PNEUMOTHORAX SEGMENTATION OF CHEST X-RAYS USING A STACKED GENERALIZATION FRAMEWORK WITH MULTIPLE CONVOLUTIONAL NEURAL NETWORKS

by

Amol Mavuduru

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Baowei Fei, Chair

Dr. Haim Schweitzer

Dr. Yee Seng Ng

Gordon Arnold

Dedicated to my mother, who continues to be an inspiration to me today.

PNEUMOTHORAX SEGMENTATION OF CHEST X-RAYS USING A STACKED GENERALIZATION FRAMEWORK WITH MULTIPLE CONVOLUTIONAL NEURAL NETWORKS

by

AMOL MAVUDURU, BS

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2020

ACKNOWLEDGMENTS

This research would not have been possible without the support and guidance that I received from many people during my time at UT Dallas. I would like to start by thanking Dr. Baowei Fei for agreeing to be my supervisor and committee chair and for offering honest feedback to guide me through this research process. I joined the Quantitative BioImaging Lab (QBIL) in May 2018 as an undergraduate with little experience in deep learning applications for medical image segmentation. Under his guidance, I was able to gain experience in this domain and publish my first conference paper at the SPIE Medical Imaging Conference.

I would like to thank Dr. Yee Seng Ng, Dr. Haim Schweitzer, and Gordon Arnold for graciously serving on my thesis committee and working with me remotely during the COVID-19 pandemic. I am grateful for their guidance and helpful comments on my thesis during such a difficult time.

My sincere gratitude goes to the members of QBIL for sharing their knowledge and feedback that allowed me to grow as a researcher in the past two years. I am especially grateful to Martin Halicek and Dr. Maysam Shahedi for working with me on my first conference paper and for contributing significantly to my development as a researcher.

Finally, I would like to thank my close friends and family, especially my parents, for believing in me and offering me support and encouragement during my years at UT Dallas.

April 2020

PNEUMOTHORAX SEGMENTATION OF CHEST X-RAYS USING A STACKED GENERALIZATION FRAMEWORK WITH MULTIPLE CONVOLUTIONAL NEURAL NETWORKS

Amol Mavuduru, MSCS
The University of Texas at Dallas, 2020

Supervising Professor: Dr. Baowei Fei, Chair

With advances in deep learning research, convolutional neural networks (CNNs) have achieved state-of-the-art results in common computer vision tasks such as image classification and image segmentation. In the past decade, most of the research in the domain of CNNs has focused on optimizing the mathematical structure or architecture of CNNs to improve performance on these tasks. Nearly a decade after the introduction of the first CNN architecture, state-of-the-art CNNs have surpassed human performance on image classification and segmentation tasks.

Given their capability to achieve super-human performance on image-recognition tasks, in recent research, CNNs have been trained to perform difficult medical imaging tasks such as cancer segmentation.

While researchers studying image segmentation have focused on optimizing specific CNN architectures to perform well on complex tasks such as medical image segmentation, there is a significant amount of unexplored potential in applying ensemble machine learning techniques to combine the predictions of multiple CNN architectures to produce more robust models.

Ensemble machine learning is an area of machine learning that involves combining the predictions of several machine learning models with techniques such as majority voting, averaging, and stacked generalization in order to produce models with lower generalization errors. Stacked generalization is a powerful technique that involves training a higher-level model that aggregates the predictions of lower models and each input instance to generate final predictions.

This research proposes and evaluates a framework for applying stacked generalization to combine the predictions of multiple CNNs and improve performance on medical image segmentation tasks. The proposed method allows researchers to combine different state-of-the-art CNN architectures into a larger neural network that uses the predictions of each individual CNN and the properties of the input image to generate a more accurate set of predictions.

We evaluate the effectiveness of this method by comparing the performance of individual CNNs and the proposed method on a dataset for medical image segmentation from the 2019 SIIM ACR Kaggle Pneumothorax Segmentation Challenge.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Ever since the introduction of AlexNet [1] in 2012, CNNs have become popular tools for computer vision tasks such as image classification and image segmentation and are now regarded as the state-of-the-art algorithms for approaching these computer vision problems. Given their success and ability to perform better than humans on image classification and segmentation tasks, CNNs have been widely applied to medical image segmentation tasks. In complex medical image segmentation tasks such as cancer segmentation, accurate models can help physicians make better decisions and address the issue of human error in diagnosis.

In the past decade, research in this domain has focused on optimizing the mathematical structure or architecture of CNNs to improve their performance on such tasks. However, a relatively unexplored area of research is the application of ensemble machine learning techniques to combine the predictions of multiple existing CNN architectures to produce more accurate models. Rather than designing new CNN architectures to produce better results in image segmentation tasks, we can use techniques such as stacked generalization to create more accurate models using multiple existing CNN architectures. These ensemble machine learning approaches have the potential to improve the generalization capability of CNNs on more difficult medical image segmentation problems where even trained physicians struggle to produce consistent and accurate results. Applying ensemble learning to CNNs is also a more straightforward and flexible approach for improving the performance on CNNs on segmentation tasks than designing novel

architectures. Ensemble learning methods for CNNs can also allow researchers to exploit the benefits of a wide variety of different CNN architectures by combining them into larger ensemble models.

