Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Using transfer learning and loss function adaptation for RNA secondary structure prediction

Master Candidate

Giovanni Faldani

Student ID 2054141

Supervisor

Prof. Loris Nanni

University of Padova

Academic Year
2023/2024

To my parents,
brothers,
niece,
and friends

Abstract

The problem of predicting RNA secondary structure is a challenging research topic, which involves various fields of computer science. Accurate solutions to this problem are helpful in the disciplines of medicine for vaccine development, to design stable mRNA molecules, or biology for discerning between different functions of various RNA molecules according to their shape.
The objective of this project is to study an emerging Machine Learning-based approach to the problem of RNA secondary structure prediction via integration of deep learning techniques like transfer learning and convolutional neural networks, aided by adaptations made for the specific problem at hand, like data representation and loss function.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

CNN	Convolutional Neural Network
DL	Deep Learning
DNA	DeoxyriboNucleic Acid
FC	Fully Connected
FLOPS	Floating-point Operations Per Second
KPI	Key Performance Indicator
LSTM	Long Short-Term Memory
MFE	Minimum Free Energy
ML	Machine Learning
MSE	Mean Squared Error
NN	Neural Network
NP	Nondeterministic Polynomial
nt	Nucleotide
ReLU	Rectified Linear Unit
RGB	Red, Green, Blue
RNA	RiboNucleic Acid
SVM	Support Vector Machine

1 Introduction & state of the art

The problem of RNA secondary structure prediction is challenging and complex, involving various notions of computer science like dynamic programming, minimization algorithms and machine learning. Accurate and reliable solutions to this problem are helpful in the fields of medicine for vaccine development, to find stable mRNA molecules to combat viruses [58], or biology for discerning between catalytic, ligand binding and scaffolding functions of various RNA molecules [18]. Currently a lot of research efforts date back to last decade [9] [47], before the current Deep Learning (DL) revolution [48] started yielding very precise models for a variety of specialized tasks, like face recognition [44].
RNA is a biomolecule composed of four basic building blocks called nucleotides (nt/nts), represented by the letters **A, C, G, U**. This allows us to encode each nucleotide with only 2 bits of information without wasting any space, a very useful property in computing. RNA differs from DNA by being single-stranded, meaning it can present a vast number of geometrical structures depending on how the nucleotides in the sequence form bonds with other nucleotides farther away in the same molecule. The secondary structure of the RNA molecule refers to its 2-dimensional folding before accounting for the three-dimensional tertiary structure. A base in an RNA sequence can be either paired or unpaired, and there are 3 types of bond in total [1]:

- **(A-U)** or **(U-A)** bonds, between the Adenine and Uracile nucleotides.
- **(C-G)** or **(G-C)** bonds, between the Cytosine and Guanine nucleotides.
- **(G-U)** or **(U-G)** bonds, a special "wobble pair" which does not follow the standard Watson-Crick base pair rules.

1.1 Problem Description

The problem can be formally described as such:

Formal problem description

Input: The sequence of nts representing a valid RNA molecule.

Output: A data structure representing the 2-dimensional folding of the molecule's secondary structure by highlighting which nts in the sequence bond together.

Typical representations of the output follow an array-like structure of the same length as the input sequence, like the one showcased in Figure 1.1, where the output array implicitly details how bases pair among each other. In this representation, the implicit pairing is between the index and the element found at that index (i.e., the base at index 3 pairs with the base at index 26, indexing from 1). When the index and its element coincide, no pairing is indicated.
Another widespread representation is the dot-bracket representation, where an open bracket represents a pairing with the respective closed bracket further in the sequence, and a dot indicates no pairing.

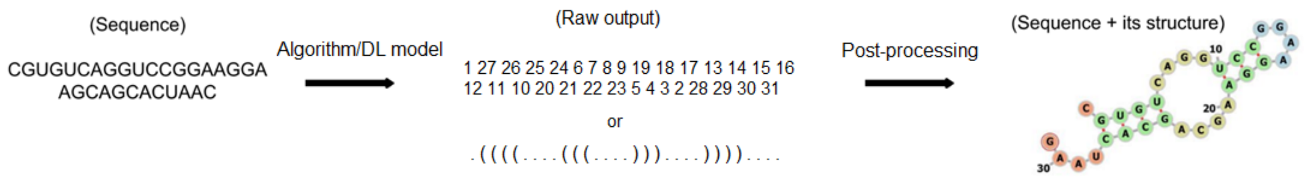


Figure 1.1: Example of the expected inputs and outputs of the problem at hand, from [1]

1.2 History and state of the art

RNA secondary structure prediction was first approached with exact mathematical and algorithmic methods, the first of which, proposed by Nussinov [40], uses a nearest neighbor thermodynamic model to predict the most likely structure following a dynamic programming algorithm that minimizes the amount of free energy in the structure. This algorithm had a complexity of $O(N^3)$ with N being the length of the RNA sequence in nucleotides, with the caveat of not allowing for prediction of pseudoknot structures. This algorithm has been known as Minimum Free Energy (MFE).
Zuker & Stiegler [62], later followed with a similar approach that allowed for prediction of suboptimal structures as well as optimal ones, however still with the constraint of not allowing pseudoknots.

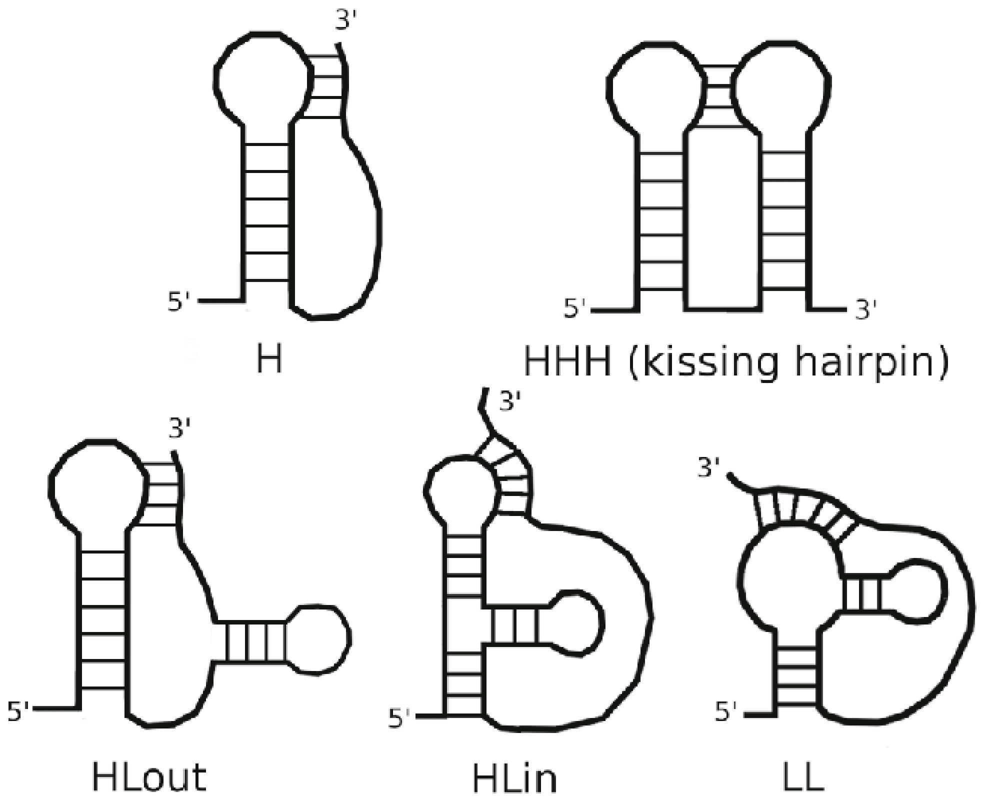


Figure 1.2: Different types of possible pseudoknotted secondary structures shown in [31]

MFE was proven to be an NP-complete problem when pseudoknots are taken into account [32], meaning that its complexity quickly becomes intractable with large input sizes. As such, heuristic methods were developed to address the high computational cost of exact methods.

Some heuristics utilized comparison to known structures with alignment algorithms [19], or statistical models with Boltzmann distribution [23], and small improvements were made throughout the 2000s and 2010s, being able to formulate reasonably tractable algorithms for restricted pseudoknot prediction [51].
One of the first DL-based approaches, SPOT-RNA [2], used a small ensemble of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks combined with the use of transfer learning, and was able to take into account pseudoknotted structures, outclassing the precision of other algorithmic heuristic approaches.
Other contemporary DL-based approaches included considering thermodynamic energy information like the MFE algorithm [33], or base-pair information [5]. Thanks to Deep Learning, steady improvements have been made over heuristic algorithmic methods in a very short time span. Current DL-based approaches have improved upon these results even further using a new kind of tensor representation for processing RNA base pair information with CNNs [1] or using length-dependent information to aid the prediction [34]. Transformer networks are a promising new architecture that also started being proposed for the problem [52].

2

RNA tensor representation

The work presented in this thesis is based on "RNA secondary structure prediction with convolutional neural networks" [1], the main idea that inspired the approach used in this project is how RNA sequences have been represented as 3-dimensional tensors of shape $L \times L \times 8$ boolean values, where L is the length of the RNA sequence in nts.
The way this representation is achieved is by arranging the RNA sequence on an LxL matrix, where entry $(i, j) \in L \times L$ represents the matching possibilities of bases at index i and j respectively. The matching possibilities are described by an 8-dimensional boolean array computed as such:

- **Bit 1**: set to `true` or 1 when $|i - j| < 2$, making the pairing infeasible, due to the bases being too close (they must be at least 2 positions apart to properly bond in the secondary structure), or simply not matching with one of the three bond types described in Section 1, which also results in an infeasible pairing.
- **Bit 2**: set to `true` or 1 when $i == j$, since a base can't be paired with itself. This means this bit will only be true in the diagonal of the tensor.
- **Bit 3**: set to `true` or 1 when the nucleotide at position i (row index) is **A** and the nucleotide at position j (column index) is **U**.
- **Bit 4**: set to `true` or 1 when the nucleotide at position i (row index) is **U** and the nucleotide at position j (column index) is **A**.
- **Bit 5**: set to `true` or 1 when the nucleotide at position i (row index) is **U** and the nucleotide at position j (column index) is **G**.
- **Bit 6**: set to `true` or 1 when the nucleotide at position i (row index) is **G** and the nucleotide at position j (column index) is **U**.

- **Bit 7**: set to `true` or 1 when the nucleotide at position i (row index) is **G** and the nucleotide at position j (column index) is **C**.
- **Bit 8**: set to `true` or 1 when the nucleotide at position i (row index) is **C** and the nucleotide at position j (column index) is **G**.

This way, every entry in the matrix is an 8-bit binary sequence, which can also be interpreted as a grayscale color value from 0 to 255. This makes this type of input naturally fit for use with CNNs.

Similarly, the desired output is also pre-processed in a matrix shape to serve as the target for training and computing Key Performance Indicators on.
The Target Matrix is simply obtained by marking each entry (i, j) with either 1 or 0 where bases are paired. If base i is unpaired then entry (i, i) is set to 1, otherwise if base i is paired with base j, both entries (i, j) and (j, i) are set to 1, ensuring symmetry in the output.
This is an important constraint for the problem, as ensuring symmetry in the output is a hard constraint on a valid solution.
A breakdown of the pipeline is found in Figure 2.1.

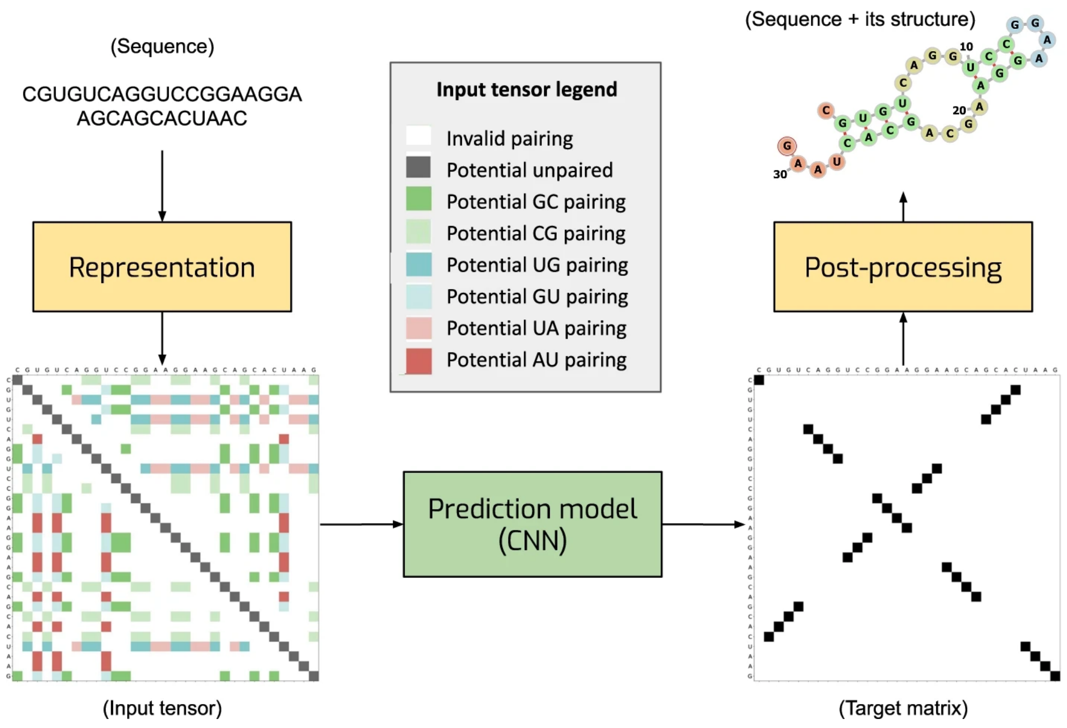


Figure 2.1: Example of the pipeline used for the application of CNNs for this problem, sourced from [1]

3 Basics of Neural Networks

Neural Networks (NNs) were introduced as far back as 1943 by McCulloch and Pitts [36], as an approximation of how the human brain processes stimuli. The field however wasn't very active until two major breakthroughs: the first was in 1989, when Hornik Et al. [20] proved that NNs are universal approximators when coupled with non-linear activation functions in Theorem 1 [8], one popular nonlinearity being the Rectified Linear Unit (ReLU, Figure 3.2);
the second breakthough arrived through the continuous improvement of computing hardware and especially graphics card technology, which allowed the training of large NNs to become feasible in the 2010s.
The basic structure of a NN is shown in Figure 3.1, composed of an input layer or column of neurons, a hidden layer and then an output layer.
Each connection performs a linear transformation with weight w_{ij} and bias b_{ij} on input x from neuron $i \in N$ of a preceding layer to neuron $j \in M$ of a successive layer, then applies a non-linear activation function σ and each connection to destination neuron j is summed as such:

$$\alpha_j = \sum_{i=1}^{|N|} \sigma(w_{ij} \cdot x_i + b_{ij})$$

When every neuron of a previous layer is connected to every neuron of the current layer, it is called a fully connected (FC) layer. FC layers can easily become very large in terms of weight parameters w_{ij}, so a lot of strategies were developed to mitigate this cost.

