Attention was originally formulated for text processing tasks, and it features many advantages in this setting. Notably, attention can process text of arbitrary length since the shape of the learnable parameters W is independent of the token count N, where N naturally grows with input text size [35]. This is vastly superior to previous approaches that encode arbitrarily long texts into fixed size representations [15]. Older methods derived from the RNN can reasonably process arbitrarily long sequences, but suffer from problems stemming from having to remember relevant information across long ranges in the sequence [6]. Attention, on the other hand, sees all tokens (i.e. the entire sequence) at all times and can prioritize whatever is relevant at every step.

2.2.3. The Transformer

The Transformer [35] is a model architecture built around attention; transformers have achieved incredible results across a vast range of Deep Learning tasks [20]. In particular, the transformer is credited as the main architectural innovation behind ChatGPT [1], a product whose release has contributed greatly to starting the subsequent AI boom. As it is largely irrelevant to Vision Transformers [10], we do not explain the entire original transformer formulation [35] designed for text-to-text tasks such as translation. Instead, we focus on the so-called decoder-only transformer backbone suited to the simple task of next-word prediction.

Next-word prediction implies considering all available words to predict the most likely next word. The process consists of three phases: embedding the input text into tokens, enriching those tokens via the transformer backbone, and decoding the resulting tokens into a probability distribution for the next token. We focus on the middle phase – using the transformer backbone to enrich and exchange information between tokens corresponding to individual words to obtain a global understanding of the text that would indicate the likely next words. Succinctly, the goal is to map the representations of individual words – tokens – into a representation of the entire text.

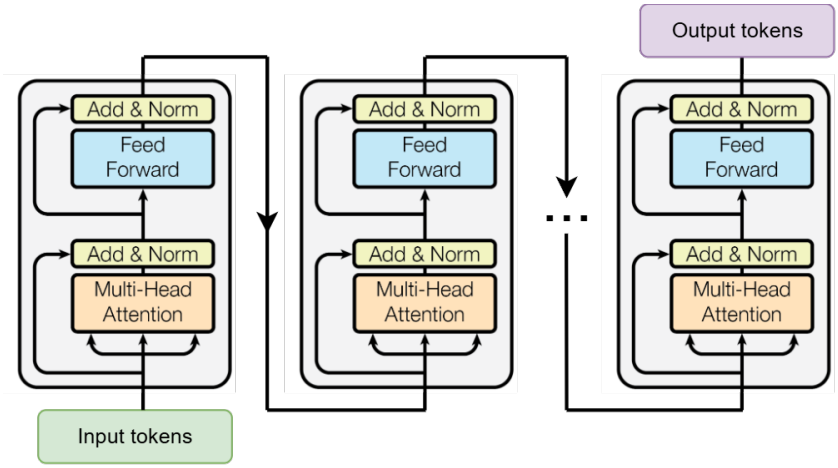


Figure 2.3: The backbone of a decoder-only transformer, featuring several blocks connected sequentially. Block diagram from [35].

Mathematically, the processing of tokens by a transformer backbone begins from an input $\boldsymbol{X} \in \mathbb{R}^{N \times D}$

of N tokens having dimension D each. This matrix $\boldsymbol{X}$ is then forwarded through a transformer *block*, which outputs a new matrix of the same dimensionality. Each transformer block partially accomplishes the task of changing the tokens from describing individual words to describing the entire text; the entire transformer backbone simply consists of multiple such blocks stacked sequentially, as seen in Figure 2.3. Each transformer block consists of the following steps:

1. Multi-Head Attention
2. Residual connection – adding the block's input
3. Layer normalization
4. Feed-forward MLP
5. Residual connection – adding the output of step 3
6. Layer normalization

We briefly explain the unintroduced concepts in the steps. Residual connections imply adding an earlier latent state to the current one, mitigating the vanishing gradient problem and increasing robustness [34]. Layer normalization [3] of a tensor x can be expressed as

$$x_{\text{norm}} = \frac{x - \mathsf{E}[x]}{\sqrt{\mathsf{Var}[x] + \epsilon}} \cdot \gamma + \beta \tag{2.8}$$

where $\epsilon \simeq 10^{-5}$ is a small positive number to prevent division by zero, γ and β are learnable parameters that increase flexibility. In the transformer block, layer normalization is applied individually on each token, preventing vanishing or exploding gradients. The MLP [17] can be described as $\phi : \mathbb{R}^D \to \mathbb{R}^D$; it is applied to each token individually, using one hidden layer and ReLU activations. While self-attention prioritizes the exchange of information between tokens, the MLP enhances individual tokens with "facts" the transformer has "memorized" [22].

The transformer architecture builds upon the many benefits of the attention mechanism. Like attention, transformers can operate on arbitrarily large sequences without involving a recurrence mechanism as used by RNNs. The ability to process the entire sequence at once implies great parallelization capabilities during training, which enables training transformers on large GPU clusters that have become increasingly accessible in recent years [20]. It coincides that the advantages of transformers are most visible at large scales – the architecture scales incredibly well with the amount of training data used [13]. Unlike other architectures, transformers converge to accuracies that grow with the amount of training data used without hitting a hard ceiling [13]. The data hunger of transformers stems partially from the absence of inductive priors describing how the input could be used – for example, RNNs are biased towards maintaining more information from recent sequence elements, whereas transformers process the entire sequence at once and do not explicitly assign importance based on tokens' ordering. The resulting increase in transformers' flexibility allows for learning intricate patterns if given enough data and training time [36]. These numerous advantages make transformer-derived architectures the go-to for modern large-scale models.

2.2.4. Repurposing Transformers for Vision - the Vision Transformer

The original transformer is defined for text processing tasks, but text is only present in the initial mapping to the latent space and the final mapping from it. Both of these mappings are small compared to the backbone that operates entirely on tokens; the advantages of the transformer stem mostly from the information exchange/enrichment that happens specifically in the backbone. Intuitively, one may repurpose the transformer for Computer Vision problems by implementing a task-appropriate method of mapping to/from tokens. This idea was implemented by the Vision Transformer (ViT) [10], achieving state-of-the-art accuracy in the widely studied ImageNet [9] classification task; future iterations have achieved remarkable results in a host of Computer Vision problems [15].

The original ViT [10] designed for ImageNet [9] classification encodes input image patches as tokens, forwards these tokens through the backbone, and decodes the tokens to obtain a classification prediction. The steps are illustrated in Figure 2.4. The initial tokenization is performed by splitting the input 224×224 image into a regular grid of 16×16 pixel patches, yielding 196 total patches. Each patch

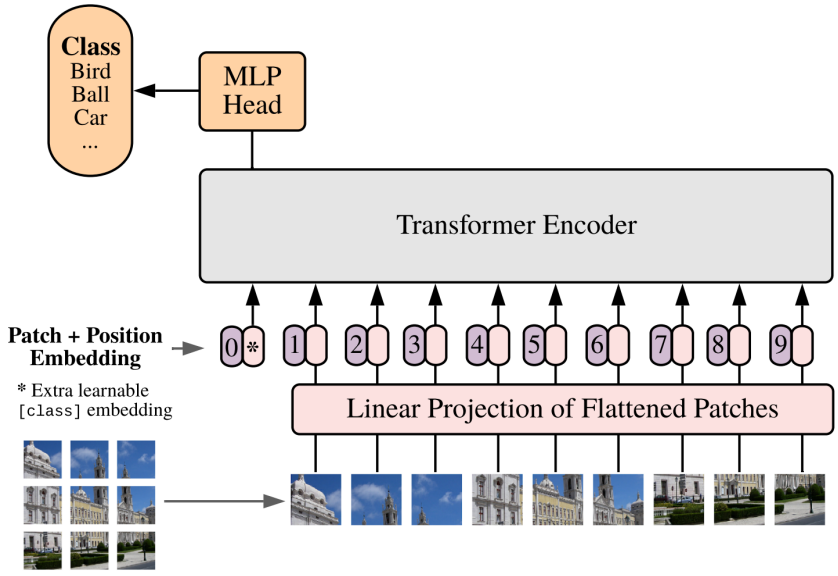


Figure 2.4: An illustration of the structure of the Vision Transformer (ViT), taken from [10].

(having shape $(3, 16, 16)$) is tokenized by being flattened and then linearly projected onto a vector, i.e. a token, of dimension 768. To maintain awareness of the patches' positions, a learnable spatial embedding is added. Since each token mirrors its respective image patch, a special 197th "classification" (CLS) token is then appended as a target to pool knowledge relevant to classification. The 197 tokens are passed through the backbone seen in Figure 2.3, having 12 layers and 12 heads in multi-head attention. Finally, an MLP is used to map the CLS token to a class probability vector. The formulation was improved by subsequent works, such as the Swin transformer [21] which introduced a hierarchical component to the backbone. Nevertheless, even the unrefined original ViT suffices to outperform earlier methods [10].

Despite the successes, the ViT, like other architectures derived from the transformer, demands strict conditions to perform optimally. Importantly, ViTs require extensive training data, caused to a large extent by ViTs' lack of inductive priors. Unlike CNNs, ViTs are not explicitly told that nearby scene elements should inform each other; similarly, ViTs are unaware of the hierarchical character of natural images [10]. Learning these ideas requires not only extensive data but also a long and carefully designed training process [32]. However, if given these resources, ViTs can learn representations that are more semantically rich than what is attainable by e.g. CNNs [8]. ViTs are at the forefront of Computer Vision, understanding their nature and addressing their limitations is a natural research direction with the goal of maximizing their likely yet-unreached potentials.

2.3. Pre-Training and Fine-Tuning

An important pillar of modern machine learning is the idea of high quality internal representations. Essentially, accurate models learn an understanding of the world that manifests itself in the presence of semantic concepts that are extractable from the internal states of models [33]. Even if not explicitly instructed to do so, powerful machine learning models do end up "modeling" the real world; it appears that this is a fundamental intermediate step towards solving any problem a model could be intended for. However, attaining a deep understanding of the world requires a large amount of training data, more than is often available when training models for specific tasks. Training with insufficient data naturally produces worse accuracy and an over-reliance on *shortcuts* [11] – simple cues that suggest the presence of a concept but distract from its fundamental identifying attributes, which, unlike the shortcuts, remain when generalizing to a broader domain. Even if extensive training data is available, fully leveraging it to train large models can be prohibitively expensive [1]. Due to the nature of the transformer, the mentioned problems are particularly relevant to Vision Transformer-based models.

Fortunately, an internal understanding of the world learned by a model trained on extensive data can

be re-used for other tasks. Deep representations learned on one task have been shown to generalize to other tasks even if they appear distinct [33]. Fundamentally, it appears that there is meaningful overlap between the abilities needed to solve a wide range of Computer Vision problems. Learning a broad understanding of the world by large-scale training on one task is referred to as *pre-training*; building task-specific ability upon the pre-trained foundation is referred to as *fine-tuning* [33]. In practical terms, one may download a powerful pre-trained model (checkpoint) from the internet and fine-tune it to a specific problem by continuing to train it on whatever task-specific data is available. Given Vision Transformers' lack of inductive biases, fine-tuning pre-trained checkpoints is a nearly unavoidable flow when seeking state-of-the-art results in niche problems characterized by limited data [8].

2.4. ViT Interpretability

To understand the conditions under which ViTs [10] do or do not work, meaningful strides have been made towards interpreting the internals of ViTs. One extensively proven [2] property is that shallow ViT layers focus on texture and spatial position, middle layers capture the scene's structure, and deep layers describe global meaning. Another relevant result is that, unlike CNNs, ViTs prioritize the structural cohesion of a scene as opposed to textures [23]. Regarding depth, it has been shown that a depth signal appears within the internal representations at later layers of a ViT even when it is trained purely using RGB [28]. Together, the results hint at one potential application of depth – to deduce scene structure. Scene structure's geometric interpretation may be most useful at earlier layers, but its semantic complexity is great enough that RGB-only models are only able to deduce it at later layers.

In the remainder of this section, we detail the mathematics of several techniques used in this thesis to interpret ViTs.

