Master of Machine Learning
2022

Universitatea Politehnica Timișoara

PaNeXt: AN IMPROVED PANOPTIC SEGMENTATION METHOD

Candidate: Caius-Ioan DEBUCEAN

Scientific coordinator: Assoc. Prof. Dr. Habil. Eng. Călin-Adrian POPA

Session: June 2022

Contents

Chapter 1

Introduction

1.1 Context

Scientific techniques that try to reproduce human vision functionalities are all part of the same field: Computer Vision. Working predominantly with image data, some of these functionalities that Computer Vision tries to recreate are Image Classification, Object Detection, Segmentation, and many others. This document presents a method for Panoptic Segmentation [14].

Segmentation is a very challenging task in Computer Vision. It represents the task of pixel-wise, voxel-wise, or point-wise classification on a multi-dimensional input by dividing it into different regions or clusters.

The existing research is predominantly concentrated on solutions using monocular vision – a single 2D RGB image [14, 15, 16, 17, 2], with some techniques leveraging additional depth information [3] to achieve better results. Other approaches are not restricted only to 2D inputs. Segmentation can also be done on 3D data, usually procured from LiDARs, and they take the form of either Voxel Segmentation or Point Cloud Segmentation.

In the case of this work, 2D images are used. There are two common segmentation tasks for RGB images: Semantic Segmentation and Instance Segmentation. The resulting segmented regions for the two tasks point to objects that are referred to as Stuff or Things in the literature. Merging these two tasks results into Panoptic Segmentation, a solution that manages to differentiate and output both Things and Stuff.

While classical segmentation solutions are effective at more straightforward tasks, such as Edge Detection, Color Mask filtering, or Noise Reduction, advanced Segmentation requires more sophisticated approaches that leverage underlying image features. In the case of Panoptic Segmentation, the most effective solutions use Neural Networks, more specifically Convolutional Deep Neural Networks [1, 2, 16] or novel Transformer Architectures [4, 5].

Segmentation has many purposes nowadays: from knowing the area of the road in a self-driving car to isolating tumors on medical scans. However, most of the time it is treated as two different and complex tasks. A Panoptic Segmentation solution is more efficient than two separate Segmentation tasks, and its use-cases can be expanded to Object Detection problems. The Panoptic approach is still relatively novel in the academic community, and

there is plenty of room for improvement.

1.2 Proposed Solution

This document tackles the problem of Panoptic Segmentation for 2D images. Because this is a complex task, a more powerful approach is employed through the use of advanced Neural Network architectures.

The proposed solution is called PaNeXt, and it combines multiple advancements from both CNNs and Transformers. A simplified use-case of the PaNeXt architecture is presented in Figure 1.1.

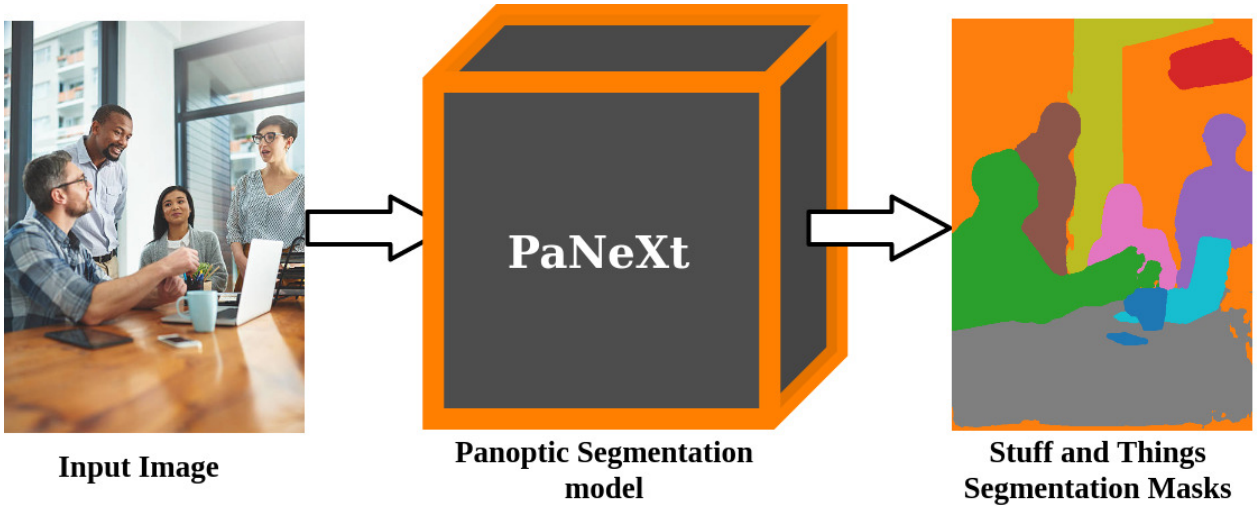


Figure 1.1: A simplified view on a panoptic segmentation performed with PaNeXt.

PaNeXt is composed of three main components:

- **The Backbone**, which encodes the input information and produces feature maps at multiple complexity levels
- **The Kernel Generator**, which locates center points for Things and regions for Stuff clusters
- **The High-Resolution Feature processor**, which encodes high-resolution information required for finer segmentations.

Existing Panoptic Segmentation solutions usually have two separate branches for the instance and segmentation subtasks [1, 2, 3, 17]. There are also solutions where instance segmentation is attempted without boxes [2]. More advanced approaches [16] manage to achieve a more unified architecture, but the models are not powerful enough. To boost performance, Transformer approaches [4, 5, 24, 25] are employed over convolutions. Unlike previous attempts, PaNeXt is a solution that remains fully convolutional, handles both Semantic and Instance Segmentation on a single branch, and also leverages Transformer aspects for improved performance.

Training such complex architectures for Panoptic Segmentation is a costly process, that requires the use of very expensive hardware for multiple days in a row, for just one complete experiment. Thus, training a novel approach to its fullest capability is not feasible, but more lightweight variants can be reached in order to show the potential of the new approach. While the State-of-the-Art landscape for panoptic approaches is full of enormous Transformer architectures, PaNeXt manages to overcome existing fully convolutional approaches and achieve satisfactory results on the COCO Panoptic dataset.

Chapter 2

Theoretical Foundations

2.1 Artificial Intelligence and its subdomains

With the growing popularity of Artificial Intelligence solutions, there has been a high level of confusion about defining and differentiating it from other concepts such as Machine Learning, Deep Learning, and other subfields, such as Computer Vision. The AI subdomains can be seen in Figure 2.1.

Artificial Intelligence can be interpreted as incorporating human logic into machines. It also includes the simplest examples of solutions implemented on a computer, such as systems defined by conditional rules.

Machine Learning is a subfield of Artificial Intelligence, representing the ability of computers to "learn" to solve a problem without explicitly programming each instruction. The more exposed the system is to a larger amount of data, the more the machine learning algorithm self-regulates.

In turn, Deep Learning is a subfield of Machine Learning and describes a problem-solving technique that uses Neural Networks – structures inspired by the human brain system. Unlike other alternative techniques, Deep Learning requires more input data and hardware resources, depending on the depth and density of the networks used.

Computer Vision is a field of computer science that aims to replicate human vision reasoning through a system that can identify features and interpret them from images.

2.2 Learning techniques in Machine Learning

Machine Learning solutions require significant amounts of data in order to provide answers to problems. There are multiple types of problems, such as Classification, Detection, Segmentation, and Reconstruction. Moreover, the data format can differ, as do possible approaches. Therefore, there are several important types of learning. Also, decentralized training methods can be used together with all types of Machine Learning depending on how the data is structured and distributed for the training process.

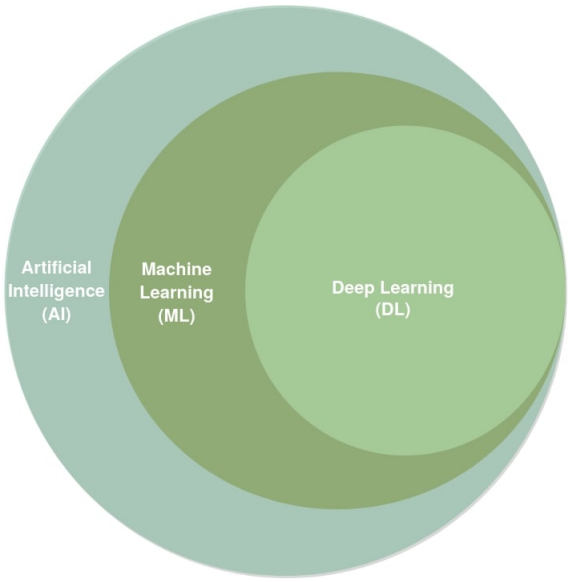


Figure 2.1: Artificial Intelligence Subdomains diagram.

2.2.1 Supervised learning

„Supervised learning involves the correlation between a set of input variables X and an output variable Y, and the application of this mapping to predict outputs for unseen data" [6]. In other words, supervised learning requires manually annotated data as a benchmark for predictions. This method implies that we are not only looking for patterns in the data but also that we know the explicit purpose of the model. Moreover, a large, diverse, and balanced dataset is needed to ensure the possibility of a proficient model. In Computer Vision, this technique is mainly used in classification, detection, segmentation, etc.

2.2.2 Unsupervised learning

Unlike the previous typology, unsupervised learning aims to determine patterns present in the dataset without having access to its labels. The main techniques related to unsupervised learning are clustering and association.

2.2.3 Semi-supervised learning

Only a small portion of the data is labeled in semi-supervised learning, while the largest portion of the data remains unlabeled. „Semi-supervised learning (SSL) is halfway between supervised and unsupervised learning. In addition to unlabeled data, the algorithm is provided with some supervision information – but not necessarily for all examples. Often, this information will be the targets associated with some of the examples" [7].

A model is trained on the small labeled subset in a supervised manner. Next, the model generates pseudo-labels for the unlabeled data. We call these "pseudo" labels because they are limited to the information of a tiny batch of data. For example, this may cause an uneven representation of classes, resulting in an unwanted bias. The labels from the

labeled training data are then linked with the pseudo labels through an iterative process, and the performance of the model will keep increasing, provided the data is suitable.

In conclusion, semi-supervised learning benefits from both supervised and unsupervised techniques.

2.2.4 Self-supervised learning

In self-supervised learning, the model trains itself by using part of the data to predict the other part. Consequently, the unsupervised problem is transformed into a supervised one using auto-generating labels. This method is mostly used in Natural Language Processing problems (e.g., sentence completion). Because the data is usually massive in such problems, it is essential to set the proper learning objectives to get supervision from the unlabeled data. To conclude, self-supervised learning tries to identify hidden parts of the input from any unhidden parts.

2.2.5 Parallel, Distributed, and Federated learning

Training large amounts of data requires powerful hardware. However, to scale a singular machine to handle enormous amounts of data can become expensive. To solve this, multiple machines can be used to offer better Performance per Price. On the other hand, when training a smaller model it is possible for a machine to be kept busy without the usage of most its resources. Such cases represent a tricky task, partly due to finding the most efficient and centralized way of handling trained gradients and when to update the model.

Parallelizing can be done in two ways: i) training locally on a single machine, but using multiple cores or GPUs of the machine to train on a bigger batch of data; ii) training with multi-core processing on a single machine (e.g., by performing SGD on multiple mini-batches).

Distributed Learning has two concepts: Data Parallelism – distributing the data into multiple machines, resulting in faster training and data computation; Model Parallelism – when the model is distributed into multiple machines due to being too big to fit a single machine.

Federated Learning is a relatively recent idea in the field of Artificial Intelligence and can be used with any other type of learning. It differs from the classical local training methods by training the neural network on several devices with different input data that cannot be shared. A system with a decentralized database, with a server that aggregates and merges device-driven features, then resets the same devices with an improved model. This technique facilitates data security as there is no centralized source of data.

2.3 Image Segmentation

In the field of Computer Vision, image segmentation is the process of separating a visual input into multiple pixel-level defined partitions, known as regions or masks. The role of segmentation is to give a more simplistic representation of the input image, by applying

certain reasonings or rules. Segmentation is usually used to create contours for areas of interest, and can be seen as a problem of pixel-level classification.

At a deeper level, segmentation is used in multiple ways: to create boundaries for countable objects – called Things (e.g., Cars, People), to create boundaries for uncountable masses – called Stuff (e.g., Sky, Road), or a mixture of the two. Subsequently, Image Segmentation splits into three approaches: Semantic Segmentation, Instance Segmentation, and Panoptic Segmentation.

2.3.1 Semantic Segmentation

Commonly referred as dense prediction, Semantic Segmentation deals with labeling each pixel in an input image with a corresponding class label. An important aspect of Semantic Segmentation is the fact that instances of the same class are not separated, but instead each class is treated as an uncountable mass, alternatively called Stuff. Refer to Figure 2.2(b) [14] for an example of a semantic segmentation map.

Because the resulting semantic segmented map is usually upsampled to the original input size, sequential architectures [34, 33] are not the most effective, as they only reduce dimensionality and produce blocky and very rough output masks. More advanced models [36, 35] use Decoders and Encoders to be able to maintain a certain level of dimensionality for the output mask, making it more refined as it was upscaled to the input size.

2.3.2 Instance Segmentation

Being a combination between Object Detection and Semantic Segmentation, Instance Segmentation deals with segmenting only the countable objects (Things) in an image, without treating uncountable masses (Stuff). In other words, Instance Segmentation detects bounding boxes for objects in an image, classifies them, and creates a separate segmentation for each one.

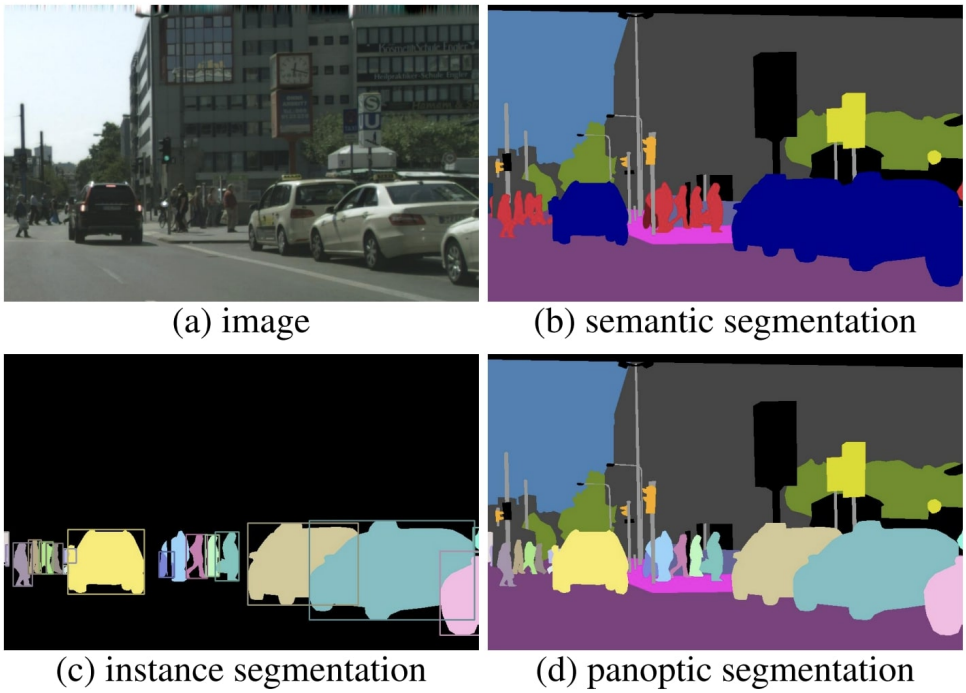
Solutions for such a problem are mostly derived from existing Object Detection [11, 12] architectures, with Mask R-CNN [13] being a landmark advancement in this direction. Refer to Figure 2.2(c) [14] for an example of an instance segmentation map.

2.3.3 Panoptic Segmentation

Introduced recently, Panoptic Segmentation [14] is a combination of Semantic and Instance Segmentation. More explicitly, Panoptic Segmentation deals with both countable objects (Things) and with uncountable masses (Stuff).

There are fewer existing solutions [15, 16, 17] for the Panoptic approach, mostly due to the high complexity requirements in data annotation and to the existence of separate datasets, for either instance or semantic approaches, that are not unified into a more comprehensive panoptic dataset. Solutions are also slower and have bigger sized models due to the increased difficulty of performing two separate sub-tasks. Furthermore, this panoptic

approach is still relatively new and there is a lot of potential research to be done, partly because existing solutions are still not achieving high enough performance to be reliably used at a production level. A comparison of segmentation tasks is presented in Figure 2.2(d) [14] for an example of a panoptic segmentation map.



(a) image (b) semantic segmentation

(c) instance segmentation (d) panoptic segmentation

Figure 2.2: For an input image (a) there are three ground-truth segmentations: (b) semantic segmentation, (c) instance segmentation, and (d) panoptic segmentation. [14]

2.4 Neural Networks

Neural Networks represent the technique that Deep Learning is based on, having the structure graph (nodes and edges), which can be associated with the synapses and neurons of the human brain. For these reasons, nodes are referred to as neurons and edges as weights or parameters. The deep appearance of neural networks is denoted by the presence of multiple hidden layers. A simple representation can be seen in Figure 2.3.

Typically, a neural network contains an input layer, some hidden layers, and an output layer. From a mathematical point of view, weights are numbers that transform input data as it passes through the network, and neurons are mathematical processes, or more commonly known as activation functions, by which information is transformed. A parameter called bias is also used to adjust the weighted amount when entering a neuron.

A layer often used in neural networks is the fully connected layer, which transforms the input data employing weights and activation functions, with formula (2.4.1).

$$\hat{y} = \sigma(Wx + b). \tag{2.4.1}$$

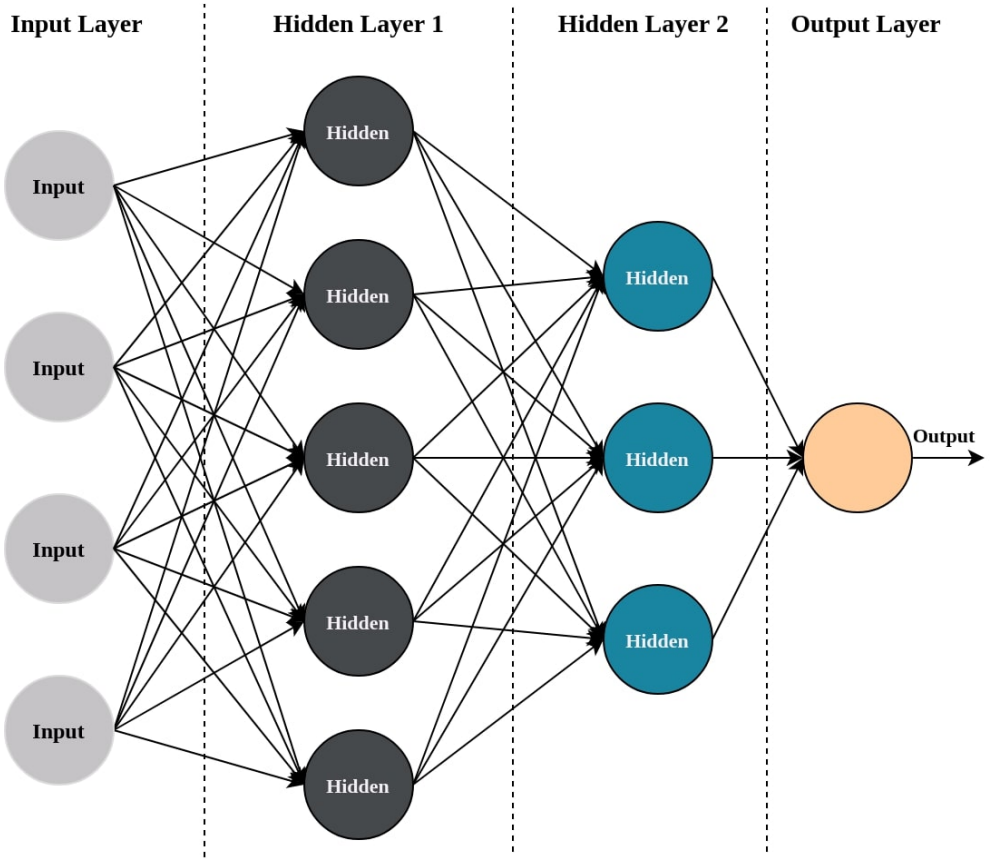


Figure 2.3: Dense Neural Network architecture with 2 hidden layers.

In the previous equation, $\hat{y}$ is the output of a neuron, σ is the activation function, W represents the weights for that neuron, b is the bias, and x is the input data for the neuron. For a network to deduce the size of the error from the desired result, a cost function is used. A common one is Binary Cross-Entropy, defined as:

$$E = -\frac{1}{n}\sum_{k=1}^{n}((1-y_k)\ln(1-\hat{y}_k) + y_k \ln(\hat{y}_k)). \tag{2.4.2}$$

In equation (2.4.2), $\hat{y}$ is the model prediction, and y is the correct targeted label. This metric is often used in supervised learning problems, such as classification.

The purpose of the network is to minimize the cost function in order to produce accurate results.