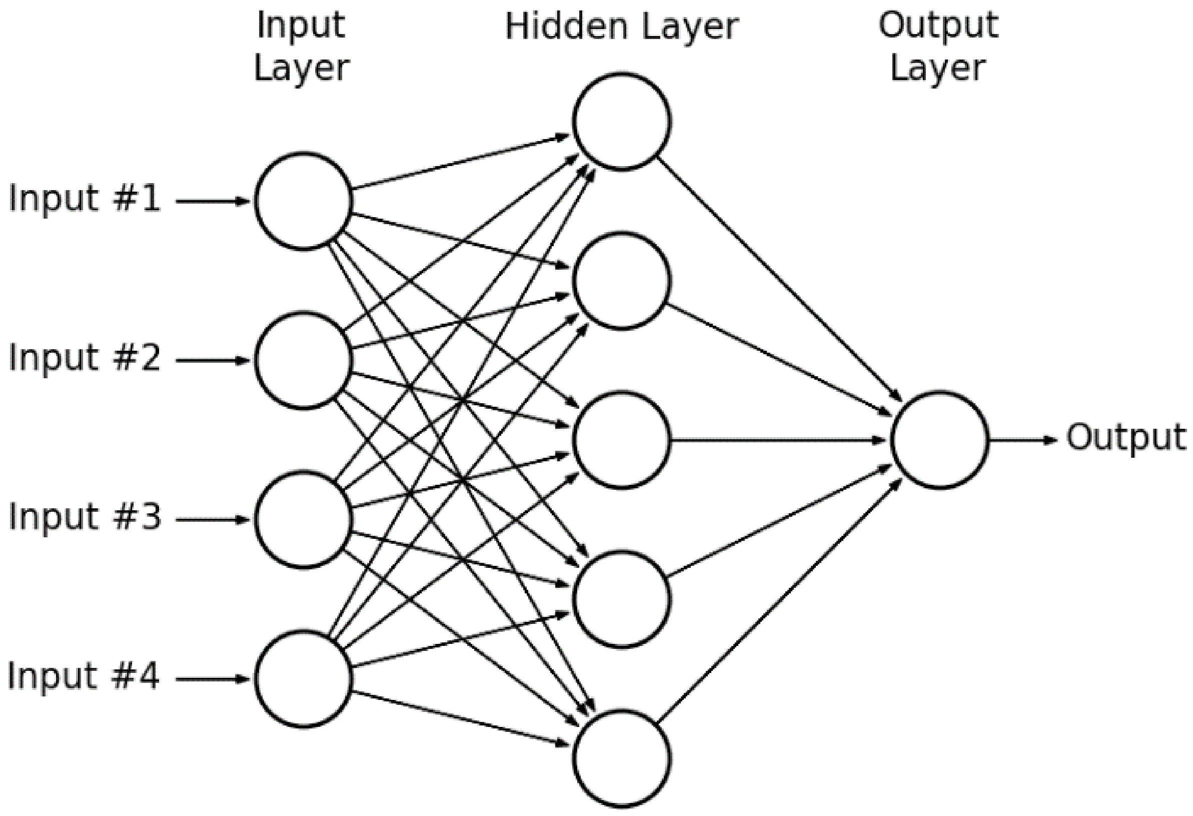


Figure 3.1: A Multilayer Perceptron schematic, the most basic neural network that can satisfy the universal approximator condition by applying a non-linear activation function σ after the input and hidden layer [16].

Theorem 1 (Universal approximation theorem) *Let $C(X, \mathbb{R}^m)$ denote the set of continuous functions from a subset X of a Euclidean $\mathbb{R}^n$ space to a Euclidean space $\mathbb{R}^m$. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$. Note that $(\sigma \circ x)_i = \sigma(x_i)$, so $\sigma \circ x$ denotes σ applied to each component of x.*
Then σ is not polynomial if and only if for every $n \in \mathbb{N}$, $m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n$, $f \in C(K, \mathbb{R}^m)$, $\varepsilon > 0$ there exist $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $C \in \mathbb{R}^{m \times k}$ such that:

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon$$

where $g(x) = C \cdot (\sigma \circ (A \cdot x + b))$.

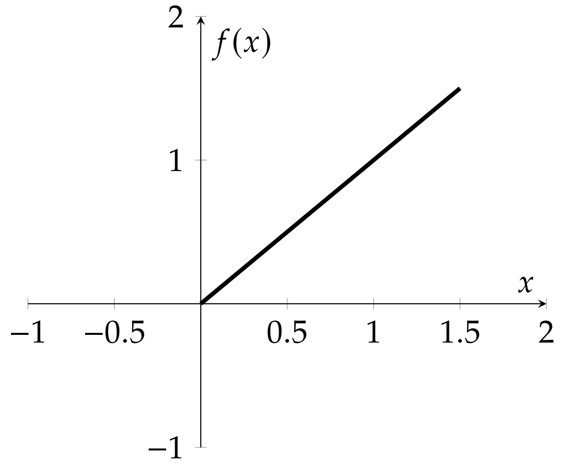


Figure 3.2: The ReLU function, effectively and identity that replaces all negative values with zero.

Deep Learning refers to the increasing trend of adding more and more hidden layers to neural network to increase expressiveness and performance. It's been observed that NNs with more hidden layers can lead to composition of features extracted at previous layers into more complex inferences. This is especially useful for image processing [12].

4 Convolutional Neural Networks

Convolutional Neural Networks are a subclass of Neural Networks that was first introduced by Yann LeCunn in 1989 [30], whose central idea is called weight sharing.

The signature layers of a CNN employ a convolution operation that takes a learnable kernel (a weight matrix, usually of shape $k \times k \times C$), and slides it across the input, usually an image of shape $H \times W \times C$, with C being the channel dimension, or how much information is stored in each pixel. Convolution performs the scalar product on every $k \times k$ subsection of the input, and saves the result in one entry of the output. Multiple output channels can be obtained by utilizing multiple kernels. This operation employs weight sharing for different zones of the output, which massively reduces the number of trainable parameters, while also reducing the variance of the operation's outcome.

Zero-padding p can be employed to prevent the size reduction of the output, and a larger stride s can be used to make the kernel "jump" farther at every step while sliding over the image.

A visualization of the operation is shown in Figure 4.1. The convolution operation results in an output with lateral dimension determined by this formula:

$$W_{out} = \left\lfloor \frac{W_{in} - k + 2 \cdot p}{s} + 1 \right\rfloor$$

Via their weight sharing, convolutional kernels have the ability to learn features and patterns of the images they are processing, and compose them in more complex combinations with deeper convolutional layers, as shown in Figure 4.2.

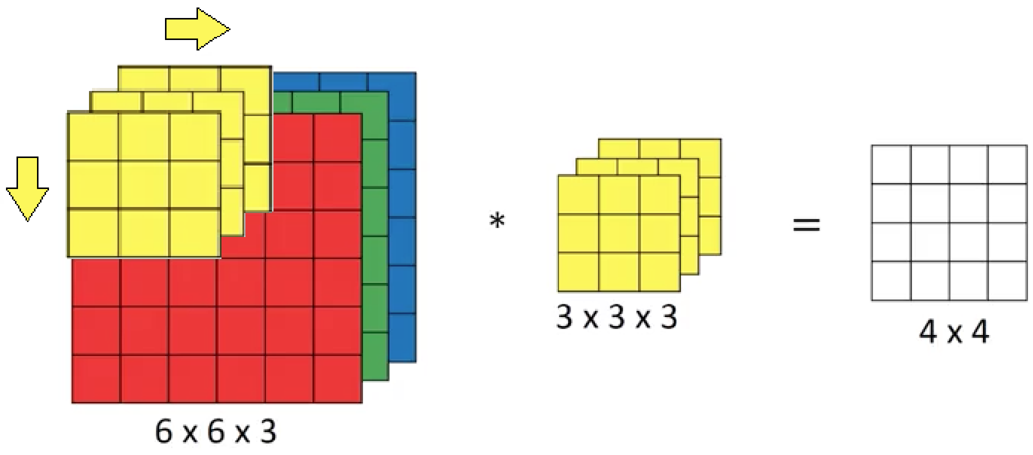
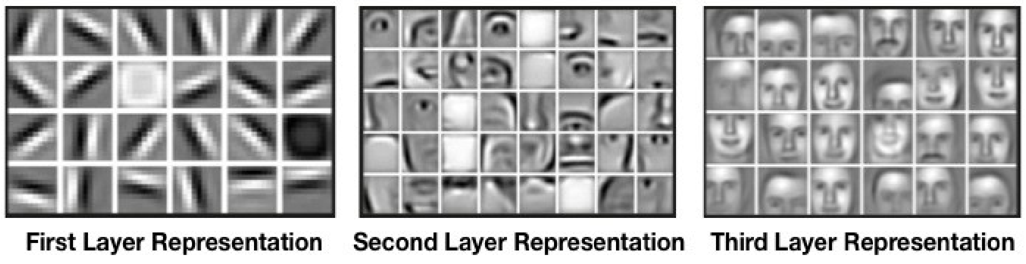


Figure 4.1: Visualization of the convolution operation, $W_{in} = 6$, $k = 3$, $p = 0$, $s = 1$, each weight in the kernel is shared with different positions of the input image [10]

Combining the convolutional layers with more standard nonlinearities like leaky ReLU forms very robust frameworks for image-related tasks.



First Layer Representation
Second Layer Representation
Third Layer Representation

Figure 4.2: Example of features observed in convolutional kernels used for facial recognition [37]

CNNs have repeatedly proven successful in image processing tasks such as classification, object detection and localization (ImageNet competition [11]), and have continued to be refined over the past decade.
Because of this, using a convolutional approach for the RNA data representation shown in Section 3 was shown capable of success in "RNA secondary structure prediction with convolutional neural networks" [1], and the possible avenues for further research on the subject are plentiful.

4.1 Transposed Convolution

While convolution is a useful operation for feature extraction, it results in the reduction of the size of the input image, meaning that to obtain a result of the same shape as the input, we need an operation to reverse this process.
The transposed convolution [24] works similarly in concept to a reverse convolution. It also employs a learnable kernel of shape $k \times k \times C$, but instead multiplies it to each pixel of the input via a scalar product to obtain a larger output, then composes these results with the given stride s, shown in Figure 4.3. As with convolution, a padding parameter p and can be employed, which inserts p zeros around the image, and then eliminates p contour lines from the output afterwards.

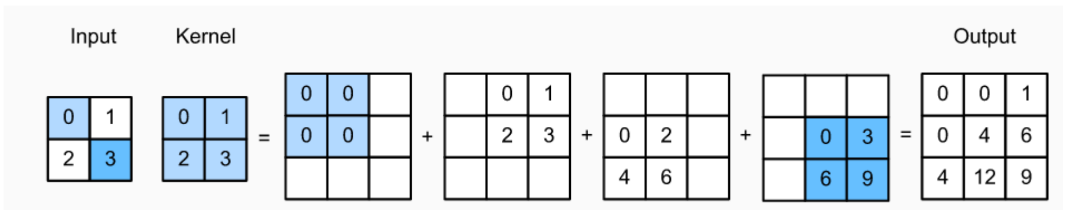


Figure 4.3: Visual example of a transposed convolution operation, $W_{in} = 2$, $k = 2, p = 0, s = 1$, sourced from [14]

The output size follows the formula:

$$W_{out} = (W_{in} - 1) \cdot s - 2 \cdot p + k$$

These types of layers have been used in Generative Adversarial Networks (GANs) to upsample low-dimensional feature vectors into images [24], achieving impressive results and kickstarting generative AI.

5 Transfer Learning

Since CNNs for image processing are a rich and active research field with a large number of architectures readily available on most modern frameworks, the main idea behind this project was to see if any of the historical and current CNN architectures that have been successful can be useful for the problem of RNA secondary structure prediction when we take the RNA tensor representation as the input.
The method through which these architectures will be tested is called transfer learning, which consists in importing previously trained CNN models for a specific problem, and using them with some small adaptations for a different problem.

Transfer learning was first introduced in 1976 by Stevo Bozinovski and Ante Fulgosi [43] for the field of machine learning and artificial intelligence, it's a technique which aims to transfer previous "knowledge" of a task to a new one, and in the context of neural network architectures can broadly be divided in two subcategories:

1. **Fine-tuning**: An existing and trained network is taken and the input or output layers are modified if necessary, the resulting model is trained on new data for a different task to adapt to it, shown in Figure 5.1. This effectively serves to give the new architecture an advantage by starting at a point where it achieved good knowledge of a certain problem, which may help in solving the new problem at hand.