2.4.1. Linear Probing

Linear probing is a method to extract a signal from the internal representations of a model. This works by freezing the model's backbone, discarding the last few layers, appending a linear projection after the last exposed layer, and training this projection to predict the desired signal. The validation error obtained upon completion of this training process indicates the degree of presence of the desired signal along a linear direction in the latent space. Like [28], we use linear probing to identify the presence of depth information across various layers of ViTs. In particular, if an input image $\boldsymbol{X}$ of size 224×224 is divided into 196 16×16 patches and each patch is mapped to a token, we are interested in whether the average depth value of each patch can be extracted from the respective token. Formally, we aim to minimize

$$\text{MeanSquaredErrorLoss}(\boldsymbol{X}_d, f_{\text{model}}(\boldsymbol{X})\boldsymbol{P}) \tag{2.9}$$

where $\boldsymbol{X}_d \in \mathbb{R}^{196\times 1}$ is the average depth of each patch, $f_{\text{model}}(\boldsymbol{X}) \in \mathbb{R}^{196\times D}$ is the output after some layer of the frozen ViT backbone (196 tokens of dimension D), and $\boldsymbol{P} \in \mathbb{R}^{D\times 1}$ is a learnable linear projection matrix. Sanghavi et al. [28] have shown that a linear mapping is better at extracting depth than higher complexity mappings. In this thesis, we use linear probing to test if there is variation, depending on model architecture, of how depth information is sustained across layers.

2.4.2. Attention Score Cosine Similarity

The cosine similarity of two equally sized vectors denotes the similarity of their direction. Mathematically, given two equally sized vectors $\vec{x}, \vec{y}$, their cosine similarity is

$$\text{CosineSimilarity}(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{|\vec{x}||\vec{y}|} \tag{2.10}$$

. The maximum value of 1.0 is attained when $\vec{x} = \vec{y}$, the minimum -1.0 is attained when $\vec{x} = -\vec{y}$, and 0.0 when the two vectors are orthogonal. A cosine similarity with a significant absolute value implies that the vectors encode correlated information [16]. We use cosine similarity to determine if the attention score matrix $\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{D}} \in \mathbb{R}^{D\times D}$, which indicates how much information each token wants from each other token, correlates with some other matrix $\boldsymbol{M}$ denoting the similarity between tokens. For example, $\boldsymbol{M}_{i,j}$

may denote the difference of the average depths of patches i, j. Effectively, we use cosine similarity to determine if the attendance between patches can be explained by their depth or related quantities.

2.4.3. Centered Kernel Alignment

Centered Kernel Alignment (CKA) [7] is a technique to express the similarity between two internal representations as a single scalar. The scalar ranges from zero, indicating no correlation between the representations, to one, indicating that the representations are identical up to semantically meaningless transformations. The algorithm takes as input two matrices $\boldsymbol{X} \in \mathbb{R}^{m \times p_1}$ and $\boldsymbol{Y} \in \mathbb{R}^{m \times p_2}$ – the internal representations computed by two models on the same m samples. The algorithm operates on the Gram matrices $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{X}^\top \in \mathbb{R}^{m \times m}$ and $\boldsymbol{L} = \boldsymbol{Y}\boldsymbol{Y}^\top \in \mathbb{R}^{m \times m}$, denoting the dot-product self-similarity of the representations $\boldsymbol{X}, \boldsymbol{Y}$ [31]. CKA computes

$$\text{CKA}(\boldsymbol{K}, \boldsymbol{L}) = \frac{\text{HSIC}(\boldsymbol{K}, \boldsymbol{L})}{\sqrt{\text{HSIC}(\boldsymbol{K}, \boldsymbol{K})\text{HSIC}(\boldsymbol{L}, \boldsymbol{L})}} \tag{2.11}$$

where HSIC (Hilbert-Schmidt independence criterion [12]) is an operation that computes the similarity of two matrices in a way that is invariant to row or column order. Specifically, given the centering matrix $\boldsymbol{H} = \boldsymbol{I_n} - \frac{1}{n}\boldsymbol{1}\boldsymbol{1}^\top$ and centered versions of $\boldsymbol{K}$ and $\boldsymbol{L}$ computed as $\boldsymbol{K'} = \boldsymbol{H}\boldsymbol{K}\boldsymbol{H}$ and $\boldsymbol{L'} = \boldsymbol{H}\boldsymbol{L}\boldsymbol{H}$,

$$\text{HSIC}(\boldsymbol{K'}, \boldsymbol{L'}) = \frac{\text{vec}(\boldsymbol{K'}) \cdot \text{vec}(\boldsymbol{L'})}{(m-1)^2} \tag{2.12}$$

. Multiplying a matrix by $\boldsymbol{H}$ from both sides yields the same output up to row or column permutations; dividing by $(m-1)^2$ creates invariance to the number of samples m used. CKA is powerful because it can be used to measure the similarity of model behavior across distinct architectures and their layers [26]. We use CKA to identify how the integration of depth into an RGB-D ViT-based model affects its internal behavior.

2.5. Datasets

In this thesis, we make use of two image datasets: ImageNet-1K [9] and NYUDepthV2 [24]; some example images from each dataset are shown in Figure 2.5. ImageNet-1K is a subset of the larger ImageNet dataset. ImageNet-1K has 1.28M training images and 50K validation images, each labeled with one of 1000 diverse class labels. The dataset was primarily gathered by scraping the internet and labeled by means of crowdsourcing. ImageNet-1K does not have real depth, but it is nevertheless virtually always used in the pre-training of ViT-based RGB-D models in the literature [39]. In this thesis, we also use ImageNet-1K for this purpose – pre-training various models before fine-tuning them on a smaller dataset having real depth.

For fine-tuning, we use the NYUDepthV2 [24] dataset, which contains real depth as it was recorded by the Kinect [29] sensor that measures depth. The dataset features a small range of indoor environments, recorded under similar conditions. NYUDepthV2 has only 795 training images and 654 testing images, which is generally insufficient to train ViTs from scratch but is satisfactory for fine-tuning [8]. Unlike ImageNet-1K which is labeled for the classification task, NYUDepthV2 is labeled for the semantic segmentation (per-pixel classification) task. Regardless, a model pre-trained on ImageNet-1K learns representations that generalize to the other datasets. In the literature, this pair of datasets is frequently used for pre-training and later fine-tuning, establishing comparability between model architectures by exposing them to the same data throughout the training and validation processes.

2.6. RGB-D ViT-based models

In this section, we present the structures and motivations behind three RGB-D ViT-based models we adapt in our experiments. The presented architectures are, in their original papers, pre-trained on ImageNet-1K [9] and fine-tuned to maximize accuracy on NYUDepthV2 [24]. We do not present the full complexity of these architectures; instead we focus on the use of depth within the encoder blocks of the ViT-derived backbones.

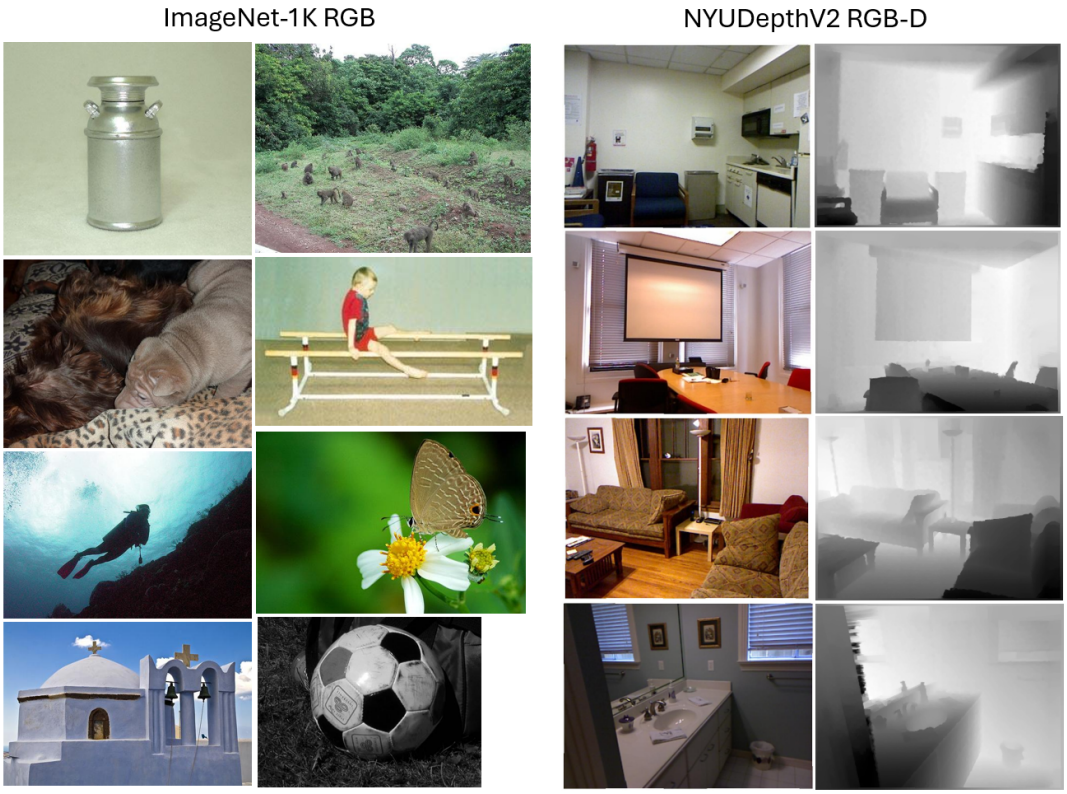


Figure 2.5: Some RGB images from the ImageNet-1K [9] dataset, and some RGB-D pairs from the NYUDepthV2 [24] dataset.

2.6.1. Dformer (v1)

Dformer [38] is, to our knowledge, the first method in the literature to pre-train on synthetic depth. The authors show that this technique is beneficial – fine-tuning the architecture when pre-trained using RGB + synthetic depth yields an accuracy that is higher than if pre-training were done purely with RGB. Additionally, Dformer leverages depth via a unique encoder block that replaces the default ViT's encoder block (Figure 2.3) in the encoder backbone. This encoder block is composed of two modules – global awareness attention (GAA) and local enhancement attention (LEA); the modules serve to enable the usage of depth for local and for global scene understanding (Figure 2.6). GAA pools tokens corresponding to adjacent image patches, using depth at the effective lower resolution to highlight the boundaries of 3D objects. On the other hand, LEA operates at the local scale, using a convolution to extract local details from the depth branch to enhance understanding of the RGB branch. The outputs of the two modules are concatenated and linearly projected onto the output RGB and depth branches.

2.6.2. GeminiFusion

GeminiFusion [14] is a model that uses distinct RGB and depth branches, but with a special method of exchanging information between them. The flow of information across the branches is illustrated in Figure 2.6. Inputs to the branches can be represented as two equally sized token collections $X_{RGB}, X_d \in \mathbb{R}^{N \times D}$ of N tokens of dimension D. The i-th token in each collection $X_{RGB,i}, X_{d,i}$ corresponds to the same patch of the input RGB-D pair. The architecture then exchanges information from tokens at equal positions. The extent to which the exchange occurs is computed by an MLP that accepts as input a concatenation of $X_{RGB,i}$ and $X_{d,i}$. Unlike Dformer, GeminiFusion uses identical RGB and depth branches that effectively embed no semantic priors about either modality's specific usage, instead focusing on a powerful and deliberate information exchange mechanism.