2.4.1 Gradient Descent

Gradient Descent is a fundamental method of optimization in Deep Learning, which aims to minimize the cost function by updating the weights of the model in the direction of the most "steep" slope.

Considering the formula (2.4.1), it can be expanded into:

$$\hat{y} = \sigma(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b). \tag{2.4.3}$$

To update the weights, we need the error function gradient, defined by:

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots, \frac{\partial E}{\partial w_n}, \frac{\partial E}{\partial b} \right). \tag{2.4.4}$$

Finally, we define a learning rate α. With all these values, a Gradient Descent step can be performed:

$$w_k^* = w_k - \alpha \frac{\partial E}{\partial w_k}, \tag{2.4.5}$$

$$b^* = b - \alpha \frac{\partial E}{\partial b}, \tag{2.4.6}$$

where w_k^* and b^* are the new values for weights w_k and bias b.

2.4.2 Feedforward

The Feedforward process is used in neural networks to turn input data into network output data. In order to better define this concept, consider the structure in Figure 2.4 with the activation function σ, weights $W^{(i)}$ for layer i and the system prediction $\hat{y}$.

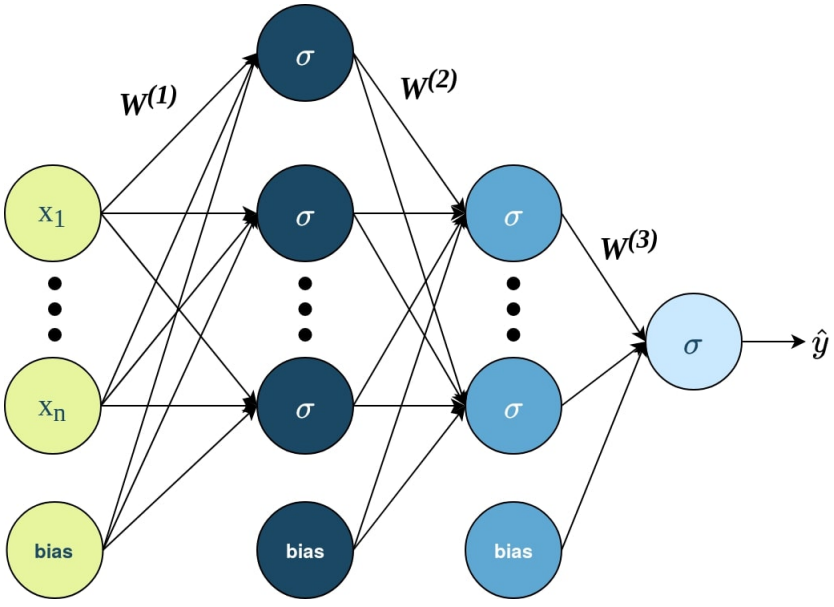


Figure 2.4: Example of a more detailed Neural Network architecture, with Weights as edges and Activations as graph nodes.

Applying the formula (2.4.1) for all layers in Figure 2.4, results in:

$$\hat{y} = \sigma \circ (W^{(3)} \sigma \circ (W^{(2)} \sigma \circ (W^{(1)} x))), \tag{2.4.7}$$

with $\circ$ as the operator for the composition of functions.

2.4.3 Backpropagation

The method used in neural networks to calculate the cost function gradients required by optimization algorithms is called Backpropagation. It is based on the use of the derivative chaining rule to calculate partial derivatives ∂ of some composite functions.

For $A = f(x)$ and $B = g \circ f(x)$, the Chain Rule is defined as:

$$\frac{\partial B}{\partial x} = \frac{\partial B}{\partial A}\frac{\partial A}{\partial x}. \tag{2.4.8}$$

By recursively using this rule, backpropagation succeeds in "producing an algebraic expression for the gradient of a scalar, depending on each node in the computational graph produced by that scalar". [8].

2.4.4 Frameworks

A framework for Deep Learning is a library that simplifies the process of creating a model by using more intelligible syntax. They are designed to interpret and optimize the interaction between the user and computing devices such as video cards or processors. For example, a framework will integrate transformations and operations with matrices, making it possible to parallelize the computational process or make it easier to view and record parameters.

Two of the most popular frameworks are:

1. TensorFlow [9]: provides a big set of useful tools such as TensorBoard for clear visualization of a model's pipeline and was the first pipeline to allow TensorRT serving, a semi-automatic optimization process of models through pruning, quantization, and weight-sharing techniques. For these reasons, TensorFlow has advantages in creating production-level models, but has the disadvantage of having a more difficult and complex syntax.

2. PyTorch [10]: based on the Torch library, it provides clarity and ease of use with "Pythonic" trends. For these reasons, the academic community tends to use this framework on a large scale.

Lately, the two frameworks can output compatible models that can be switched between them through the use of a common model format extension (e.g., ONNX, wts).

2.5 Convolutional and Attention Neural Networks

„CNN is a type of Deep Learning model for processing data that has a grid pattern, such as images, which is inspired by the organization of animal visual cortex and designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns. CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers." [18]

The convolution is a landmark linear mathematical operation in Computer Vision and Deep Learning, which was imported from the Digital Signal Processing field. It works in a sliding window approach, applying a set of filters, resulting in a set of feature maps made out of multiple channels. The resulting set is also called a layer. Mathematically, they are

represented in the form of tensors, a generalized matrix form. Unlike matrices, tensors are N-dimensional.

While the convolution dominated the Computer Vision field in the 2010s with a constant flow of improved architectures and comprehensive research, attention mechanisms [20, 23] applied on image-based problems have started to quickly dominate the field in the 2020s. Initially used in Natural Language Processing problems, attention mechanisms (or Transformers) were adapted to work for images [24], with later improvements even introducing alternatives [25] for backbone architectures that are capable of producing new State-of-the-Art results in many Computer Vision problems. Currently, convolutional improvements inspired from the transformer-based methods [26] are making a comeback, with more research opportunities opening the door to a new era of Computer Vision solutions.

2.5.1 Multidimensional Convolutions and Terminology

A layer contains a multitude of channels. The kernel is a weight matrix that is multiplied by the underlying patch of the input layer, the resulting values being summed into a single final value representative of that portion. A convolution filter is a set of stacked kernels. The stride of a convolution is the number of pixels by which the filter slides over the image. A stride of one ensures complete coverage, while a stride greater than one leads to a drastic reduction in dimensionality, with a smaller resulting feature set. A representation of the different types of kernels is shown in Figure 2.5.

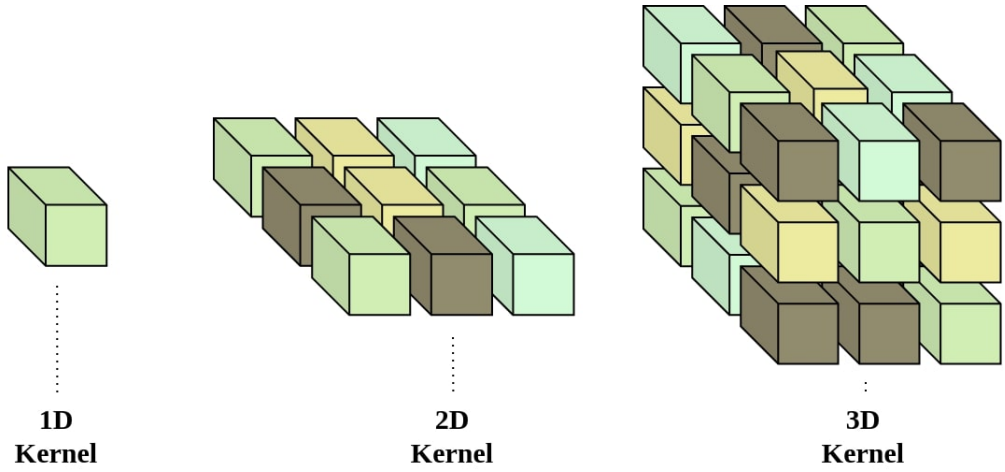


Figure 2.5: Types of Convolution kernels (1D, 2D, and 3D).

For a 2D convolution, a kernel is applied depthwise to each channel, the results being gathered and aggregated into a new feature map. The number of kernels in the filter represents the number of channels that will result from the convolution. While performing a convolution, the sliding window can only move filters only on the height and width of the layer. Figure 2.6 shows a simple convolution with a 2×2 kernel and six output channels. 3D convolutions contain 3D kernels, which are applied independently in the depth of the input layer. Thus, 3D convolutions move the filters both on the height and width of the layer, as well as on its depth.

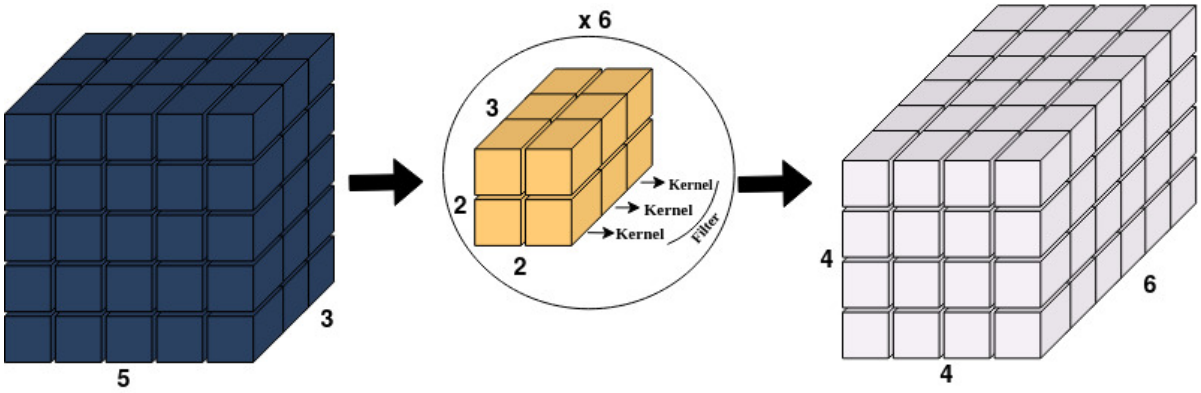


Figure 2.6: 2D Convolution with 6 output channels.

In terms of dimensionality, the opposite operation to the convolution is the transposed convolution. Different from a classical convolution, maintains or lowers the feature size, a transposed convolution aims to increase the size by adding padding or using expansion. This reduction or increase of dimensionality is often referred as downsampling and upsampling.

2.5.2 Separable convolution

Another concept used in various architectures is an efficient convolution, called separable convolution. [27] It divides the classic operation into a depthwise convolution, that only reduces the size of the input layer, and another pointwise convolution, that increases the number of channels. The result is a substantial reduction in the number of operations. An example is shown in Figure 2.7 as an alternative to the convolution shown in Figure 2.6.

In the case presented in Figure 2.6, the input layer has filters of size $2 \times 2 \times 3$ applied 4×4 times for a single output layer. The previous steps are applied 6 times, the final dimensions being $4 \times 4 \times 3$. In total, there are $2 * 2 * 3 * 4 * 4 * 6 = 1152$ multiplication operations.

Applying the separable convolution in Figure 2.7, the input layer has 3 filters of size $2 \times 2 \times 1$ applied 4×4 times. This is called a depthwise convolution and it is performed only once. The number of operations in this depthwise convolution is $3 * 2 * 2 * 1 * 4 * 4 = 192$ multiplication operations.

A pointwise convolution is applied to the intermediary result of the depthwise convolution. with a filter of size $1 \times 1 \times 3$ applied 4×4 times. This step is repeated 6 times. In total, the pointwise convolution performs $1 * 1 * 3 * 4 * 4 * 6 = 288$ multiplications.

Thus, the separable convolution performs only $192 + 288 = 480$ multiplications, as opposed to the 1152 multiplications performed by the classic convolution. This difference becomes more significant with the increase of the input set of features.

The disadvantage of separable convolutions is the reduction in the number of weights in the convolution, possibly limiting the learning threshold compared to a normal convolution. However, the efficiency of separable convolutions compensates for potential downsides.

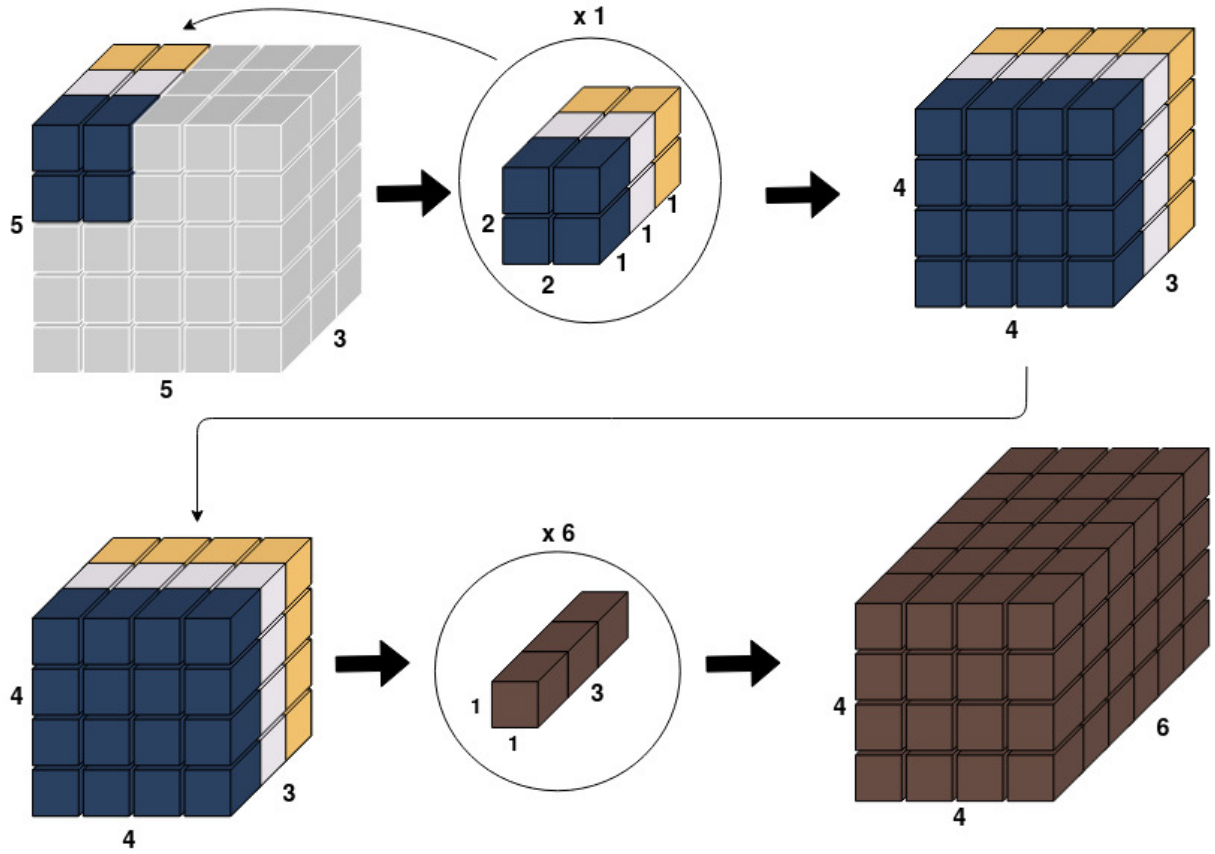


Figure 2.7: Separable Convolution composed of a Depthwise Convolution and a Pointwise Convolution.

2.5.3 Deformable convolutions

Deformable convolutions [28] have been introduced to have different receptive fields by adding 2D offsets to the grid sampling locations from normal convolutions. These help to factor multiple scales of objects. A visual representation of Deformable convolutions is presented in Figure 2.8. To be noted that the offsets are learnable parameters from preceding feature maps.

2.5.4 Residual blocks (ResNet)

A classic residual block [29] was introduced in the famous ResNet architecture, and it consists of a 2D convolution with a 1×1 kernel that reduces the number of filters, a second 2D convolution with a 3×3 kernel that does not reduce dimensionality, followed by a third 2D convolution with a 1×1 kernel that restores the initial number of filters. The features from the last convolution are summed up with the input of the first convolution. This type of connection between the input and the output is called a residual connection (or skip connection) and can be applied only when the input and output have the same dimensions. All convolutions have a ReLU activation function (described in Section 2.6.2). The residual connection combined with this set of convolutions form the Residual Block, presented in Figure 2.9. If reduced dimensionality is desired, the first convolution is strided and a skip connection is not used.

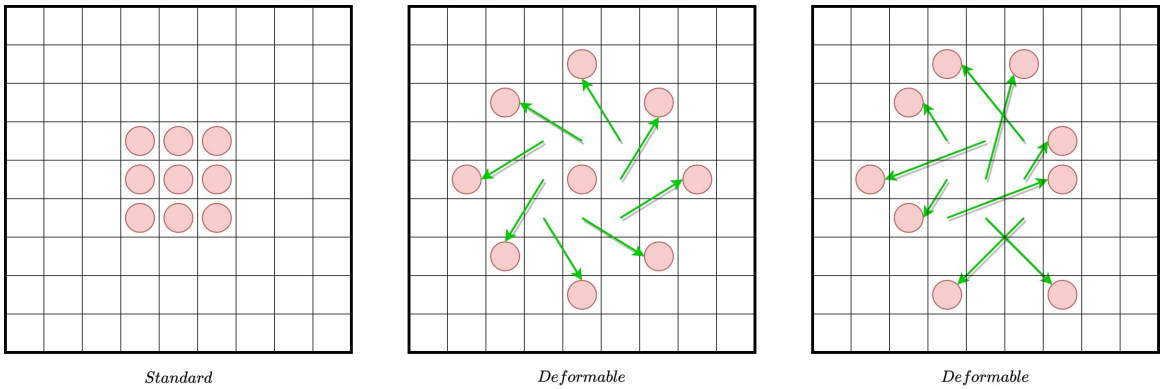


Figure 2.8: Visualisation for: a normal convolution kernel (left); example $\#1$ of deformable convolution (center); example $\#2$ of deformable convolution (right).

The residual blocks were created to combat the problem of gradients with very low values. The block does not add a new level of abstraction but only refines the information, and has the ability of maintaining dimensionality. Thus, stacking more residual blocks can increase computational power, making the model able to learn the existing features better. The ResNet block can be interpreted as an output that can only be influenced by intermediate features but cannot be overridden by them. The name "residual" comes from the fact that the output of this block is equal to the input plus the residual information captured.

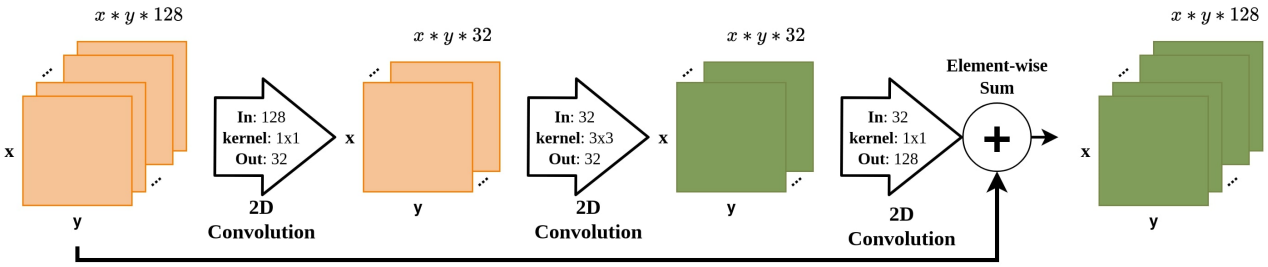


Figure 2.9: An example of a Residual Block.

2.5.5 Inverted Residual Blocks with Linear Bottlenecks

Introduced in [31], they started from the premise that feature maps can be encoded in small layers (with fewer channels) and that non-linear activations (e.g., ReLU) cause information loss, even if they have the ability to represent the complexity of the data. Linear activations $f(x) = x$ (identity function) and non-linear ReLU6 activations are used in this structure (which are actually ReLU activations with upper limit 6).

This block receives a tensor as input and applies a pointwise convolution in the first phase, which increases the number of channels. This aims to prepare the tensor for non-linear operations, which require an increase in size. After a ReLU6 activation, the second convolution is applied: one depthwise with a 3×3 kernel, followed by another ReLU6 activation, with the role of filtering the increased dimensions of the tensor. The third convolution

is a pointwise one that projects a large number of channels into a small size, equal to that of the input data.

After the third convolution, the information returned to a reduced space from a dimensional point of view, applying a non-linear activation would cause too much loss of information. Consequently, the linear activation $f(x) = x$ is used. If the feature maps are not downsampled, a skip connection can be made between the input and the output. Otherwise, no such connection is applied. The structure of this block is shown in Figure 2.10.