1.2 Contribution of Thesis

This research proposes and evaluates a framework for using stacked generalization [2] to combine the predictions of multiple CNNs for image segmentation tasks. While the proposed framework is designed for medical imaging tasks it can also be applied to general image segmentation tasks. In addition, this framework can be extended to include additional CNNs and is not limited to include only the combinations of CNNs used in this paper.

In order to evaluate the effectiveness of this framework, we compare the performance of this method to that of individual CNNs on a challenging medical imaging dataset. The image segmentation dataset we used is a publicly available dataset from the Kaggle SIIM ACR Pneumothorax Challenge. This dataset focuses on the task of detecting pneumothorax, or a collapsed lung, in chest X-rays.

To demonstrate the flexibility of this framework, we propose two variations of the stacked generalization framework with different aggregator models for combining the predictions of each CNN. We evaluate each variation separately on the pneumothorax segmentation dataset to compare their effectiveness.

CHAPTER 2

BACKGROUND

2.1 Neural Networks

Neural networks, first introduced in 1958 and popularized since the 1980s in artificial intelligence and machine learning research, are biologically inspired mathematical models that can learn patterns from training data and make predictions on instances from unseen data. Neural networks are loosely based on the structure of neurons in the nervous system and have become popular algorithms for approaching a wide variety of machine learning problems such as text classification, text translation, and image classification and segmentation. While there are many different types of neural networks, this section focuses on the theory and concepts of neural networks that are necessary to understand the material presented in this work. There are three main concepts that can be defined for all neural networks and allow them to learn patterns in data – the architecture or mathematical structure of the network, the loss function used to evaluate the network's predictions, and the optimization algorithm used to optimize the parameters of the network to minimize the loss function.

2.1.1 The Perceptron

The perceptron, proposed in 1958 by Frank Rosenblatt [3], was designed to serve as a mathematical model of the functionality of neurons. A perceptron consists of a set of input "neurons" that store input values and have numerical weights attached to them. In order to compute the output of a perceptron for a given set of inputs, the weighted sum of the inputs is passed to an activation function that returns a binary value depending on whether or not the

weighted sum is greater than a predefined threshold [3]. Mathematically, the perceptron is essentially a simple linear model for solving binary classification problems, which are problems that involve categorizing input examples into one of two possible classes.

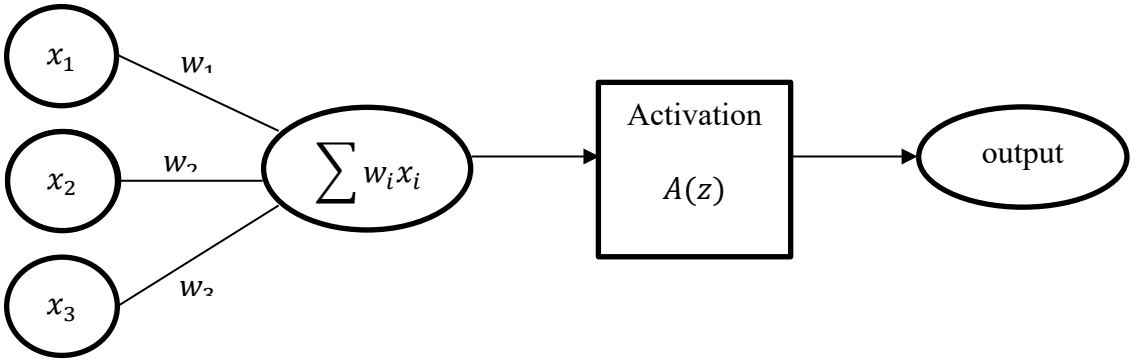


Figure 1: A perceptron with three inputs.

Referring to Figure 1, the threshold activation function $A(z)$ is defined as follows for a given threshold θ:

$$A(z) = \begin{cases} 0, & z < \theta \\ 1, & z \geq \theta \end{cases}$$

The output of a perceptron can only be 1 or 0 and each value corresponds to a different class in a binary classification problem. The weights, $w = \{w_1, w_2, w_3\}$, are parameters that the perceptron must learn for each input example. The i-th example in the training dataset for a perceptron contains inputs $x_1^{(i)}, \ldots, x_n^{(i)}$ and the target output $y^{(i)}$. For a single training example, we can compute the perceptron output $\hat{y}^{(i)}$ and use the perceptron learning rule defined below to update each weight:

$$\Delta w_j = \ell(y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

In the previous equation, Δw_j represents the weight update which will be added to weight w_j to produce the updated weight. The parameter ℓ is the learning rate, and generally ranges from 0 to 1, with higher values producing more volatile weight updates. For a linearly separable dataset, where training examples belonging to class 1 and the training examples belonging to class 0 can be separated by a straight line, or in higher-dimensional space, a hyperplane, the perceptron will always converge to a specific weight configuration using this learning rule. However, the perceptron cannot learn patterns in data that is not linearly separable, and for this reason, the first true deep neural networks contained more layers in order to model more complex data.