2.6.3. Dformerv2

Dformerv2 [39] is unique because it does not use depth in a complex way, i.e. it does not pass depth to an encoder that uses depth to generates the initial set of tokens. Instead, the core innovation behind Dformerv2 is to modify the attention mechanism used in the vanilla ViT to be

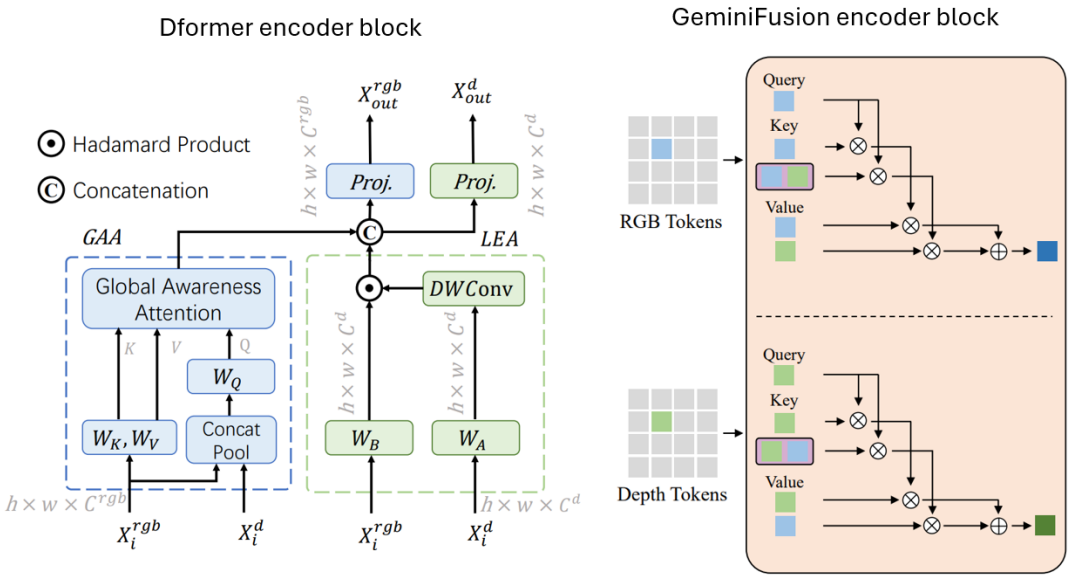


Figure 2.6: Encoder blocks of Dformer [38] and GeminiFusion [14]. Dformer uses asymetric depth and RGB branches to emphasize the modalities' distinct meanings in perceiving global and local information; GeminiFusion instead uses a symetric backbone but opts for deliberately guided fusion. Illustrations taken from the respective papers.

$$\text{MaskedAttention}(\boldsymbol{K}, \boldsymbol{Q}, \boldsymbol{V}) = \text{softmax}\left(\boldsymbol{K}\boldsymbol{Q}^{\top} - \boldsymbol{G}\right)\boldsymbol{V} \tag{2.13}$$

where $\boldsymbol{G}$ is a geometry prior expressed as

$$\boldsymbol{G} = \alpha|\boldsymbol{\Delta}^{depth}| + \beta|\boldsymbol{\Delta}^{distance}| \tag{2.14}$$

where $\alpha, \beta > 0$ are learnable scalars, and $|\boldsymbol{\Delta}^{depth}_{i,j}|$ and $|\boldsymbol{\Delta}^{distance}_{i,j}|$ are respectively the absolute average depth and Manhattan distances between patches i, j. Intuitively, using $\boldsymbol{G}$ as a bias to the attention scores decreases the extent to which patches interact with each other if they are far away in the 3D world. Effectively, distant patches that probably correspond to different objects do not conflate each other. The learnable parameters help modulate the effect. The authors claim that linear weighting works best without providing details about alternatives. The fine-tuning results achieved by DformerV2 on the NYUDepthV2 [24] dataset are higher than the results obtained by other models, including Dformer and GeminiFusion. It is therefore questionable whether the high complexity of these architectures is necessary to optimally leverage the value of depth.

3

Scientific Paper

Towards Understanding RGB-Depth Pre-Training in ViT-based Models

Per Skullerud
Delft University of Technology

ABSTRACT

Adding depth to RGB inputs (RGB-D) is known to improve model accuracy. State-of-the-art RGB-D models routinely adapt the Vision Transformer (ViT), but training ViTs purely on RGB-D is infeasible given the scarcity of depth data. A solution is using large RGB datasets to pre-train before fine-tuning on RGB-D, leveraging depth estimators to add complementary pseudo-depth to RGB datasets. We investigate the characteristics of models trained in this setup. We find that models, regardless of RGB-D fusion architecture, consistently learn simple patterns of depth utilization in the attention mechanism and across encoder layers. Our conclusions motivate the need to justify proposed depth fusion architectures against simple baselines, and to use depth fusion modules suited to the value of depth at each layer. We also show that after pre-training on pseudo-depth, fine-tuning favors pseudo- as opposed to real depth, highlighting the importance of minimizing their differences.

1. INTRODUCTION

In Computer Vision, the depth modality can be thought of as an image whose pixels denote the distance from the camera to the respective point of the imaged 3D scene. Intuitively, depth should be a valuable alternate view on RGB data as it provides geometric understanding of the scene as well as robustness against variable illumination. Numerous works have shown that adding depth improves model accuracy compared to RGB-only baselines [28].

Recent works have focused on integrating depth into architectures derived from the Vision Transformer (ViT) [5], given its state-of-the-art accuracy across a range of Computer Vision tasks [7]. However, there is a core challenge when using depth with ViTs – real depth data is scarce [28], but ViTs only outperform comparable backbones when trained on large datasets [7]. Recently, a novel solution has been shown promising – enhancing large RGB pre-training datasets with "fake" pseudo-depth produced by a specialized model that estimates depth given an RGB image as an input [23, 24]. Leveraging the knowledge within a depth estimator, it is then possible to (pre-)train on an RGB-D dataset of the same size as a source RGB dataset. Pre-training on pseudo-depth promises a small distribution shift when proceeding to fine-tune on real depth. However, the consequences of this hypothesized advantage are little understood, as models in the literature feature distinct backbones, training recipes, and other tweaks that complicate quantifying a good way to use depth [28]. Nevertheless, integrating RGB with depth (RGB-D fusion) is the core of RGB-D research and is as such the focus of this paper.

We aim to understand the value of pseudo-depth pre-training as an avenue towards better RGB-D models, with observed depth utilization patterns serving as recommendations for future works. Using pseudo-depth, in consistent conditions, we pre-train several diverse RGB-D models to identify the extent of architectural intricacy needed to meaningfully leverage depth. Since the ultimate goal is better results after fine-tuning, we also test how our pre-trained models, depending on their depth fusion technique, respond to the shift from pseudo- to real depth when proceeding to fine-tune on a dataset with real depth. Our three main contributions, the first two visualized in Figure 1, are that:

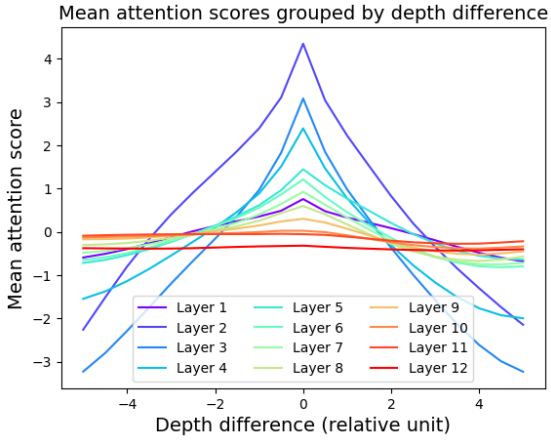


Figure 1. The use of depth in a ViT [5]-based model that fuses depth by adding it as a 4th channel to be used by the initial convolution. At all layers, there is a linear dependence between the absolute depth difference of two patches and their attention score. The strength of the dependence, expressed as the magnitude of the gradients of the plotted lines, peaks at layer 2 and wanes afterwards. Future RGB-D fusion techniques should exploit the simple dependence, and operate with differing purpose/intensity across layers.

- Models consistently learn a linear relationship between the attention scores $QK^{\top}_{i,j}$ of token pairs i, j and the difference of average depths of the corresponding patches. Future models should deliberately emphasize this pattern or detract from it.

- The intensity with which depth is utilized follows a consistent pattern – peaking at early/middle layers and gradually decreasing thereafter, implying that optimal depth fusion techniques may differ by layer.

- Fine-tuning with real depth yields worse accuracies than with pseudo-depth; the comparative advantage of pseudo-depth grows with model complexity. For better accuracy, real depth should be aligned with pseudo-depth when fine-tuning.

2. RELATED WORKS

2.1 ViT-based RGB-D Models

Recent RGB-D architectures introduce intricate depth fusion modules that highlight the consequences of appending a depth modality [24, 28]. For example, GeminiFusion [8] introduces a learnable way to emphasize the exchange of semantically distinct tokens between the modalities. Irrespective of their architectural details, RGB-D ViT [5]-based models must overcome the incompatibility of their data hunger with the scarcity of real depth data [28]. A widespread solution involves pre-training on a large RGB-only dataset, then fine-tuning on an RGB-D dataset with real depth [8, 23, 24, 27]. Models built for this setup often feature separate RGB and depth branches produced by encoding either modality with an encoder pre-trained purely on RGB; fine-tuning then implies training a network that fuses the representations within the two branches [8, 27]. The downside is that an encoder trained purely on RGB undergoes a domain shift when used for depth data [24]. Dformer [24] bypasses this domain shift by directly pre-training on a large RGB-D dataset created by complimenting a large RGB dataset with pseudo-depth. The authors show that this approach yields higher accuracies compared to a setup where depth is used only during fine-tuning [24]. Dformerv2 [23] also pre-trains on pseudo-depth and achieves a new benchmark for speed and accuracy by not encoding depth, instead simply masking attention scores based on the relative depth difference between patches. The success of Dformerv2 [23] motivates our interest in whether the true utilization of pseudo-depth motivates a need for complex fusion backbones.

2.2 ViT Interpretability

Meaningful strides have been made towards understanding the internal properties of ViTs [5]. An extensively proven property is that shallow ViT layers focus on texture and spatial position, middle layers capture the scene's structure, and deep layers describe global meaning [2]. Raghu et al. [17] use Centered Kernel Alignment (CKA) [10] to show that ViTs need fewer layers than Convolutional Neural Networks (CNNs) [11] to learn semantically rich representations. By exposing ViTs to various types of structured noise, Naseer et al. [14] gather evidence that ViTs are more robust to local distortions than CNNs [11] because ViTs prioritize a scene's structure over textures. Regarding the depth modality, Sanghavi et al. [19] use linear probing [1] of feature maps to show that depth information is actively maintained throughout all layers and can be accurately extracted via linear mappings at later layers. These observations guide our investigation by hinting that understanding depth is synonymous with achieving a thorough understanding the scene itself; the strong presence of depth at later layers suggests that scene comprehension may be eased at earlier layers by explicitly inputting depth.

2.3 Pseudo-Depth Estimation

The scarcity and noisiness of depth data measured by specialized sensors drives the use of depth estimators in RGB-D research [28]. Pseudo-depth used to pre-train Dformer [24] was generated using OmniData [6], a modality prediction pipeline trained on 3D assets labeled with depth by a rendering engine. To better leverage limited RGB-D training data, one may adapt a pre-trained RGB backbone for the task of depth estimation, e.g. Marigold [9] is a depth estimator that adapts Stable Diffusion [18]. ViTs [5] can be used to build powerful depth estimators if the scarcity of training data can be overcome. DepthAnythingV2 [22] achieves this by using exclusively synthetic data to train a teacher model, which then enables a student model to generalize on real-world data. Despite the appealing crisp edges and smooth surface gradients within pseudo-depth images, estimators are fundamentally limited to predicting relative depth (which of two pixels is closer). Hardware such as the Kinect [20] can see "real" absolute depth (distance in meters) [15], although it is uncertain whether this advantage outweighs the abundant noise in its measurements. The reaction of models to the differences between pseudo- and real depth are of interest in this paper.