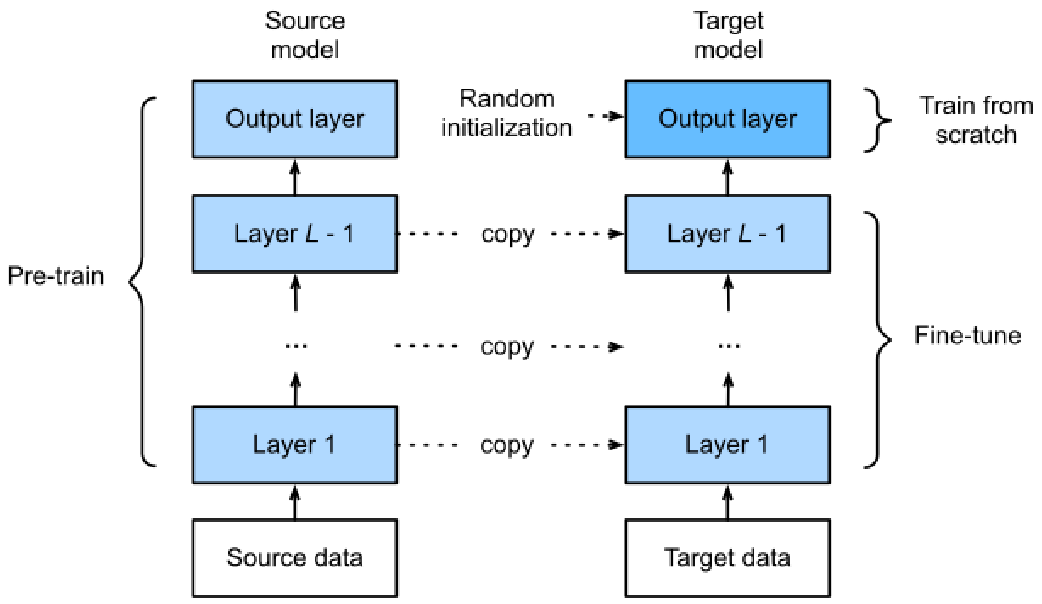


Figure 5.1: Illustration of the methodology behind fine-tuning from [15]

2. **Deep Features**: An existing and trained network is taken as is and used as is, but cut off to generate deep feature vectors, which are then classified by another ML algorithm like a Support Vector Machine, like in Figure 5.2. The weights of the existing network are not updated during the training process, just the post-processing scheme that uses its deep features as inputs.

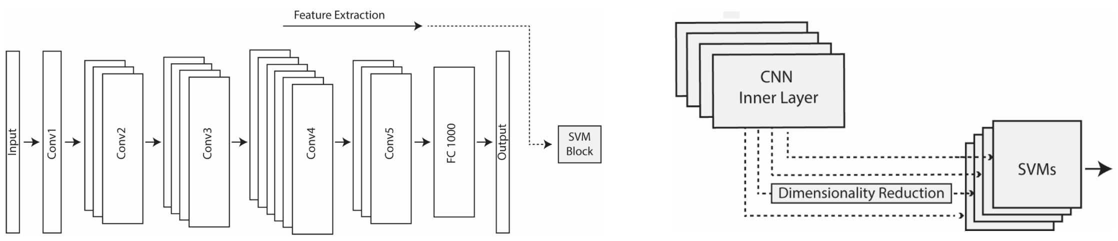


Figure 5.2: Example of deep features transfer learning taken from [39]

Transfer learning has been demonstrated to posses some significant benefits [61], and has been shown to be able to achieve good results on this specific problem [2].
In general the benefits can be summed up as such: better performance in relevant KPIs for the problem, like F_1 score, when the networks are trained for the same amount of time, especially when the available training set is small and might not be enough for a randomly initialized network to converge.
Convolutional Neural Networks trained on the ImageNet dataset have also

demonstrated strong proficiency in transfer learning tasks [22], mostly due to the massive size of the ImageNet dataset itself, which sports over 14 million labeled images, making its knowledge very general.

In our case, the fine-tuning approach is taken, with the pre-trained architectures taken as a starting point for further training on a labeled dataset of RNA tensors and target matrices.
Fine-tuning has been shown to help when there isn't enough data available for a newly trained model to converge, which is very valuable here as there isn't a lot of labeled RNA secondary structure data, and the data can exhibit huge variation in size, detailed in Section 8.

6

Loss function for symmetry

The choice of loss function is one of the most important parts of training a neural network [57], since it dictates what the network is going to focus on learning. The loss function needs to provide a good error model for the type of output we desire, must be a differentiable function so that gradient descent can be applied to the weights of the network during back-propagation, and the network can learn to perform its task better. The idea is to move along the loss function's surface to try to find the lowest point, by following the direction of the "slope" of the function, which is given by its gradient, illustrated in Figure 7.2.

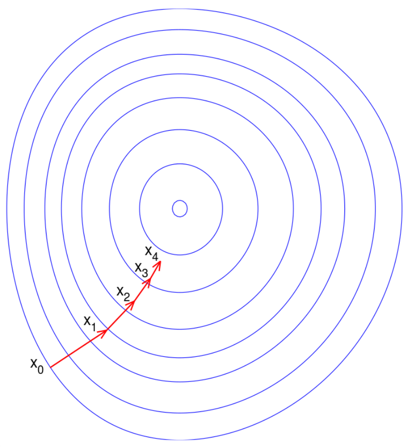


Figure 6.1: An illustration of gradient descent from [59], where the function is lower the closer to the innermost circumscription.

There are two main ways to implement gradient descent that were contemplated, which are:

1. **Stochastic Gradient Descent (SGD)**: A stochastic approximation of the standard Gradient Descent algorithm for optimization [45], tailored to run faster by being computed on small pieces of the training set (called minibatches) in various iterations. Splitting up the computation in minibatches greatly improves speed and probabilistically converges to the same value that standard GD would, pseudocode for the algorithm is found in Algorithm 1.

Algorithm 1 Stochastic Gradient Descent

Input: **S**: the training set, **net**: the neural network, **NumEpoch**: number of training epochs, **m**: size of the minibatches, **L**: loss function η: learning rate.

```
for $i \in NumEpoch$ do
    $R \leftarrow random\_shuffle(S)$
    divide $R$ in $\frac{|R|}{m}$ minibatches
    grad $\leftarrow 0$
    for each $B$ minibatch of $S$ do
        for $x \in B$ do
            grad $\leftarrow compute\_gradient(L, x, net.weights)$
            $net.weights \leftarrow net.weights - \eta \cdot$ grad
        end for
    end for
end for
```

2. **Adaptive Moment Estimation (ADAM)**: Introduced in 2014 [26], it is similar in concept to Stochastic Gradient Descent, but utilizes two moment terms to speed up and slow down the distance moved each iteration in accordance to how steep the gradient is. Each learnable parameter has its own moment terms, so each parameter gets updated differently. The algorithm for ADAM is shown in Algorithm 2 [60].

Between the two, ADAM was chosen because of its ability to converge faster, allowing us to adapt the pre-trained architectures to the new problem with fewer iterations. Despite this, as will be detailed in Section 9, some architectures still did not manage to converge within the given epochs.

The problem of RNA prediction applied to the tensor representation imposes hard constraints on the output of the computation. Since the networks used for the ImageNet competition are trained to process images of the same height and width, specifically 224×224 pixels, the output will also be constrained to be

Algorithm 2 ADAM

Input: **S**: the training set, **net**: the neural network, **NumEpoch**: number of training epochs, **L**: loss function η: learning rate, ρ_1, ρ_2: parameters, ϵ: small positive number to avoid division by 0.

```
1: $m_0 \leftarrow 0$
2: $u_0 \leftarrow 0$
3: $i \leftarrow 0$
4: while $i < NumEpoch$ do
5:     $i \leftarrow i + 1$
6:     $grad_i \leftarrow compute_gradients(L, S, net.weights)$
7:     $m_i \leftarrow \frac{\rho_1 m_{i-1} + (1-\rho_1) grad_i}{1-\rho_1}$
8:     $u_i \leftarrow \frac{\rho_2 u_{i-1} + (1-\rho_2) grad_i^2}{1-\rho_2}$
9:     $net.weights \leftarrow net.weights - \eta \cdot \frac{m_i}{\sqrt{u_i} + \epsilon}$
10: end while
```

some sort of N-dimensional tensor. Since the representation for the problem's solution is a symmetrical target matrix, and there is no layer in the architectures used to ensure a symmetrical output with respect to its diagonal, the only way to encourage the network to learn to enforce this constraint is to implement a custom loss function to penalize asymmetrical output. As such, all the architectures in this project were trained with a custom loss function composed of two terms:

- Mean Squared Error (MSE) between the neural network's output $\hat{Y}$ and the target matrix T for the folding with length n.

$$\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} (\hat{Y}[i,j] - T[i,j])^2$$

- Symmetry Error computed on the output $\hat{Y}$. Once again the mean square error between each component $\hat{Y}[i,j]$ and $\hat{Y}[j,i]$. In order to optimize the computation, the rows to the right of the matrix diagonal of $\hat{Y}$ and the columns below it are taken as arrays on which the MSE is computed, rather than iterating on each entry.

$$\sum_{i=1}^{n-1} \mathbf{MSE}(RightRow(\hat{Y}, i), LowerColumn(\hat{Y}, i))$$

The two terms are summed together to compute the final loss function, which encourages symmetrical output alongside closeness to the target matrix T, so that positive predictions will be close to a value of 1 and negative ones will be close to 0 in the final score matrix output.

7 Architecture and pre-trained backbones

In order to adapt previously trained CNNs for this problem, some degree of modification is necessary to the original architectures. For this, the general outline used for adapting pre-trained networks was to encode every RNA sequence in a 500×500, since the largest sequence in the dataset is 499 nts long. All sequences are zero-padded into 500×500 tensors as described in Section 3. The first adapatation that needs to be made is downsampling these inputs to coincide with the input size the original networks used. Since all of the pre-trained architectures that were developed for the ImageNet [11] competition at various points in time, they are all meant for processing of 224×224 size RGB inputs. All of the architectures were also tailored to perform classification on 1000 different categories, so the output layers are also removed and cut off at the point where the most deep features are preserved, usually of shape $7 \times 7 \times C$ where C is variable depending on the architecture. A general outline of the network is found in Figure 7.1, note that to help preserve as much information as possible through the convolutions, the input is copied 3 times over the 3 RGB channels.

The output layer rebuilds the target matrix with three progressive transposed convolutional layers that gradually increase the size of the image and reduce the number of channels, to balance the expansion of information. Each layer uses a Leaky ReLU activation function since it has been shown to work better when post-processing features when using transfer learning in [28].

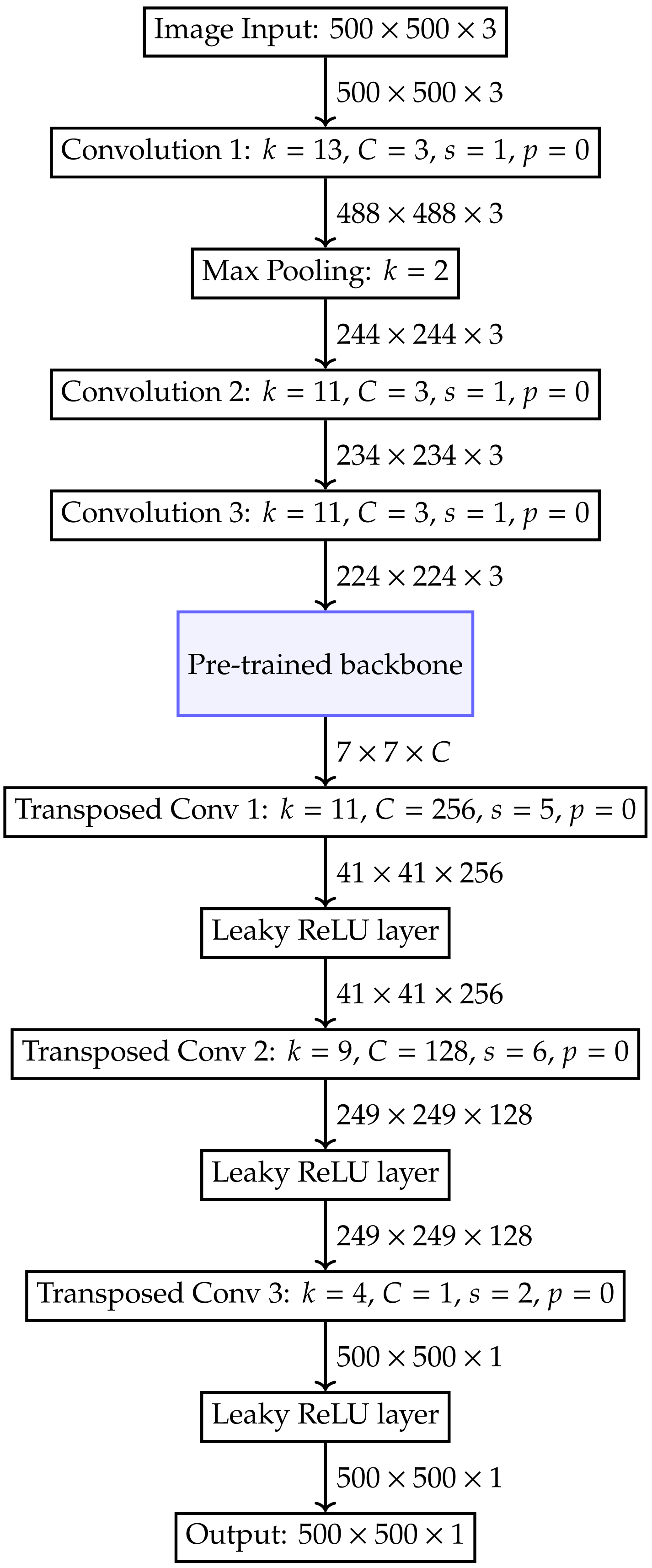
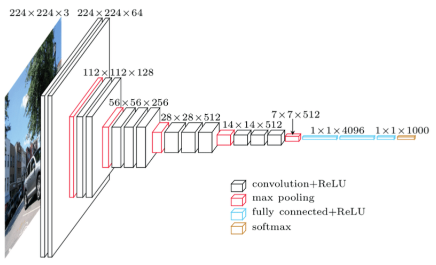