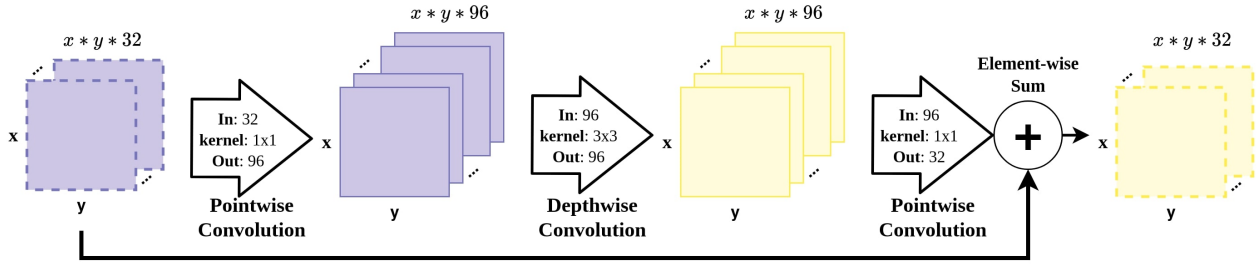


Figure 2.10: Example of an Inverted Residual Block with Linear Bottleneck.

2.5.6 Aggregated Residual Transformations Block (ResNeXt)

A modified version of the Residual Block, the ResNeXt [30] was introduced as a modernized version of the ResNet [29] block, with a multi-branch architecture that was able to aggregate a set of transformations that have the same topology. Two ResNet-inspired rules are employed: the first one is to produce same-size spatial maps and to maintain the same width and filter sizes between blocks, and the second rule is to multiply by 2 the width of the blocks each time that the dimensionality is halved. This second rule maintains complexity throughout the ResNeXt architecture. The thought process behind this refined block is that instead of a single convolution with a lot of filters, a grouped convolution is formed by splitting the original convolution filters into multi-branch groups of smaller convolutions. The number of groups is called cardinality.

In the example from Figure 2.11, the input features are passed through 16 branches. Each branch is comprised of a 1×1 convolution with 4 channels, meaning that instead of a convolution with $4 * 16 = 64$ channels, 16 convolution with 4 channels are done in parallel. Next on the branch, there is a 3×3 convolution that maintains the number of channels, followed by a 1×1 convolution with the initial number of input feature channels. The resulting 16 branches are added element-wise and then the result is added with the input feature map through a skip connection. For the first 2 convolutions on each branch, ReLU activation functions are used along with Batch Normalization. The third convolution is only followed by a Batch Normalization layer, maintaining the negative values for the first feature merge. After the skip connection, a ReLU activation is applied to the resulting feature map.

The ResNeXt block introduced a new hyperparameter for similar convolutional blocks – cardinality. The authors stated that „increasing cardinality is more effective than going deeper or wider when we increase the capacity" [30]. Moreover, they have empirically proven

that even when forced to maintain model complexity, increasing cardinality improved the classification results. As an observation, the authors provided equivalent blocks of ResNeXt. One alternative is to concatenate the feature maps from all branches after the second convolution, and then to pass the information to a single convolution that increases the concatenated feature map size to the input size, before finally adding the skip-connection. Another alternative is to do a grouped convolution [32].

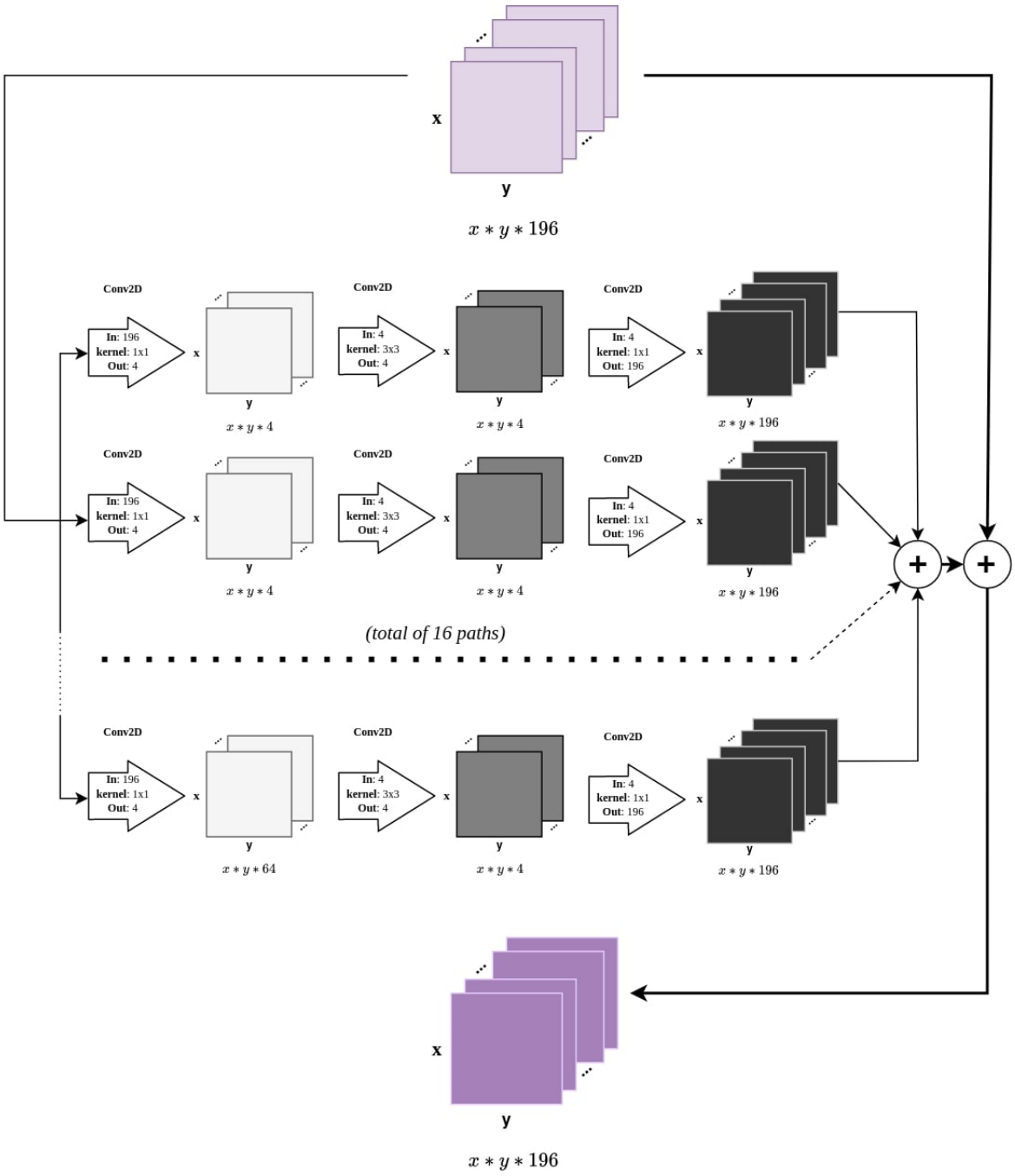


Figure 2.11: Example of a ResNeXt block with a cardinality of 16.

2.5.7 Attention and Transformers

Compared to convolutions, which have a sliding window approach on the whole image, human perception does not tend to process the entirety of a scene at once. Instead, the human perception focuses „selectively on parts of the visual space to acquire information [...] and combine information from different fixations over time [...] guiding future eye movements and decision making" [20]. In other words, the human brain does not process an entire visual snapshot all at once, but extracts information selectively from sequentially looking at a scene.

2.5.7.1 Sequence to Sequence Models

Sequence to Sequence models are defined by their task of converting sequences from one domain to sequences in another domain. Mostly used in the Natural Language Processing (NLP) field, these types of models are more popularly used for machine translation, generating answers to a question (chatbots), or text summarization. They can also be used in combination with images, to generate Image Captions. In NLP, Sequence to Sequence models got traction through the use of Recurrent Neural Networks [19].

On a higher level, Sequence to Sequence models are formed by two parts: an Encoder and a Decoder. A machine translation (Romanian to English) sequence to sequence model will be used as an architectural example, with each language having a different Embedding Space. In this case, the Embedding Space is usually a numerical representation of the words in a certain language. In classical approaches, the Encoder processes the entire input sequence and generates a context to the Decoder, which creates the output sequence. The encoded information, often called Context or State, is a fixed size tensor, independent on the input size. The ability to process entire sequences (split into multiple steps/timesteps) is given by the use of RNNs [19], which have a closed loop. In the RNN Encoder, the model processes recurrently on the next input and the intermediary context information until a final context representation is generated for the RNN Decoder, as seen in Figure 2.12.

This is one of the major limitations of a classical Sequence to Sequence model: the Encoder is forced to generate a single vector, independent on the size of the input. In this case there is a trade-off that has to be made when setting the context vector fixed size (the number of the hidden units in the RNN). A smaller size will have problems with long input sequences, while a bigger size will have problems with overfitting on short input sequences.

2.5.7.2 Attention Encoder and Decoder

A Sequence to Sequence model with attention mechanisms is also comprised of an Encoder and a Decoder.

Continuing the same example of machine translation as before, in an Attention model the Encoder works the same, processing and producing one hidden state at a time. Unlike the model without attention, the context that is generated by the encoder is composed not only by the last hidden state, but by all of the hidden states together. It is to be noted that hidden states produced at later timestamps will also contain some information about the previously generated hidden states. With the bigger context, the Decoder learns during the

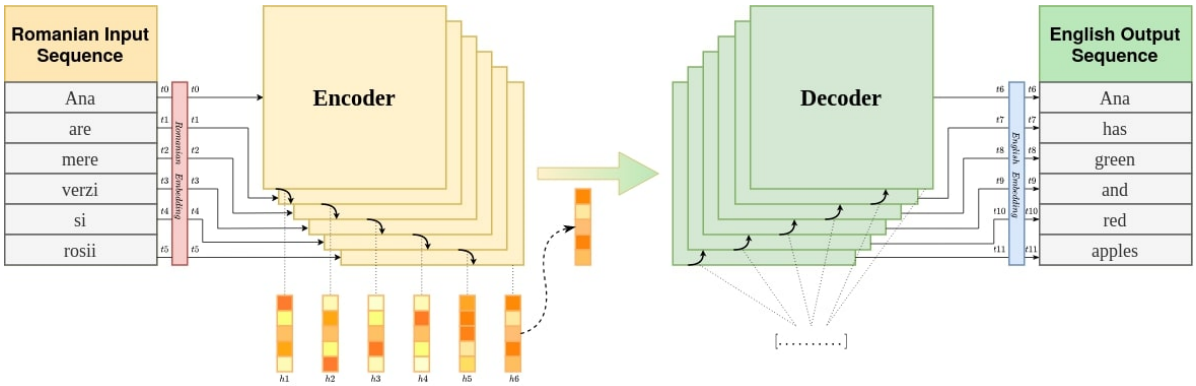


Figure 2.12: A Sequence to Sequence model with two RNNs for machine translation between Romanian and English.

training phase, with the help of a scoring function, on which parts of the context to pay most attention at a given timestep. This results in attention weight vectors, which are used amplify the most relevant parts of the input and to diminish the irrelevant parts.

There are two main types of attention: Additive Attention [21] – often called Bahdanau Attention, and Multiplicative Attention [22] – called Luong Attention.

2.5.7.3 Multiplicative Attention

As mentioned previously, Attention weights (known as the score vector) are computed at each timestep in the Decoder by the scoring function. The scoring function is usually taking as inputs: i) the hidden state of the Decoder at the current or previous timestamp and ii) the whole set of hidden states (the whole context) generated by the Encoder. The Multiplicative Attention [22] authors propose three different alternatives for the score function, as shown in (2.5.1).

$$score(\mathrm{h}_t, \overline{\mathrm{h}}_s) = \begin{cases} \mathrm{h}_t^\top \overline{\mathrm{h}}_s & dot \\ \mathrm{h}_t^\top \mathrm{W}_a \overline{\mathrm{h}}_s & general \\ \mathrm{v}_a^\top \tanh(\mathrm{W}_a \left[\mathrm{h}_t; \overline{\mathrm{h}}_s\right]) & concat \end{cases} . \tag{2.5.1}$$

Normally, $softmax$ is applied over the resulting scoring vector. Each element of the new $softmax$-ed scoring vector is multiplied with its corresponding Encoder vector. Finally, the resulting vectors are added, resulting in the Context Vector.

The most basic scoring function is the $dot\ product$ of the two input vectors, which produces a single number. Geometrically, the $dot\ product$ of only two vectors is computed by multiplying the lengths of the two input vectors by the cosine of the angle between them. With the intuition that the $dot\ product$ is used as a similarity measurement between two vectors of the same length, it can be inferred that the smaller the angle between them, the bigger the $dot\ product$ becomes. In practice, the $dot\ product$ is computed between the decoder hidden state h_t and all of the encoder hidden states $\overline{\mathrm{h}}_s$ at once, resulting in a scoring vector of single values with the length of the number of hidden encoder states. The $dot\ product$ is usually limited to having the same embedding space for the input and the output. This scoring function alone cannot be applied in machine translation, as normally the input language has a

different embedding space than the output language. A visual representation example of the $dot\ product$ operations is presented in Figure 2.13.

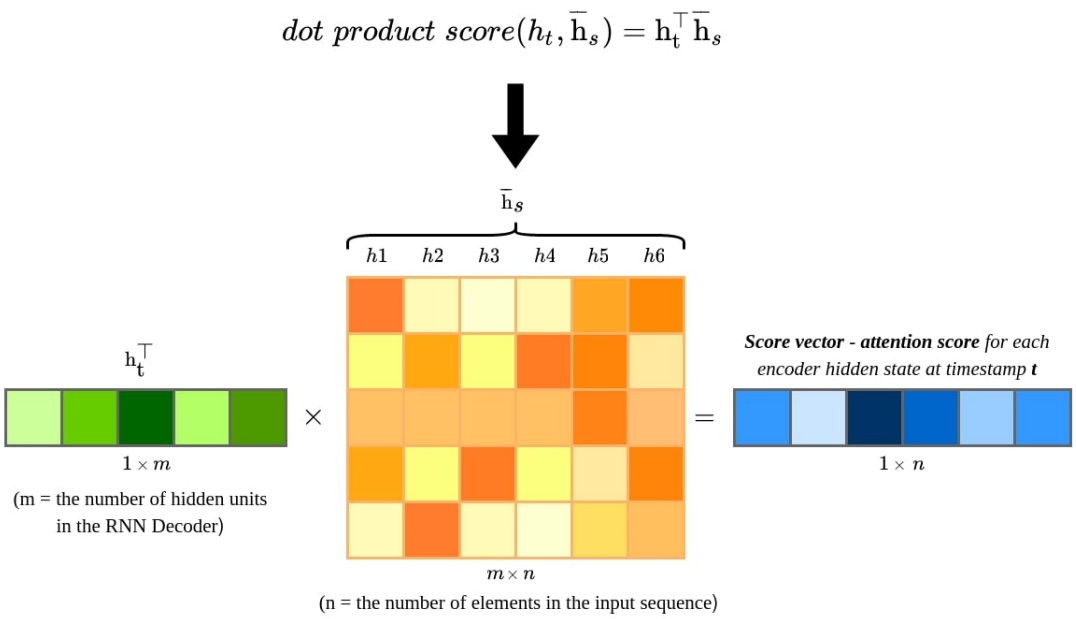


Figure 2.13: Example of Dot Product score operation.

The $general$ score function is similar to the $dot\ product$, introducing only a weight matrix W_a between the multiplication of the hidden state $\mathrm{h}_t^\top$ at a certain timestamp in the Decoder and the whole context $\bar{\mathrm{h}}_s$ of the Encoder. For machine translation, this solves the problem of having different embedding spaces. A visual representation example of the $general$ operations is presented in Figure 2.14.

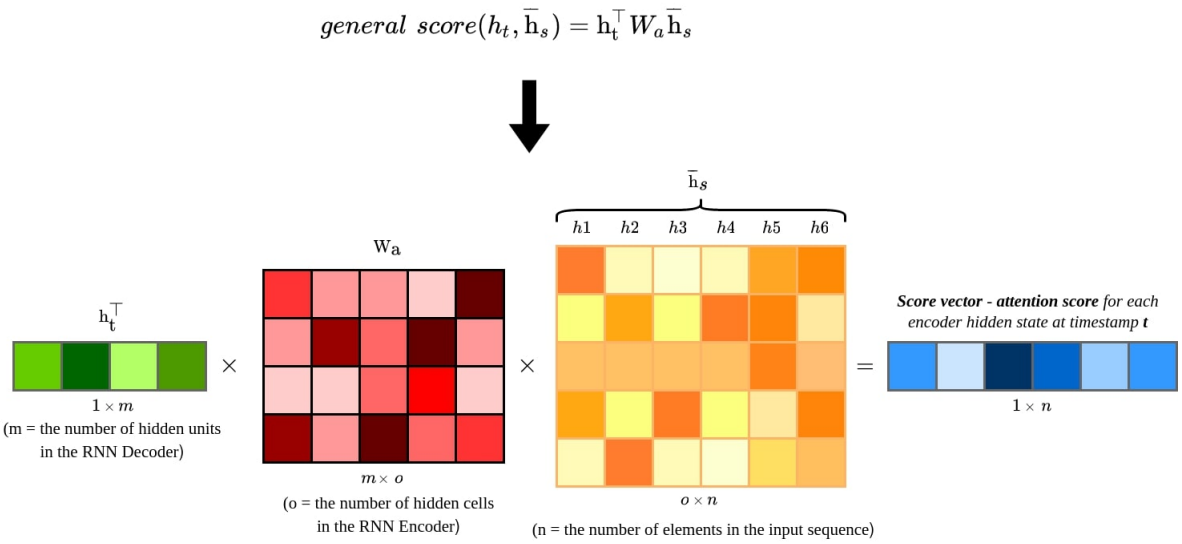


Figure 2.14: Example of General score operation.

The $concat$ score function concatenates the Encoder hidden state h_t at timestep t and the Encoder hidden states $\bar{\mathrm{h}}_s$, then it multiplies the resulting vector with a Weight matrix W_a. Finally, the result is put through a $tanh$ activation function, and then it is multiplied by

another learned parameter $\mathrm{v_a^\top}$. An example of the $concat$ score function at timestep t for a single Encoder hidden state indexed x is presented in Figure 2.15.

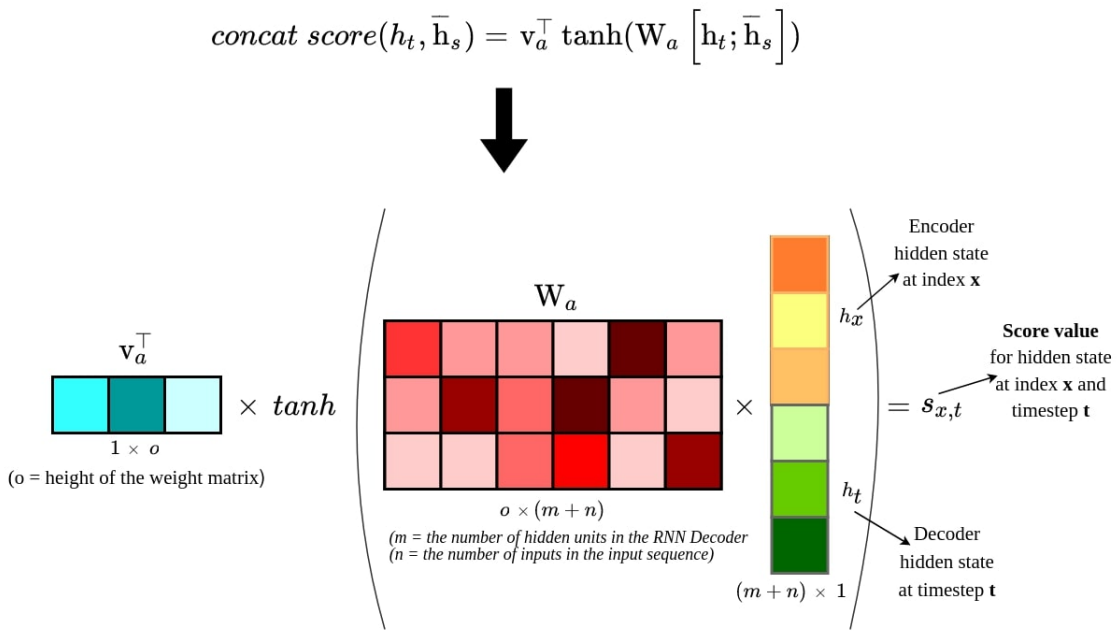


Figure 2.15: Example of Concat score operation for a single Encoder hidden state indexed x.

An example of a machine translation Sequence to Sequence model with Multiplicative Attention (with the $dot\ product$ as the scoring function) is presented in Figure 2.16. The processing steps are the following: 1) Encoder hidden states are computed, as well as the Decoder hidden state at timestamp $\boldsymbol{t}$; 2) the Score vector (or Attention weights) is computed by calculating the $dot\ product$ between the transposed Decoder hidden state $\mathrm{h}_t^\top$ and the Encoder states $\overline{\mathrm{h}}_s$; 3) the score vector is passed through a Sigmoid function; 4) the values of the new score vector are multiplied with their corresponding hidden states. The resulting vectors are added together, resulting in the Context Vector; 5) the Context Vector is concatenated with the Decoder state h_t, and then multiplied with the Weight matrix W_a of a Neural Network. Then, $\tanh$ is applied over the result; 6) the result is interpreted using the Output Embeddings, and a word is generated.