3. METHOD

The goal is to understand the internals of pseudo-depth pre-training and accuracies after subsequent real depth fine-tuning, depending on RGB-D fusion architecture in ViT [5]-based models. Regarding pre-training, we pre-train six models using depth fusion techniques of various complexity. Our main baseline, from which all other models are derived, is the official PyTorch ViT implementation [16] that closely mirrors the original formulation. Since virtually all ViT-based architectures involve a sequential stack of encoder blocks, we introduce depth into the vanilla ViT in a straightforward way by swapping its encoder blocks with ones from methods [8, 23, 24] that do depth fusion. We pre-train on ImageNet-1K [4] classification, as is widely done by RGB-D models in the literature for pre-training [8, 23, 24, 27]. We follow a broadly accepted training recipe: 300 training epochs, 0.001 learning rate, 192 batch size, MixUp ($\alpha = 0.8$) [26] and CutMix ($\alpha = 1.0$) [25] augmentations. All pseudo-depth referenced in our experiments is generated by DepthAnythingV2 [22]; all depth (real or pseudo-) used is z-standardized to facilitate basic transferability. Given that RGB-D models in the literature use widely differing dropout techniques, we opt not to use any dropout to compare the models under equal conditions even if they may not align with the papers' original settings. Also, we do not (pre-)pre-train on a larger dataset like ImageNet-21K [4], or use miscellaneous improvements introduced by e.g. Swin [13] or DeiT [21]. We avoid these improvements to maintain alignment with the original ViT [5], given its ubiquity as a basis for derived models. Naturally, our models thus converge to lower scores than what has been shown possible. Nevertheless, the goal is to create a uniform environment for comparison as opposed to maximizing scores; identifying the optimal conditions for each model individually is beyond the scope of our work.

We now present the models we train with performance-related numbers summarized in Table 1. We motivate our approach towards extracting the depth fusion mechanism

Model	Params↓ (vs. M3CH)	FLOPs↓ (vs. M3CH)
M3CH	87M	17.58B
MLIN	87M (+12)	17.59B (+12M)
MMLP	87M (+1K)	17.62B (+40M)
M4CH	87M (+200K)	17.62B (+40M)
MDF1	103M (+16M)	19.96B (+2.4B)
MGEM	72M (-15M)	17.31B (-280M)

Table 1. Parameter counts and FLOPs of our trained models. The models are comparable w.r.t. their FLOPs and parameter counts being of a similar order of magnitude.

from chosen sources, seeking to maintain comparability across models while aligning with the theoretical motivation behind each architecture. For conciseness, we assign a moniker to each of our described models. In order of increasing intricacy of the depth fusion mechanism:

M3CH: vanilla ViT with 3 channel (RGB) inputs. As a reference point for what is gained by adding depth, we train a vanilla ViT closely following the original formulation [5]. We use 16×16 patches from an input image of size 224×224, 12 encoder blocks/layers, 12 attention heads per layer, and a hidden size of 768. We use these settings for all trained models unless otherwise specified.

MLIN: vanilla ViT with a patch depth difference attention mask weighted linearly. Inspired by Dformerv2 [23], this model adds a linearly weighed mask of differences between average patch depths as described by

$$\text{softmax}(QK^\top - \alpha|\Delta|)V \tag{1}$$

where $\alpha > 0$ is a learnable scalar and $|\Delta| \in \mathbb{R}^{N\times N}$ is an attention mask of the same shape as $QK^\top \in \mathbb{R}^{N\times N}$, where $|\Delta_{i,j}|$ is the absolute difference between the average depths of patches $1 \leq i \leq N$ and $1 \leq j \leq N$. The motivation is that patches having distinct depth values probably depict different objects, so those patches should inform each other less strongly to avoid confounding between unrelated scene elements. With only one learnable parameter per attention block and therefore also per layer, this model represents our simplest examined depth fusion technique.

MMLP: vanilla ViT with a patch depth difference attention mask weighed by an MLP. The authors of Dformerv2 [23] claim that, according to their experiments, it is optimal to weight $|\Delta|$ linearly in Equation 1; no ablation studies investigating alternatives are given. The Swin transformer [13] uses a similar mechanism (relative position bias) to reduce the interaction between patches based on their spatial distance, but weighted using an MLP. Since both spatial and depth relations denote distance in the 3D world, it is surprising that they are best weighted according to different formulations. To test if it is actually worse to weight depth using an MLP, we modify Equation 1 to be

$$\text{softmax}(QK^\top - \phi(\Delta))V \tag{2}$$

where $\phi : \mathbb{R} \to \mathbb{R}$ is an MLP with one hidden layer of size 32 using ReLU activations. We maintain the sign of Δ in case the MLP's flexibility finds benefit in distinguishing between patches being in front of or behind each other.

M4CH: vanilla ViT with 4 channel (RGB-D) inputs. Exactly like M3CH, but with depth appended as a 4^{th} input channel to be used by the initial convolution. This model showcases a naive integration of depth that makes use of all pixels in the input depth image, while not giving the model any prior knowledge on how depth could be used. The results acquired using this model set the precedent for the necessity of more complex depth fusion architectures.

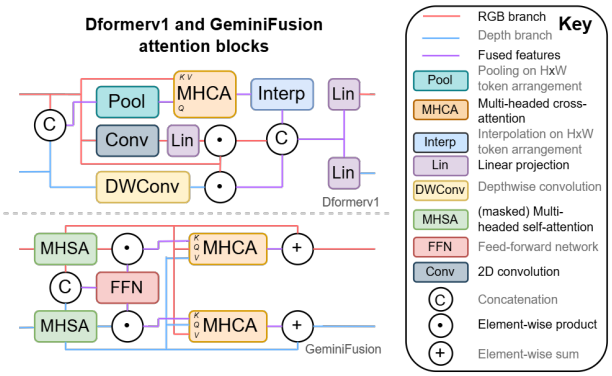


Figure 2. Abridged view on structures of depth fusion mechanisms used by Dformer [24] and GeminiFusion [8]. Dformer [24] asymmetrically uses RGB and depth branches, whereas GeminiFusion [8] uses a combination of self- and weighted cross-attention to mix tokens.

MDF1: using the encoder block of Dformer [24]. We directly copy the encoder block of Dformer [24]. This is our reference for a recent successful architecture from literature that uses separate branches for RGB and depth with complex modules to fuse them. As seen in Figure 2, the Dformer [24] block asymmetrically treats its two branches, embedding the inductive prior that depth can be used to refine details in RGB, and also to provide global context of the scene's shape. The added complexity makes Dformer [24] blocks more expensive than vanilla ViT blocks. To keep model sizes (Table 1) comparable, we only replace the first 4 encoder blocks with Dformer [24] blocks; the other 8 remain as vanilla ViT blocks. Given this architecture's intricacy, one would expect it to achieve better accuracy than the other simpler models described.

MGEM: using the encoder block of GeminiFusion [8]. Similar to MDF1, we directly copy the encoder block of GeminiFusion [8]. The GeminiFusion [8] block does not embed inductive priors, instead emphasizing an intense and deliberate fusion of information (Figure 2). A GeminiFusion [8] block uses four attention operations and is thus more expensive than a vanilla ViT block. To maintain comparability, we halve the size of the hidden state used by the RGB and depth branches, achieving the same total shape of encoder block inputs/outputs as when using one branch

at the original hidden state size. As in MDF1, we only replace the first 4 encoder blocks with GeminiFusion [8] blocks. Like MDF1, MGEM represents a model from the literature achieving near-state-of-the-art accuracies.

To identify what shared or differing properties the models exhibit, we investigate internal representations using a range of techniques. To see what self-attention prioritizes, we compute cosine similarities between $QK^\top$ and other matrices D denoting various distances between patches. We use Centered Kernel Alignment (CKA) [3] to compare internal representations across distinct models. Like [19], we use linear probing [1] to identify the presence of depth information throughout the models' layers. We rely on evidence from multiple techniques to increase confidence in the patterns we present.

Figure 3. Image 1015 from NYUdepthv2 [15]: RGB, real depth, pseudo-depth. Real depth is noisier but displays absolute distance unlike pseudo-depth, which consequently has a different value distribution. Semantic differences also exist – the mirror is seen as noise in real depth, whereas pseudo-depth suggests the room extends into the mirror. Neither depth image is obviously strictly better.

Looking beyond the pre-training process, we are also interested in how the shift from pseudo- to real depth affects the models. We investigate the robustness of (pre-)trained models to the unstructured noise and the value distribution shift displayed in Figure 3. Following the work of [12], we expose the models to Gaussian noise of varying σ; we separately do this per-pixel to emulate unstructured noise, and per 16×16 patch to emulate the distribution shift. Naturally, we also want to gauge the gap between these simple noises and the true pseudo-/real depth gap. To this end, we use RGB-D data from NYUDepthV2 [15] to compare how the models' output classification tensor changes when discarding real depth and instead using pseudo-depth estimated from RGB. Finally, we fine-tune models according to the training recipe used by Dformerv2 [23] on the NYUDepthV2 semantic segmentation (per-pixel classification) task. By doing so, we hope to answer how the pseudo-/real depth gap affects fine-tuning outcomes depending on model architecture.

4. EXPERIMENTS

The training process is summarized in Table 2; the table is sorted according to the architectural intricacy (complexity) of the models. More complex models achieve higher accuracies, though there are exceptions – MLIN beats MMLP, M4CH ties with MDF1. Training time scales unfavorably with FLOPs, implying that in practice, a simpler architecture runs faster than a complex one even at the same FLOP and parameter counts. Furthermore, Figure 4 shows that simpler models experience less accuracy loss when exposed to noise in the depth modality, which may make it easier for these simpler models to adjust from pseudo- to real depth when later fine-tuning. To answer whether the more complex architectures are justified, we analyze the internals of all trained models.

Model	Top-1%↑	Top-5%↑	Training time (h)↓	FLOPs↓
M3CH	66.0	86.6	26	17.58B
MLIN	68.3	88.0	40	17.59B
MMLP	67.9	87.4	44	17.62B
M4CH	71.0	89.5	32	17.62B
MDF1	71.2	88.5	55	19.96B
MGEM	71.2	89.8	48	17.31B

Table 2. Validation accuracies and pre-training times of our models on the ImageNet-1K [4] dataset. Accuracy tends to increase with model complexity, though training times scale unfavorably with FLOP count.

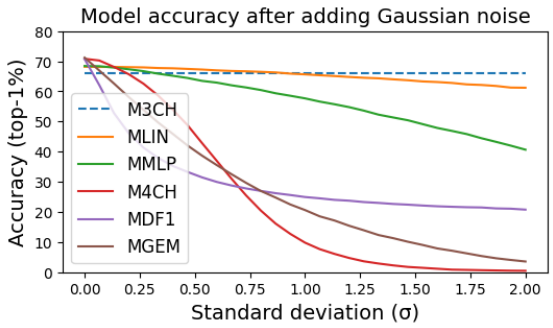


Figure 4. ImageNet-1K [4] validation accuracy of our trained models upon adding per-pixel Gaussian noise to the input depth channel. Simpler models are more robust.

4.1 Model Internal Representation Analysis

M3CH is our baseline without depth. To gauge the value added by introducing depth to the input, we are interested in the extent to which a depth signal appears passively. We compute the cosine similarity of the $KQ^\top$ (attention score) matrix and the between-patch depth difference matrix Δ at each layer. The peak in Figure 5 shows that depth partially explains the interaction between tokens at middle layers, where ViTs are known to extract structural features. However, the Figure also shows that $KQ^\top$ correlates even more strongly with a matrix of the Euclidean distances between all pairs of patch centers, as seen by the deeper peak. The correlation is probably caused by spatially close-by patches likely belonging to the same object and thus also having similar depth. In Figure 6 we disentangle depth from distance and find that M3CH has limited understanding of depth beyond distance, as expressed by the absence of patterns in columns, i.e. at the same spatial

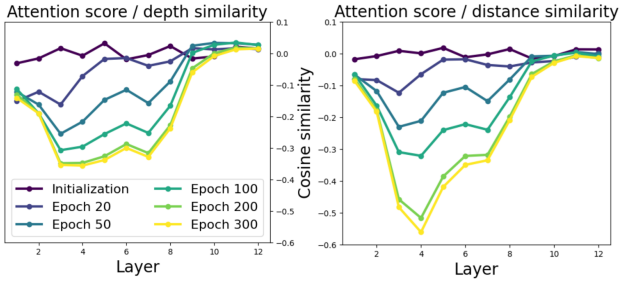


Figure 5. The cosine similarity between attention scores and patch-wise relative depth differences or Euclidean spatial distances across layers at various epochs. Negative values imply that increasing either distance decreases the extent to which patches attend with each other. Depth and distance consistently correlate, as expressed by cosine similarity differing from zero.

distance. Regardless, it must be noted that by explicitly adding depth to a model's input, we introduce information that is already partially available via a spatial embedding.