Image Input: $500 \times 500 \times 3$
$500 \times 500 \times 3$
Convolution 1: $k = 13, C = 3, s = 1, p = 0$
$488 \times 488 \times 3$
Max Pooling: $k = 2$
$244 \times 244 \times 3$
Convolution 2: $k = 11, C = 3, s = 1, p = 0$
$234 \times 234 \times 3$
Convolution 3: $k = 11, C = 3, s = 1, p = 0$
$224 \times 224 \times 3$
Pre-trained backbone
$7 \times 7 \times C$
Transposed Conv 1: $k = 11, C = 256, s = 5, p = 0$
$41 \times 41 \times 256$
Leaky ReLU layer
$41 \times 41 \times 256$
Transposed Conv 2: $k = 9, C = 128, s = 6, p = 0$
$249 \times 249 \times 128$
Leaky ReLU layer
$249 \times 249 \times 128$
Transposed Conv 3: $k = 4, C = 1, s = 2, p = 0$
$500 \times 500 \times 1$
Leaky ReLU layer
$500 \times 500 \times 1$
Output: $500 \times 500 \times 1$

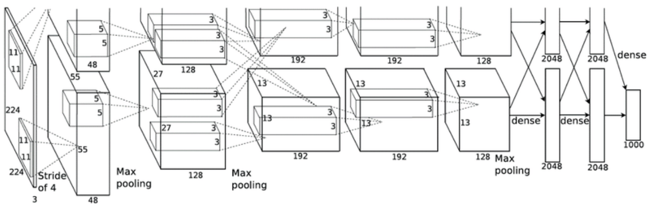
Figure 7.1: General outline of the structure of the networks used.

7.1 VGG16

VGG16 is the simplest architecture used and functions as the baseline with regard to how a CNN can be implemented. First proposed in 2014 [49], it is a straight-forward succession of 16 learnable layers, with its main innovation being the use of small 3×3 convolution kernels to reduce image dimensions very little and achieve far larger depth than something like AlexNet [27], which used larger kernels and as such could not be as deep.
This type of kernel is also the smallest possible that can maintain a notion of spatial relativity like up, down, left or right.
Being a deeper neural network provides an advantage in how it allows for more nonlinear activation functions and a more expressive network as a result [28].
Thanks to these intuitions, VGG16 was able to achieve more depth than its competitors, at the cost of a notably higher number of learnable parameters and as such long training and inference time.
A comparison between its architecture and that of AlexNet can be found in Figure 7.2, where we can see the 16 learnable layers of VGG16 (13 convolutional and 3 fully connected, the FC layers were removed from the backbone used in this project to keep the hidden features two-dimensional) contrasted with the 7 learnable layers of AlexNet (5 convolutional and 2 fully connected).
We can see that for this network, the C in output of the backbone is 512.



(a) VGG16 network architecture



(b) AlexNet network architecture

Figure 7.2: Comparison of the two architectures, images taken from [4]

7.2 GoogLeNet

GoogLeNet is a contemporary network to VGG16 [50], and its main idea is that of the inception block, a different stratagem to increase the depth of a convolutional network.
The inception block is composed of four parallel operations, which are:

1. a 1×1 convolution with ReLU activation.
2. a sequence of a 1×1 convolution with ReLU activation to reduce the number of channels and a 3×3 convolution with ReLU activation, with padding and stride that keep the same dimensions as the input.
3. a sequence of a 1×1 convolution with ReLU activation to reduce the number of channels and a 5×5 convolution with ReLU activation, with padding and stride that keep the same dimensions as the input.
4. a sequence of a 3×3 Max pooling layer and a 1×1 convolution layer with ReLU activation, with padding and stride that keep the same dimensions as the input.

The results of these four branches are the same size as the input matrix, and are then concatenated depth-wise into a larger output, as in Figure 7.3. To avoid the explosion of the hidden feature space, projections to reduce dimensionality are used to keep the hidden feature matrix under a certain size.
The effect of these inception blocks, placed after a series of regular convolution

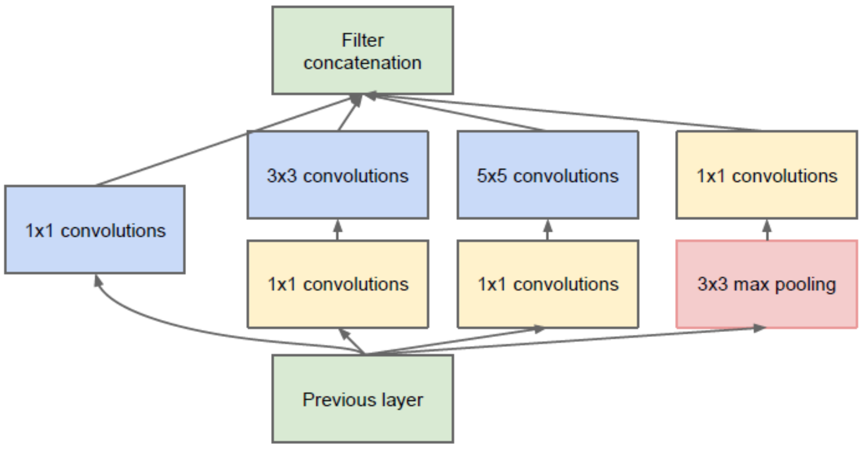


Figure 7.3: GoogLeNet's inception block visualized from [50]

layers to reduce the size of the input, is that they analyze receptive fields of different sizes in parallel, and as such have a more complete understanding of the whole picture at different levels of locality.

type	patch size/stride	output size	depth	#1 × 1	#3 × 3 reduce	#3 × 3	#5 × 5 reduce	#5 × 5	pool proj	params	ops
convolution	7 × 7/2	112 × 112 × 64	1							2.7K	34M
max pool	3 × 3/2	56 × 56 × 64	0								
convolution	3 × 3/1	56 × 56 × 192	2		64	192				112K	360M
max pool	3 × 3/2	28 × 28 × 192	0								
inception (3a)		28 × 28 × 256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28 × 28 × 480	2	128	128	192	32	96	64	380K	304M
max pool	3 × 3/2	14 × 14 × 480	0								
inception (4a)		14 × 14 × 512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14 × 14 × 512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14 × 14 × 512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14 × 14 × 528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14 × 14 × 832	2	256	160	320	32	128	128	840K	170M
max pool	3 × 3/2	7 × 7 × 832	0								
inception (5a)		7 × 7 × 832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7 × 7 × 1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7 × 7/1	1 × 1 × 1024	0								
dropout (40%)		1 × 1 × 1024	0								
linear		1 × 1 × 1000	1							1000K	1M
softmax		1 × 1 × 1000	0								

Table 7.1: A table representing the breakdown of GoogLeNet's layers from [50]

A full breakdown of GoogLeNet is found in Table 9.1.
For the project presented here, the network was cut off at the output of the final inception block, preserving a deep feature matrix of shape $7 \times 7 \times 1024$.

7.3 ResNet

ResNet architectures provided another big innovation on CNNs for ImageNet challenge [17], proposing its main idea of a residual block (shown in Figure 7.4), in which the input of a convolutional block is summed to its output with a skip connection, resulting in a capacity of expressing a much larger function class than a regular CNN.
Since each residual block essentially computes $\mathcal{F}(x) + x$, if the output of $\mathcal{F}(x)$ is zero, the function still returns information, making it so that, in terms of function classes, $\mathscr{F}_{non-res} \subseteq \mathscr{F}_{res}$.
If we see the whole output after the sum as $f(x)$, then the residual block is essentially computing $f(x) - x$, hence the name.

Residual blocks also allow the network to reach further depth, since it helps address the performance degradation of optimizing more and more parameters, and the vanishing gradient problem [3]. By connecting later layers to earlier activations, the magnitude of this effect is reduced, since larger values from previous layers are preserved.

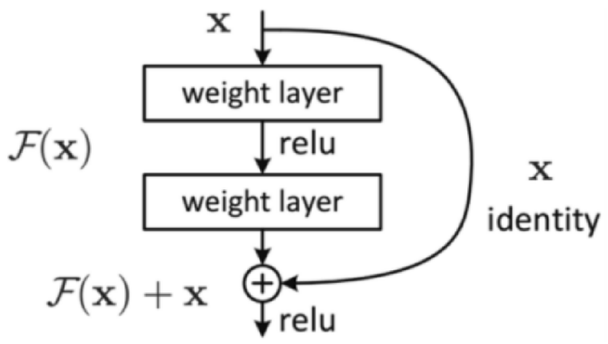


Figure 7.4: A schematic of the residual block from [17]

The second big innovation ResNets brought to the table is the use of Batch Normalization [25] which, instead of applying normalization by computing the whole training set S mean value $\hat{\mu}$ and variance $\hat{\sigma}^2$ and applying the transformation $\frac{x-\hat{\mu}}{\hat{\sigma}}$ for each $x \in S$, takes a minibatch split of the dataset $B \subset S$, and computes the mean $\hat{\mu}_B$ and variance $\hat{\sigma}_B^2$ on each x in the batch as such:

- **batch mean**:
$$\hat{\mu}_B = \frac{1}{|B|} \sum_{x \in B} x$$
- **batch variance**:
$$\hat{\sigma}_B^2 = \frac{1}{|B|} \sum_{x \in B} (x - \hat{\mu}_B)^2 + \epsilon$$
where ϵ is a small positive number to avoid division by zero.
- **batch normalization**:
$$BN(x) = \gamma \odot \frac{x - \hat{\mu}_B}{\hat{\sigma}_B} + \beta$$
where γ and β are learnable parameters that get updated during backpropagation. In CNNs, each channel has its own γ and β learnable parameters.

This operation serves to reduce the *internal covariate shift* of huge hidden feature vectors, which helps keep the distribution of inputs more consistent at deeper layers. It also allows for higher learning rates, and thus faster convergence, and reduced overfitting, as well as providing a small contribution to regularization.
For this project, the ResNet18 network is cut off just before the average pooling layer, outputting features of shape $7 \times 7 \times 512$.
Two more ResNets are used as well, with 50 and 101 weighted layers respectively, implemented with the same criteria.

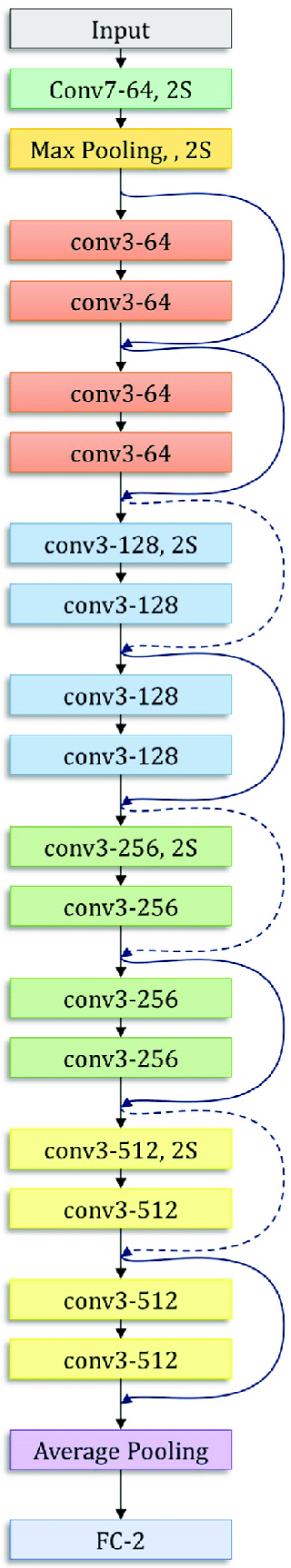


Figure 7.5: Architecture of ResNet18, composed of 18 weighted layers as the name suggests [29].

7.4 Xception

Xception is an iteration on the idea of the inception block started by GoogLeNet [7], with its main idea being that of a block that forms a middle ground between an inception block and a depth-wise separable convolution.
A depthwise separable convolution is a different implementation of the convolution operation described in Section 4, which splits the operation in two parts:

- **Depthwise convolution**: during this step, C_{in} kernels of shape $k \times k \times 1$ are used on each channel of the input image, and a convolution is performed separately for each channel. The output has the same number of channels as the input, illustrated in Figure 7.6.

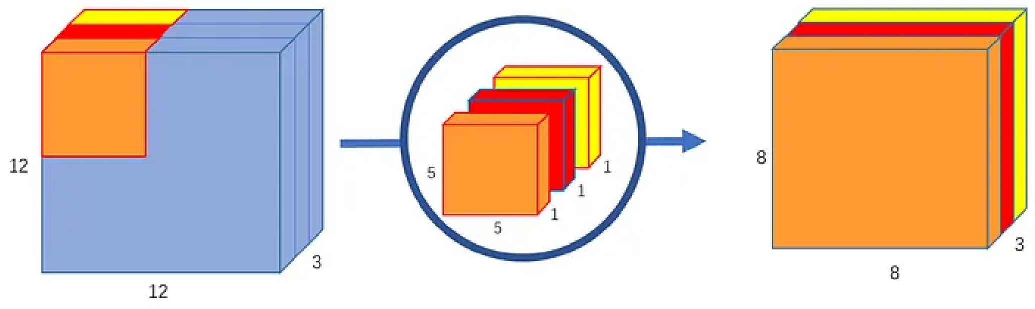


Figure 7.6: A depthwise convolution operation [55]

- **Pointwise convolution**: after the depthwise convolution, C_{out} kernels of shape $1 \times 1 \times C_{in}$ are applied to the intermediate result, and the final result is computed as in Figure 7.7.