2.5.7.4 Additive Attention

The $concat$ scoring method from the Luong Attention [22] is very similar to the Bahdanau (Additive) Attention[21], with a few differences. The Additive Attention score formula is presented in (2.5.2).

$$score(\mathrm{h}_{t-1}, \overline{\mathrm{h}}_s) = \mathrm{v}_a^\top \tanh(\mathrm{W}_a \mathrm{h}_{t-1} + \mathrm{U}_a \overline{\mathrm{h}}_s). \tag{2.5.2}$$

One difference from the $concat$ attention formula is the fact that the Decoder hidden state h_{t-1} is from the previous timestep $(t-1)$ instead of the current timestep t. The other difference is the fact that the Weight matrix is split into two parts, one for the Decoder hidden state h_{t-1} and another for the Encoder hidden states$\overline{\mathrm{h}}_s$.

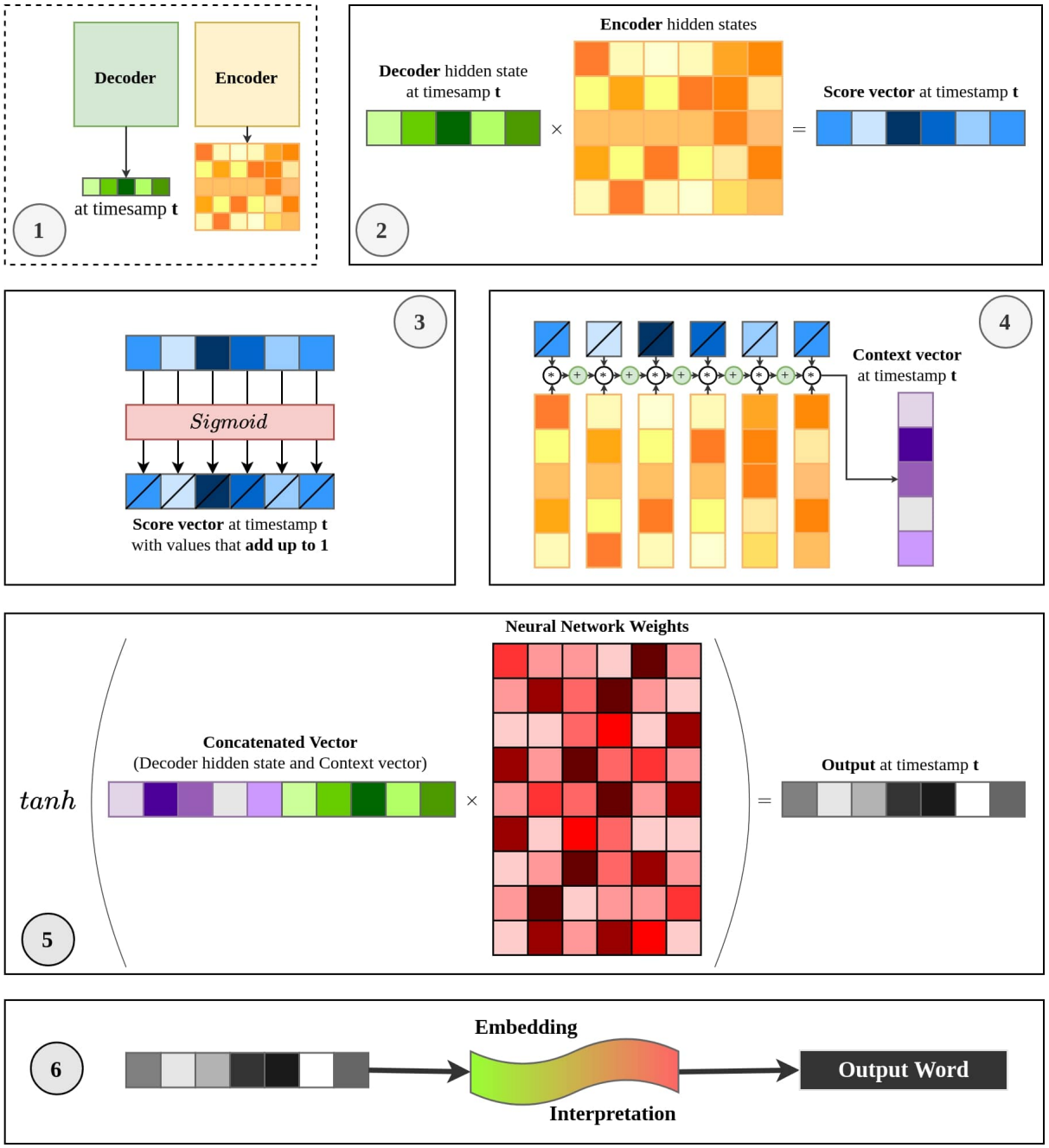


Figure 2.16: Steps for a machine translation model with $dot\ product$ as the scoring attention.

2.5.7.5 Transformers and Self-Attention

A landmark paper in the field is “Attention is all you need” [23] – that simplified Sequence to Sequence architectures by introducing a type of model that uses only Attention, without RNNs, that proved to be both faster and better performing than previous methods. This type of new model is called the Transformer. Compared to the RNN, which can only process the input sequence by feeding it one part at a time, the Transformer can process the whole input sequence in parallel. The main aspect about the Transformer architecture is the fact that they us Feed-Forward Neural Networks and Self-Attention.

The Transformer is composed of a stack of identical Encoders and Decoders, which allows parallelization. On a high-level view, the Encoder is made of a Multi-headed Self-

Attention layer and a Feed-Forward layer. Compared to previous solutions with Attention, which used attention mechanisms in the Decoder, the Transformer uses this new attention mechanism (self-attention) also in the Encoder. This helps the Encoder to better understand the current input by focusing on both previous and future parts of the input sequence that are relevant. The Decoder is composed of three components: a Masked Multi-headed Self-Attention layer that can only focus on previously decoded outputs, an Encoder-Decoder attention component that allows it to focus on previous and future relevant parts of the input as well as the previously generated outputs, and a Feed-Forward layer.

The Self-Attention mechanism relies on the use of three learnable layers: the query layer, the key layer, and the values layer. Each has a weight matrix and they represent different information to be learned about the word embeddings. The steps for preforming Self-Attention are: 1) generate the queries vector by multiplying the embedding layer with the query weight matrix; 2) generate the keys vectors for each word by multiplying the embeddings by the key weight matrix; 3) generate the values vectors by multiplying with the value weight matrix; 4) compute the score for all of the words (except the one it is processing) by comparing the query vector against the keys vectors (the word we are performing self-attention has a score of 1); 5) scale the scores and apply softmax on the resulting values. The layer is called Multi-headed because the attentions for each word in the process are independent from each other, so the transformer can look to multiple words in parallel. Normally, the resulting values from the Self-Attention layer are multiplied with the corresponding value vectors, added up and fed to the next layer.

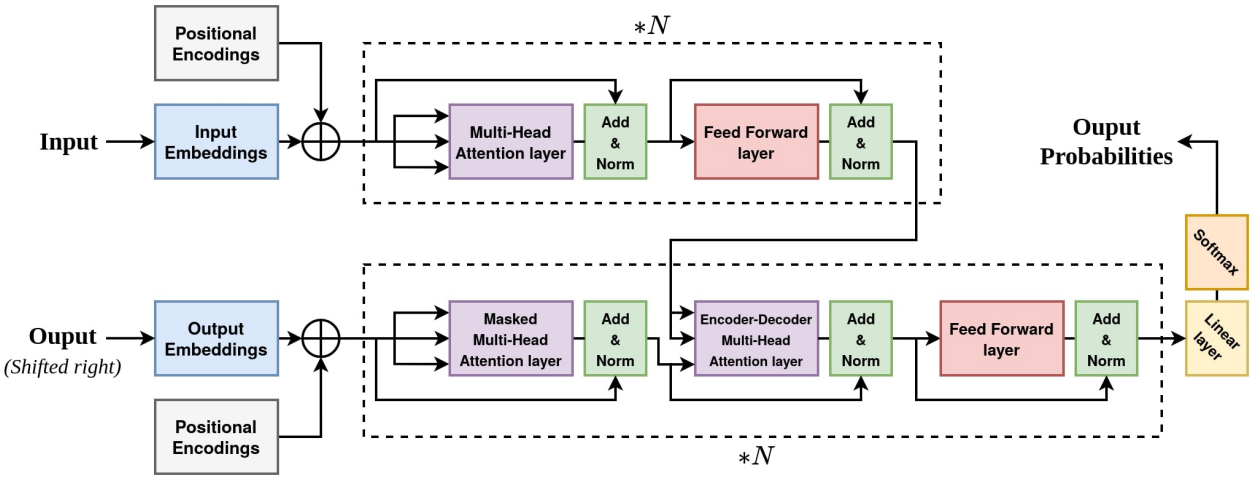


Figure 2.17: The Transformer Architecture.

The architecture of the Transformer is presented in Figure 2.17. First, it can be observed that the Input Embeddings are combined with Positional Encodings, which give context passed on the position that a word has in the sentence. Then, the new word embeddings are passed through the Encoder Multi-Head Attention layer and the Feed-Forward layer to generate a set of encoded vectors for every word in the input sequence. Next, the previous outputs are put through a different word embedding and combined with Positional Encodings. The resulting output embeddings are passed through the Decoder, which first

processes them with the Masked Multi-Head Attention layer. The masked property refers to the fact that there is no attention to output words that have not yet been generated, thus masking half of the attention matrix. Next, the information is combined with the encoded vectors and passed to the Encoder-Decoder Attention layer, which generates attention vectors for every input and output word embedding. Then, the information is put through a Feed-Forward layer. Finally, the resulting features are passed to another fully connected layer, which is the size of the output dictionary. Next, the generated vector is passed through a softmax activation to generate a probability distribution for the output word dictionary.

2.5.7.6 Vision Transformers

While the Transformer represents the standard for Natural Language Processing tasks nowadays, there were not a lot of applications for it in Computer Vision tasks. The Vision Transformer [24] (ViT) is an alternative to Convolutional Neural Networks, treating the input as a sequence of non-overlapping image patches. This new approach managed to perform comparable State of the Art results with much fewer computational resources during training.

An image can be interpreted as a sequence of pixels. For Sequence to Sequence models with Attention, each part of the input is scored against all of the other parts. Scoring against all the other parts is referred to as Global Attention, and scoring only against parts in the neighborhood is called Local Attention. For example, a sequence of length n will have n^2 attention comparisons. Applying this reasoning to an image of size $m \times m$, there are m^{2^2} pixel comparisons. Computations of such size for Global Attention are difficult to perform, and Vision Transformers introduced a solution for this – splitting the image into multiple parts (or patches).

The architecture of the Vision Transformer is presented in Figure 2.18. First, the image is split into patches that are unrolled and transformed into vectors of an arbitrary size C (Capacity). The vectors are multiplied by an Embedding matrix E, and combined with the Patch and Position Embeddings, which are learnable parameters that are used to give context to the position of the patches in the original image. There is a special input $*$ which is an extra learnable class embedding. Then, the information is passed to a standard Transformer Encoder, with a Feed-Forward layer (MLP) of size C. The output of the Transformer MLP Head corresponding to the learnable encoding $*$ is considered as the final result.

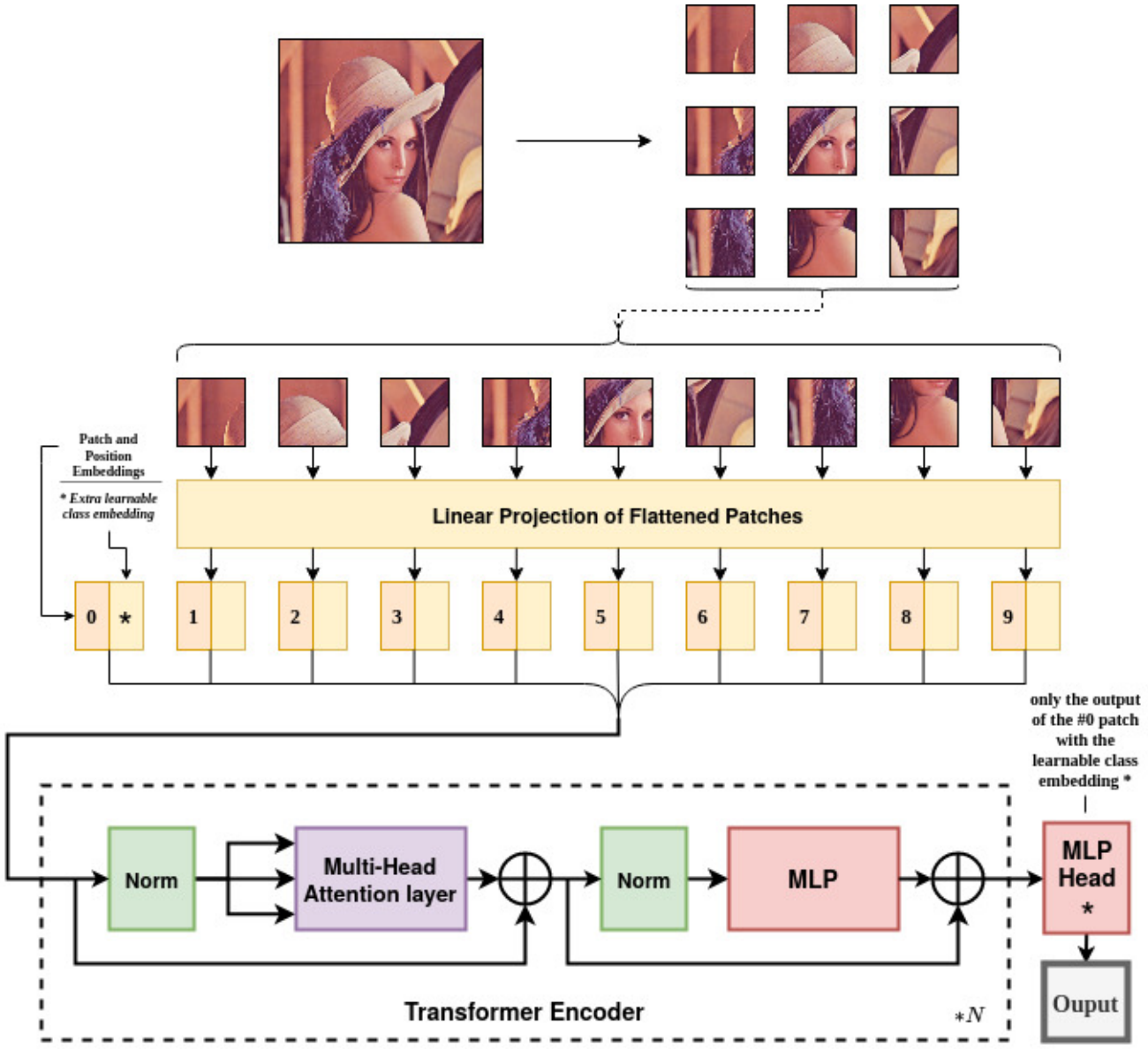


Figure 2.18: A Vision Transformer Classification architecture with 3×3 patches.

2.5.7.7 Swin Transformer

The Swin (Shifted Windows) Transformer [25] is a Vision Transformer variant with a Hierarchical approach to processing the image. This architecture got traction in the academic community, overcoming the State of the Art in many Computer Vision tasks such as Object Detection, Image Classification, and Semantic Segmentation. The Swin Transformer aimed to solve some of the problems with the Vision Transformers, such as not being able to analyze fine-grained features at higher resolutions, and being too computationally expensive to look at all the other patches in the image (if the resolution is high and the patch size is small).

The Swin Transformer starts by using smaller patches in the first Transformer layer, then it merges them into bigger patches into deeper layers, thus processing the image attention on multiple levels of features. Just like the Vision Transformer, the patches are unrolled, linearly

projected into vectors of an arbitrary size C, which also determines the size (capacity) of the Feed-Forward layer. The Transformer blocks processes the patches in a Shifted Windows manner, where the attention span for each patch is limited to N neighbors, thus drastically reducing the complexity of the self-attention mechanism.

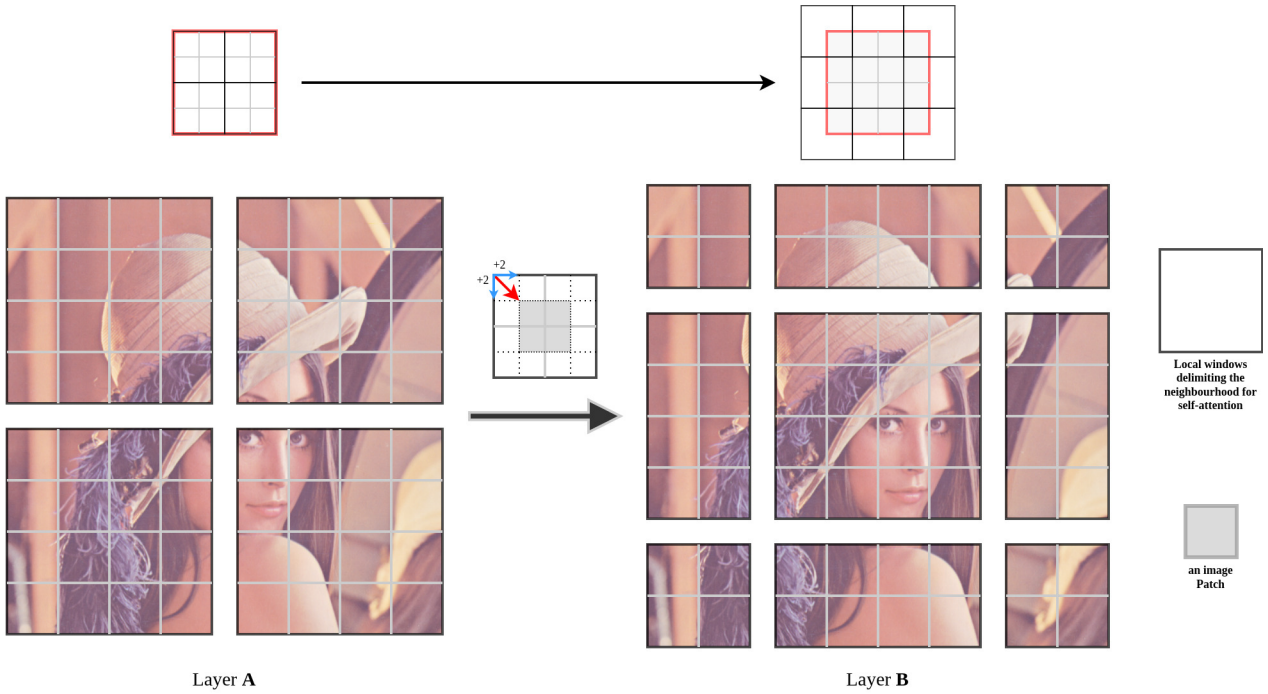


Figure 2.19: The Shifted Windows approach for self-attention. In Layer A the regular window partitioning is presented, with the respective neighborhood areas delimited. In Layer B, the neighborhood areas are shifted with 2 patches in both axes.

The outputted encoded vectors are then passed through a Merging Layer, which concatenates the vectors of 2×2 groups of neighboring patches from the input image into a single bigger patch of size $4 \times C$. This greater capacity allows information to be captured from a larger region. This resulting patch is in turn passed through a Linear Projection with its dimension reduced to a size of $2 \times C$. Then, the process is repeated until the merging process is not possible anymore, or earlier – depending on the architecture and image size. Each time the process is repeated and another pass is made through the Swin Transformer, the neighborhoods regions – for preforming limited (or local) self-attention – are shifted by two patches both vertically and horizontally. The resulting neighborhood areas are non-uniform and allow previous parts of the image to interact with other parts. This method is called the Shifted Windows approach for self-attention, and it is presented in Figure 2.19. There are multiple variations of the Swin Transformer architecture, that use different values for C and different number of transformer stacks. All of them can be used as independent backbones in other existing architectures, which proved to improve results over standard CNN backbones.

2.5.8 ConvNeXt

As Transformers [23, 24] started to sweep the Computer Vision landscape, the authors of ConvNeXt [26] aimed to revive the Convolutional approach by modernizing the ResNet [29] architecture with multiple improvements from both transformers and other existing Convolutional approaches.

Convolutions induce shifting equivariance, which refers to the fact that if the information in the input is shifted, then the convolution will also capture the information shift when outputting a feature map. In CNNs, pooling layers induce shift invariance, which means that if information is shifted in the input, applying pooling (e.g., max pooling) with a big enough pool size, the operation will output the same feature map as if the input information wasn't shifted, or will diminish the shifting effect. Without the inductive bias introduced by sliding window approaches in convolutions, the Transformers are not translation equivariant – mostly due to their global attention. This gave them more flexibility, but also limited them when inductive bias would have brought improvements. However, to achieve new State of the Art results, the Swin Transformer [25] inspired from the convolutional techniques and introduced a variant of sliding window with the Shifted Window Self-Attention approach. All these being said, the ConvNeXt architecture is an attempt to take inspiration from the Transformers – just as the Swin Transformer did with convolutions, and revive CNN approaches.

The ConvNeXt architecture was achieved by compiling a set if improvements, such as:

1. **Similar Tranining Techniques to the Transformer** – use of the AdamW optimizer, newer augmentation methods (e.g., RandAugment, Cutmix, Mixup, Random Erasing), better regularization (e.g., Label Smoothing and Stochastic Depth), and more training.