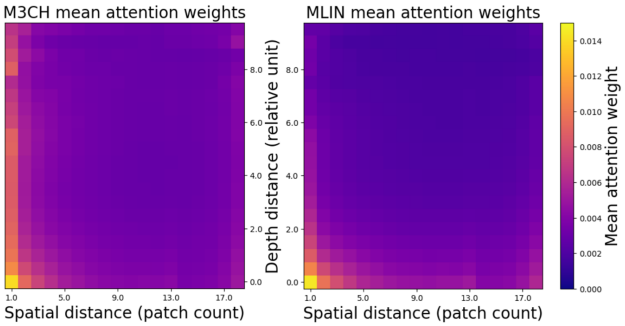


Figure 6. Mean attention weight (softmax($KQ^\top$)) grouped by corresponding patch spatial distance and absolute depth distance. Compared to M3CH, MLIN features weaker attending between patches having a large depth distance as seen by the lower values in the top region.

MLIN uses depth, and achieves a 2.3% higher top-1% score than M3CH despite only having 12 more parameters. In line with the model's theoretical motivation, masking attention based on depth weakens the interaction between patches having a large depth difference (Figure 6). Figure 7 shows that the parameters weighting the depth difference attention masks Δ cleanly converge to a pattern, implying that the model clearly understands the applicability of depth. The converged-at pattern reveals that the utilization of depth, expressed as the value of the parameters, gradually peaks at early/middle layers and decreases thereafter. Given that these particular layers are known to perform recognition of a scene's structure, it appears that one benefit of depth is easing this task.

MMLP achieves lower accuracy than M3CH despite having strictly more flexibility. In Figure 8, we again observe the same pattern of earlier layers showing a stronger utilization of depth. More crucially, the MLP learns a largely

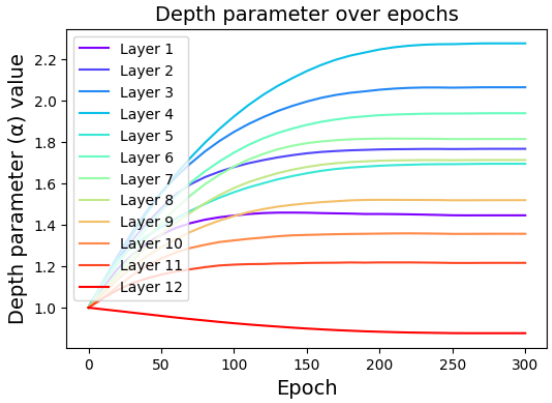


Figure 7. Of model MLIN, the training trajectory of the parameters used to weight the absolute depth difference between patches when forming an attention mask. The parameter converges without jitter and describes a clear pattern – the intensity of depth use gradually peaks at layer 4 and gradually decreases thereafter.

linear response across the domain of possible depth differences between patches, only changing direction near the origin. Near the origin is thus where the flexibility of the MLP is most apparent, learning a smooth transition between the leftmost and rightmost linear segments. Flexibility is probably most important near the origin, since depth differences are heavily concentrated within this region. Regardless, a simpler function could attain the same shape. Also, unlike MLIN which only handles the absolute value of Δ, MMLP learns to distinguish between positive and negative depth differences, and occasionally even learns a negative response, i.e. that patches should attend more strongly the greater their depth difference. Ultimately, the usefulness of the unique behavior of MMLP is questionable, as it encodes a weaker depth signal than MLIN (Figure 9), which is bad under the assumption that learning and retaining a depth signal is a consequence of understanding the scene. Overall, MMLP is inferior to MLIN in accuracy and runtime; also it is less robust to noise despite less consistently encoding the depth input.

M4CH is our simplest model that "sees" all pixels in the depth image as opposed to just the averages of each 16×16 patch. Also, unlike MLIN and MMLP, M4CH's design implies no inductive priors on how depth could be used. Curiously, without explicit guidance M4CH learns patterns that align with the behavior of MLIN. Most notably, M4CH learns a linear correlation between the depth differences of patches and entries in the $QK^\top$ matrix (Figure 1). Even when disentangling depth from the spatial distance between patches, we see the same pattern (Figure 10). This is the exact behavior enforced by MLIN via its attention mechanism expressed by Equation 1. However, as seen in Figure 11, the difference in internal representations between M4CH and M3CH is greater than between M3CH and MLIN in early/middle layers. It is implied that M4CH learns to leverage details within depth images as opposed to just the average of each 16×16 patch like MLIN. The utilization of depth within the early/middle

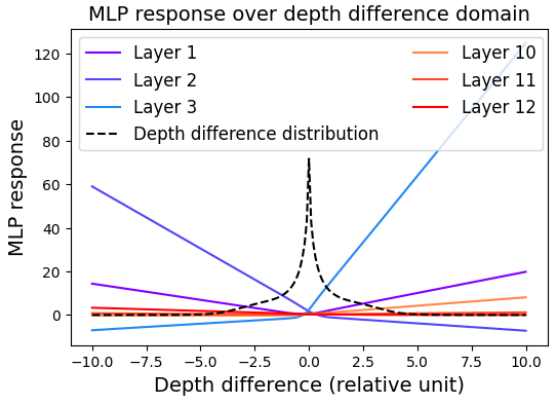


Figure 8. In MMLP, the input/output relation of the MLP weighting depth difference attention masks; the distribution of between-patch depth differences on pseudo-depth produced by DepthAnythingV2 [22] on ImageNet-1K [4]. The curves are straight across much of the distribution of between-patch depth differences, implying that the MLP's flexibility is not fully utilized.

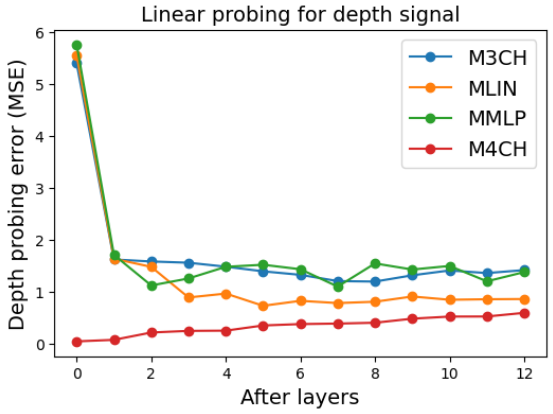


Figure 9. Linear probing to extract depth signals; lower values indicate a stronger signal. M4CH encodes the strongest signal since it has depth in the encoder input, MLIN encodes a stronger signal than MMLP despite being less flexible.

layers has been shown to peak in all models investigated so far, and the same also applies for M4CH as evidenced by Figure 1. Linear probing (Figure 9) further confirms the usefulness of depth in earlier layers as evidenced by strength of the depth signal. However, the signal remains stronger than in the other models across all layers, implying its usefulness throughout. In conclusion, M4CH confirms the soundness of MLIN's motivation, while enhancing its capabilities by utilizing details in depth images.

MDF1 and **MGEM** are slower than M4CH, but their accuracies are not much better. Looking into their internal representations reveals concerning trends about both Dformer's and GeminiFusion's encoder blocks. As seen in Figure 12, MDF1's Dformer blocks show a stronger presence of the depth signal in the RGB branch. This is incompatible with the motivation for the asymmetric design of the branches supposedly emphasizing the unique

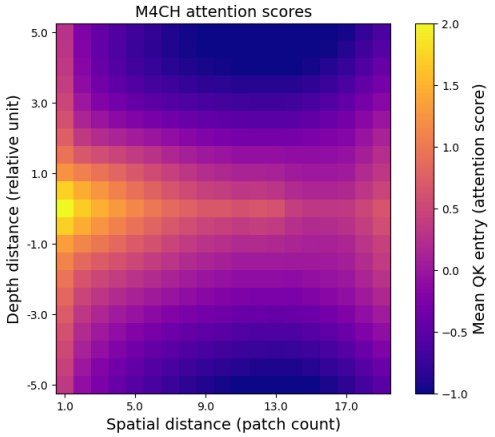


Figure 10. Mean attention score $QK^{\top}_{i,j}$ entry of tokens i, j grouped by depth and spatial distances of corresponding image patches, of model M4CH. At every spatial distance, attention scores correlate linearly with depth differences, suggesting that a simple mechanism could suffice to capture the relationship i.e. utilize depth.

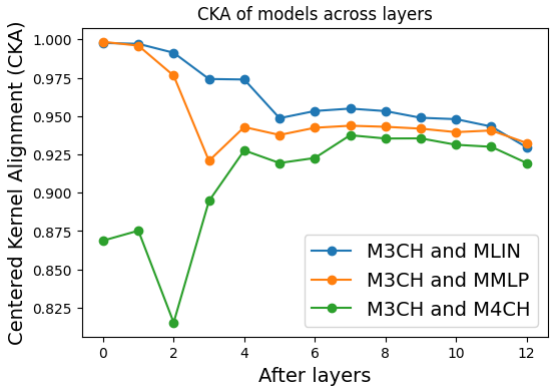


Figure 11. CKA between a vanilla RGB ViT (M3CH) and other models at equal layers. Only M4CH sees texture in the depth image, making this model's representations less similar to M3CH than MLIN's or MMLP's in early layers.

properties of RGB and depth. Conversely, MGEM's GeminiFusion blocks hardly change the presence of the depth signal in either branch, which contradicts the supposed active fusion accomplished by these blocks. The utilization of depth is hard to extract and fairly compare from the model's respective Dformer and GeminiFusion blocks. However, subsequent ViT blocks show the same pattern of depth utilization as the other examined models (Figure 13), i.e. linearly weigthing attention scores based on patch depth difference. Overall, in our particular scenario of pre-training on pseudo-depth, we find little benefit in the specific designs of the encoder blocks used in MDF1 and MGEM.

4.2 Model Generalization to Real Depth

Ultimately, the purpose of pre-training using pseudo-depth is to ease the transition to real depth when later fine-tuning on a dataset that has it. For simplicity, we begin with a theoretical investigation on MLIN and M4CH. We are inter-

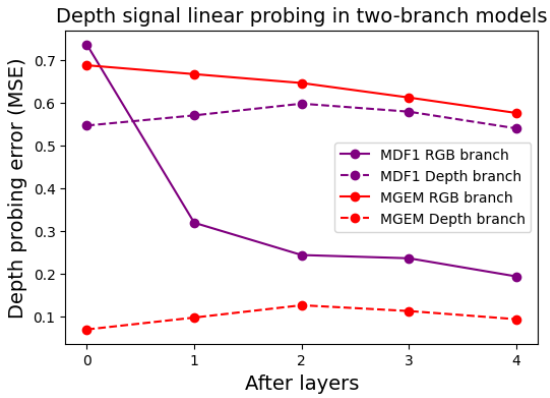


Figure 12. Linear probing to find the depth signal in the RGB and depth branches of models using Dformer [24] and GeminiFusion [8] encoder blocks. Dformer's RGB branch puzzlingly retains a stronger depth signal than the depth branch, whereas the presence of a depth signal hardly changes in both GeminiFusion's branches.