2.1.2 Deep Neural Networks

Deep neural networks, which form the basis of the subfield of machine learning known as deep learning, expand on the concepts from the perceptron to produce models that can learn complex, nonlinear relationships in data.

Structure of Deep Neural Networks

A deep neural network consists of a collection of neurons organized in multiple layers connecting the inputs to the final outputs. Each neuron is essentially a perceptron with a set of incoming inputs and weights and an activation function that is applied to produce an output. Between consecutive layers, each neuron in the previous layer is connected to each neuron in the following layer as demonstrated in Figure 2. For this reason, these layers are often referred to as *fully connected layers*. Each connection is also associated with a weight, which is parameter that the network must learn. A deep neural network contains one or more such hidden layers between the input layer and the output layer.

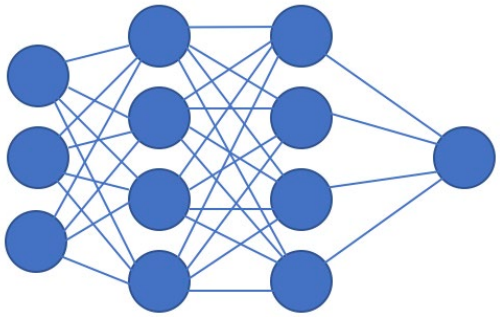
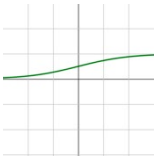
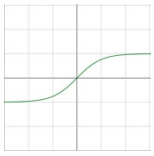
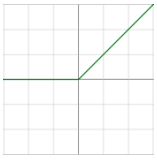


Figure 2: Architecture of a deep neural network with two hidden layers.

Typically, we define a universal activation function for all neurons in the network. Table 1 defines some of the most common activation functions [4] for neural networks.

Table 1: Common activation functions

Activation Function	Definition	Graph
Sigmoid	$A(z) = \frac{1}{1+e^{-z}}$	
Hyperbolic Tangent	$A(z) = \tanh(z) = \frac{2}{1+e^{-2z}} - 1$	
Rectified Linear Unit (ReLU)	$A(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$	

For conventional deep neural networks, the sigmoid activation is commonly used, whereas the ReLU activation appears more often in convolutional neural networks, which will be discussed layer in this section. The value at each neuron in a neural network is the result of applying the activation function to the weighted sum of the values from the incoming inputs. Figure 3 provides an example with real numbers to demonstrate how the values in a fully connected layer are computed.

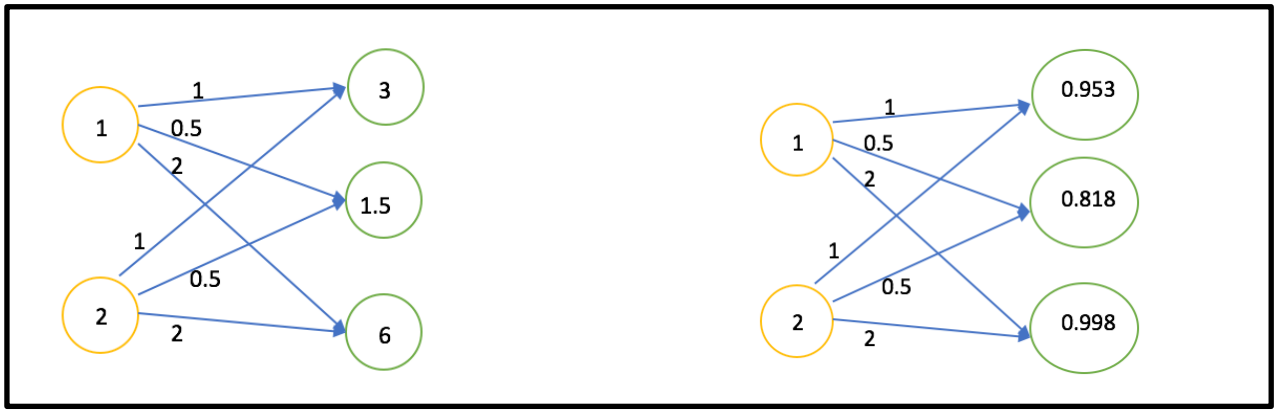


Figure 3: Values in hidden layer before (left) and after (right) applying sigmoid activation.