2. **ResNet restructuring** – adjusting the ratio of blocks in each stage, and modifying the standard ResNet stem cell (a 7×7 convolution with stride 2, followed by a pooling layer) to be more similar to the "Patchify" layer of the Swin Transformer, which uses smaller patches. The new stem cell is implemented with a 4×4 convolution with stride 4.

3. **ResNeXt-ifying the ResNet blocks** – using depthwise convolutions (a variant of grouped convolutions where the number of groups equal the number of channels). The depthwise convolution is quite similar with the weighted sum that is used in self-attention, operating on a per-channel basis. Furthermore, they increase the network width (64 channels) to the same number of channels (the variable C) presented in the Swin paper (96 channels).

4. **Using Inverted Bottlenecks** – just as the Swin Transformer took inspiration from the MobileNetV2 [31] Inverted Bottlenecks and merged its features to the size of $4C$, the ConvNeXt are using Inverted Bottlenecks with an expansion ratio of 4 to extract residual information from a higher-dimensional channel space and fit it to a smaller dimension channel space.

5. **Larger Kernel sizes** – Transformers are looking at the whole image to extract information. Swin Transformers use limited attention however, but the standard 3×3 convolutional kernel is too small to replicate that. However, increasing the kernel size

in the building blocks to 7×7 proved to be closer to the ability of Swin Transformers to use a higher space of local attention. Higher kernel sizes did not improve the results substantially.

6. **Modifying the micro design of the blocks** – using Layer Normalization instead of Batch Normalization, which is inspired from what RNNs did; reducing the number of normalizations – inspired from the Swin MLP blocks; using GELU activations instead of ReLU; allowing some linearity by using fewer activations – inspired from the Swin MLP blocks; using separate 2×2 downsampling convolutional layers after building blocks (compared to using the first convolution in a ResNet block with a stride).

With these improvements, the ConvNeXt has managed to overcome the power of Swin Transformers by giving insights as to how many improvements can still be made to the convolutional models, before they are overshadowed in popularity. The architecture of a ConvNeXt block compared to a ResNet block can be seen in Figure 2.20.

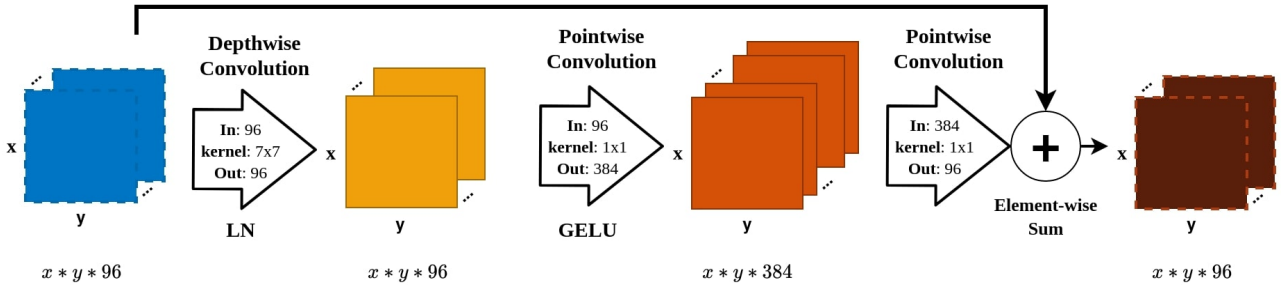


Figure 2.20: Example of a ConvNeXt block.

2.6 General Neural Network aspects

The performance of a Neural Network model depends on a multitude of aspects. These can refer to the overall composition of layers in the architecture, or to the training, validation, and testing processes.

2.6.1 Hyperparameters

There are multiple types of parameters in Deep Learning. First, there are the commonly learned parameters: model weights and biases, specific learnable parameters (such as β and γ in Batch Normalization), and others. Then, there are the arbitrary parameters which are set in the beginning. These are called Hyperparameters, and they can define training, validation, and testing processes, the architecture, data distribution, hardware usage, etc. Some hyperparameters can be adaptable and even trained for a better performance. Some general Hyperparameters are:

- Learning rate – refers to the scaling of gradients in the weight update process. The learning rate can vary during training, depending on the chosen optimization method.

- Architectural parameters – in some cases (e.g., ResNet), the architecture can vary in terms of the building blocks used. The architectural parameters define the structure of the model.
- Optimizer parameters – depending on the optimizer, they define how aggressive or passive the process of cost function minimization is.
- Batch size – the training hardware is limited in memory and computational resources. The batch size refers to the number of samples that are trained at a single iteration.
- Epochs – since Neural Networks require intensive training, the epochs denote the number of times a model trains over the whole dataset.
- Max iterations – alternative to the epochs, Max iterations refer to the number of batches that are processed during the training process.
- Other parameters – there are multiple custom parameters that can refer to the output's structure, logging, and many other aspects.

The hyperparameters are usually discovered empirically by the authors by following the training process closely. For a more comprehensive research, the logging process needs to be clearly defined and the use of an experimentation backlog API is recommended (e.g., Weight and Biases), where the results for each set of hyperparameters can be linked and observed.

2.6.2 Activation Functions

The Activation Functions stand at the core of Neural Network models. They are mathematical equations that filter information that passes through them, and most importantly, they add non-linearity in the model. Mathematically speaking, to apply Gradient Descent algorithms, the activation function should be differentiable for all possible input values. The activation functions relevant to this work are the following:

- **Sigmoid** – used for binary classification, it outputs values between 0 and 1. Its graph can be observed in Figure 2.21(left).

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}. \tag{2.6.1}$$

- **Softmax** – the general form of Sigmoid. Its outputs sum up to 1. This is a core function for multi-class classification, which means that it also applies to Image Segmentation – which is a pixel-level classification task.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}. \tag{2.6.2}$$

- **ReLU** (Rectified Linear Unit) [38] – it is the most popular activation function, due to its simplicity – which leads to fast computations. It partially solves the vanishing gradient problem, and stands at the base of most new activation functions. In a few words, ReLU lets positive information pass through, while reducing all negative values to 0. Being unbounded above means that in the case of large values, the output will not be saturated to a maximum value (such is the case with Sigmoid), making the gradient non-zero for any positive value. Its graph can be observed in Figure 2.21(right).

$$\text{ReLU}(x) = \begin{cases} x, & x > 0 \\ 0, & \text{otherwise} \end{cases}. \tag{2.6.3}$$

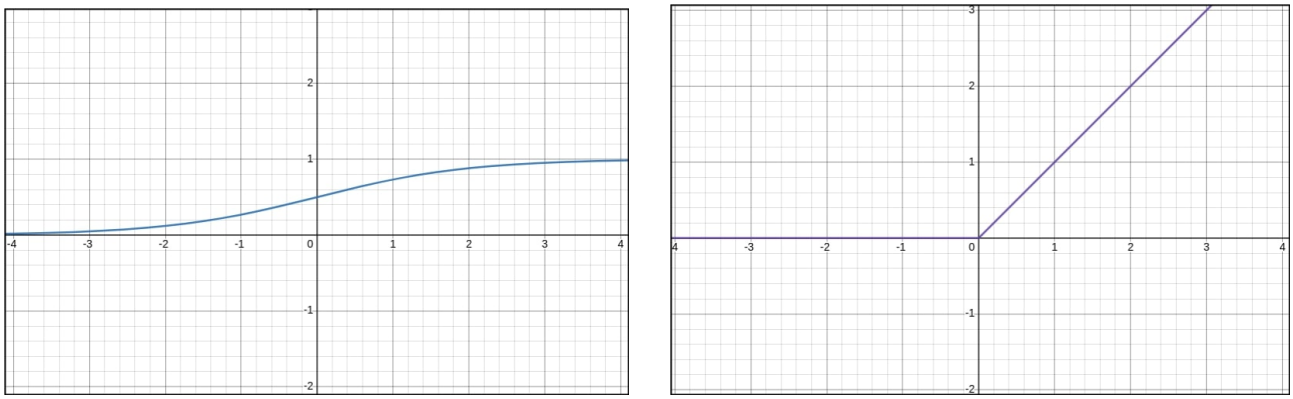


Figure 2.21: Sigmoid activation (left) and ReLU activation (right).

- **Leaky-ReLU** (Leaky Rectified Linear Unit) [39] – a brute solution to the "dying ReLU" problem (Subsection 2.8.3), it treats positive solutions the same as ReLU, but it also takes negative values into consideration in order to avoid getting neurons stuck on values of 0. The problem with Leaky ReLU is that, as values become greater negatives, the activation functions gives them more importance, which can lead to slower training. Its graph, with multiple values for α – which denote the impact of negative values, can be observed in Figure 2.22(left).

$$\text{LeakyReLU}(x) = \begin{cases} x, & x > 0 \\ \alpha x, & \text{otherwise} \end{cases}. \tag{2.6.4}$$

- **ELU** (Exponential Linear Unit) [40] – similarly to Leaky-ReLU, the ELU activation also treats negative values, but limits the effect of negative values to a threshold α. Its graph, with multiple values for α, can be observed in Figure 2.22(right).

$$\text{ELU}(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}. \tag{2.6.5}$$

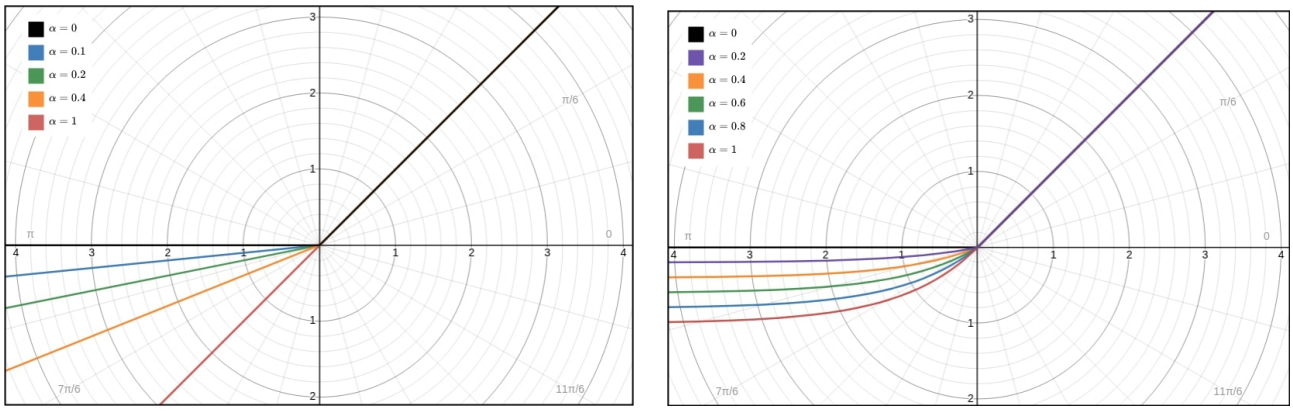


Figure 2.22: Leaky-ReLU activation (left) and ELU activation (right).

- **Swish** [41] – a function similar to ReLU, but that is non-monotonic in nature, presenting a degree of fluctuation for negative values. Similarly to ELU, it is bounded in the negative domain – meaning that as input values reach $-\infty$, the output will converge to some small non-zero constant. In other words, Swish deals with moderate negative values, and tends to forget very large negative values, leading to deactivations. The downside of this activation function is that it's very computationally expensive. This feature has a regularizing effect, and can be fine-tuned through a β parameter. When β is 0, Swish becomes a linear function. Then β tends towards ∞, Swish becomes ReLU. The default value of β is 1. Its graph, with multiple values for β, can be observed in Figure 2.23(left).

$$\text{Swish}(x) = x * \text{Sigmoid}(\beta x). \tag{2.6.6}$$

- **Mish** [42] – inspired by Swish, it has the same properties, with the addition of it being continuously differentiable with infinite order. It also has the same disadvantage as Swish, with it being computationally expensive when used in bigger architectures. The graph of the function can be observed in Figure 2.23(right).

$$\text{Mish}(x) = x * \tanh(\ln(1 + e^x)). \tag{2.6.7}$$

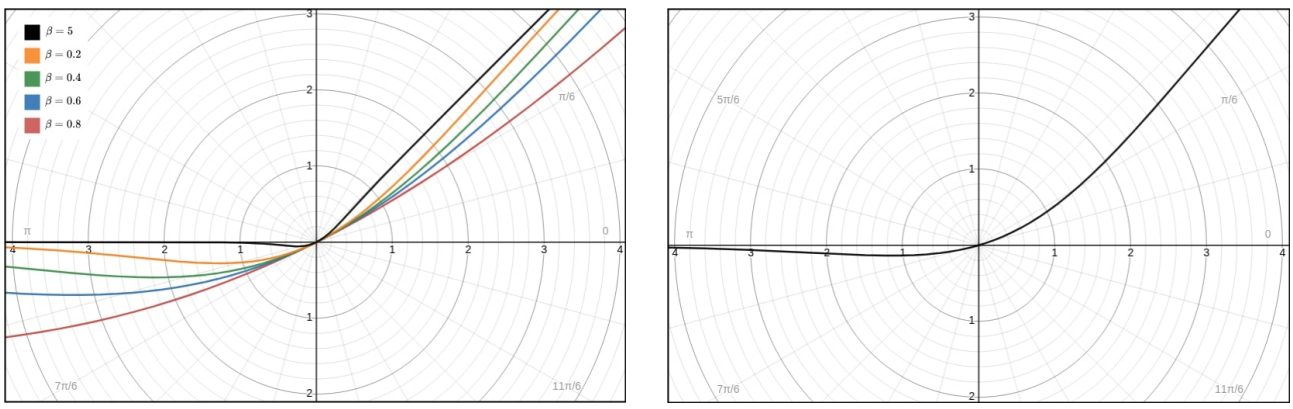


Figure 2.23: Swish activation (left) and Mish activation (right).

- **SmeLU** [43] – viewed as an alternative to GELU, the Smooth ReLU activation function splits delimits three zones with the β parameter. For values smaller than $-\beta$ and bigger than β, SmeLU behaves exactly like ReLU. Between $-\beta$ and β however, a quadratic equation is employed to create a smoother transition between the two delimiting zones, constraining continuous gradients. SmeLU can be interpreted as a convolution of ReLU with a box, with it being rather fast in its implementation. To maintain better linearity, the transition zone delimited by β must not be too smooth. Its graph, with multiple values for β, can be observed in Figure 2.24(left).

$$SmeLU(x) = \begin{cases} 0, & x \leq \beta \\ \frac{(x+\beta)^2}{4\beta}, & |x| \leq \beta \\ x, & x \geq \beta \end{cases} . \tag{2.6.8}$$

- **GELU** [44] – known as the Gaussian Error Linear Unit, GELU gained traction with attention solutions in NLP, most popularly in the Transformer architectures. Composed with the standard Gaussian cumulative distribution function, the GELU activation is also non-monotonic and has a nonlinearity that gates weight inputs by their percentile instead of their sign (compared to ReLU). It's formula has quite some interesting constants, and is sufficiently fast in its current code implementations. Its graph can be viewed in Figure 2.24(right).

$$GELU(x) = 0.5x(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3))). \tag{2.6.9}$$

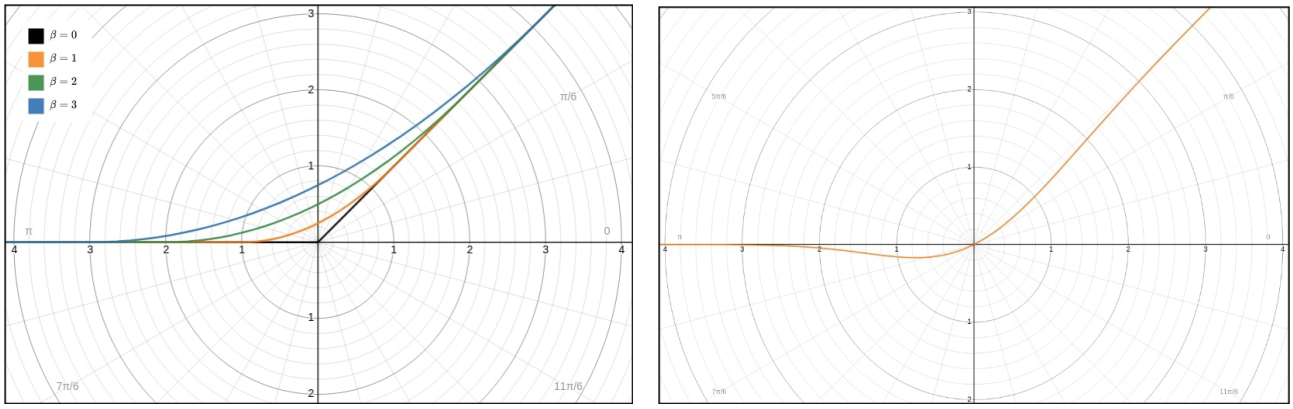


Figure 2.24: SmeLU activation (left) and GELU activation (right).

2.6.3 Loss Functions

Minimizing the loss function is the main objective of a Deep Learning model during training. The loss function is a way of evaluating the performance of our model for the available data. In the case of supervised learning, the loss function is computed by measuring the error of the prediction from the true answer. In other words, the model weights are optimized

towards a smaller value for the loss function. There are two main loss functions used in PaNeXt: the Focal Loss [45] and the Dice Loss [46].

A commonly used Loss function in Classification problems is the Cross-Entropy Loss, as shown in (2.6.10):

$$CrossEntropyLoss = -\sum_{j=1}^{N} gt_j \ln(p_j). \tag{2.6.10}$$

In equation (2.6.10): N is the number of classes, gt_j is the class indicator (or the true class) at j, and p_j is the predicted probability.

The Focal Loss [45] is an improved variant of the Cross-Entropy Loss that treats class imbalance by assigning more weights to hard or easily misclassified objects. In other words, the Focal Loss reduces contributions from easy samples and increases the impact for correcting misclassified examples. It implements a scaling factor that decays to zero as confidence in the groundtruth class increases. This factor is introduced as $(1-p_j)^\gamma$ to the standard Cross-Entropy Criterion. A value of $\gamma > 0$ reduces the relative loss for correctly-classified examples of probability $p_j > 0.5$. The formula for the Focal Loss for one example j is presented in (2.6.11).

$$FocalLoss(p_j) = -(1-p_j)^\gamma \ln(p_j). \tag{2.6.11}$$

The Dice Loss [46] (or Dice Coefficient) that is used in this work has the role of optimizing the predicted segmentations, and is similar to the Intersection-over-Union heuristic. The Dice Loss was also introduced to improve the Cross-Entropy Loss – which, in the context of segmentation, calculated the average of per-pixel loss. The per-pixel loss is computed discretely, without knowing if adjacent pixels are boundaries. In other words, the loss is not considered in a global approach – which facilitates image segmentation.

$$DiceCoefficient = 1 - \frac{1 + 2y\widehat{y}}{1 + y + \widehat{y}}. \tag{2.6.12}$$

In equation (2.6.12): y represents the target segmentation mask and $\widehat{y}$ is the predicted segmentation mask.

2.6.4 Metrics

For Segmentation problems, a popular metric is the mean Intersection-over-Union. The formula for mIoU is presented in (2.6.13).

$$mIoU = \frac{1}{N} \sum_{c=1}^{N} \frac{TP_c}{TP_c + FP_c + FN_c}. \tag{2.6.13}$$

For Object Detection problems, the mean Average Precision metric is commonly used. Average Precision represents the area under the Precision-Recall curve. The formula for the standard Average Precision is presented in equation (2.6.14), where c is the class, p represents the Precision ($\frac{TP}{TP+FP}$), and r is the Recall ($\frac{TP}{TP+FN}$). The mean Average Precision metric is the average AP over all the classes, at different arbitrary thresholds of confidence.

Its formula is presented in equation (2.6.15).

$$AP_c = \int_0^1 p_c(r_c)dr, \tag{2.6.14}$$

$$mAP = \frac{1}{N}\sum_{c=1}^{N} AP_c. \tag{2.6.15}$$

The problem of Panoptic Segmentation can be split into 2 subtasks: Semantic Segmentation and Instance Segmentation (by leveraging Object Detection). The metrics introduced for these are called Segmentation Quality (SQ) and Recognition Quality (RQ). Combining the two results in the novel Panoptic Quality metric [14], that is popular for this type of segmentation. Its formula is reproduced from the original paper [14] and is presented in Equation (2.6.16), where g is the groundtruth segment and p is the predicted segment.

$$PQ = \underbrace{\frac{\sum_{(p,g)\in TP} IoU(p,g)}{|TP|}}_{Segmentation\ Quality\ (SQ)} \times \underbrace{\frac{|TP|}{|TP| + \frac{1}{2}|FP| + \frac{1}{2}|FN|}}_{Recognition\ Quality\ (RQ)}. \tag{2.6.16}$$

2.6.5 Other common layers

There are a multitude of layers commonly used in Neural Network Architectures, that have a wide variety of roles.

The pooling layer is usually used to reduce the dimensions of a feature map. It's implemented in a sliding window approach, with two popular variants: average pooling – which outputs the average integer value within the sampling window; max pooling – which outputs the maximum value from the sliding window.