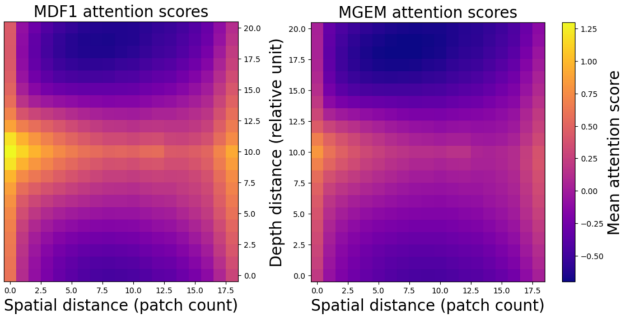


Figure 13. Mean $QK^{\top}_{i,j}$ entry of tokens i, j grouped by depth and spatial distances of corresponding image patches, of models MDF1 and MGEM. Results from the ViT blocks following the models' initial intricate encoder blocks. As with all models, we see linear weighting depending on the depth difference between patches.

ested in whether MLIN could possibly outperform M4CH in fine-tuning, as MLIN is more robust to input depth modality variations (Figure 4), which encompass the shift from pseudo- to real depth. We expose MLIN and M4CH to per-pixel Gaussian noise to emulate the unstructured noise visible in Figure 3, and per-patch Gaussian noise to emulate the distribution shift also shown in the Figure. The results in Figure 14 show that while M4CH is more accurate when no noise is added, MLIN is more accurate at higher noise levels. Of course, it is crucial whether the transition from pseudo- to real depth corresponds to a sufficiently high level of noise that MLIN would be more accurate than M4CH. Ideally, we would test how accuracy changes when validating with real depth, but there is no real depth available for ImageNet-1K [4]. As a proxy, we can use real depth from NYUDepthV2 [15] and pseudo-depth generated from this dataset's RGB images. We input images from this dataset (resized to 224×224) into our models and compute the Mean Absolute Error (MAE) between the classification tensors outputted when inputting real or

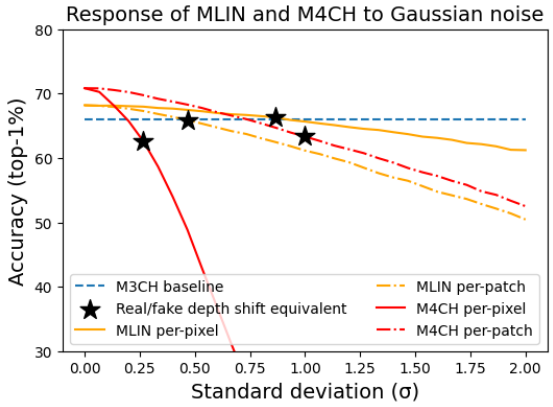


Figure 14. Robustness of MLIN and M4CH to per-pixel and per-patch noise; the noise levels that are comparable to the transition from pseudo- to real depth on NYUdepthv2 [15], depending on the type of noise and the model. For both per-pixel and per-patch noise types, the more complex model M4CH is more accurate at no noise, but MLIN is more accurate at the level of noise that is comparable to the move from pseudo- to real depth.

pseudo-depth. We then compute a comparable MAE between outputs produced using clean pseudo-depth and pseudo-depth with Gaussian noise applied. Effectively, for each model we can identify a Gaussian noise level that produces an error in model outputs comparable to moving from pseudo- to real depth. As seen in Figure 14, at this noise level MLIN is more accurate. The question remains whether MLIN, or other simple models, therefore give better accuracy when fine-tuning on real depth after having transitioned from pre-training on pseudo-depth.

Depth trained on	Real		Pseudo	
Depth tested on	Real	Pseudo	Pseudo	Real
Model	**mIOU%↑**			
MLIN*	26.67	24.40	26.73	24.18
MMLP*	26.87	24.99	27.60	24.47
M4CH	29.16	23.72	30.35	21.80
MDF1	31.19	25.43	32.66	21.87
MGEM	29.16	27.51	34.96	23.12
Dformer-Tiny	51.23	47.64	51.92	43.69
Dformerv2-Small*	56.73	55.62	56.67	55.19

*Depth used simply – not passed to an encoder.

Table 3. Fine-tuning results on our pre-trained models and models from literature. Models get fitted to their fine-tuning distributions, i.e. accuracy drops when testing on a model trained on pseudo-depth using real depth or vise versa. Fine-tuning with pseudo-depth yields higher accuracies, the difference is larger in the more complex models.

In Table 3, we present accuracies obtained after fine-tuning several pre-trained checkpoints on the NYUDepthV2 semantic segmentation task. All initial checkpoints are pre-trained using pseudo-depth; we compare subsequent fine-tuning using real and pseudo-depth. For completeness, we

include the results of the Dformer-Tiny [24] and Dformerv2-Small [23], since they are purposefully designed to maximize scores on the task at hand. One notable finding is that fine-tuning results vary more depending on model architecture than pre-training results (Table 2). More interesting is that models using depth simply, i.e. not encoding it, obtain similar accuracies when trained on pseudo- or real depth, but models that do encode depth prefer pseudo-depth. We also see that testing a model fine-tuned on pseudo- depth using real depth, or vise versa, yields a significant accuracy drop. The drop is greater for models encoding depth, and is also greater when moving specifically from pseudo- to real depth. It appears that while the fine-tuning process fits every model to the type of depth used, it is easier to fit to pseudo-depth. Overall, the results reject the hypothesis that models not encoding depth would perform better in fine-tuning as they are more resistant to the pseudo-/real depth shift. In reality, it appears that the best scores are achieved on pseudo-depth by models encoding depth in a complex way; the behavior of pre-trained checkpoints is not an ideal predictor of fine-tuning outcomes.

5. DISCUSSION AND CONCLUSIONS

One of our objectives was to identify the similarities and differences in the internal behavior of models when pre-trained on RGB-D. To investigate this, we pre-train six models closely following the ViT [5] but featuring distinct depth fusion modules. We analyze the internal behaviors of these pre-trained models. Then, we fine-tune these models to evaluate how their architectures affect their ability to generalize from pseudo- to real depth.

Our most striking finding is that the models consistently learn a linear relationship between the attention score of two tokens and the difference of the average depths of the corresponding input image patches. In addition to that, models learn that extracting details from depth images is beneficial. The consistency of this simple pattern contradicts the need of complex inductive priors or constructions directing models to utilize depth in a supposedly "better" way. In any case, future RGB-D fusion architectures should be validated to demonstrate that they solve a problem posed by the simple internal use of depth found.

Another simple and consistent pattern we find is that the utilization of depth, expressed as the correlation between inter-patch depth and attention scores, is strongest at early to middle layers. These layers are known to prioritize extracting structural information from the scene; depth possibly aligns strongly with this goal. This idea is further supported by the finding that the use of depth closely correlates with the use of spatial distance. These findings are highly actionable, as they clearly imply that RGB-D architectures should prioritize fusion in early layers where the semantic meaning of depth is most relevant.

Finally, we show that the shift from pre-training with pseudo-depth to fine-tuning with real depth impacts model accuracies. We show that simpler depth fusion techniques are more robust to changes in the depth modality, conversely the more complex a model is the more it favors fine-tuning with pseudo-depth. The highest accuracies are achieved by complex models regardless of the type of depth used in fine-tuning, though pseudo-depth is preferred. However, it is not true that fine-tuning with pseudo-depth fundamentally yields more accurate models since validating a model fine-tuned using real depth on pseudo-depth (or vise versa) yields a significant accuracy drop.

Our experiments are devalued by certain limitations. For one, we do not achieve near-state-of-the-art accuracies and do not determine the optimal conditions for each investigated architecture. The architectures address semantic segmentation and classification; we do not study a broad range of downstream tasks. The behavior of models meant for certain tasks, or models achieving higher accuracies, may be more complex. Importantly, we do not investigate the root causes of observed patterns, as we subscribe to the wide yet uncertain principle that ViTs learn arbitrarily complex patterns given enough data. On the topic of pseudo-/real depth, we align these distributions using z-standardization and do not investigate other statistical methods that may work better. Irrespective of the scale of our results' generalizability to other domains, we present several sensible recommendations to future RGB-D ViT-based model authors stemming from our work:

- Ensure that your model is competitive against a comparable baseline of simply inputting depth as a 4^{th} channel alongside RGB.
- Evaluate your proposed depth fusion module's effectiveness depending on the layer it is positioned in.
- Consider that your fusion architecture may be limited in generalizing from pseudo- to real depth.

Zooming out, in practical settings the choice between using pseudo- or real depth hinges on distinct costs – obtaining real depth requires specialized hardware, whereas generating pseudo-depth requires additional compute. If compute is not a limitation, it may be better to simply merge the depth estimator and the subsequent RGB-D model into one, avoiding the likely information loss stemming from needlessly mapping to/from a depth image. So, we believe that practical RGB-D deployments are only sensible if using real depth, in which cases the adversarial pseudo-/real depth gap must be accounted for in model choice.

Future works could delve deeper into the degrees of freedom within ViTs, i.e. to gather nuance on how distinct conditions could affect the consistency of the patterns we showcase in this paper. On a practical note, we do not investigate the realities of building better models using the reported results. Future works could, for example, investigate the feasibility of achieving better accuracies by constraining the relation between depth and distance in the attention mechanism. The optimal complexity of depth fusion modules depending on the layer of the ViT should also be investigated. In broader contexts, the fundamental value of pseudo-depth in (pre-)training remains little understood despite its wide-reaching promise of enabling large RGB-D datasets. Looking into the future, we believe that the path towards better RGB-D fusion is hidden inside models exposed to depth since their initial training epochs; it is just a matter of uncovering this path via careful inspection.

6. REFERENCES

[1] G. Alain and Y. Bengio, "Understanding intermediate layers using linear classifier probes, 2018," *URL https://arxiv. org/abs/1610.01644*, vol. 1610, 2018.

[2] S. Amir, Y. Gandelsman, S. Bagon, and T. Dekel, "Deep vit features as dense visual descriptors," *arXiv preprint arXiv:2112.05814*, vol. 2, no. 3, p. 4, 2021.

[3] C. Cortes, M. Mohri, and A. Rostamizadeh, "Algorithms for learning kernels based on centered alignment," *The Journal of Machine Learning Research*, vol. 13, pp. 795–828, 2012.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[6] A. Eftekhar, A. Sax, J. Malik, and A. Zamir, "Omnidata: A scalable pipeline for making multi-task mid-level vision datasets from 3d scans," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 786–10 796.

[7] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu *et al.*, "A survey on vision transformer," *IEEE transactions on pattern analysis and machine intelligence*, vol. 45, no. 1, pp. 87–110, 2022.

[8] D. Jia, J. Guo, K. Han, H. Wu, C. Zhang, C. Xu, and X. Chen, "Geminifusion: Efficient pixel-wise multimodal fusion for vision transformer," *arXiv preprint arXiv:2406.01210*, 2024.

[9] B. Ke, A. Obukhov, S. Huang, N. Metzger, R. C. Daudt, and K. Schindler, "Repurposing diffusion-based image generators for monocular depth estimation," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2024, pp. 9492–9502.

[10] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton, "Similarity of neural network representations revisited," in *International conference on machine learning*. PMlR, 2019, pp. 3519–3529.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[12] Q. Liu, Y. Liu, J. Wang, X. Lyu, P. Wang, W. Wang, and J. Hou, "Modgs: Dynamic gaussian splatting from casually-captured monocular videos with depth priors," in *International Conference on Learning Representations*, vol. 2025, 2025, pp. 97 048–97 074.

[13] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 10 012–10 022.

[14] M. M. Naseer, K. Ranasinghe, S. H. Khan, M. Hayat, F. Shahbaz Khan, and M.-H. Yang, "Intriguing properties of vision transformers," *Advances in Neural Information Processing Systems*, vol. 34, pp. 23 296–23 308, 2021.

[15] P. K. Nathan Silberman, Derek Hoiem and R. Fergus, "Indoor segmentation and support inference from rgbd images," in *ECCV*, 2012.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[17] M. Raghu, T. Unterthiner, S. Kornblith, C. Zhang, and A. Dosovitskiy, "Do vision transformers see like convolutional neural networks?" *Advances in neural information processing systems*, vol. 34, pp. 12 116–12 128, 2021.

[18] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10 684–10 695.

[19] J. Sanghavi, "From edges to depth: Probing the spatial hierarchy in vision transformers," *arXiv preprint arXiv:2604.23452*, 2026.

[20] H. Sarbolandi, D. Lefloch, and A. Kolb, "Kinect range sensing: Structured-light versus time-of-flight kinect," *Computer vision and image understanding*, vol. 139, pp. 1–20, 2015.

[21] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International conference on machine learning*. PMLR, 2021, pp. 10 347–10 357.

[22] L. Yang, B. Kang, Z. Huang, Z. Zhao, X. Xu, J. Feng, and H. Zhao, "Depth anything v2," *Advances in Neural Information Processing Systems*, vol. 37, pp. 21 875–21 911, 2024.

[23] B.-W. Yin, J.-L. Cao, M.-M. Cheng, and Q. Hou, "Dformerv2: Geometry self-attention for rgbd semantic segmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025, pp. 19 345–19 355.