Upon closer inspection, we can see that this operation is nothing more than a matrix multiplication between a matrix of input values from the previous layer and the weight matrix containing the weights connecting the neurons between the previous and the next layer. The values of successive layers in neural networks are computed in this fashion until the output is finally computed. For classification problems, the output is usually a vector corresponding to the probability of each class, whereas for regression problems, the output is usually the continuous value that the network is trying to predict.

Training Deep Neural Networks

Once the output is computed for a given training instance, the next step is to compare the predicted output with the actual output and adjust the weights of network accordingly using procedures known as backpropagation and gradient descent. In order to quantify how bad a neural network's predictions are, we need to define a loss function. A loss function takes two inputs – the network's predictions and the correct outputs and compares them to return a scalar value indicating the magnitude of the error between the two values. For classification problems, the cross-entropy loss function is most commonly used to train neural networks, whereas for regression problems, most loss functions are variations of the mean squared error (MSE) between the predictions and expected output. The training process for a neural network involves computing the loss for multiple training instances and adjusting the weights of the network in each iteration to minimize the loss values. The algorithm that adjusts the weights of the network to minimize the loss function is known as an optimization algorithm.

Gradient Descent and Backpropagation

Gradient descent [5] was the first optimization algorithm used to train neural networks and is still widely used to train neural networks in practice today. Gradient descent computes the derivatives of the loss function with respect to each weight and updates the weights accordingly. The gradient descent weight update rule is defined as follows:

$$\Delta w_i = -\ell \frac{\partial L}{\partial w_i}$$

In the gradient descent weight update rule, L represents the loss function, w_i represents the weight that is being updated, and ℓ is the learning rate. The weights are updated in the direction

opposite of the derivative of the loss with respect to the weights in order to minimize the loss. This procedure involves computing a partial derivative, which can become difficult and computationally expensive if we take a standard approach that involves defining the output of the network as a function of several weights and compute each derivative separately.

Traditionally, an algorithm known as backpropagation [6] is used in gradient descent to simplify the process of computing weight updates by making use of the chain rule for derivatives. Assume that we have two parameters, w_1 and w_2. If the two parameters are involved in the computation of the output, we can define the derivative of one with respect to the loss as follows:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial w_2}\frac{\partial w_2}{\partial w_1}$$

This equation is simply the chain rule, stated in terms of the weights in a neural network and the loss function. Rather than computing the derivative of each weight directly with respect to the complex function defining the relationship between the weights and the loss, we can use the chain rule to compute the derivatives of adjacent weights in relation to each other. The backpropagation algorithm involves moving from the end of the network to the input layer, computing the relationships between each weight derivative in this fashion and then *back-propagating* from the input layer to the output layer, adjusting the weights accordingly by substituting the values of the gradients that have already been computed from previous layers.

2.1.3 Convolutional Neural Networks

Convolutional neural networks [7] use the same basic concepts from deep neural networks, but involve special layers known as convolutional layers. These layers are designed to extract information from spatial input data such as text or images. Convolutional neural networks have gained immense popularity in image classification and segmentation problems due to their ability to extract complex features related to the spatial properties of image data. Unlike simple deep neural networks that receive vectors as input, convolutional neural networks will often receive image data in the form of matrices or tensors. A standard image can be represented using a stack of three numerical matrices, each corresponding to a separate RGB channel. For example, a 256 x 256 pixel RGB image contains three 256 x 256 pixel channels. The total number of pixel values to be considered in this image is 196,608 and it is impractical to flatten the image tensor and treat each of these values as an individual input for a standard deep neural network. For this reason, convolutional layers were designed to take advantage of the spatial properties of image data.

2.1.4 Convolutional Layers

Convolutional layers operate on spatial data, where the relative position of input values is important [7]. A standard 2-D convolutional layer contains a series of convolutional filters (also known as kernels) that each operate on a 3-D image volume. Convolutional filters can be visualized as a square matrix with numerical weights that passes over an input image, performing a pixel-wise dot product with the set of pixel values that it is placed over. A convolutional filter has two parameters – the size of the filter, and the stride length. The size represents the size of

the square matrix corresponding to each filter and the stride length represents the number of pixels that the filter is shifted by in each iteration. Figure 4 provides a visualization of a convolution operation with a 3x3 filter with a stride of 1. Each dot product from the convolutional filter and the original image pixels forms the values of a new feature map containing the result of the convolution operation. A standard convolutional layer will contain multiple filters with different values, and each filter produces a separate convolutional feature map, transforming the image into a stack or volume of feature maps. In a convolutional neural network, several convolutional layers may be stacked together, acting on successive feature maps to extract deeper and more complex features from the input image.