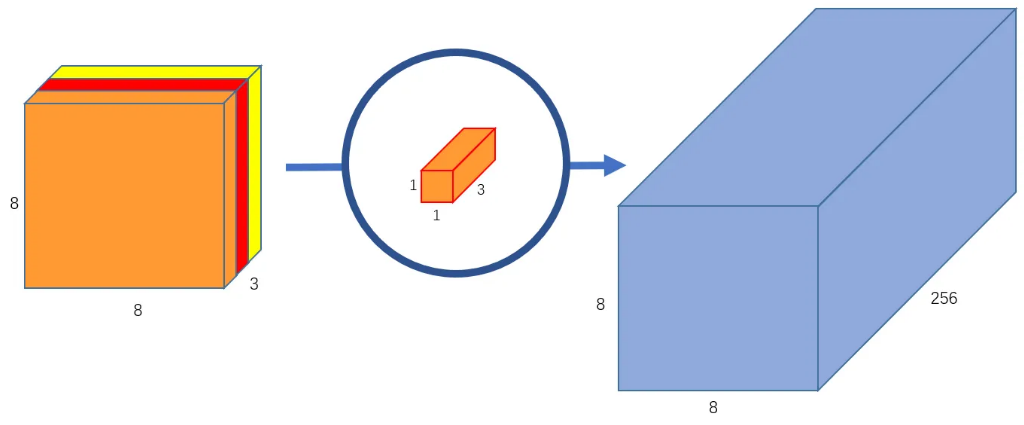
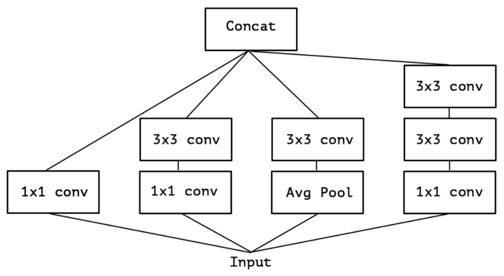
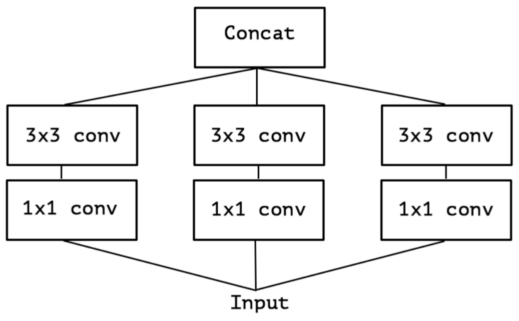


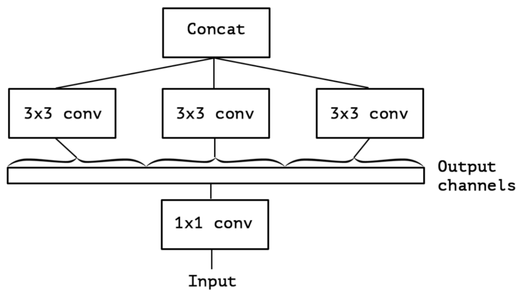
Figure 7.7: A pointwise convolution operation with 256 kernels [55]



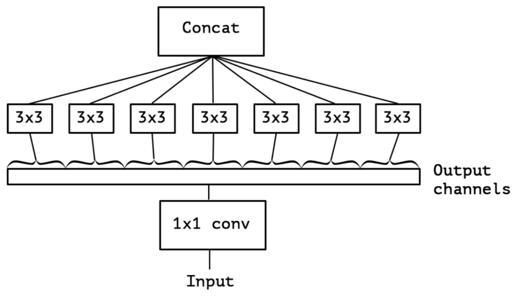
(a) The regular inception block from [50].



(b) An inception block modified by removing pooling and homogenizing the other branches.



(c) An equivalent formulation of (b)



(d) An "extreme" version of (c), a very large depthwise separable convolution.

Figure 7.8: Steps that can be taken to transform an inception block into a depthwise separable convolution [7]

These types of convolutions offer very similar performance and potential as regular convolution, with a much lower computational cost, for example taking an image of height, width and depth h_i, w_i and d_i respectively and convolving it with d_j kernels of shape $k \times k \times d_i$ would see regular convolution achieve a cost of $h_i \cdot w_i \cdot d_i \cdot d_j \cdot k^2$, while a depthwise separable convolution would only cost $h_i \cdot w_i \cdot d_i \cdot (d_j + k^2)$ [46]. The authors of [7] note that an inception block from GoogLeNet with the pooling branch removed and every other branch changed to the same kernel size is not far removed from the depthwise separable convolution operation, shown in Figure 7.8.

Xception adds some key modifications to the depthwise separable convolution operation to find a middle ground behind these two different but similar operations, which are:

1. The order of the depthwise and pointwise convolutions is swapped.
2. both the depthwise and pointwise convolution are followed by a ReLU layer.

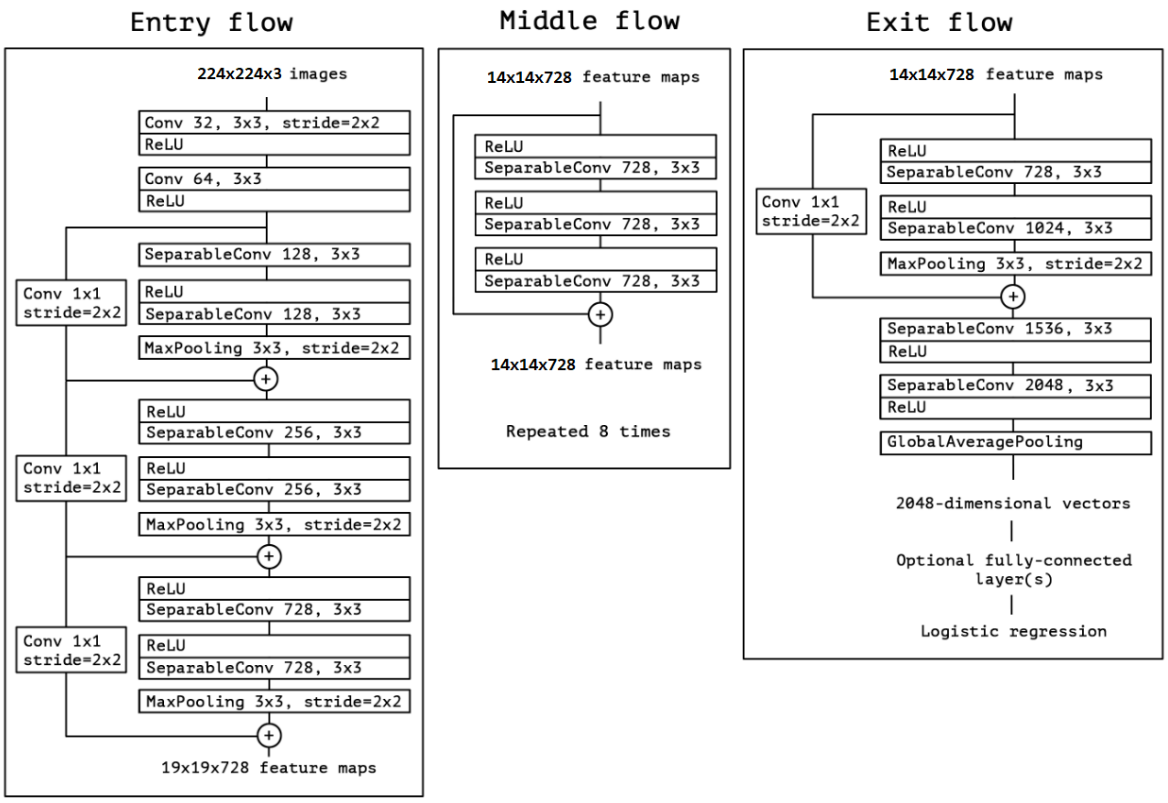
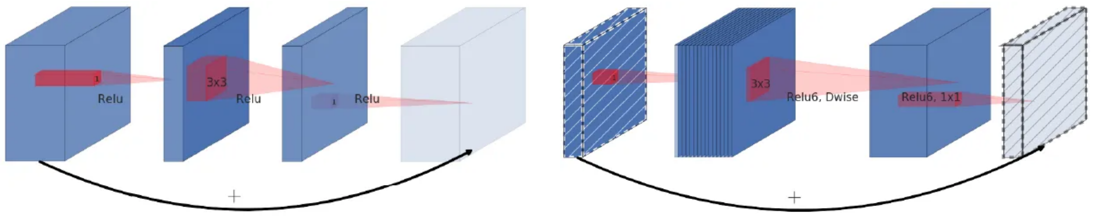


Figure 7.9: Full architecture of the Xception network [7]

The authors argue that the first point is unimportant, but the second is of relevance. The architecture of Xception is shown in Figure 7.9. Each depthwise separable convolution is implemented with the aforementioned modifications. In this project, the architecture is cut off before the fully connected layers, outputting $7 \times 7 \times 2048$ deep features.

7.5 MobileNet v2

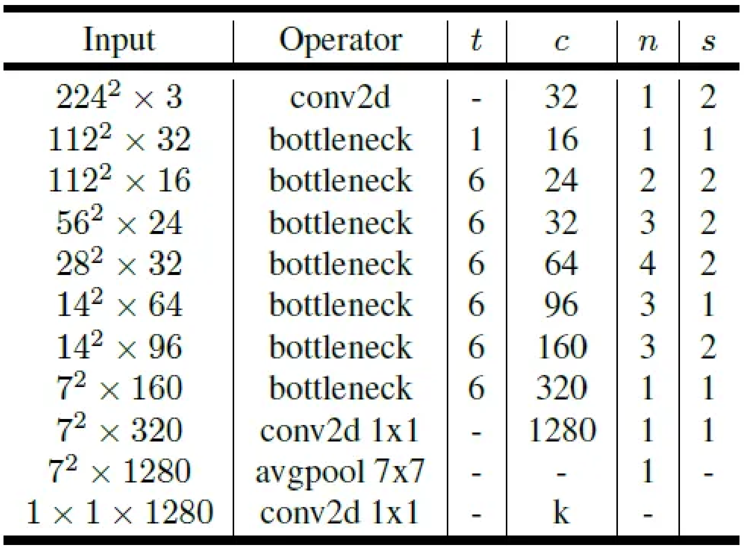
MobileNet v2 [46] is the second iteration on the idea of depthwise separable convolutions, used by MobileNet v1, with the main intuition being the idea of linear bottleneck layers. A linear bottleneck layer functions as an inverse of a residual layer from ResNet [17], operating off the basic assumption that all CNNs make that highly information dense inputs can be encoded into some low-dimensional subspace. As seen in Figure 7.10, while a residual block's inner layers reduce the number of channels in the inner steps, linear bottlenecks expand the channels and then contract them again at the output.



(a) The regular residual block. (b) The linear bottleneck layer.

Figure 7.10: Showcase of the differences between residual blocks and linear bottleneck blocks [46].

The residual block compresses the input with a 1×1 convolution first that reduces the number of channels, which means reducing the number of parameters the following 3×3 kernel needs to learn, after which the channels are expanded again with another 1×1 convolution.
Furthermore the authors argue that while ReLU allows for greater expression thanks to its non-linear properties, it still results in an overall loss of information from the previous layers.
Conversely, the linear bottleneck block multiplies the number of channels by an expansion factor t, so the input is greatly expanded before the non-linearity forces some information loss. After increasing the channels with a 1×1 convolution, a 3×3 convolution is applied, and the output is then reduced to the same number of channels as the input with another 1×1 convolution.
A residual connection is used to once again facilitate the backpropagation of the gradient and ensure greater depth.
Another important thing to note is that after every convolution, batch normalization is applied and the activation function used is ReLU6, a variant of ReLU that only allows values in the $[0, 6]$ interval to pass through, since it's been observed to be more robust in low-precision computation [21], which uses less bits to encode weights to save on space.



Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 7.11: The architecture of the MobileNet v2 network [56]

For the purposes of transfer learning, the architecture, shown in Figure 7.11, is cut off before the average pooling layer, resulting in features of shape $7 \times 7 \times 1280$.

7.6 EfficientNet

The latest of the networks used, EfficientNet is a vast optimization of previous architectures [53], here used in its B0 architecture.
The key intuition behind EfficientNet is reducing the number of learnable parameters of CNNs in a way that still keeps most of the proficiency intact.
The focus of it is the idea of scaling a CNN optimally in order to find the best balance between depth, width and resolution using a compound scaling method.

In general, different ways of scaling are illustrated in Figure 7.12, and can be divided in three types:

- **Depth scaling**: increasing the number of layers in the network. Intuitively this helps networks capture more complex features, but has the drawback of increasing the risk of vanishing gradient [3] and reaches a point of diminishing returns even when the problem is addressed.
- **Width scaling**: increasing the number of channels of the hidden convolutional layers. It's helpful for smaller sized models and can make them easier to train, but it can make it difficult to capture higher level features and saturate quickly in accuracy.

- **Resolution scaling**: increasing the size of the input images can offer an advantage in recognizing features. This type of scaling is very expensive and it also reaches a point of diminishing returns for very high resolutions.