[24] B. Yin, X. Zhang, Z. Li, L. Liu, M.-M. Cheng, and Q. Hou, "Dformer: Rethinking rgbd representation learning for semantic segmentation," *arXiv preprint arXiv:2309.09668*, 2023.

[25] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "Cutmix: Regularization strategy to train strong classifiers with localizable features," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 6023–6032.

[26] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," *arXiv preprint arXiv:1710.09412*, 2017.

[27] J. Zhang, H. Liu, K. Yang, X. Hu, R. Liu, and R. Stiefelhagen, "Cmx: Cross-modal fusion for rgb-x semantic segmentation with transformers," *IEEE Transactions on intelligent transportation systems*, vol. 24, no. 12, pp. 14 679–14 694, 2023.

[28] T. Zhou, D.-P. Fan, M.-M. Cheng, J. Shen, and L. Shao, "Rgb-d salient object detection: A survey," *Computational Visual Media*, vol. 7, no. 1, pp. 37–69, 2021.

References

[1] Josh Achiam et al. “Gpt-4 technical report”. In: *arXiv preprint arXiv:2303.08774* (2023).

[2] Shir Amir et al. “Deep vit features as dense visual descriptors”. In: *arXiv preprint arXiv:2112.05814* 2.3 (2021), p. 4.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[5] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2026. URL: https://www.blender.org.

[6] Sneha Chaudhari et al. “An attentive survey of attention models”. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 12.5 (2021), pp. 1–32.

[7] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. “Algorithms for learning kernels based on centered alignment”. In: *The Journal of Machine Learning Research* 13 (2012), pp. 795–828.

[8] Mostafa Dehghani et al. “Scaling vision transformers to 22 billion parameters”. In: *International conference on machine learning*. PMLR. 2023, pp. 7480–7512.

[9] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[10] Alexey Dosovitskiy et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020).

[11] Robert Geirhos et al. “Shortcut learning in deep neural networks”. In: *Nature Machine Intelligence* 2.11 (2020), pp. 665–673.

[12] Arthur Gretton et al. “A kernel statistical test of independence”. In: *Advances in neural information processing systems* 20 (2007).

[13] Jordan Hoffmann et al. “Training compute-optimal large language models”. In: *arXiv preprint arXiv:2203.15556* 10 (2022).

[14] Ding Jia et al. “Geminifusion: Efficient pixel-wise multimodal fusion for vision transformer”. In: *arXiv preprint arXiv:2406.01210* (2024).

[15] Salman Khan et al. “Transformers in vision: A survey”. In: *ACM computing surveys (CSUR)* 54.10s (2022), pp. 1–41.

[16] Alfirna Rizqi Lahitani, Adhistya Erna Permanasari, and Noor Akhmad Setiawan. “Cosine similarity to determine similarity measure: Study case in online essay assessment”. In: *2016 4th International conference on cyber and IT service management*. IEEE. 2016, pp. 1–6.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[18] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: *http://yann.lecun.com/exdb/mnist/* (2010).

[19] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[20] Tianyang Lin et al. “A survey of transformers”. In: *AI open* 3 (2022), pp. 111–132.

[21] Ze Liu et al. “Swin transformer: Hierarchical vision transformer using shifted windows”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 10012–10022.

[22] Kevin Meng et al. “Locating and editing factual associations in gpt”. In: *Advances in neural information processing systems* 35 (2022), pp. 17359–17372.

[23] Muhammad Muzammal Naseer et al. “Intriguing properties of vision transformers”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 23296–23308.

[24] Pushmeet Kohli Nathan Silberman Derek Hoiem and Rob Fergus. “Indoor Segmentation and Support Inference from RGBD Images”. In: *ECCV*. 2012.

[25] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[26] Maithra Raghu et al. “Do vision transformers see like convolutional neural networks?” In: *Advances in neural information processing systems* 34 (2021), pp. 12116–12128.

[27] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. 1985.

[28] Jainum Sanghavi. “From Edges to Depth: Probing the Spatial Hierarchy in Vision Transformers”. In: *arXiv preprint arXiv:2604.23452* (2026).

[29] Hamed Sarbolandi, Damien Lefloch, and Andreas Kolb. “Kinect range sensing: Structured-light versus Time-of-Flight Kinect”. In: *Computer vision and image understanding* 139 (2015), pp. 1–20.

[30] Alex Sherstinsky. “Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network”. In: *Physica d: Nonlinear phenomena* 404 (2020), p. 132306.

[31] Victor Sreeram and P Agathoklis. “On the properties of Gram matrix”. In: *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 41.3 (2002), pp. 234–237.

[32] Andreas Steiner et al. “How to train your vit? data, augmentation, and regularization in vision transformers”. In: *arXiv preprint arXiv:2106.10270* (2021).

[33] Chen Sun et al. “Revisiting Unreasonable Effectiveness of Data in Deep Learning Era”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[34] Christian Szegedy et al. “Inception-v4, inception-resnet and the impact of residual connections on learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 31. 1. 2017.

[35] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[36] Jason Wei et al. “Emergent abilities of large language models”. In: *arXiv preprint arXiv:2206.07682* (2022).

[37] Lihe Yang et al. “Depth anything v2”. In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 21875–21911.

[38] Bowen Yin et al. “Dformer: Rethinking rgbd representation learning for semantic segmentation”. In: *arXiv preprint arXiv:2309.09668* (2023).

[39] Bo-Wen Yin et al. “Dformerv2: Geometry self-attention for rgbd semantic segmentation”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2025, pp. 19345–19355.

A

Model Codes

In this Appendix, we provide the code for the 6 models we train described in the Scientific Paper. All codes are adapted from the official Pytorch Implementation of the Vision Transformer [25]. Specifically, we modify the version of the file in commit a095de1. Our baseline 3 channel RGB ViT *M3CH* constitutes this exact implementation. For the other models, we provide the snippet of the code adapted, alongside with the line numbers where the code snippets would be if individually copy-pasted into the Pytorch implementation. A short description of the changes made is given. Some unchanged lines are included for a better understanding of the context of the changed lines.

A.1. MLIN – linearly weighted depth difference attention mask

We pass depth as a second parameter to the *forward* function of the Vision Transformer. Only the averages of each 16×16 patch are kept. The averages are passed into the encoder, where they are used to generate a between-patch attention mask as in DformerV2 [39].

```
103         self.depth_param = nn.Parameter(torch.full((1, 1, 1), 1.0), requires_grad=True)
104         depth_scale = torch.log(
105             1 - 2 ** (-initial_value - heads_range * torch.arange(num_heads, dtype=torch.
                float) / num_heads)
106         )
107         self.register_buffer("depth_scale", depth_scale)
```

```
109     def generate_depth_decay(self, grid_d):
110         mask_d = grid_d.unsqueeze(2) - grid_d.unsqueeze(1)
111         mask_d = mask_d.abs() * self.depth_param
112         mask_d[:, 0, :] = 0
113         mask_d[:, :, 0] = 0
114         mask_d = mask_d.unsqueeze(1) * self.depth_scale.view(1, -1, 1, 1)
115         return mask_d
116 
117     def forward(self, input: torch.Tensor):
118         input, d = input
119         torch._assert(input.dim() == 3, f"Expected (batch_size, seq_length, hidden_dim) got {
                input.shape}")
120         HW = input.size(1)
121         ddc = self.generate_depth_decay(d)
122         attn_mask = ddc.view(-1, HW, HW)
123 
124         x = self.ln_1(input)
125         x, _ = self.self_attention(x, x, x, need_weights=False, attn_mask=attn_mask)
126         x = self.dropout(x)
127         x = x + input
128 
129         y = self.ln_2(x)
130         y = self.mlp(y)
131         return (x + y, d)
```

```
    def forward(self, input: torch.Tensor, d):
        torch._assert(input.dim() == 3, f"Expected (batch_size, seq_length, hidden_dim) got {
            input.shape}")
        input = input + self.pos_embedding
        for i, l in enumerate(self.layers):
            input, _ = l((input, d))
        return self.ln(input)
```

```
    def forward(self, x: torch.Tensor, d: torch.Tensor):
        parts = F.interpolate(d, size=(14, 14), mode="bilinear", align_corners=False)
        parts = parts.view(-1, 196)
        d = torch.cat((torch.zeros((x.size(0), 1), device=x.device, dtype=x.dtype), parts),
            dim=1)
        # Reshape and permute the input tensor
        x = self._process_input(x)
        n = x.shape[0]

        # Expand the class token to the full batch
        batch_class_token = self.class_token.expand(n, -1, -1)
        x = torch.cat([batch_class_token, x], dim=1)

        x = self.encoder(x, d)

        # Classifier "token" as used by standard language architectures
        x = x[:, 0]

        x = self.heads(x)

        return x
```

A.2. MMLP – MLP weighted depth difference attention mask

Similar to the code for MLIN, but we instead use an MLP to weight the between-patch attention mask.

```
        self.depth_mlp = nn.Sequential(
            nn.Linear(1, 32, bias=True),
            nn.ReLU(inplace=True),
            nn.Linear(32, 1, bias=False)
        )
        depth_scale = torch.log(
            1 - 2 ** (-initial_value - heads_range * torch.arange(num_heads, dtype=torch.
                float) / num_heads)
        )
        self.register_buffer("depth_scale", depth_scale)
```

```
    def generate_depth_decay(self, grid_d):
        mask_d = grid_d.unsqueeze(2) - grid_d.unsqueeze(1)
        mask_d = self.depth_mlp(mask_d)
        mask_d[:, 0, :] = 0
        mask_d[:, :, 0] = 0
        mask_d = mask_d.unsqueeze(1) * self.depth_scale.view(1, -1, 1, 1)
        return mask_d
```

A.3. M4CH – a 4-channel RGB-D ViT

In the convolution that produces the initial tokens, the input channel count is increased from 3 to 4. The change represents appending depth as a 4^{th} input channel.

```
            self.conv_proj = nn.Conv2d(
                in_channels=4, out_channels=hidden_dim, kernel_size=patch_size, stride=
                    patch_size
            )
```

A.4. MDF1 – using the encoder block of Dformer (v1)

We replace the first four encoder blocks from the vanilla ViT with the encoder block from Dformer [38], referred to in the code below as *Block*. The implementation of this block (and its dependencies) can be found in the authors' repository at lines 24 – 200.

```
class Encoder(nn.Module):
    """Transformer␣Model␣Encoder␣for␣sequence␣to␣sequence␣translation."""

    def __init__(
        self,
        seq_length: int,
        num_layers: int,
        num_heads: int,
        hidden_dim: int,
        dropout: float,
        norm_layer: Callable[..., torch.nn.Module] = partial(nn.LayerNorm, eps=1e-6),
        mlp_dim: int = 0,
    ):
        super().__init__()
        # Note that batch_size is on the first dim because
        # we have batch_first=True in nn.MultiAttention() by default
        self.pos_embedding = nn.Parameter(torch.empty(1, seq_length, hidden_dim).normal_(std
            =0.02))  # from BERT
        self.dropout = nn.Dropout(dropout)
        layers: OrderedDict[str, nn.Module] = OrderedDict()
        for i in range(4):
            layers[f"encoder_layer_{i}"] = Block(
                index=i,
                dim=hidden_dim,
                num_head=num_heads,
            )
        for i in range(4, 12):
            layers[f"encoder_layer_{i}"] = EncoderBlock(
                num_heads,
                hidden_dim,
                mlp_dim,
                dropout,
                0.0,
                norm_layer,
            )
        self.layers = nn.Sequential(layers)
        self.ln = norm_layer(hidden_dim)
        self.class_token = nn.Parameter(torch.zeros(1, 1, hidden_dim))

    def forward(self, x: torch.Tensor, d: torch.Tensor):
        torch._assert(x.dim() == 3, f"Expected␣(batch_size,␣seq_length,␣hidden_dim)␣got␣{x.
            shape}")
        x = x + self.pos_embedding
        B, _, C = x.shape
        x = x.view(B, 14, 14, C)
        d = d.view(B, 14, 14, C // 2)
        for i, f in enumerate(self.layers):
            if i == 4:
                x = x.view(B, 196, C)
                batch_class_token = self.class_token.expand(B, -1, -1)
                x = torch.cat([batch_class_token, x], dim=1)
            if i < 4:
                x, d = f((x, d))
            else:
                x = f(x)
        return self.ln(x)
```

```
    def forward(self, x: torch.Tensor, d: torch.Tensor):
        # Reshape and permute the input tensor
        x = self._process_input(x)
        n = x.shape[0]

        # Expand the class token to the full batch
        batch_class_token = self.class_token.expand(n, -1, -1)
        x = torch.cat([batch_class_token, x], dim=1)

        x = self.encoder(x, d)

        # Classifier "token" as used by standard language architectures
        x = x[:, 0]