Another popular layer used in Neural Networks is the Batch Normalization layer. Its purpose is to increase stability and smooth out the features by normalizing the input values x. These are normalized by subtracting the average and dividing by the standard deviation, resulting in $\hat{x}$. This process has the role of minimizing the outlier effect on the other features. The two learnable parameters of Batch Normalization are $\gamma - (gamma)$ and $\beta - (beta)$.

Alternatives to the Batch Normalization are the Group Normalization [48] – which performs normalization over groups of channels for each input sample. Combining all channels into a single group results in Layer Normalization [47] – which only normalizes across features instead of the batch dimension. A visual representation of the three is presented in Figure 2.25.

2.7 Optimizers

In the field of Deep Learning, the optimizer is the algorithm that updates the trainable parameters of a neural network, in order to reach the global minimum of the cost function. Choosing the optimizer and adjusting its hyperparameters is essential in applying neural networks.

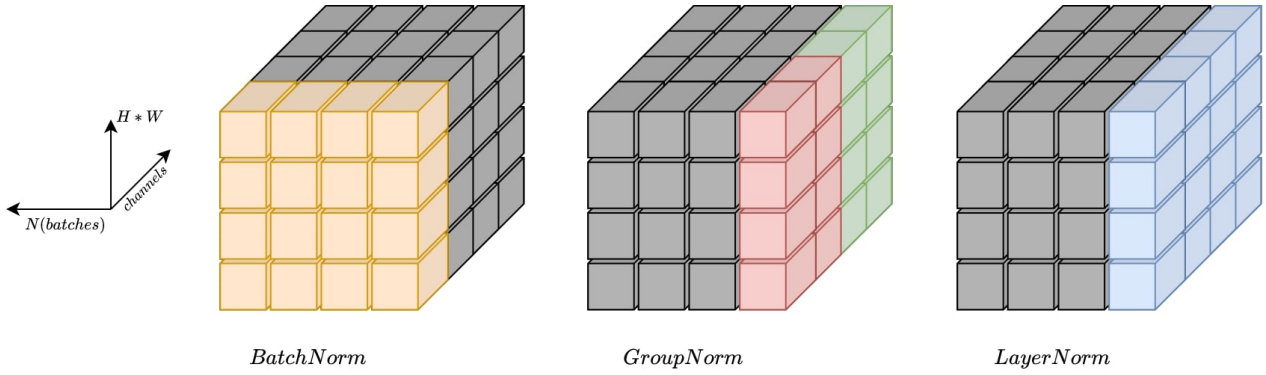


Figure 2.25: Visual representation for Batch Normalization (left), Group Normalization (center), and Layer Normalization (right).

2.7.1 Batching variants of Gradient Descent

Stochastic Gradient Descent [49] (or SGD) is an approximation of the Gradient Descent Optimization process. SGD computes the error gradient and updates the weights using them, for each sample in the dataset. The stochastic aspect is defined by the randomness employed when picking examples from the dataset, in order to estimate the values of the gradients.

Batch Gradient Descent is a variation that updates the error after each example in the dataset, but only updates the weights after all of the data has been iterated.

Mini-Batch Gradient Descent comes with a solution to the Batch Gradient Descent – instead of calculating the gradients for each example until updating the weights once, the mini-batch approach splits the dataset into multiple batches, and the error and weights are updated for each batch. The particularities of Gradient Descent are presented in Section 2.4.1.

2.7.2 Momentum

Momentum [50] is an improvement to the Gradient Descent algorithm. Apart from the current gradient, Momentum considers past gradients and uses an exponentially decreasing parameter β to set the impact for them in the current optimization process. This approach can be interpreted as having an exponentially decreasing weighted local average of the gradients that are used to update the current weights. The steps for Momentum are displayed in Algorithm 2.1.

2.7.3 Root Mean Square Propagation (RMSProp)

The Root Mean Square Propagation [52] (or RMSProp) shares the learning rate with a local average of the gradients' magnitudes for each weight. There are two parameters introduced by RMSProp: a forgetting factor γ and a scalar ϵ to prevent division by 0. The steps for RMSProp are displayed in Algorithm 2.2.

Algorithm 2.1 The steps for the SGD with Momentum Optimizer.

for iteration i:

compute gradients $\nabla E_i(\omega)$ and $\nabla E_i(b)$ of mini-batch i
compute local averages $\Delta\omega$ and Δb:

$$\Delta\omega \leftarrow \beta\Delta\omega + (1-\beta)\nabla E_i(\omega)$$
$$\Delta b \leftarrow \beta\Delta b + (1-\beta)\nabla E_i(b)$$

update the weights ω^* and biases b^*:

$$\omega^* \leftarrow \omega - \alpha\Delta\omega$$
$$b^* \leftarrow b - \alpha\Delta b$$

end for
return ω^*, b^*

Algorithm 2.2 The steps for the RMSProp Optimizer.

for iteration i:

compute gradients $\nabla E_i(\omega)$ and $\nabla E_i(b)$ of mini-batch i
compute the local averages of gradient magnitudes $\Theta\omega$ and Θb:

$$\Theta\omega \leftarrow \gamma\Theta\omega + (1-\gamma)(\nabla E_i(\omega))^2$$
$$\Theta b \leftarrow \gamma\Theta b + (1-\gamma)(\nabla E_i(b))^2$$

update the weights ω^* and biases b^*:

$$\omega^* \leftarrow \omega - \alpha\frac{\nabla E_i(\omega)}{\sqrt{\Theta\omega+\epsilon}}$$
$$b^* \leftarrow b - \alpha\frac{\nabla E_i(b)}{\sqrt{\Theta b+\epsilon}}$$

end for
return ω^*, b^*

2.7.4 Adaptive Moment Estimation (Adam)

The Adaptive Moment Estimation [53] (or Adam) optimizer is the combination of Momentum and RMSProp with Gradient Descent. Furthermore, Adam adds a bias correction method to decrease the big errors in the initial training steps. The steps for Adam are displayed in Algorithm 2.3.

2.7.5 Rectified-Adam (RAdam)

Rectified-Adam [54] is an improved version of Adam that introduces a warm-up for the learning rate. In other words, the learning rate is adaptable – meaning that in the early stages, during the warm-up period, the learning potential is limited.

In the process of minimizing the cost function, thus finding the global minima, there are a lot of local minima where the optimizer can become stuck. Usually, an optimizer makes bigger learning steps in the beginning, and starts to decrease the step size when it gets close to a minimum. However, if the initial steps are random, without knowing “the direction towards the global minimum”, they can actually achieve the opposite effect of learning.

Algorithm 2.3 The steps for the Adam Optimizer.

for iteration i:

compute gradients $\nabla E_i(\omega)$ and $\nabla E_i(b)$ of mini-batch i (with $L2$)
compute $\Delta\omega, \Delta b, \Theta\omega$ și Θb (from previous Algorithms):

$\Delta\omega \leftarrow \beta\Delta\omega + (1-\beta)\nabla E_i(\omega)$
$\Delta b \leftarrow \beta\Delta b + (1-\beta)\nabla E_i(b)$
$\Theta\omega \leftarrow \gamma\Theta\omega + (1-\gamma)(\nabla E_i(\omega))^2$
$\Theta b \leftarrow \gamma\Theta b + (1-\gamma)(\nabla E_i(b))^2$

do bias correction:

$\Delta\omega^{corect} \leftarrow \frac{\Delta\omega}{1-\beta^i}$
$\Delta b^{corect} \leftarrow \frac{\Delta b}{1-\beta^i}$
$\Theta\omega^{corect} \leftarrow \frac{\Theta\omega}{1-\gamma^i}$
$\Theta b^{corect} \leftarrow \frac{\Theta b}{1-\gamma^i}$

update the weights ω^* and the biases b^*:

$\omega^* \leftarrow \omega - \alpha\frac{\Delta\omega^{corect}}{\sqrt{\Theta\omega^{corect}}+\epsilon}$
$b^* \leftarrow b - \alpha\frac{\Delta b^{corect}}{\sqrt{\Theta b^{corect}}+\epsilon}$

end for
return ω^*, b^*

Minimizing the learning steps in the beginning ensures that the "learning direction" is known before performing larger learning steps.

To achieve this, a maximal value ρ_∞ is computed together with the value for ρ_k – the exponentially weighted and corrected local average, with the formulas (2.7.1) and (2.7.2). The rectification term r_k for iteration k is defined by the formula (2.7.3). The rectification parameter is used until $\rho_k \leq 4$ is true. The value of 4 was found empirically. Moreover, if $\rho_\infty \leq 4$ and $\beta \leq 0.6$, Rectified-Adam downgrades into SGD with Momentum.

$$\rho_\infty = \frac{2}{1-\beta} - 1, \tag{2.7.1}$$

$$\rho_k = \rho_\infty - \frac{2k\beta^k}{1-\beta^k}, \tag{2.7.2}$$

$$r_k = \sqrt{\frac{(\rho_k - 4)(\rho_k - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_k}}. \tag{2.7.3}$$

2.7.6 AdamW

Another improved version of Adam, AdamW [55] is a stochastic method that modifies the weight decay in regular Adam by decoupling weight decay from the gradient update. Weight decay is a type of regularization that reduces the chance of overfitting. In the Adam algorithm, the $L2$ regulating factor $\lambda\omega_i$ was added to the gradients $\nabla E_i(\omega)$ and $\nabla E_i(b)$. In

AdamW, this is added in the weight updating step, as seen in formula (2.7.4). The same operation is done for the bias b^*.

$$\omega^* \leftarrow \omega - \alpha \left(\frac{\Delta\omega^{corect}}{\sqrt{\Theta\omega^{corect}} + \epsilon} + \lambda\omega_i \right). \tag{2.7.4}$$

2.7.7 Lookahead

Lookahead [56] is a macro optimizer, meaning that, in order to update the weights, it uses other optimizer methods (SGD by default).

It is defined by two types of weights: slow weights ϕ and fast weights θ. Using a memorized starting point ϕ_t at step t, Lookahead performs k fast steps with SGD, computing the fast weights θ. After the k steps, we take the last fast weight θ_{k+5}, and perform a partial learning state from ϕ_t to θ_{k+5}. The size of the step is determined by a hyperparameter α. By "looking ahead" with another optimizer, Lookahead manages to perform a partial learning step towards the new direction. This approach has proven to achieve faster learning in certain Computer Vision applications, being more effective in the later stages of learning, where smaller learning steps are necessary, and overshooting becomes a problem. The steps for the Lookahead are displayed in Algorithm 2.4. A visual example is presented in Figure 2.26.

Algorithm 2.4 The steps for the Lookahead Optimizer.

initialize ϕ_0, k and α
initialize the loss L and the fast optimizer SGD
for slow steps $t = 1, 2, \ldots$:
    synchronize $\theta_{t,0} \leftarrow \phi_{t-1}$ parameters
    for fast steps $i = 1, 2, \ldots$:
        pick mini-batch d
        update the fast weights $\theta_{t,i} \leftarrow \theta_{t,i-1} + SGD(L, \theta_{t,i-1}, d)$
    end for
    update the slow weights with a partial step $\phi_t \leftarrow$
    $\phi_{t-1} + \alpha(\theta_{t,k} - \phi_{t-1})$
end for

2.7.8 Ranger / Ranger21

Ranger and Ranger21 [57] are variations of the Lookahead optimizer. Ranger uses Rectified-Adam instead of SGD, while Ranger21 uses AdamW together with other improvements such as: adaptive gradient clipping, gradient normalization, stable weight decay, linear learning rate warm-up (to achieve similar effects to Rectified-Adam), and others. Ranger21 has been proved to achieve better and faster results than other optimizers in the same timeframe.

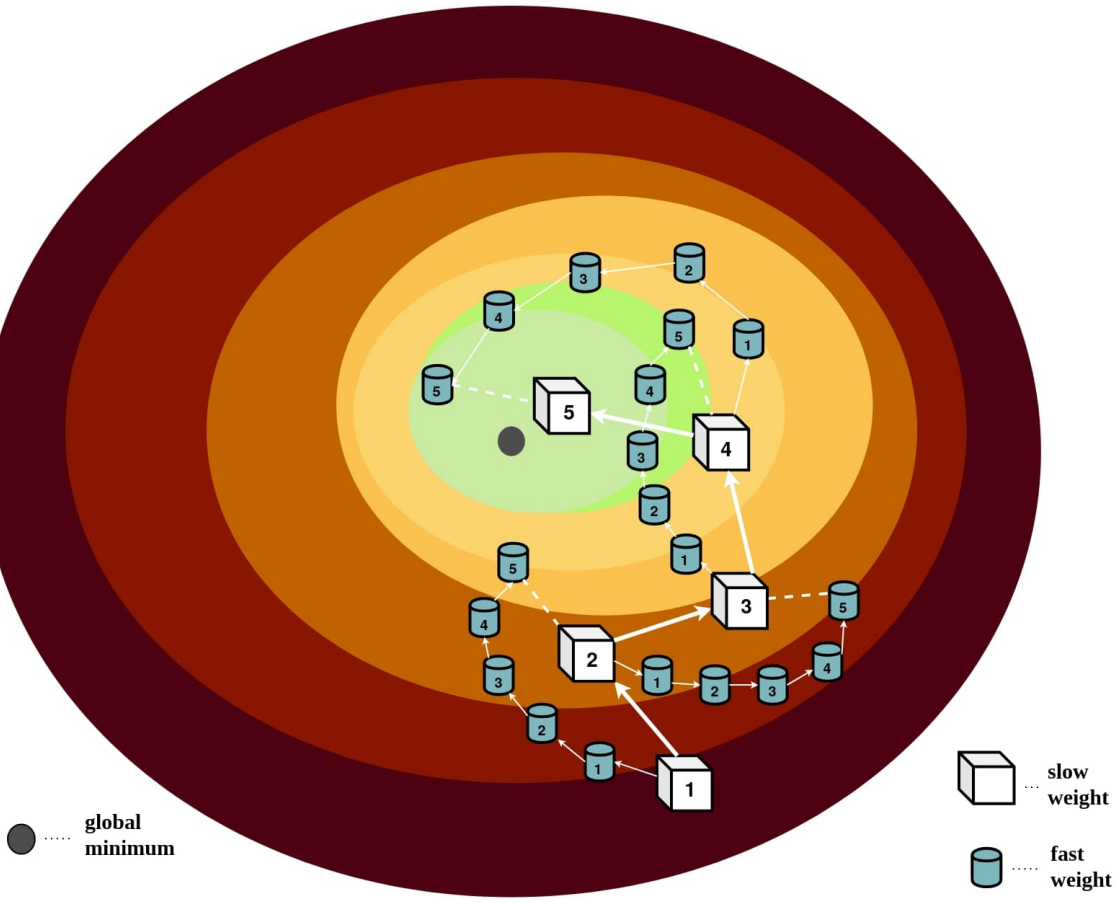


Figure 2.26: Performing 25 fast steps and 5 slow steps with the Lookahead optimizer.

2.8 Neural Network problems and solutions

Neural Networks are complex models that can reach up to billions of parameters, tens of tunable hyperparameters and many types of mathematical principles applied as layers. This wide of a variety comes with numerous possible problems that can appear during the training process. A few of those problems will be covered in this work, along with possible solutions.

2.8.1 Underfitting and Overfitting (Bias and Variance)

The Variance measures how much a prediction can vary for an unseen input instance during the training process. If the Variance is large, then the model has learned the dataset too well during training and cannot generalize to new data. This phenomenon is called Overfitting and is manifested by a small error during training and a large error during testing.

On the other hand, Bias represents the degree of simplicity that the model applies in its predictions, like some predetermined assumptions for the data. In the case of a significant bias, the model makes many assumptions about the desired result without considering the features present in the dataset, thus simplifying the solution too much and failing to make a reasonable assumption. This consequence is called Underfitting and is manifested in a large error for both training and test data.

Therefore, a Bias-Variance trade-off has to be made in order to obtain a solution that has low Variance or low Bias. Usually, it is easier to detect Underfitting simply by observing the training loss. However, Overfitting is a more costly error, as, during training, the loss cannot always show signs of Overfitting. The Validation process is important because it reflects the model's performance on new data and catches Overfitting patterns earlier.

The problem of Overfitting occurs typically late into the training process. In order to detect and avoid this problem, it is a good practice to keep a history of the error during training as well as during testing. Early-stopping is a simple solution which detects if the model starts to Overfit and stops the whole training process early. However, the occurrence of a local minima can give the impression of a global minimum. Early-stopping is not optimal, and other methods exist for avoiding Overfitting.

2.8.2 Regularization

Regularization solutions are meant to combat Overfitting and improve a model's ability to generalize. The other solution for Overfitting is having a large and well balanced dataset. Since the latter is usually expensive, this work presents some existing Regularization techniques. A common solution is the $L2$ Regularization, which works by penalizing big weights when computing the loss function. More specifically, it adds the product between a regularization term λ and the sum of the squared weights $W = (w_1^2, w_2^2, \ldots, w_m^2)$. Nowadays, architectural regularization techniques have gained popularity.

2.8.2.1 Dropout

Dropout [58], presented in Figure 2.27, is a type of layer in Neural Network models. Dropout is usually used on the hidden states of a model, and has a probability of shutting down random nodes for a specific iteration. Usually, the model will focus more on the primary features, and train some neurons more than others. Dropout solves this by temporarily shutting down primary feature nodes and forcing the model to learn more underlying patterns. While Dropout works good in Multi-Layer Perceptron architectures, its usage in images is fairly limited, as the feature maps are bigger.

2.8.2.2 DropBlock and DropPath

Inspired by Dropout, DropBlock [59] works by canceling whole regions in a feature map. This method introduces two hyperparameters: $block\_size$ – size of the removed region; $drop\_prob$ – probability that a point in the feature map is chosen.

Dropblock works by taking into consideration the feature-rich regions. A Bernoulli distribution with a probability of γ is applied over the important feature maps. The γ variable is used to balance the lack of choosing unimportant points. The formula for it is presented in equation (2.8.1). A visual example of DropBlock regularization is presented in Figure 2.28.

$$\gamma = \frac{drop\_prob}{block\_size^2} \frac{feat\_size^2}{(feat\_size - block\_size + 1)^2}. \tag{2.8.1}$$

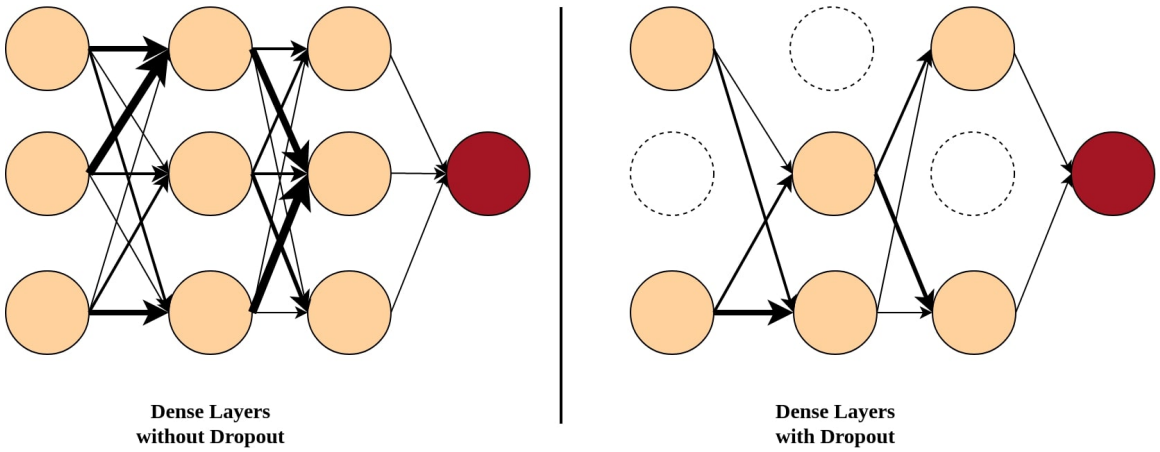


Figure 2.27: The Dropout effect on a small Fully Connected architecture.

An example of how DropBLock works is presented in Figure 2.28.

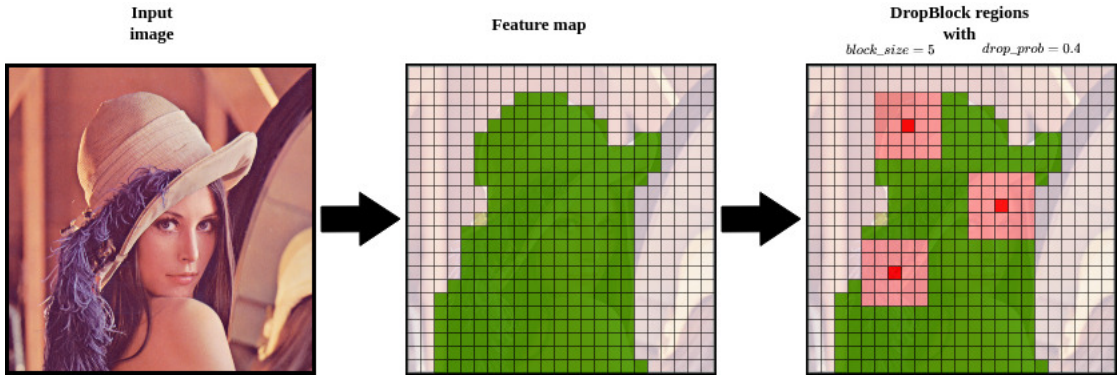


Figure 2.28: An example of chosen regions to be cancelled with DropBlock.