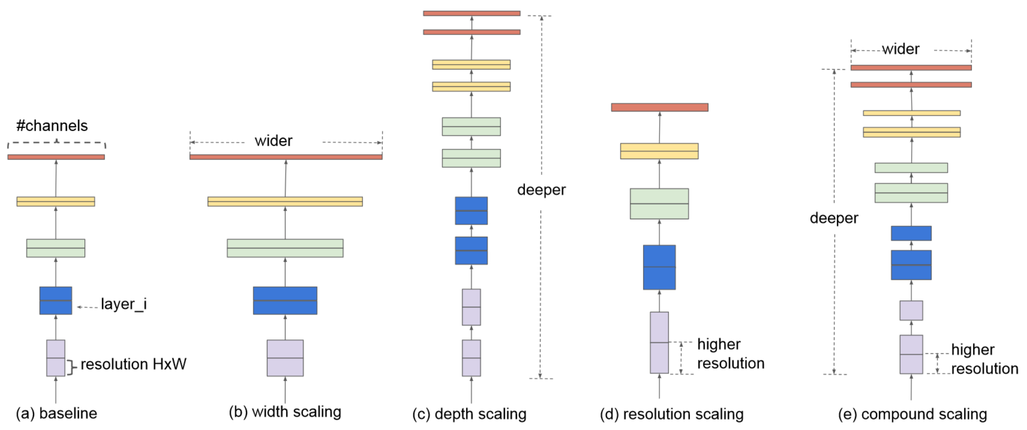


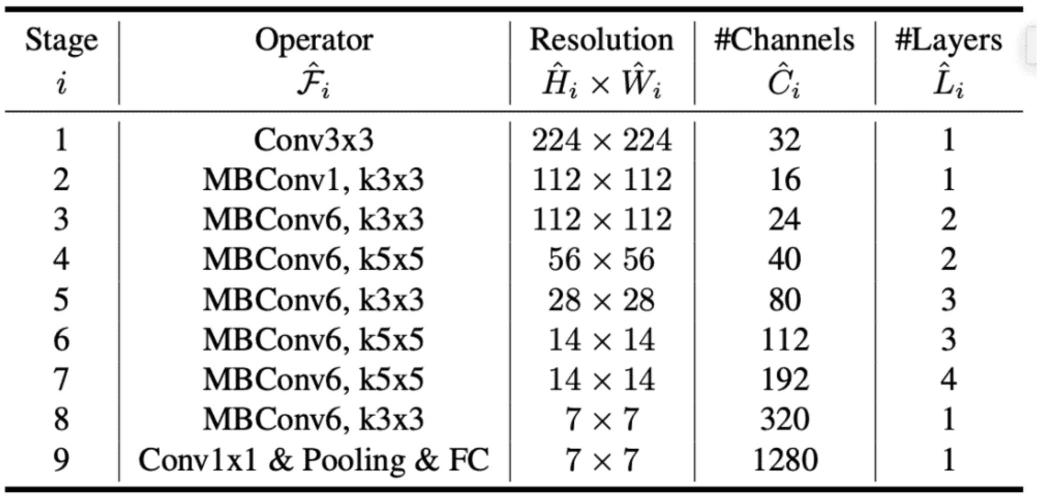
Figure 7.12: Different ways of scaling a CNN [53]

The key intuition behind compound scaling is to use this policy to scale up a smaller CNN:

- **depth**: $d = \alpha^{\phi}$
- **width**: $w = \beta^{\phi}$
- **resolution**: $r = \gamma^{\phi}$

such that $\alpha + \beta^2 + \gamma^2 \approx 2$, with $\alpha \geq 1, \beta \geq 1$ and $\gamma \geq 1$.
ϕ is a hyperparameter that controls how large the scaling will be and the resources to be employed. This ensures that the Floating-point Operations Per Second (FLOPS) will only increase by a factor of 2^{ϕ}.

EfficientNet-B0 is a CNN architecture that has been optimally scaled up with this rule in mind, with Mnas-Net as a baseline architecture [54], which is targeted to run on mobile phone devices and as such very lightweight.
Its efficiency is several times that of a ResNet50 model, reducing the computational load in FLOPS by a factor of 16, while achieving better accuracy on the ImageNet dataset [53].



Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Figure 7.13: The architecture of EfficientNet-B0 [53]

The architecture of EfficientNet-B0 can be seen in Figure 7.13, where the MBConv layers are inverted bottlenecks like in the MobileNet v2 architecture above. The features are extracted right before the final fully connected (FC) layer, and have shape $7 \times 7 \times 1280$.

8 Dataset & Methods Overview

The dataset used to benchmark the performance of this project is obtained from the bpRNA [41], split into the TR0 and TS0 datasets for training and testing respectively, composed of 12119 RNA sequences of length ranging from 22 to 499 nucleotides long. The dataset is split in approximately 90% (10814) sequences for training in TR0 and 10% (1305) sequences for testing in TS0, using the same split as the reference papers [1] and [6] to ensure that the results are comparable.

The dataset was obtained by combining over 100.000 known RNA structures from various databases, using an automatic annotation algorithm, and reducing it to a subset of high quality non-redundant sequences, by cutting off the sequences with over 80% sequence identity [2]. The result is a dataset of highly differentiated RNA secondary structures like the ones illustrated in Figure 8.1. On top of that, the dataset's size is also reduced due to computational limitation, retaining only sequences shorter than 500 nt.

The dataset contains all types of RNA secondary structures including pseudoknots, making it a particularly challenging benchmark.

The source files for this dataset were obtained from the github page of the "RNA secondary structure prediction with convolutional neural networks" project [1] at this link:
`https://github.com/mehdi1902/RNA-secondary-structure-prediction-using-CNN/tree/master/datasets`.

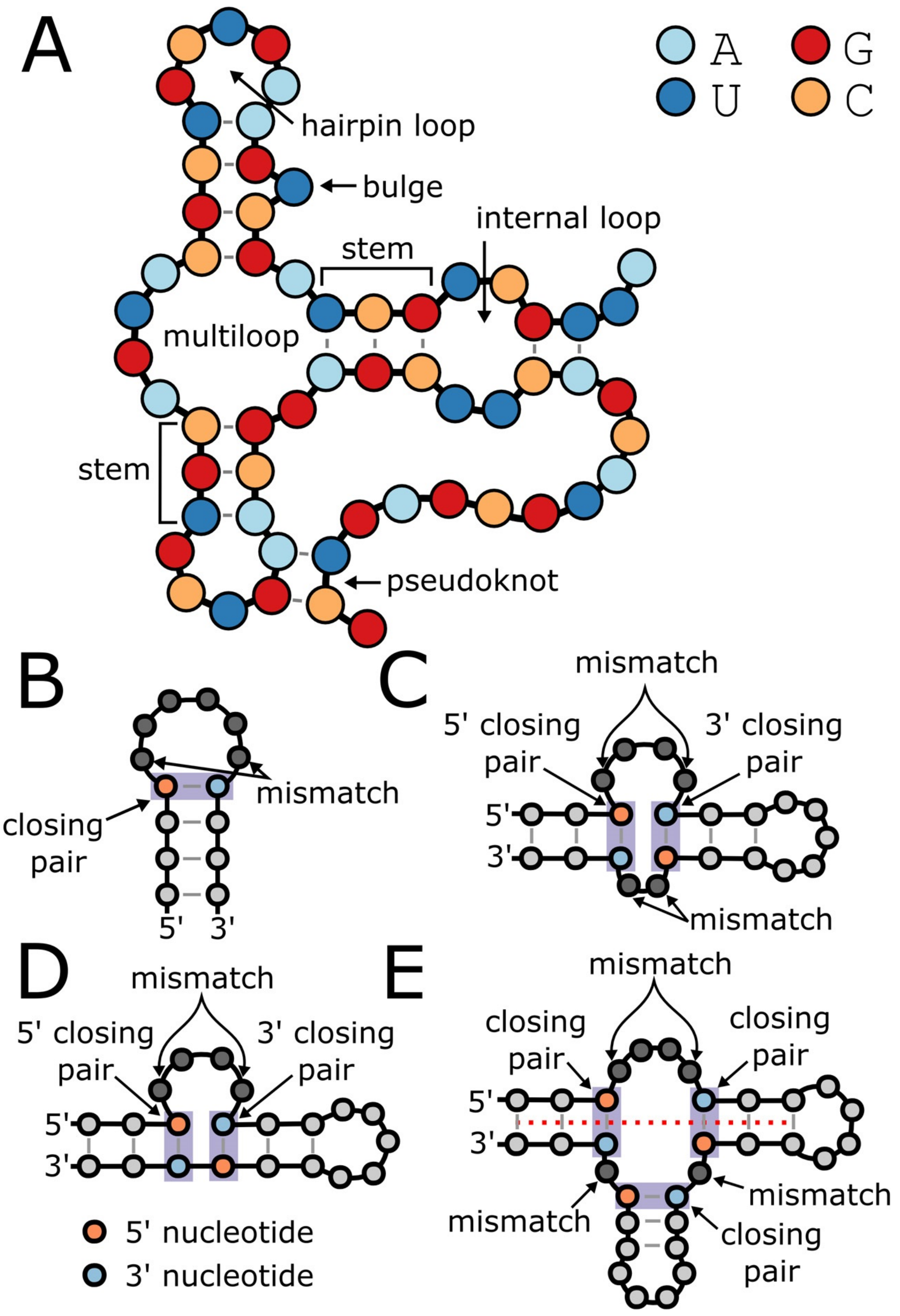
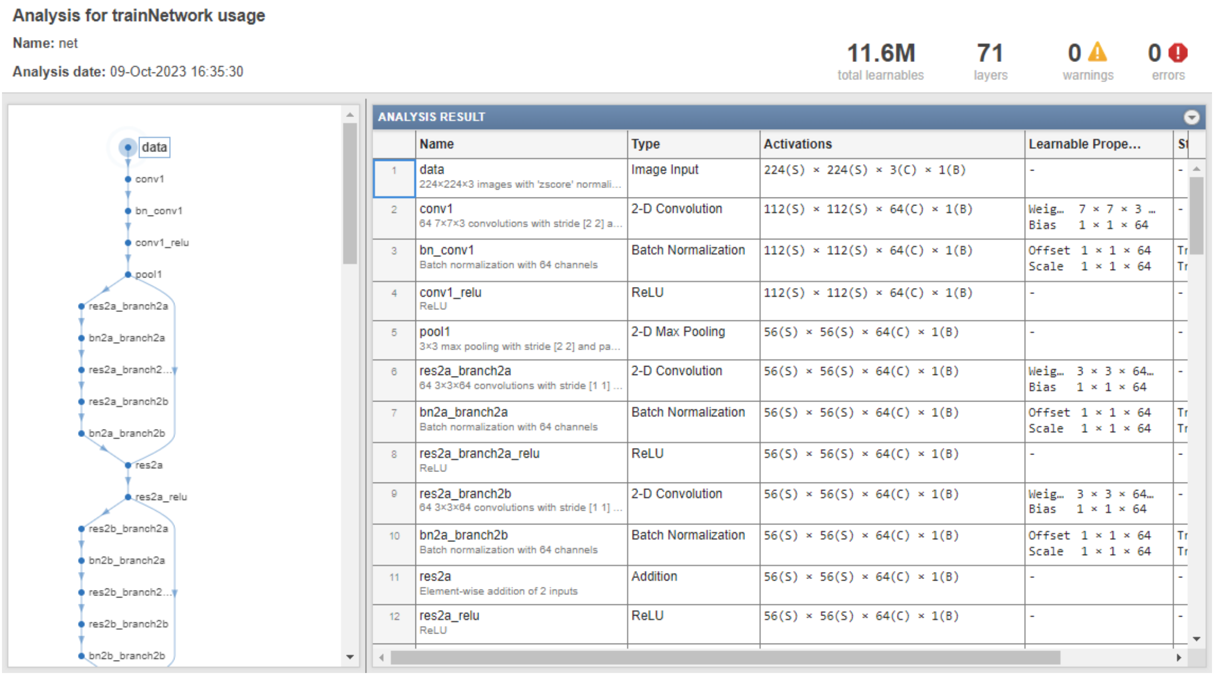


Figure 8.1: Distinctive types of RNA structures present in bpRNA:
(A): schematic of all structure types, (B): hairpin structure, (C): internal loops, (D): bulges, (E): multiloops [41].



Analysis for trainNetwork usage

Name: net

Analysis date: 09-Oct-2023 16:35:30

11.6M total learnables | 71 layers | 0 warnings | 0 errors

ANALYSIS RESULT

	Name	Type	Activations	Learnable Prope...	St
1	data 224×224×3 images with 'zscore' normali...	Image Input	224(S) × 224(S) × 3(C) × 1(B)	-	-
2	conv1 64 7×7×3 convolutions with stride [2 2] a...	2-D Convolution	112(S) × 112(S) × 64(C) × 1(B)	Weig... 7 × 7 × 3 ... Bias 1 × 1 × 64	-
3	bn_conv1 Batch normalization with 64 channels	Batch Normalization	112(S) × 112(S) × 64(C) × 1(B)	Offset 1 × 1 × 64 Scale 1 × 1 × 64	Tr Tr
4	conv1_relu ReLU	ReLU	112(S) × 112(S) × 64(C) × 1(B)	-	-
5	pool1 3×3 max pooling with stride [2 2] and pa...	2-D Max Pooling	56(S) × 56(S) × 64(C) × 1(B)	-	-
6	res2a_branch2a 64 3×3×64 convolutions with stride [1 1] ...	2-D Convolution	56(S) × 56(S) × 64(C) × 1(B)	Weig... 3 × 3 × 64... Bias 1 × 1 × 64	-
7	bn2a_branch2a Batch normalization with 64 channels	Batch Normalization	56(S) × 56(S) × 64(C) × 1(B)	Offset 1 × 1 × 64 Scale 1 × 1 × 64	Tr Tr
8	res2a_branch2a_relu ReLU	ReLU	56(S) × 56(S) × 64(C) × 1(B)	-	-
9	res2a_branch2b 64 3×3×64 convolutions with stride [1 1] ...	2-D Convolution	56(S) × 56(S) × 64(C) × 1(B)	Weig... 3 × 3 × 64... Bias 1 × 1 × 64	-
10	bn2a_branch2b Batch normalization with 64 channels	Batch Normalization	56(S) × 56(S) × 64(C) × 1(B)	Offset 1 × 1 × 64 Scale 1 × 1 × 64	Tr Tr
11	res2a Element-wise addition of 2 inputs	Addition	56(S) × 56(S) × 64(C) × 1(B)	-	-
12	res2a_relu ReLU	ReLU	56(S) × 56(S) × 64(C) × 1(B)	-	-

Figure 8.2: Analysis of the ResNet18 neural network within Matlab.