```

```
303         x = self.heads(x)
304
305         return x
```

A.5. MGEM – using the encoder block of GeminiFusion

Similar to MDF1, except that the encoder block *Block* is taken from GeminiFusion [14]. The implementation of this block can be found in lines 14 – 322 of the authors' official code repository.

```
139         self.gemini_blocks = 4
140         for i in range(self.gemini_blocks):
141             layers[f"encoder_layer_{i}"] = Block(
142                 dim=hidden_dim // 2,
143                 num_heads=num_heads,
144                 last_block=False,
145             )
146         for i in range(self.gemini_blocks, 12):
147             layers[f"encoder_layer_{i}"] = EncoderBlock(
148                 num_heads,
149                 hidden_dim,
150                 mlp_dim,
151                 dropout,
152                 0.0,
153                 norm_layer,
154             )
155         self.layers = nn.Sequential(layers)
156         self.ln = norm_layer(hidden_dim)
157         self.class_token = nn.Parameter(torch.zeros(1, 1, hidden_dim))
158
159     def forward(self, x: torch.Tensor, d: torch.Tensor):
160         torch._assert(x.dim() == 3, f"Expected (batch_size, seq_length, hidden_dim) got {x.
                shape}")
161         x = x + self.pos_embedding
162         d = d + self.pos_embedding
163         B, _, C = x.shape
164         for i, f in enumerate(self.layers):
165             if i == self.gemini_blocks:
166                 x = torch.cat([x, d], dim=2) # (B, N, C)
167                 batch_class_token = self.class_token.expand(B, -1, -1)
168                 x = torch.cat([batch_class_token, x], dim=1)
169             if i < self.gemini_blocks:
170                 x, d = f([x, d], 14, 14)
171             else:
172                 x = f(x)
173         return self.ln(x)
```

B

Training Code

Code to train the models, specifically M4CH in the example below, though the format is compatible with all models.

```python
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader, DistributedSampler
from torchvision.transforms import v2
from torchvision.models import vit_b_16, ViT_B_16_Weights
from datasets import load_dataset
from transformers import AutoModelForDepthEstimation, AutoImageProcessor
from tqdm import tqdm
import time

train_transforms = v2.Compose([
    v2.RandomResizedCrop(224, antialias=True),
    v2.RandomHorizontalFlip(p=0.5),
    v2.RandAugment(num_ops=2, magnitude=9),
    v2.ToImage(),
    v2.ToDtype(torch.float32, scale=True),
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

val_transforms = v2.Compose([
    v2.Resize(256, antialias=True),
    v2.CenterCrop(224),
    v2.ToImage(),
    v2.ToDtype(torch.float32, scale=True),
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

def train_transform_fn(examples):
    # Convert PIL images to Tensors and apply transforms
    examples["pixel_values"] = [train_transforms(image.convert("RGB")) for image in examples[
        "image"]]
    return examples

def val_transform_fn(examples):
    # Convert PIL images to Tensors and apply transforms
    examples["pixel_values"] = [val_transforms(image.convert("RGB")) for image in examples["
        image"]]
    return examples

def cleanup():
```

```
    dist.destroy_process_group()

def collate_fn(batch):
    pixel_values = torch.stack([example["pixel_values"] for example in batch])

    # Extract labels and convert to a long tensor
    labels = torch.tensor([example["label"] for example in batch])

    return (pixel_values, labels)

def depth_values(depth_model, images):
    with torch.inference_mode(), torch.amp.autocast('cuda'):
        outputs = depth_model(pixel_values=images)
    parts = F.interpolate(outputs.predicted_depth.unsqueeze(1), size=(224, 224), mode="
        bilinear", align_corners=False)
    parts = (parts - 2.5785) / 1.6296
    return parts

def main(rank, world_size):
    torch.cuda.set_device(rank)
    dist.init_process_group(backend="nccl", rank=rank, world_size=world_size)
    device = torch.device(f"cuda:{rank}")
    print(f"I␣am␣process␣{rank}/{world_size}")

    # DDP-specific environment variables
    depth_model = AutoModelForDepthEstimation.from_pretrained(
        "depth-anything/Depth-Anything-V2-Small-hf",
        dtype=torch.float16
    ).to(device).eval()
    depth_model = torch.compile(depth_model, mode="max-autotune")

    BATCH_SIZE = 192      # Adjust based on your VRAM
    NUM_WORKERS = 16       # CPU threads for dataloading
    EPOCHS = 300          # ViTs require long training schedules (300 is standard)
    LR = 1e-3             # Base Learning Rate
    if rank == 0:
        print(f"initial␣LR:␣{LR}")
    WEIGHT_DECAY = 0.05    # High weight decay is standard for ViT

    torch.backends.cudnn.benchmark = True

    cutmix_mixup = v2.RandomChoice([
        v2.CutMix(num_classes=1000, alpha=1.0),
        v2.MixUp(num_classes=1000, alpha=0.8)
    ])

    train_dataset = load_dataset("ILSVRC/imagenet-1k", split='train')
    train_sampler = DistributedSampler(train_dataset, shuffle=True, num_replicas=world_size,
        rank=rank)
    val_dataset = load_dataset("ILSVRC/imagenet-1k", split='validation')
    val_sampler = DistributedSampler(val_dataset, shuffle=False)
    train_dataset.set_transform(train_transform_fn)
    val_dataset.set_transform(val_transform_fn)

    train_loader = DataLoader(
        train_dataset, batch_size=BATCH_SIZE, sampler=train_sampler,
        num_workers=NUM_WORKERS, pin_memory=True, drop_last=True, collate_fn=collate_fn
    )
    val_loader = DataLoader(
        val_dataset, batch_size=BATCH_SIZE, sampler=val_sampler,
        num_workers=NUM_WORKERS, pin_memory=True, collate_fn=collate_fn
    )
    model = vit_b_16(weights=None).to(device)

    print("Compiling␣model...")
    model = torch.compile(model, mode="max-autotune")
    model = DDP(model, device_ids=[rank])
    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

    optimizer = optim.AdamW(model.parameters(), lr=LR, weight_decay=WEIGHT_DECAY)

```

```
    warmup_epochs = 10
    warmup_scheduler = torch.optim.lr_scheduler.LinearLR(
        optimizer, start_factor=0.01, end_factor=1.0, total_iters=warmup_epochs
    )

    main_epochs = EPOCHS - warmup_epochs
    cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=main_epochs, eta_min=1e-6
    )

    scheduler = torch.optim.lr_scheduler.SequentialLR(
        optimizer,
        schedulers=[warmup_scheduler, cosine_scheduler],
        milestones=[warmup_epochs]
    )

    scaler = torch.amp.GradScaler('cuda')

    if rank == 0:
        checkpoint = {
            'model_state_dict': model.module.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'scheduler_state_dict': scheduler.state_dict(),
            'epoch': 0
        }
        torch.save(checkpoint, f"vit_b_16_epoch_0_full_4CH.pth")

    print("Starting training...")
    for epoch in range(EPOCHS):
        model.train()
        running_loss = 0.0
        train_loader.sampler.set_epoch(epoch)

        since_last = time.time()
        for i, (inputs, targets) in enumerate(train_loader):
            inputs, targets = inputs.to(device, non_blocking=True), targets.to(device,
                non_blocking=True)

            # Apply CutMix or MixUp dynamically on the GPU
            inputs, targets = cutmix_mixup(inputs, targets)
            depths = depth_values(depth_model, inputs)
            inputs = torch.cat((inputs, depths), dim=1)

            optimizer.zero_grad(set_to_none=True) # Slightly faster than standard zero_grad()

            # Forward pass with AMP
            with torch.amp.autocast('cuda', dtype=torch.bfloat16):
                outputs = model(inputs)
                loss = criterion(outputs, targets)

            # Backward pass & Optimizer step
            scaler.scale(loss).backward()

            # Gradient clipping is often helpful for ViTs early in training
            scaler.unscale_(optimizer)
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

            scaler.step(optimizer)
            scaler.update()

            running_loss += loss.item()

            if rank == 0 and i % 100 == 99:
                now = time.time()
                print(f"Epoch [{epoch+1}/{EPOCHS}], Step [{i+1}/{len(train_loader)}], Loss: {
                    running_loss/100:.4f}, Since last: {now - since_last}")
                running_loss = 0.0
                since_last = now

        scheduler.step()
        model.eval()
```

```
183
184         # 1. Initialize metrics as tensors on the current device
185         top1_tensor = torch.tensor(0.0).to(device).to(torch.float64)
186         top5_tensor = torch.tensor(0.0).to(device).to(torch.float64)
187         total_tensor = torch.tensor(0.0).to(device).to(torch.float64)
188
189         # Ensure the sampler knows which epoch we are in (if applicable)
190         val_loader.sampler.set_epoch(epoch)
191
192         # Disable gradients and compilation overhead
193         with torch.no_grad():
194             for inputs, targets in tqdm(val_loader, disable=(rank != 0)): # Only show
                    progress bar on rank 0
195                 inputs, targets = inputs.to(device, non_blocking=True), targets.to(device,
                        non_blocking=True)
196
197                 with torch.inference_mode(), torch.amp.autocast('cuda'):
198                     depths = depth_values(depth_model, inputs)
199                     inputs = torch.cat((inputs, depths), dim=1)
200                     outputs = model(inputs)
201
202                 # Metrics calculation
203                 _, pred = outputs.topk(5, 1, True, True)
204                 pred = pred.t()
205                 correct = pred.eq(targets.view(1, -1).expand_as(pred))
206
207                 # Update tensors instead of python floats
208                 top1_tensor += correct[:1].reshape(-1).float().sum()
209                 top5_tensor += correct[:5].reshape(-1).float().sum()
210                 total_tensor += targets.size(0)
211
212                 # 2. Synchronize results across all GPUs
213                 dist.all_reduce(top1_tensor, op=dist.ReduceOp.SUM)
214                 dist.all_reduce(top5_tensor, op=dist.ReduceOp.SUM)
215                 dist.all_reduce(total_tensor, op=dist.ReduceOp.SUM)
216
217         # 3. Only Rank 0 handles printing and saving
218         if rank == 0:
219             t1_acc = 100 * top1_tensor.item() / total_tensor.item()
220             t5_acc = 100 * top5_tensor.item() / total_tensor.item()
221
222             print(f"Top-1␣Accuracy:␣{t1_acc:.2f}%")
223             print(f"Top-5␣Accuracy:␣{t5_acc:.2f}%")
224             with open("t1accs_4CH.txt", "a") as file:
225                 file.write(f"{t1_acc}␣{t5_acc}\n")
226
227             if epoch == 4 or (epoch + 1) % 10 == 0:
228                 checkpoint = {
229                     'model_state_dict': model.module.state_dict(),
230                     'optimizer_state_dict': optimizer.state_dict(),
231                     'scheduler_state_dict': scheduler.state_dict(),
232                     'epoch': epoch
233                 }
234                 torch.save(checkpoint, f"vit_b_16_epoch_{epoch+1}_full_4CH.pth")
235         dist.barrier()
236     cleanup()
237
238 if __name__ == '__main__':
239     main(int(os.environ["LOCAL_RANK"]), torch.cuda.device_count())
```

C

AI Statement

Generative AI tools were used in the writing of this thesis. Google Gemini and Anthropic Claude were the models used. The models were used to assist in training code generation, and for researching relevant related works. Generative AI tools were not used in writing, except to discover terminology or to refine already written sentences. The work in this thesis constitutes original ideas by the authors.