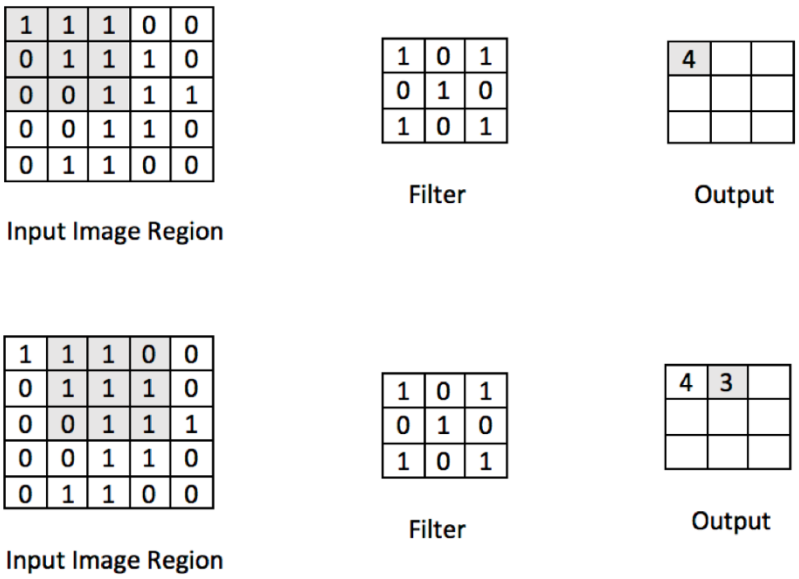


Figure 4: Visualizing a convolution operation with a 3x3 filter with a stride of 1.

Sometimes, convolutional layers can be designed to add a padding of zeros around the original image. Zero padding is a common procedure that can be used to control the shape of the output volume. It is often important to know the shape of the output volume that results from a convolutional layer. Given an input volume of size $W \times H \times D$, a convolutional layer with F filters of size $N \times N$ with a stride of S and zero padding of P on all sides will produce an output volume of size $W_{out} \times H_{out} \times D_{out}$ as defined in the equations below [7].

$$W_{out} = \frac{W - N + 2P}{S} + 1$$

$$H_{out} = \frac{H - N + 2P}{S} + 1$$

$$D_{out} = F$$

Convolutional layers are usually followed by ReLU activation functions computed on each value of the output volume. As a result, some of the values in the output may be set to zero.

2.1.5 Pooling Layers

The amount of information contained in a single image or a convolutional feature map is quite large and it can be difficult for a neural network to distinguish the important information from noise or irrelevant information. This dilemma inspired the creation of pooling layers. The main purpose of a pooling layer is the reduce the size of a convolutional volume by only selecting certain values from local regions to create a smaller volume. Pooling layers [8] are similar to convolutional filters because they involve a filter of a specific size that passes over the input

volume and selects specific values. There are several types of pooling, but the most common form of pooling is max pooling.

Max Pooling

A max pooling layer [8] selects the maximum value from each region of a certain size (such as 2 x 2 pixels) from the input volume and inserts these values into a new volume. Figure 5 demonstrates the concept of max pooling using a pooling layer with a 2 x 2 pooling layer.

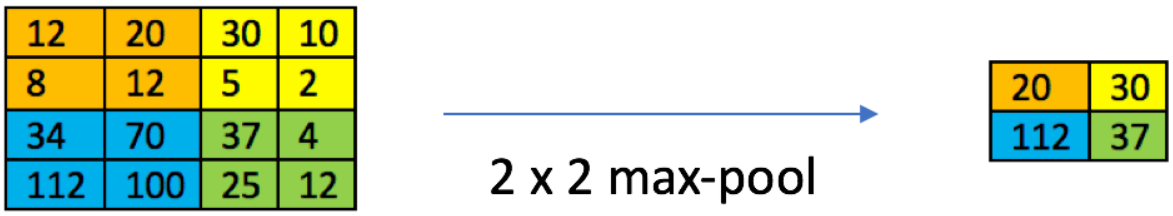


Figure 5: Demonstration of max pooling.

Max pooling reduces the size of the input volume and only keeps the largest values, which can be used to separate signal from noise when learning patterns in image data.

Other Types of Pooling

The same concept of pooling can be applied differently in different types of pooling such as average pooling or min pooling. Average pooling selects the average values from each pooling region while min pooling selects the minimum values. While these types of pooling are not as common as max pooling, they can still be used effectively on certain types of image data.

2.1.6 Batch Normalization

Batch normalization [9] is a technique used to scale the input data passed between layers. For each batch passed through a neural network, batch normalization fixes the means and variances of each layer's inputs. Given an input vector x from the k-th neuron, batch normalization produces a normalized vector $\hat{x}$ defined as follows.

$$\hat{x} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

The resulting vector has a zero mean and unit variance. The main benefit of batch normalization is that it increases the stability of a neural network by limiting the range of values in each layer, thus also limiting the *internal covariate shift* or change in the distribution of the network activations due to weight updates [9]. As a result, batch normalization allows for the use of higher learning rates and can allow the optimization algorithm to converge faster.