Similar to Dropout [58], DropPath [60] randomly drops operands of the join layers. A problem that usually happens is that when two layers join, the network will most likely use one as an anchor and the other as a corrective term. DropPath has two strategies. The Global strategy is to have one single path selected for the network. This path is restricted to one column, thus encouraging individual columns to have a stronger effect. The Local strategy has paths drop at an arbitrary probability rate, maintaining at least one path open.

2.8.3 The „dying ReLU" problem

In the Deep Learning field, saturation usually refers to the horizontal tendency of a function. The closer a function is to the value at which it converges, the more it looses its learning potential. For activation functions, a non-saturated property is desired for positive values in the upper bounds of the function (e.g., ReLU [38]). The network should only detect the positive characteristics that determine the desired result, not the incidentally present ones – which are represented by negative values. These instances that exist in the lower bounds point to current irrelevant features. However, these should not be totally discarded, as it is possible that current irrelevant features might become relevant later. Still, saturating the

lower bounds of a function can limit the effect of Overfitting, an effect observed in many newer activation functions [40, 41, 42, 44].

The main advantages of using ReLU are that it is being computationally simple and having a non-saturated property for the positive domain. The big disadvantage of ReLU is that it is fully saturated in the negative values range. This is one of the main proposed causes of the "dying ReLU" problem. In other words, values that are categorized as negative at some point during the learning process will have a value of 0. If the derivative of ReLU becomes 0, the whole gradient will multiply by 0, meaning that it will learn nothing. Consequently, once a neuron with ReLU activation "dies", it will almost always remain "dead", as the derivative will continuously be 0 and no learning will be possible. As neurons are usually inter-connected, this also limits the learning power of posterior and anterior neurons.

As much as 40% of neurons can suffer from this problem, and it has been shown that a very big learning rate in the early training process can lead to the premature "death" of neurons. Existing solutions for this are to also treat negative values, making the derivative no longer 0 and thus making it possible to bounce back into the learning process. However, it can become computationally expensive to give too much attention to negative values, which is why the saturation property is wanted in the lower bounds.

Some existing activation functions that treat negative values are either non-saturated [39] and can lead to big negative values having too much of an impact. Other activation functions are saturated in the lower bounds [40, 44], but still remain monotonous. The more advanced activation functions are non-monotonous but still saturated in the lower bounds, meaning that smaller negative values are seen as being more probable to become relevant later on, and thus are given more importance than the very big negative values. Out of all of these, only the GELU activation [44] treats negative values (up to a threshold) with positive values, potentially learning some anti-features. All of the aforementioned functions are less prone to the "dying ReLU" problem.

2.8.4 Data Imbalance

Most Machine Learning solutions require massive amounts of data to achieve a good performance. However, the quality of the data is also a driving factor in performance. An imbalanced dataset is a problem that exists everywhere, and it can give the illusion of good results. This is why using metrics that take into account data imbalance (such as the F1-score, mAP, etc.) are recommended in very big datasets.

Picking a more suitable metric is necessary for data imbalance, but it only corrects the performance interpretation, not the performance itself. For better performance, augmentation techniques are used to diversify and multiply the amount of trainable data. Some fundamental augmentations include:

1. **Random Resize** – modifying the scale of objects, and their ratios.
2. **Random Flipping** – diversifying the potential location of features on an object by flipping it horizontally or vertically. It is useful to reduce bias effects (in some cases).

3. **Random Crop** – cutting out a part of the image to train on it. It is useful because it forces the model to learn features from fewer pixels.
4. **Random Rotation** – rotates the image. It a similar effect to the Random Flipping, it is useful to reduce bias effects for some objects.
5. **Introducing Noise** – forces the model to better learn general features.

Chapter 3

Implementation Details

PaNeXt is a fully convolutional Panoptic Segmentation approach, similar to other convolutional based architectures [1, 2, 16]. However, PaNeXt brings improvements through advanced activation functions [43, 44], improved convolutional building blocks [26] that take inspiration from powerful Transformer architectures [25], better optimizers during training [57] and additional regularization techniques [59].

The General Architecture of PaNeXt is composed of three main components: the Feature Encoder – composed of a Backbone and an FPN, to create context at different feature levels; a Kernel Generator – composed of a Positional Head and a Kernel Head, in order to localize center-points for Things and also Stuff regions; a High-Resolution Feature Processor – composed of another FPN and a Convolutional structure, this module passes high-resolution features for finer segmentations. To extract such rich information, PaNeXt leverages Neural Network architectures, more specifically CNNs, in order to perform Panoptic Segmentation.

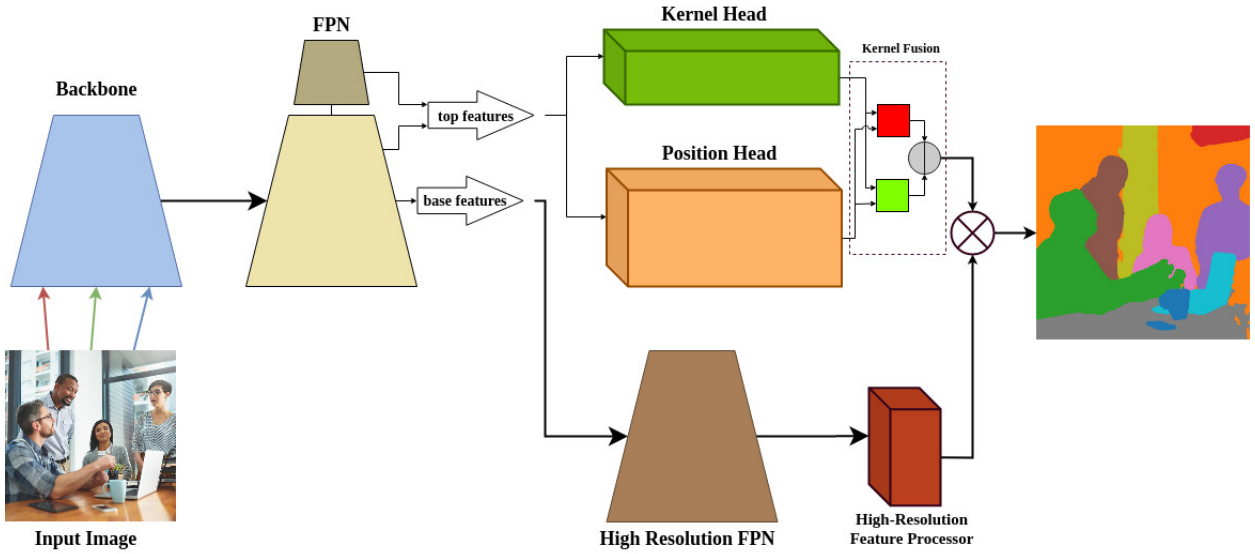


Figure 3.1: A general overview of the **PaNeXt** architecture.

Even though modern cameras can capture photos in 4K and higher, in the case of Image Segmentation there is no need for super-resolution – yet. Early Segmentation approaches outputted masks of 32×32 pixels. As techniques got more advanced, the output resolution

increased. PaNeXt preprocesses input images to sizes between 400 and 854 pixels. Other existing convolutional architectures use both lower resolutions, such in our case, and higher resolutions, up to 1333 pixels. The computational requirements increase exponentially with the input size, thus PaNeXt trains on smaller resolutions that can still produce good quality segmentations. Despite this aspect, PaNeXt achieves comparable (and even better) results with higher resolution alternatives.

3.1 Architecture

The main pipeline of PaNeXt is the following: the input image is passed through the Backbone and the FPN. The top 5 level features of the FPN are passed to: 1) the Kernel Head, which captures spatial cues and generates kernel weights; 2) the Position Head, which locates and classifies Thing instances and Stuff regions. Both generate outputs for each of the 5 feature levels. Then, the Kernel weights and the positional and class outputs are merged between the different FPN stages. At the same time, the base features of the FPN are passed to a High-Resolution Merging FPN (introduced by [1]). This structure manages to encode features from multiple levels of complexity of the FPN into a single feature map. This in turn is passed to a Convolutional stack for High-resolution refinement. The output of the High Resolution Convolutional stack is then combined with the Things and Stuff outputs to create a Panoptic Mask.

The detailed PaNeXt architecture is presented in Figure 3.2, where N represents the number of blocks on the 3^{rd} level of the Backbone. The legend for the used layers is presented in the dashed rectangles.

PaNeXt comes in two variants: a) A simpler, experimental one, where the backbone is ConvNeXt-tiny [26]; b) A more advanced one, which uses an improved version of ConvNeXt-tiny, better activations, an extended High Resolution stack, and DropBlock regularization.

3.1.1 The Feature Extractor

PaNeXt's Feature Extractor is vital to the model's performance, as it is the component that processes the most information. The Feature Extractor is composed of two parts:

1. the **Backbone**, which outputs features on multiple levels $[\frac{Input}{4}, \frac{Input}{8}, \frac{Input}{16}, \frac{Input}{32}]$.
2. the **FPN**, which takes the multi-level information from the Backbone and processes it with respect to previous feature levels.

There are two variants of Backbones for PaNeXt.

The first is ConvNeXt-tiny [26], which uses 4 stages of blocks. The standard ConvNeXt block is composed of a Depthwise Convolution with a 7×7 kernel that maintains the number of channels. A Layer Normalization is applied, and then a Pointwise Convolution is employed to quadruple the number of channels. After a GELU activation, another Pointwise convolution is applied to bring the number of channels to the original size. Finally, a skip connection between the input and the output of the last Pointwise convolution is used. The

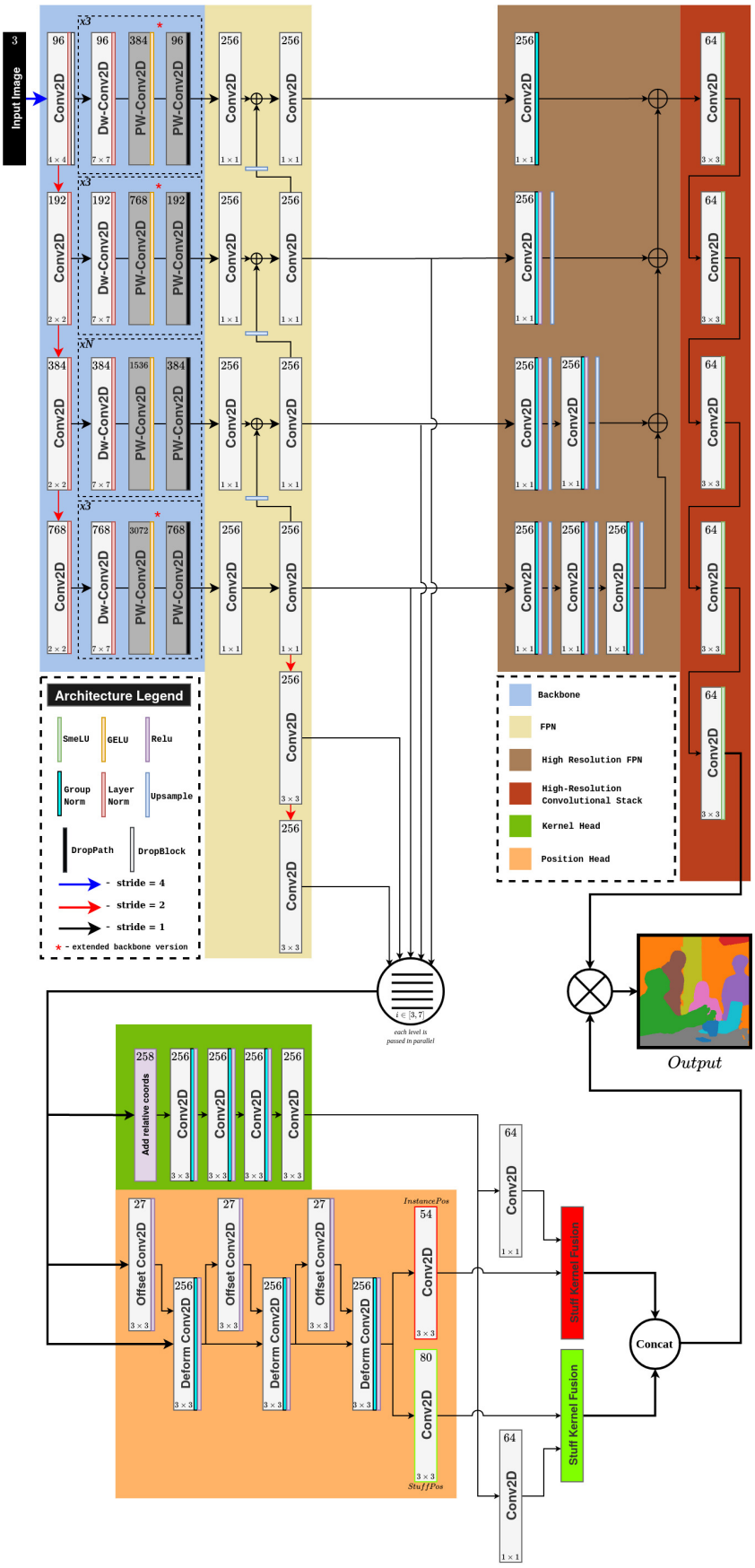


Figure 3.2: A detailed architecture of **PaNeXt**. The modifications for **PaNeXt-e** are highlighted in red.

ConvNeXt block is an improved version of the ResNet [29] block. Most improvements are inspired from the Transformer approaches [25, 24], such as increasing the kernel size to reproduce the "Patchify" layer effect, replacing Batch Normalization with Layer Normalization [47], and replacing ReLU [38] with GELU [44] activations. Another improvement is inspired by the ResNeXt [30] block, which uses grouped convolutions. The Depthwise convolution is a special case of grouped convolution that has the number of channels equal to the number of groups, and also operates in a per-channel basis. Combining this with Inverted Pointwise convolutions is also inspired from Transformers, where spatial and channel mixing is done separately. Similar to ResNets, ConvNeXt is composed of 4 stages that stack multiple ConvNeXt blocks, each stage with a channel size pair. For ConvNeXt-tiny, the stages $[1, 2, 3, 4]$ contain $(3, 3, 9, 3)$ blocks with $(96, 192, 384, 768)$ channels.

The second variant of Backbone in PaNeXt is an improved version of ConvNeXt-tiny. The first modification is the replacement of GELU activations with SmeLU [43]. Even though Transformers employed GELU activations in the past, SmeLU is the only activation function that allows negative values to positively impact the learning process. In a way, it can be seen as paying attention to small-impact "anti-features". These "anti-features" refer to relational information, or 2^{nd} degree features. The introduction of SmeLU does not impact computation efficiency. Another improvement is an additional 1×1 Pointwise convolution, that allows more in-depth learning from the high dimensionality space of $4X$ channels – similar to a bigger Feed-Forward layer in Transformers. This modification is computationally expensive, as multiplications are quadrupled. To reduce complexity, only Stages $[1, 2, 4]$ benefit from the extended blocks. Additionally, as a middle ground between ConvNeXt-tiny and ConvNeXt-small, 19 blocks can be used in Stage 3, but the small increase in performance was not worth for a $\sim 42\%$ increase in backbone complexity.

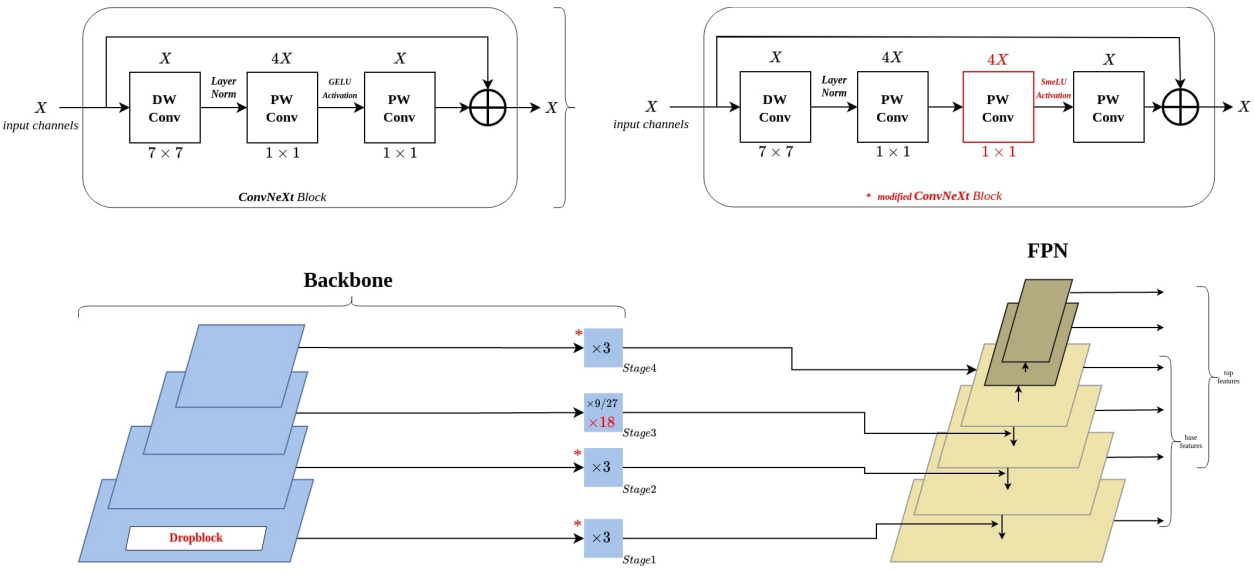


Figure 3.3: A representation for the Feature Extractor, composed of the Backbone (left) and the FPN (right). The modifications for **PaNeXt-e** are highlighted in red.

The second Feature Extractor component is the Feature Pyramid Network [37] (or FPN). Its architecture is capable of taking the information from multiple stages of the Backbone

and aggregating it with previous FPN feature levels, in order to obtain refined multi-stage features. In the case of PaNeXt, the base FPN has 4 levels $(1, 2, 3, 4)$ – referred to as base features. Two additional levels are expanded from the last base FPN level. The features from the $(2, 3, 4, 5, 6)$ levels are referred as the top features. A more comprehensive representation of the Feature Extractor is presented in Figure 3.3.

3.1.2 The Kernel Generator

The Kernel Generator is tasked with creating kernel weight maps Stuff and Things with different classes [16]. This component is made out of the Kernel Head, which produces the kernel weights, and the Position Head, which outputs the class and location for Thing instances and Stuff regions. Both the Kernel and Position Head take as input the top features from the Feature Encoder. For each feature level, a separate stack of outputs are produced.

For a single feature level, the Kernel Head first adds two additional channels in order to encode relative coordinates for features. Then, a convolutional stack with Group Normalization and ReLU activations is adopted to generate the kernel weight map. Combined with predictions from the Position Head, the kernel weights are correlated with the respective Things or Stuff predictions.

Using the same level feature stack, the Positional Head encodes it and produces locations and classes for each Thing and Stuff instance in the feature map. However, Stuff is uncountable, and thus cannot be bound to a center point. To solve this, background regions with the same semantic meaning are grouped together as one instance. To build this, deformable convolutions [28] are used. These are composed of a learnable offset layer that is then applied with a normal convolution. The last deformable convolution feature map is then branched to create Instance Positions and Stuff Regions. All of the stuff regions and only the best 100 object kernels are produced.

The kernel weights with class and location encodings is produced by using Kernel Fusion between the Kernel weights and the positional and class outputs. Multiple sets of kernel weights are generated, for each different level of feature maps. The Kernel Fusion is designed to merge the kernel weights with the same identity from multiple stages, through the use of Average Clustering.

The more detailed representation for the Kernel Generator is presented in Figure 3.2.

3.1.3 The High-Resolution Feature Processor

Similar to other components, the High Resolution Feature Processor is made of two convolutional structures: a special FPN architecture [1] and a Convolutional Stack.