The project was implemented in MathWorks' Matlab environment [35], which offers powerful tools for deep learning and transfer learning, and allows for custom training loops and implementation of layers and loss functions from scratch.
Matlab also offers intuitive tools to visualize neural network architecture like in Figure 8.2, making editing and debugging more convenient.

8.1 Implementation details

In order to have a baseline for comparison, the smallest version of CNNFold [1] was implemented in Matlab from scratch and trained on the TR0 training set, while testing protocols used the TS0 test set to compute key performance indicators such as precision, recall and F_1 score.
The output of the neural network $\hat{Y}$ is still not guaranteed to be symmetrical or boolean, so some amount of post-processing is still needed to ensure it can be compared to the target matrix T.
To do so, the first step is to enforce binary output by applying this algorithm:

```
function out = Binarize(Y)
    L = length(Y);
    % Y is a dlarray, this line turns it back into an array
    Y = extractdata(Y);
    out = zeros(L,L,"logical");
    [~,maxColIndex] = max(Y);
    for i=1:length(maxColIndex)
        out(maxColIndex(i),i)=1;
    end
end
```

And so $\hat{Y}_B$ is obtained by picking the highest score in each column of $\hat{Y}$, setting it to 1, and then setting the rest of the column to 0. This is called ArgMax post-processing and it has been proven to perform fastest for this type of problem without significant downsides [1].
Afterwards, the matrix $\hat{Y}_B$ is made symmetrical using this formula:

$$\hat{T} = \left\lceil 0.5 \cdot (\hat{Y}_B + \hat{Y}_B^T) \right\rceil$$

The way these metrics are computed on the target matrix T and the output matrix $\hat{T}$ is as such:

- True Positives (TP): count of the (i, j) positions where $T = 1$ & $\hat{T} = 1$.
- False Positives (FP): count of the (i, j) positions where $T = 0$ & $\hat{T} = 1$.
- False Negatives (FN): count of the (i, j) positions where $T = 1$ & $\hat{T} = 0$.

The test set TS0 is divided in 3 subdivisions according to sequence length, where each subdivision is determined with a step given by the formula $s = \lfloor \frac{499-22}{3} \rfloor$, where 22 is the minimum sequence length, 499 the maximum sequence length and 3 the number of subdivisions.
This way we can analyze the results for sequences shorter than $22 + s$, $22 + 2s$ and the rest, plus the entire dataset to see the impact sequence length has on performance.
The subdivisions were as such:

1. **First subdivision**: sequences shorter than 181 nt, 1060 in total, the large majority of the test set.
2. **Second subdivision**: sequences between 182 and 340 nt long, 166 in total.

3. **Third subdivision**: sequences longer than 340 nt, 79 in total.
4. **Fourth subdivision**: the entire test set of 1305 sequences.

Then all the TP, FP and FN are separately summed for each corresponding pair $(T, \hat{T}) \in$ (TS0 , Predictions(TS0)), and also for the three smaller subdivisions. Once the total statistics of TP, FP and FN are computed, the precision, recall and F_1 score metrics are computed with these formulas:

- **Precision** = $\frac{TP}{TP+FP}$, intuitively, the rate of correct positive predictions among all positive predictions.
- **Recall** = $\frac{TP}{TP+FN}$, intuitively, the rate of correct positive predictions among all true positives.
- **F_1 score** = $2 \cdot \frac{Precision \cdot Recall}{Precsion+Recall}$, intuitively the harmonic mean of precision and recall.

All the CNNs used for this project were trained with the same parameters, which were:

- **Epochs**: 30 epochs of training on the training set TR0.
- **Mini-batch size**: each mini-batch contained 5 sequences from the training set.
- **Optimizer**: ADAM optimizer for faster convergence on the comparatively small RNA training set.
- **Shuffle**: The training set was randomly shuffled before processing.
- **Learning rate**: constant learning rate of value 0.001.
- **Loss Function**: MSE with symmetry penalty from Section 6.

The training was performed on the University of Padova's Blade cluster [13], utilizing an RTX3090 graphics card for the computation. The results are compiled in Section 9.

9 Results and Discussion

All the different architectures were tested against the baseline of the smallest CNNFold model [1] implemented in Matlab and trained with the same parameters as the ones detailed in Section 8, minus the custom loss function, which was instead standard MSE.
The final results for all subdivisions and metrics can be seen in Table 9.1

Architecture	**L<=181**			**181<L<=340**			**L>340**			**whole test set**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
Baseline (CNNFold) [1]	0.549	0.549	0.549	0.532	0.532	0.532	0.488	0.488	0.488	0.535	0.535	0.535
GoogLeNet backbone	0.529	0.541	0.535	0.236	0.332	0.276	0.113	0.186	0.140	0.356	0.433	0.390
ResNet18 backbone	0.553	0.553	0.553	0.540	0.541	0.540	0.491	0.494	0.492	0.539	0.540	0.539
ResNet50 backbone	0.553	0.553	0.553	0.542	0.542	0.542	0.495	0.496	0.495	0.540	0.540	0.540
ResNet101 backbone	0.552	0.552	0.552	0.542	0.542	0.542	0.485	0.490	0.487	0.538	0.539	0.538
Xception backbone	0.474	0.509	0.491	0.196	0.289	0.234	0.180	0.259	0.212	0.340	0.417	0.375
MobilenNet v2 backbone	0.553	0.553	0.553	0.418	0.473	0.444	0.321	0.391	0.353	0.475	0.507	0.490
EfficientNet-B0 backbone	**0.553**	**0.553**	**0.553**	**0.542**	**0.542**	**0.542**	**0.496**	**0.497**	**0.497**	**0.540**	**0.540**	**0.540**

Table 9.1: The results of the tests performed on the Blade cluster [13].

The first immediate observation is that "old school" CNNs struggle severely with the problem: VGG16 does not converge to any meaningful degree and does not surpass 10% F_1 score in any category.
GoogLeNet shows vast improvements over VGG16 in terms of KPIs on sequences of length less than 181 bp, but still quickly falls off on longer sequences. This could be due to the proportional lack of longer sequences in the training set, making this network unable to adapt to them.
The first real improvement comes from the ResNet18 architecture, managing to outclass all the baseline metrics by a few points.

Interestingly, the ResNet50 architecture manages to perform slightly better, but this advantage falls off when extended even further with ResNet101, which ends up with worse performance than the ResNet18 model. Residual architectures seem nonetheless to be best suited to this problem, seeing as how EfficientNet-B0 also obtains the best performance in every KPI, outclassing the baseline, even if by a little amount, and could potentially perform even better when scaled up to a larger architecture like B3 to B7.
Xception shows that inception architectures aren't as well suited to the problem either, performing even worse than GoogLeNet on all fronts, perhaps due to the lack of data on larger sequences.
The impact of transfer learning in this instance seems to be minimal, although the state of the art is still fairly inaccurate on this problem, with the best results obtained being those shown by SPOT-RNA [2], achieving an F_1 score of 0.597, while the best version of CNNFold [1] managed an F_1 score of 0.582.
SPOT-RNA uses a combination of transfer learning and ensemble learning using ResNet models and also Long Short-Term Memory (LSTM) models to build a more diverse pipeline, so the intuition was to try to do the same with the various architectures presented here.

9.1 Ensemble tests

Ensemble learning refers to the idea of combining the predictions of different machine learning models in order to obtain a more "democratic" prediction by having different and diverse algorithms pool their results together to come to a conclusion [42].
To combine the knowledge of multiple models, the way it was implemented in this project, it was sufficient to obtain the final score output matrices $\hat{Y}$ from each model and sum them together to generate an aggregate result, a process which is called sum rule, illustrated in Figure 9.1.
Once all model results were aggregated, the standard post-processing pipeline described in Section 8 was used.

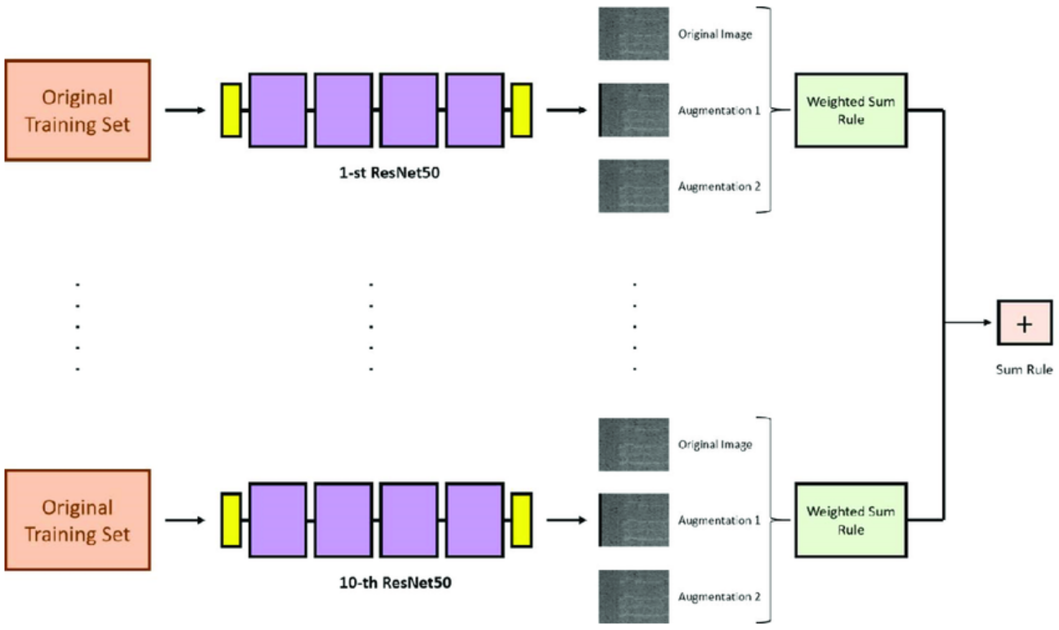


Figure 9.1: An example of a meta-ensemble (an ensemble of ensembles) using the sum rule [38]

Architecture	**L<=181**			**181<L<=340**			**L>340**			**whole test set**		
	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
MobileNet v2+GoogLeNet	0.553	0.553	0.553	0.428	0.480	0.452	0.307	0.381	0.340	0.474	0.507	0.490
ResNet18+GoogLeNet	0.553	0.553	0.553	0.540	0.540	0.540	0.491	0.494	0.493	0.539	0.540	0.539
EfficientNet-B0+GoogLeNet	0.553	0.553	0.553	0.542	0.542	0.542	0.496	0.497	0.497	0.540	0.541	0.540
ResNet18+EfficientNet-B0	0.553	0.553	0.553	0.542	0.542	0.542	0.497	0.497	0.497	0.541	0.541	0.541
ResNet50+EfficientNet-B0	0.553	0.553	0.553	0.542	0.542	0.542	0.497	0.497	0.497	0.541	0.541	0.541
EfficientNet-B0+MobileNet v2	0.553	0.553	0.553	0.460	0.499	0.479	0.489	0.492	0.491	0.520	0.530	0.525
ResNet18+EfficientNet-B0+GoogLeNet	0.553	0.553	0.553	0.542	0.542	0.542	0.497	0.497	0.497	0.541	0.541	0.541
ResNet18+EfficientNet-B0+MobileNet v2	0.553	0.553	0.553	0.479	0.509	0.493	0.493	0.494	0.493	0.525	0.533	0.529
ResNet18+EfficientNet-B0+MobileNet v2+GoogLeNet	0.553	0.553	0.553	0.483	0.511	0.496	0.493	0.494	0.493	0.526	0.533	0.530
CNNFold+ResNet18	**0.563**	**0.569**	**0.566**	**0.544**	**0.546**	**0.545**	**0.497**	**0.499**	**0.498**	**0.548**	**0.552**	**0.550**
CNNFold+ResNet50	0.561	0.565	0.563	0.543	0.544	0.544	0.497	0.499	0.498	0.546	0.549	0.547
CNNFold+ResNet101	0.560	0.564	0.562	0.543	0.544	0.544	0.496	0.499	0.498	0.545	0.548	0.547
CNNFold+MobileNet v2	0.563	0.568	0.565	0.455	0.499	0.476	0.480	0.495	0.487	0.523	0.540	0.531
CNNFold+EfficientNet-B0	0.562	0.566	0.564	0.544	0.545	0.544	0.498	0.500	0.499	0.547	0.550	0.548

Table 9.2: The results of the ensemble tests performed on the Blade cluster [13].

From Table 9.2 we can see that ensembles of the architectures that use transfer learning don't improve performance in any significant way, oftentimes being detrimental to the better performing individual element. From this we can gather that these CNN models are not statistically diverse enough between each other to provide useful contributions to the ensemble, and need to be integrated with other network architectures.
Particularly, all metrics seem to plateau around those of the best performing architecture, to the point that every ensemble of pretrained backbones achieves identical performance on the first subdivision of sequences shorter than 181 nt, always mirroring the results of the best individual part.
However, fusions with the CNNFold method trained from scratch without the use of transfer learning perform a fair bit better, outclassing the results of the individual parts and performing the best out everything presented, surpassing the best single-architecture network by a full percentage with an F_1 score of 55% on the whole dataset.
This suggests that there is a difference between transfer learning and learning from scratch that leads to the networks focusing on different aspects of the data representation, which then serves to strengthen the overall fusion.