2.1.7 Fully Connected Layers

The standard layers in deep neural networks are referred to as fully connected layers and convolutional neural networks designed for image classification tasks will usually have a set of consecutive fully connected layers at the end that use the output of the convolutional and pooling layers to classify the image. While convolutional and pooling layers extract complex features from image data, the fully connected layers actually use these features to perform image classification tasks. The output of the sequence of fully connected layers at the end of a convolutional neural network is a vector with the same number of components as the number of classes in the image classification problem. Each value in the vector is a score related to the

likelihood of the image belonging to the corresponding class. However, to transform these scores into probabilities, we need to use a special activation function that operates on vectors rather than scalar values.

2.1.8 Softmax Activation Function

The softmax activation function [4] operates on a vector of values and can be viewed as a generalization of the sigmoid function to vector inputs. Mathematically, the softmax function transforms a vector of real values into a vector of probabilities. Given an input vector in the form $z = (z_1, \dots, z_n)$, the softmax function is defined as follows.

$$Softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

The function essentially computes the sum of the exponentials of each component and divides each individual exponential value by this sum. The denominator in this equation ensures that all of the values in the resulting output vector sum to 1, which allows the results to be interpreted as probabilities. In classification problems, the softmax activation function is applied to output of the last fully connected layer to produce the predicted class probabilities.

2.1.9 Regularization Techniques

A common problem that occurs not just when training deep learning models, but when training machine learning models in general, is the problem of overfitting. Overfitting occurs when a model essentially memorizes training examples and seems to perform well on the training data, but is unable to generalize what it has learned to unseen testing or validation data. In practice, for

any machine learning experiment or research investigation, the original dataset will be split into three sets – a training set, a validation set, and a testing set that is not touched until the very end.

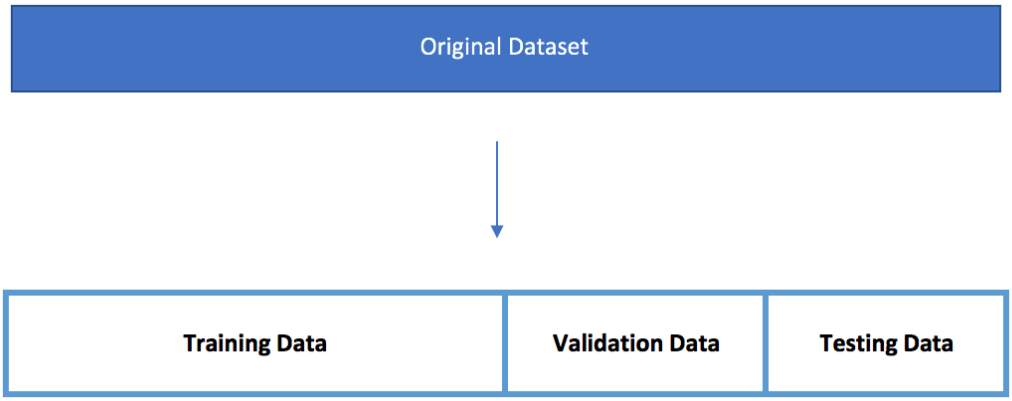


Figure 6: The standard split of a dataset into three separate groups in machine learning studies.

In machine learning experiments, the training set is used to train models, whereas the validation set is used to evaluate the model's performance on unseen data. Usually, the model performs best on the training data, but if the gap between the performance on the training and validation data is unusually high, then the model likely suffers from overfitting because it is unable to generalize what it has learned from the training data to the validation data. Generally, a model that is too complex will suffer from overfitting and a model that is too simple will suffer from the reverse problem, known as underfitting.

Regularization refers to the use of techniques that are used to prevent models from overfitting. In deep learning research, three popular regularization techniques are dropout, weight decay, and data augmentation.

Dropout

Dropout [10] is a procedure that involves randomly setting the activations of certain neurons in a layer to zero in order to the prevent the network from relying too much on specific features when making predictions. Dropout is equivalent to "dropping out" or randomly removing certain neurons. A dropout layer will have a parameter indicating the percentage of neuron activations that should be set to zero. For example, a dropout value of 0.4 indicates that 40 percent of the activations from the preceding layer will randomly be set to zero.

Weight Decay

Weight decay [11] is a technique that involves modifying the loss function to control the magnitudes of the weights and discourage the network from assigning extremely high values to specific weights. In practice, this technique can combat overfitting by producing simpler models. There are two types of weight decay, and they are often referred to as L2 and L1 regularization. L2 regularization adds a term with the sum of the squares of the weights multiplied by a constant to the original loss function as demonstrated in the following equation [11].

$$L_{new}(w) = L_{original}(w) + \lambda \sum_{i=1}^{n} w_i^2$$

In the previous equation, λ is the weight decay hyper-parameter that controls the regularization strength. Larger values of λ will produce simpler models, whereas smaller values will produce more complex models that may be prone to underfitting. This value is ultimately a hyper-parameter that must be tuned through trial and error for different deep learning tasks.