To allow finer segmentations, a separate branch for High-Resolution features is created from the Feature Encoder. The first convolution in the Feature Encoder reduces the image size by a factor of 4. Subsequently, it outputs feature maps of sizes $[\frac{Input}{4}, \frac{Input}{8}, \frac{Input}{16}, \frac{Input}{32}]$, that encode both finer details and more general features. While this allows the Kernel Generator to process information at different dimensionalities, only High Resolution features are needed. An option is to use only the first feature stack of size $\frac{Input}{4}$, ignoring the coarser

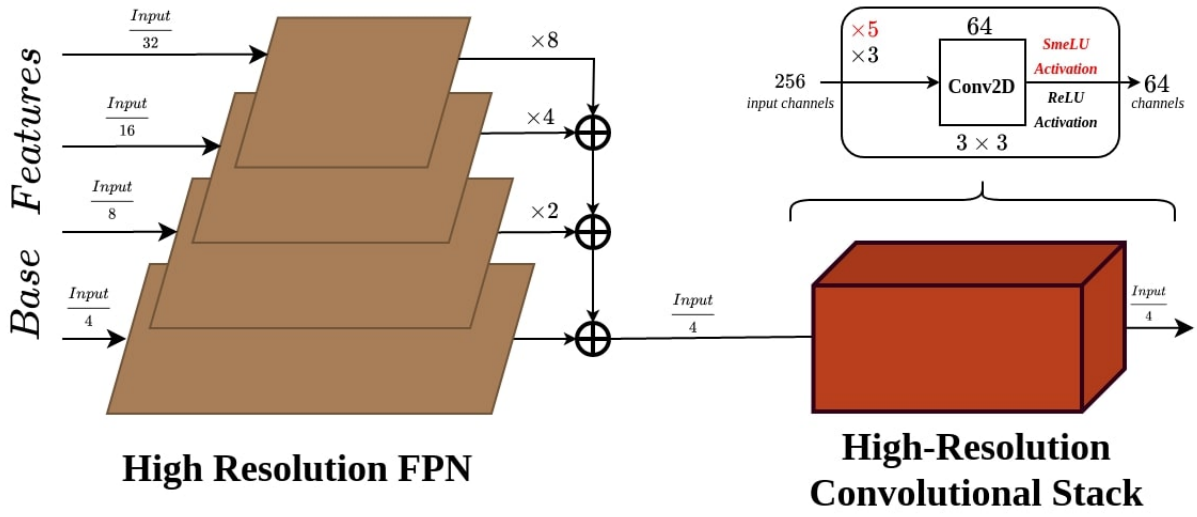


Figure 3.4: A representation for the High Resolution Feature Processor, composed of an FPN and a Convolutional stack. The modifications for **PaNeXt-e** are highlighted in red.

features from higher levels. However, a special FPN structure introduced by [1] would avoid waste and merge the coarser features into the first feature stack. As shown in Figure 3.4., this FPN structure takes the base features from the Feature Encoder begin to upsample the smaller feature levels to the base feature level. More specifically, the features from the top of the FPN of size $\frac{Input}{32}$ are upsampled to $\frac{Input}{16}$ and appended to the next feature level of the same new size. This process is repeated until a High-Resolution $\frac{Input}{4}$ feature map is produced.

Then, an extended convolutional stack further refines these features. Similar to the ConvNeXt block improvements, SmeLU [43] activations are employed instead of the standard ReLU activations. The resulting refined High-Resolution Stack is combined with the Stuff and Things kernel weights to produce a Panoptic mask. For the extended variant of PaNeXt, the number of convolutions in the convolutional stack is expanded from 3 to 5, and SmeLU [43] activations are employed. A more in-depth visualization of the High-Resolution Feature Processor is presented in Figure 3.4.

3.2 COCO Dataset

The Common Object in Context (COCO) [61] dataset is one of the few open-source Panoptic datasets. COCO has been widely accepted as a suitable benchmark, as it contains a total of 143K annotated images, split into balanced sets for training, validation, and testing. It contains 53 Stuff classes and 80 Thing classes. The COCO Panoptic dataset was created by merging the annotations for COCO Instance Segmentation over the masks of COCO Semantic Segmentation.

A few Panoptic maps from the COCO dataset can be observed in Figure 3.5.

Figure 3.5: Examples of COCO Panoptic Annotations.

3.3 Training

The quality of the training process is determined by a handful of factors, such as choosing the suitable Hyperparameters, using an effective Optimizer, Preprocessing the data, choosing the evaluation dataset, and suitable metrics. PaNeXt has customizable settings, as it is integrated within the detectron2 [62] framework. Checkpoints during the training of PaNeXt are created at every 5% progress increment.

3.3.1 Data Augmentation

Due to expensive computational requirements, advanced Augmentation methods (such as the ones used in ConvNeXt [26]) lead to an observable increase in total training time. Consequently, only Random Flipping and Random Rescaling are employed as Augmentation methods.

3.3.2 Hyperparameters

There are multiple types of hyperparameters used in training PaNeXt.

In terms of Augmentation, Random Rescaling will resize by the short edge to one random size from $(352, 384, 416, 448, 480, 512)$. Previous one-stage approaches [63] leveraged an increased training schedule. Similarly, PaNeXt uses triple the amount of iterations $(270K)$ during training, with a batch size of 16, bringing an 2.78% increase in performance. In terms of the Optimizer, Ranger21 [57] is applied with the default parameters, except for replacing adaptive gradient clipping with standard gradient clipping and initializing the learning rate with 0.01. Furthermore, the DropBlock [59] regularization probability in the Backbone is set to 0.05 with a size of 5, and the DropPath [60] rate is set to 0.2.

3.3.3 Loss function and Metrics

A composite loss using the Focal Loss [45] and Dice Loss [46] is employed during the training process.

More specifically, the Positional Loss is computed for both Things and Stuff using the Focal Loss with $\gamma = 2$. The exact equations are presented in (3.3.1) and (3.3.2), where th and st are short for Things and Stuff, k is the stage index from the FPN features, $\boldsymbol{L}$ represents the positions for Thing instances and Stuff Regions, $\boldsymbol{Y}$ is a groundtruth heatmap for Things and a groundtruth one-hot embedded map for Stuff, N_{th} is the number of Thing classes, W_k and H_k are the Width and the Height of the feature maps at stage k in the FPN. The overall Positional Loss is obtained by adding the two, as shown in equation (3.3.3).

$$L_{pos}^{th} = \sum_{k=2}^{6} \frac{FocalLoss(\boldsymbol{L}_k^{th}, \boldsymbol{Y}_k^{th})}{N_{th}}, \tag{3.3.1}$$

$$L_{pos}^{st} = \sum_{k=2}^{6} \frac{FocalLoss(\boldsymbol{L}_k^{st}, \boldsymbol{Y}_k^{st})}{(W_k H_k)}, \tag{3.3.2}$$

$$L_{pos} = L_{pos}^{th} + L_{pos}^{st}. \tag{3.3.3}$$

For the Segmentation Loss, a weighted Dice Loss [46, 16] is adopted. More specifically, the Segmentation Loss (3.3.5) equation is computed using the (3.3.4) equation, where $\boldsymbol{Y}_i^{seg}$ is the groundtruth for the i-th prediction $\boldsymbol{P_i}$, N, and M are the existing number of Stuff and Things, k is the number of chosen positions from within each object ($k = 1$ for the Stuff class, as all same-category points are treated equally), w_k is the k-th weighted score.

$$WeightedDiceLoss(\boldsymbol{P}_i, \boldsymbol{Y}_i^{seg}) = \sum_k w_k DiceLoss(\boldsymbol{P}_{i,k}, \boldsymbol{Y}_i^{seg}), \tag{3.3.4}$$

$$L_{seg} = \sum_i \frac{WeightedDiceLoss(\boldsymbol{P}_i, \boldsymbol{Y}_i^{seg})}{M + N}. \tag{3.3.5}$$

Combining the Positional Loss (3.3.3) and the Segmentation Loss (3.3.5) results in equation (3.3.6), where $\lambda_{pos} = 1$ and $\lambda_{seg} = 3$.

$$L = \lambda_{pos} L_{pos} + \lambda_{seg} L_{seg}. \tag{3.3.6}$$

The primary Metric used for measuring model performance is the Panoptic Quality (PQ) [14], as shown in equation (2.6.16). Additional secondary metrics are also used, such as the Segmentation Quality (SQ) and Recognition Quality (RQ).

3.3.4 Validation and Testing

The COCO [61] Panoptic Dataset contains $143K$ images split into $118K$ for training, $5K$ for validation and $20K$ for testing. The annotations for the training and validation split are public. However, there are no public groundtruth labels for the testing split. In order to

benchmark on the testing split, one must enter the timely process of submitting the solution to a 3rd party that will fairly perform the benchmark. For these reasons, the benchmarks will be performed on the $coco - val$ split.

3.4 Training Environment

3.4.1 Hardware Resources

The PaNeXt architecture was trained on a Microsoft Azure Standard NC24rs, that uses a 14-core Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz processor, $4 \times V100$ GPUs that add up to 64Gb of Video Memory, 20480 CUDA cores and 2560 Tensor Cores. Additionally, the NC24rs machine uses 448Gb of RAM memory. Training using the powerful NC24rs machine is a costly process. Because of that, short-term experiments were performed on a NC6s v3 machine, which only owns a quarter of the computational resources.

The model was trained using PyTorch's NCCL for Distributed Learning over the 4 GPUs. The training process lasted between around 2 days for the standard PaNeXt architecture, and 3 days for the extended PaNeXt architecture. A batch of 16 and a tripled training schedule of $270K$ iterations was used. For a standard training schedule of $90K$, training lasted between 16 and 23 hours, depending on the PaNeXt architecture.

3.4.2 Software Resources

The development of PaNeXt was conducted on a Ubuntu 18.04 operating system with NVIDIA CUDA 11.2, using Python 3.8 together with PyTorch 1.8.0, TorchVision 0.9.0, detectron2 0.6+cu111, and their subsequent required packages. Additionally, separate packages for the implementations of SmeLU [43] and Ranger21 [57] have been installed.

Chapter 4

Experiments and Results

Two variants of the PaNeXt architecture are proposed:

1. **PaNeXt** – the standard architecture with the ConvNeXt-tiny [26] backbone, combined with an enhanced [43] High Resolution Feature Processor and trained using the advanced Ranger21 [57] optimizer.

2. **PaNeXt-e** – an extended version that improves the ConvNeXt block with SmeLU [43] activations and an additional Pointwise convolution, together with an extended convolutional stack in the High-Resolution Feature Processor.

Regarding the training schedule employed for different models, $\times 1$ will be used for $90K$ iterations at 16 images per batch, and $\times 3$ will be used for $270K$ iterations at 16 images per batch. Due to hardware limitations, PaNeXt-e was only trained on the $\times 1$ schedule. However, considering the improvement for the standard PaNeXt from the $\times 1$ schedule to the $\times 3$ schedule, it can be reasonable to say that PaNeXt-e has the potential for a Panoptic Quality of at least 44.

The values for model size and training time (on a NC24rs Azure machine) is presented in Table 4.1.

	PaNeXt	**PaNeXt**	**PaNeXt-e**	**PaNeXt-e** *(*est.)*
train schedule	$\times 1$	$\times 3$	$\times 1$	$\times 3$
Size (Mb)	156	156	165	*165**
Training time	14h	44h	19h	*60h**
FPS (on $V100$)	22.1	21.6	18.2	*18.0**

Table 4.1: The size, speed and training times for PaNeXt variations.

The results for the COCO val set are presented in Table 4.2. PaNeXt was trained on images of size 400 and it is comparable to PanopticFCN-400 in terms of input resolution. However, PaNeXt manages to overcome existing convolutional approaches. Moreover, PaNeXt achieves the fastest results, at 22.1 fps. Comparing the results, PaNeXt is better at Segmentation overall, but is lacking in recognizing Things in particular.

Visualizations of panoptic masks from the COCO val set are presented in Figure 4.1.

Architecture	**PanopticFPN[1]**	**PanopticFCN[16]**	**PanopticFCN**	**PanopticFCN-400**	**PaNeXt**	**PaNeXt**	**PaNeXt-e**
train schedule	$\times 1$	$\times 1$	$\times 3$	$\times 3$	$\times 1$	$\times 3$	$\times 1$
PQ	39.4	41.1	43.6	40.7	41.1	**44.2**	43.8
SQ	77.8	79.8	80.6	80.5	79.6	**83.5**	81.2
RQ	48.3	49.9	52.6	49.3	49.9	**52.9**	52.8
PQ^{th}	45.9	-	**49.3**	44.9	45.9	48.6	48.6
SQ^{th}	80.9	-	82.6	82.0	81.6	**83.6**	82.6
RQ^{th}	55.3	-	**58.9**	54.0	55.4	58.2	58.2
PQ^{st}	29.6	-	35.0	34.3	33.6	**37.6**	36.5
SQ^{st}	73.3	-	77.6	78.1	76.5	**83.3**	79.2
RQ^{st}	37.7	-	42.9	42.1	41.6	**45.1**	44.8
FPS (on V100)	17.5	13.6	12.5	20.9	**22.1**	21.6	18.2

Table 4.2: Result comparisons on the COCO val dataset for convolutional panoptic solutions.

Figure 4.1: Examples of panoptic maps from the COCO val set.

Chapter 5

Conclusion

The Panoptic Segmentation problem is still relatively new when compared to other Computer Vision Tasks. Furthermore, panoptic data is complex, hard to annotate and expensive. Consequently, there are not too many datasets to train or benchmark solutions, compared to Object Detection or Semantic Segmentation. However, there are promising upcoming projects [64] that aim to expand the existing data significantly.

The Panoptic task is a complex one, and the only solutions for it employ some sort of a Neural Network architecture. Up until the 2020s, Convolutional architectures have dominated the Computer Vision field. The "Transformer revolution", however, managed to overcome convolutional methods and hold the State-of-the-Art over the existing Computer Vision tasks. It is the same case with Panoptic Segmentation solutions, where convolutions have more or less been discarded in favor of the Transformers. While this work does not attempt to overcome Transformer State-of-the-Arts, its aim is to revitalize Convolutional approaches through improving newer convolutional structures [26] and taking inspiration from Transformer techniques.

This work introduces PaNeXt, a fully convolutional method for Panoptic Segmentation that manages to overcome most of the previous CNN approaches. Its innovation comes from the use and improvement of the ConvNeXt [26] block, adopting SmeLU [43] activations for learning relational "anti-features", and expanding on existing convolutional structures. PaNeXt comes also in an extended variant, with more drastic improvements. Even though training on Panoptic data is a timely and computationally expensive process, PaNeXt manages to achieve satisfactory results even when training for less than its competition. There is still room for improvement, and the fact that PaNeXt-e managed to achieve relatively close results with PaNeXt – while being trained on a third of the iterations – is proof of that. While Transformers point to the future, there are still plenty of advancements made by convolutions that can be transferred over, and with the introduction of new data, there is plenty of potential.

In terms of use-cases, Panoptic Segmentation is designed to replace both Object Detection and Semantic Segmentation for a faster inference. Thus, its applications can range from the autonomous driving systems to mobility assistive devices for the blind.

To conclude, this work introduces a method for Panoptic Segmentation using improved CNNs, while acknowledging the power of Transformers.

Bibliography

[1] A. Kirillov, R. Girshick, K. He, and P. Dollár, Panoptic Feature Pyramid Networks. arXiv, 2019

[2] Bowen Cheng, Maxwell D Collins, Yukun Zhu, Ting Liu, Thomas S Huang, Hartwig Adam, and Liang-Chieh Chen. Panoptic-deeplab: A simple, strong, and fast baseline for bottom-up panoptic segmentation. In CVPR, 2020

[3] S. Qiao, Y. Zhu, H. Adam, A. Yuille, and L.-C. Chen, ViP-DeepLab: Learning Visual Perception with Depth-aware Video Panoptic Segmentation. arXiv, 2020

[4] Z. Li et al., Panoptic SegFormer: Delving Deeper into Panoptic Segmentation with Transformers. arXiv, 2021

[5] B. Cheng, I. Misra, A. G. Schwing, A. Kirillov, and R. Girdhar, Masked-attention Mask Transformer for Universal Image Segmentation. arXiv, 2021

[6] P. Cunningham, M. Cord and S. Delany, Supervised Learning, Machine Learning Techniques for Multimedia, Springer, 2008.

[7] O. Chapelle, B. Schölkopf, and A. Zien, Semi-Supervised Learning (Adaptive Computation and Machine Learning), Page 2. 2006

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. 2016. MIT Press.

[9] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems, https://www.tensorflow.org/, 2015

[10] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch. În monitorul NeurIPS Autodiff Workshop, 2017. https://pytorch.org/

[11] R. Girshick. Fast R-CNN. In ICCV, 2015

[12] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In NIPS, 2015

[13] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. Mask r-cnn. In ICCV, 2017

[14] A. Kirillov, K. He, R. Girshick, C. Rother, and P. Dollar, “Panoptic segmentation,” in Proceedings of IEEE CVPR, 2019

[15] L.-C. Chen, H. Wang, and S. Qiao, Scaling Wide Residual Networks for Panoptic Segmentation. arXiv, 2020.

[16] Y. Li et al., Fully Convolutional Networks for Panoptic Segmentation. arXiv, 2020.

[17] Yuwen Xiong, Renjie Liao, Hengshuang Zhao, Rui Hu, Min Bai, Ersin Yumer, and Raquel Urtasun. Upsnet: A unified panoptic segmentation network. In CVPR, 2019.

[18] Yamashita, R., Nishio, M., Do, R.K.G. et al. Convolutional neural networks: an overview and application in radiology. Insights Imaging, 2018, 9, 611–629.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning Internal Representations by Error Propagation,” in Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations, D. E. Rumelhart and J. L. Mcclelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 318–362.

[20] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, “Recurrent Models of Visual Attention,” CoRR, vol. abs/1406.6247, 2014

[21] D. Bahdanau, K. Cho, and Y. Bengio, Neural Machine Translation by Jointly Learning to Align and Translate. arXiv, 2014.

[22] M.-T. Luong, H. Pham, and C. D. Manning, Effective Approaches to Attention-based Neural Machine Translation. arXiv, 2015.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszko- reit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In NeurIPS, 2017.

[24] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In ICLR, 2021.

[25] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. 2021

[26] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, “A ConvNet for the 2020s,” CoRR, vol. abs/2201.03545, 2022

[27] F. Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. Published at CVPR, 2017

[28] J. Dai et al., Deformable Convolutional Networks. arXiv, 2017

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. K. Deep residual learning for image recognition. In CVPR, 2016

[30] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In CVPR, 2017

[31] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. MobilenetV2: Inverted residuals and linear bottlenecks. In CVPR, 2018

[32] A. Krizhevsky, I. Sutskever, and G. Hinton. Im- agenet classification with deep convolutional neural networks. In NIPS, 2012

[33] K. Simonyan and A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014

[34] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, 2012.

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Internal Representations by Error Propagation, Parallel distributed processing vol. 1: foundations. 1986

[36] Ronneberger, Olaf & Fischer, Philipp & Brox, Thomas. U-Net: Convolutional Networks for Biomedical Image Segmentation. Published in MICCAI, 2015

[37] Lin, Tsung-Yi et al. Feature Pyramid Networks for Object Detection. În monitorul CVPR, 2017

[38] Xavier Glorot, Antoine Bordes, Yoshua Bengio. In the Fourteenth International Conference on Artificial Intelligence and Statistics, PMLR 15:315-323, 2011

[39] Maas, Andrew L, Hannun, Awni Y, and Ng, Andrew Y. Rectifier nonlinearities improve neural network acoustic models. În monitorul ICML, 2013

[40] Clevert, Djork-Arné & Unterthiner, Thomas & Hochreiter, Sepp. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2016

[41] Prajit Ramachandran, Barret Zoph, & Quoc V. Le. Searching for Activation Functions, 2017

[42] D. Misra, Mish: A Self Regularized Non-Monotonic Activation Function. 2020

[43] G. I. Shamir, D. Lin, and L. Coviello, Smooth activations and reproducibility in deep networks. 2020

[44] D. Hendrycks and K. Gimpel, Gaussian Error Linear Units (GELUs). 2020

[45] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Doll´ar. Focal loss for dense object detection. In ICCV, 2017

[46] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In 3DV, 2016

[47] J. L. Ba, J. R. Kiros, and G. E. Hinton, Layer Normalization. arXiv, 2016

[48] Y. Wu and K. He, Group Normalization. arXiv, 2018

[49] H. Robbins and S. Monro. A stochastic approximation method. Annals of Mathematical Statistics, 22:400–407, 1951

[50] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, Learning representations by backpropagating errors, 1988

[51] Andrew Yan-Tak Ng. Deep Learning Specialization, https://www.deeplearning.ai/, 2017

[52] G. E. Hinton, N. Srivastava, and K. Swersky. Neural networks for ML: Lecture 6e RMSProp, https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, Cited on, 2012

[53] D. P. Kingma and J. Ba, Adam: A Method for Stochastic Optimization. arXiv, 2014

[54] Liu, Liyuan & Jiang, Haoming & He, Pengcheng & Chen, Weizhu & Liu, Xiaodong & Gao, Jianfeng & Han, Jiawei. On the Variance of the Adaptive Learning Rate and Beyond, 2019

[55] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In ICLR, 2019

[56] Michael R. Zhang and James Lucas and Geoffrey E. Hinton and Jimmy Ba. Lookahead Optimizer: k steps forward, 1 step back. În monitorul NIPS, 2019

[57] L. Wright and N. Demeure, Ranger21: a synergistic deep learning optimizer. arXiv, 2021

[58] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012

[59] Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. DropBlock: A regularization method for convolutional networks. În monitorul NIPS, 2018

[60] G. Larsson, M. Maire, and G. Shakhnarovich, FractalNet: Ultra-Deep Neural Networks without Residuals. arXiv, 2016

[61] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll´ar, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In ECCV, 2014

[62] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github. com/facebookresearch/detectron2, 2019

[63] Xinlei Chen, Ross Girshick, Kaiming He, and Piotr Doll´ar. Tensormask: A foundation for dense object segmentation. In ICCV, 2019

[64] J. Lambert, Z. Liu, O. Sener, J. Hays, and V. Koltun, MSeg: A Composite Dataset for Multi-domain Semantic Segmentation. arXiv, 2021