10 Conclusions and Future Works

This thesis examined the efficacy and potential of transfer learning applied to various types of convolutional neural networks for the purposes of RNA secondary structure prediction. The results show some promise in the pursuit of transfer learning with compact architectures like EfficientNet, and some further research could be made studying an optimal way to scale the network for this specific problem. Another improvement was obtained by combining networks that make use of transfer learning with networks that were trained from scratch for just this problem, which seems to lead to good results when fusing their knowledge together through an ensemble.

Residual networks seem to be best suited to the task among CNNs, with the other architectures falling behind by a wide margin, so another potential direction to take the research could be adding more diverse architectures, not just convolutional, to assist in the building of an ensemble.

This could be achieved through different representations for the input data, different backbones for transfer learning or from-scratch training, or other adaptations like new loss functions to incentivize learning of the constraints on the output.

As demonstrated by the state of the art's performance still being unable to break 60% F_1 score on the bpRNA benchmark [1], there is still much room for improvement when it comes to fast, ML-based algorithms for RNA secondary structure prediction comprehensive of pseudoknots, although the length of potential RNA sequences can still be a large obstacle to overcome, seeing how this project only focused on small molecules no longer than 500 nt.

References

[1] Saman Booy M. Ilin A. and Orponen P. "RNA secondary structure prediction with convolutional neural networks". In: *BMC Bioinformatics* 23 (2022), p. 58. DOI: 10.1186/s12859-021-04540-7.

[2] Singh J. Hanson J. Paliwal K. et al. "RNA secondary structure prediction using an ensemble of two-dimensional deep neural networks and transfer learning". In: *Nature Communications* 10 (2019), p. 5407. DOI: 10.1038/s41467-019-13395-9.

[3] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[4] Baanin Dakula N. Bezdan T. "Convolutional Neural Network Layers and Architectures". In: *Sinteza 2019 - International Scientific Conference on Information Technology and Data Related Research*. 2019, pp. 445–451. DOI: 10.15308/Sinteza-2019-445-451.

[5] Wang L. Liu Y. Zhong X. Liu H. Lu C. Li C. and Zhang H. "DMfold: A Novel Method to Predict RNA Secondary Structure With Pseudoknots Based on Deep Learning and Improved Base Pair Maximization Principle". In: *Frontiers in genetics* 10 (2019), p. 143. DOI: 10.3389/fgene.2019.00143.

[6] Xinshi Chen et al. "RNA Secondary Structure Prediction By Learning Unrolled Algorithms". In: (2020). DOI: 10.48550/arXiv.2002.05810. arXiv: 2002.05810 [cs.LG].

[7] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 1800–1807. DOI: 10.1109/CVPR.2017.195.

[8] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Math. Control Signal Systems* 2 (1989), pp. 303–314. DOI: https://doi.org/10.1007/BF02551274.

[9] Richard M. Watson David H. Mathews Douglas H. Turner. “RNA Secondary Structure Prediction”. In: *Current protocols in nucleic acid chemistry* 67 (2016), pp. 11.2.1–11.2.19. DOI: 10.1002/cpnc.19. URL: https://doi.org/10.1002/cpnc.19.

[10] *Deep Learning Specialization course*. https://www.deeplearning.ai/program/deep-learning-specialization/. Accessed: July 2019.

[11] Jia Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[12] Li Deng and Dong Yu. “Deep Learning: Methods and Applications”. In: *Foundations and Trendső in Signal Processing* 7.34 (2014), pp. 197–387. ISSN: 1932-8346. DOI: 10.1561/2000000039. URL: http://dx.doi.org/10.1561/2000000039.

[13] *Dipartimento di Ingegneria dell'Informazione*. https://www.dei.unipd.it/bladecluster. Accessed: September 2023.

[14] *Dive Into Deep Learning*. https://d2l.ai/chapter_computer-vision/transposed-conv.html. Accessed: September 2023.

[15] *Dive Into Deep Learning*. https://d2l.ai/chapter_computer-vision/fine-tuning.html. Accessed: September 2023.

[16] Hassan Hassan et al. “ASSESSMENT OF ARTIFICIAL NEURAL NETWORK FOR BATHYMETRY ESTIMATION USING HIGH RESOLUTION SATELLITE IMAGERY IN SHALLOW LAKES: CASE STUDY EL BURULLUS LAKE.” In: *International Water Technology Journal* 5 (Dec. 2015).

[17] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. DOI: 10.48550/arXiv.1512.03385. arXiv: 1512.03385 [cs.CV].

[18] Tetsuro Hirose, Yuichiro Mishima, and Yukihide Tomari. “Elements and machinery of non-coding RNAs: toward their taxonomy”. In: *EMBO reports* 15.5 (2014), pp. 489–507. DOI: https://doi.org/10.1002/embr.201338390. eprint: https://www.embopress.org/doi/pdf/10.1002/embr.201338390. URL: https://www.embopress.org/doi/abs/10.1002/embr.201338390.

[19] Ivo L. Hofacker, Martin Fekete, and Peter F. Stadler. "Secondary Structure Prediction for Aligned RNA Sequences". In: *Journal of Molecular Biology* 319.5 (2002), pp. 1059–1066. ISSN: 0022-2836. DOI: https://doi.org/10.1016/S0022-2836(02)00308-X. URL: https://www.sciencedirect.com/science/article/pii/S002228360200308X.

[20] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[21] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[22] Minyoung Huh, Pulkit Agrawal, and Alexei A. Efros. *What makes ImageNet good for transfer learning?* 2016. DOI: 10.48550/arXiv.1608.08614. arXiv: 1608.08614 [cs.CV].

[23] Miklós I. Meyer I.M. and Nagy B. "Moments of the Boltzmann distribution for RNA secondary structures". In: *Bulletin of Mathematical Biology* 67 (2005), pp. 1031–1047. DOI: 10.1016/j.bulm.2004.12.003.

[24] Daniel Jiwoong Im et al. *Generating images with recurrent adversarial networks*. 2016. DOI: 10.48550/arXiv.1602.05110. arXiv: 1602.05110 [cs.LG].

[25] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/arXiv.1502.03167. arXiv: 1502.03167 [cs.LG].

[26] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980 [cs.LG].

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[28] Nalinda Kulathunga et al. "Effects of Nonlinearity and Network Architecture on the Performance of Supervised Neural Networks". In: *Algorithms* 14.2 (2021). ISSN: 1999-4893. DOI: 10.3390/a14020051. URL: https://www.mdpi.com/1999-4893/14/2/51.

[29] Rohit Kundu et al. "Pneumonia detection in chest X-ray images using an ensemble of deep learning models". In: *PLOS ONE* 16 (Sept. 2021), e0256630. DOI: 10.1371/journal.pone.0256630.

[30] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[31] Angel E. Tahi F. Legendre A. "Bi-objective integer programming for RNA secondary structure prediction with pseudoknots". In: *BMC Bioinformatics* 19 (2018), p. 13. DOI: 10.1186/s12859-018-2007-7.

[32] Rune B. Lyngsø and Christian N. S. Pedersen. "Pseudoknots in RNA Secondary Structures". In: RECOMB '00. Tokyo, Japan: Association for Computing Machinery, 2000, pp. 201–209. ISBN: 1581131860. DOI: 10.1145/332306.332551. URL: https://doi.org/10.1145/332306.332551.

[33] Sato K. Akiyama M. and Sakakibara Y. "RNA secondary structure prediction using deep learning with thermodynamic integration". In: *Nature Communications* 12 (2021), p. 941. DOI: 10.1038/s41467-021-21194-4.

[34] Kangkun Mao, Jun Wang, and Yi Xiao. "Length-Dependent Deep Learning Model for RNA Secondary Structure Prediction". In: *Molecules* 27.3 (2022). ISSN: 1420-3049. DOI: 10.3390/molecules27031030. URL: https://www.mdpi.com/1420-3049/27/3/1030.

[35] *MathWorks*. https://it.mathworks.com/products/matlab.html. Accessed: September 2023.

[36] Pitts W. McCulloch W.S. "A logical calculus of the ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[37] Agnieszka Mikoajczyk and Micha Grochowski. "Data augmentation for improving deep learning in image classification problem". In: *2018 International Interdisciplinary PhD Workshop (IIPhDW)*. 2018, pp. 117–122. DOI: 10.1109/IIPHDW.2018.8388338.

[38] Loris Nanni, Daniela Cuza, and Sheryl Brahnam. "Building Ensemble of Resnet for Dolphin Whistle Detection". In: *Applied Sciences* 13 (July 2023), p. 8029. DOI: 10.3390/app13148029.

[39] Loris Nanni, Stefano Ghidoni, and Sheryl Brahnam. "Deep Features for Training Support Vector Machines". In: *Journal of Imaging* 7.9 (2021). ISSN: 2313-433X. DOI: 10.3390/jimaging7090177. URL: https://www.mdpi.com/2313-433X/7/9/177.

[40] Jacobson AB Nussinov R. "Fast algorithm for predicting the secondary structure of single-stranded RNA". In: *Proceedings of the National Academy of Sciences of the United States of America* 77.11 (1980), pp. 6309–6313. DOI: 10.1073/pnas.77.11.6309.

[41] Danaee P et al. "bpRNA: large-scale automated annotation and analysis of RNA secondary structure." In: *Nucleic acids research* 46.11 (2018), pp. 5381–5394. DOI: 10.1093/nar/gky285.

[42] R. Polikar. "Ensemble based systems in decision making". In: *IEEE Circuits and Systems Magazine* 6.3 (2006), pp. 21–45. DOI: 10.1109/MCAS.2006.1688199.

[43] Riyad Bin Rafiq and Mark V. Albert. "Transfer Learning: Leveraging Trained Models on Novel Tasks". In: *Bridging Human Intelligence and Artificial Intelligence*. Ed. by Mark V. Albert et al. Cham: Springer International Publishing, 2022, pp. 65–74. ISBN: 978-3-030-84729-6. DOI: 10.1007/978-3-030-84729-6_4. URL: https://doi.org/10.1007/978-3-030-84729-6_4.

[44] Rajeev Ranjan et al. "Deep Learning for Understanding Faces: Machines May Be Just as Good, or Better, than Humans". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 66–83. DOI: 10.1109/MSP.2017.2764116.

[45] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. DOI: 10.48550/arXiv.1609.04747. arXiv: 1609.04747 [cs.LG].

[46] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474.

[47] Matthew G. Seetin and David H. Mathews. “RNA Structure Prediction: An Overview of Methods”. In: *Bacterial Regulatory RNA: Methods and Protocols*. Ed. by Kenneth C. Keiler. Totowa, NJ: Humana Press, 2012, pp. 99–122. ISBN: 978-1-61779-949-5. DOI: 10.1007/978-1-61779-949-5_8. URL: https://doi.org/10.1007/978-1-61779-949-5_8.

[48] Terrence J. Sejnowski. “The Deep Learning Revolution”. In: *The Deep Learning Revolution*. 2018, pp. 1–10.

[49] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. DOI: 10.48550/arXiv.1409.1556. arXiv: 1409.1556 [cs.CV].

[50] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. DOI: 10.48550/arXiv.1409.4842. arXiv: 1409.4842 [cs.CV].

[51] Sato K. Kato Y. Hamada M. Akutsu T. and Asai K. “IPknot: fast and accurate prediction of RNA secondary structures with pseudoknots using integer programming”. In: *Bioinformatics (Oxford, England)* 27.13 (2011), pp. i85–i93. DOI: 10.1093/bioinformatics/btr215.

[52] Cheng Tan, Zhangyang Gao, and Stan Z. Li. *RFold: RNA Secondary Structure Prediction with Decoupled Optimization*. 2023. DOI: 10.48550/arXiv.2212.14041. arXiv: 2212.14041 [q-bio.BM].

[53] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. DOI: 10.48550/arXiv.1905.11946. arXiv: 1905.11946 [cs.LG].

[54] Mingxing Tan et al. *MnasNet: Platform-Aware Neural Architecture Search for Mobile*. 2019. DOI: 10.48550/arXiv.1807.11626. arXiv: 1807.11626 [cs.CV].

[55] *Towards Data Science*. https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728. Accessed: September 2023.

[56] *Towards Data Science*. https://towardsdatascience.com/mobilenetv2-inverted-residuals-and-linear-bottlenecks-8a4362f4ffd5. Accessed: September 2023.

[57] Zhao K. et al. Wang Q. Ma Y. “A Comprehensive Survey of Loss Functions in Machine Learning”. In: *Ann. Data. Sci.* 9 (2022), pp. 187–212. DOI: 10.1007/s40745-020-00253-5.

[58] Hannah K Wayment-Steele et al. “Theoretical basis for stabilizing messenger RNA through secondary structure design”. In: *Nucleic Acids Research* 49.18 (Sept. 2021), pp. 10604–10617. DOI: 10.1093/nar/gkab764. eprint: https://academic.oup.com/nar/article-pdf/49/18/10604/40537863/gkab764.pdf. URL: https://doi.org/10.1093/nar/gkab764.

[59] *Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Gradient_descent. Accessed: September 2023.

[60] Jing Yuan and Ying Tian. “An Intelligent Fault Diagnosis Method Using GRU Neural Network towards Sequential Data in Dynamic Processes”. In: *Processes* 7 (Mar. 2019), p. 152. DOI: 10.3390/pr7030152.

[61] Fuzhen Zhuang et al. “A Comprehensive Survey on Transfer Learning”. In: *Proceedings of the IEEE* 109.1 (2021), pp. 43–76. DOI: 10.1109/JPROC.2020.3004555.

[62] Michael Zuker and Patrick Stiegler. “Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information”. In: *Nucleic Acids Research* 9.1 (Jan. 1981), pp. 133–148. ISSN: 0305-1048. DOI: 10.1093/nar/9.1.133. eprint: https://academic.oup.com/nar/article-pdf/9/1/133/6201945/9-1-133.pdf. URL: https://doi.org/10.1093/nar/9.1.133.