L1 regularization is similar to L2 regularization, except that the sum of the squares of the weights is replaced with the sum of the absolute values of the weights when defining the new loss function, as presented in the equation below [11].

$$L_{new}(w) = L_{original}(w) + \lambda \sum_{i=1}^{n} |w_i|$$

In practice, L1 regularization is a stricter form of regularization and often produces more sparse weights, with values closer to zero for less important features [10]. Sometimes, weight decay may be used with both L1 and L2 regularization with two separate regularization parameters as demonstrated below.

$$L_{new}(w) = L_{original}(w) + \lambda_1 \sum_{i=1}^{n} w_i^2 + \lambda_2 \sum_{i=1}^{n} |w_i|$$

Data Augmentation

Data augmentation [12] is a technique that involves altering the existing training examples to create more diverse training examples for a neural network to learn from. Data augmentation is a common technique used when training CNNs on image data because images can be altered in many different ways to produce a wide variety of training examples for a network to learn from. A wide variety of techniques can be applied to produce additional images from an existing set of training images. These techniques include, but are not limited to:

- Horizontal and vertical flips of the original images.
- Random rescaling of the original images.
- Random cropping of the original images.

- Perturbing the brightness and contrast of the images.
- Modifying the hue, saturation, and value of the images.

When all of these augmentations are randomized and combined, data augmentation can occur on a batch-by-batch basis to produce a virtually infinite amount of unique training images that are slightly different from the original images. Data augmentation allows the network to learn invariance to properties of images that may not be related to the expected output that it must produce. For example, an image of an object such as car, is still an image of the car even if the brightness is altered or if it is flipped horizontally. Through data augmentation a neural network could learn to recognize the features that truly define an image of a car, rather than finding incorrect patterns and overfitting to the training data.

2.1.10 CNN Architectures

Over the last decade, research in deep learning for computer vision has focused on developing the best configurations of convolutional layers, pooling layers, fully connected layers, and similar components for designing networks that can perform well on tasks such as image classification. These configurations are referred to as CNN architectures. These CNN architectures are usually evaluated based on their performance on standard benchmark datasets for image classification such as CIFAR-10 [13], CIFAR-100 [13], and more popularly, ImageNet [14]. The CIFAR-10 dataset consists of 6000 32 x 32 pixel images in 10 classes, with 6000 images for each unique class [13]. The CIFAR-100 dataset is the same size as the CIFAR-10 dataset, except that it has 100 unique classes with 600 images per class [13]. While both the CIFAR-10 and CIFAR-100 datasets have been used for evaluating CNNs, the ImageNet dataset

is a more universal benchmark. The ImageNet dataset is much larger, consisting of millions of human-annotated images, and has been used for several years for the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [14]. The ILSVRC challenges research institutions and scientists to train algorithms using a subset of ImageNet to classify images belonging to any one of 1000 image categories [14]. Improvements in CNN architectures have often come from teams who competed in these challenges.

AlexNet

AlexNet [1] was one of the first CNN architectures, and the first CNN architecture to win the ILSVRC challenge by a huge margin. The AlexNet architecture was designed for 224 by 224 pixel images in the 2012 ILSVRC. AlexNet is relatively simple, consisting of only five convolutional layers, two batch normalization layers, three max pooling layers, and three fully connected layers as described in Table 2.

Table 2: Summary of AlexNet layers from [1].

Layer Name	Description	Output Size
Input	Input Image	224 x 224 x 3
CONV1	96 11 x 11 filters with a stride of 4	55 x 55 x 96
MAXPOOL1	3 x 3 max pooling with a stride of 2	27 x 27 x 96
NORM1	Batch Normalization Layer	27 x 27 x 96
CONV2	256 5 x 5 filters with zero padding of 2	27 x 27 x 256
MAXPOOL2	3 x 3 max pooling with a stride of 2	13 x 13 x 256
NORM2	Batch Normalization Layer	13 x 13 x 384
CONV3	384 3 x 3 filters with zero padding of 1	13 x 13 x 384
CONV4	384 3 x 3 filters with zero padding of 1	13 x 13 x 384
CONV5	256 3 x 3 filters with zero padding of 1	13 x 13 x 256
MAXPOOL3	3 x 3 max pooling with a stride of 2	6 x 6 256
FC6	Fully connected layer with 4096 neurons	4096
FC7	Fully connected layer with 4096 neurons	4096
FC8	Fully connected layer with 1000 neurons followed by softmax activation	1000

In the original paper, the AlexNet architecture was actually distributed across two GPUs due to memory limitations. The output of the MAXPOOL3 layer in Table 2 is flattened before being transferred to FC6, the first fully connected layer. The final output of the architecture is a vector of probabilities corresponding to each of the 1000 classes in the ImageNet Challenge. Using the

training configuration described in the paper with dropout and data augmentation, AlexNet achieved a top-5 test error rate of 15.3% [1]. The top-5 error rate refers to the percent of images in which the correct class was not within the top five most probable classes predicted by the network. One of the key assertions in this paper was that deeper convolutional neural networks can potentially achieve better performance.

VGG

The VGG architecture [15], developed by Simonyan and Zisserman from the Visual Geometry Group in the Department of Engineering Science from the University of Oxford in 2015, took the concept of deep convolutional networks to a new level at the time. Simonyan and Zisserman's work [15] proposed several configurations of VGG, with the largest ones, referred to as VGG16 and VGG19, having 16 and 19 layers with network weights respectively. Each convolutional layer contains multiple 3 x 3 filters, and the final fully-connected layers follow the same configuration as AlexNet.

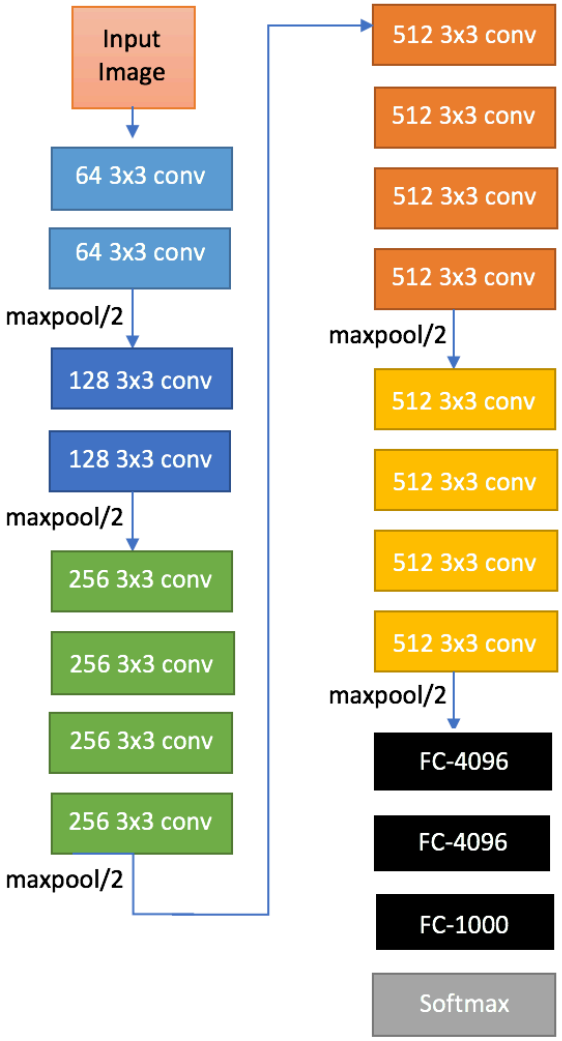


Figure 7: VGG19 CNN architecture described in [15].

The authors of the VGG paper asserted that very deep convolutional networks can achieve better results than their shallower counterparts. The team behind VGG achieved a top-5 classification error of 7.32 percent on the 2014 ILSVRC [15], which is a huge improvement over the AlexNet architecture.

Inception/GoogLeNet

The Inception architecture [16], also known as GoogLeNet, introduced the concept of designing modules or blocks of convolutional layers that could be repeated and combined several times to produce CNN architectures. The authors of the paper that introduced this architecture described the following Inception module in Figure 8.

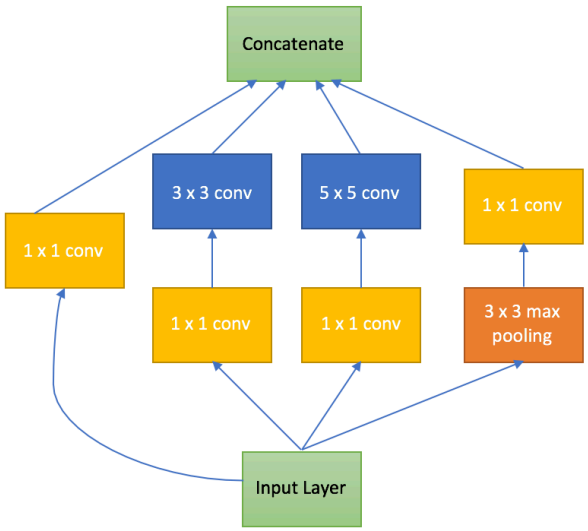


Figure 8: Visualization of the Inception module described in [16].

These Inception modules are stacked together in several blocks to create the Inception architecture. Table 3 describes the layers and modules in the architecture. An interesting feature of the Inception architecture is that it has intermediate "auxiliary classifiers" [16] in the middle of the network that are used in the process of optimizing the loss function. This feature facilitates the propagation of gradients and addresses the *vanishing gradient* problem that frequently occurs in deep networks. The network is 22 layers deep and was originally designed for 224 x 224 RGB input images. The GoogLeNet team achieved a top-5 error of 6.67 percent on the 2014 ILSVRC [16], beating the VGG team by a small margin.

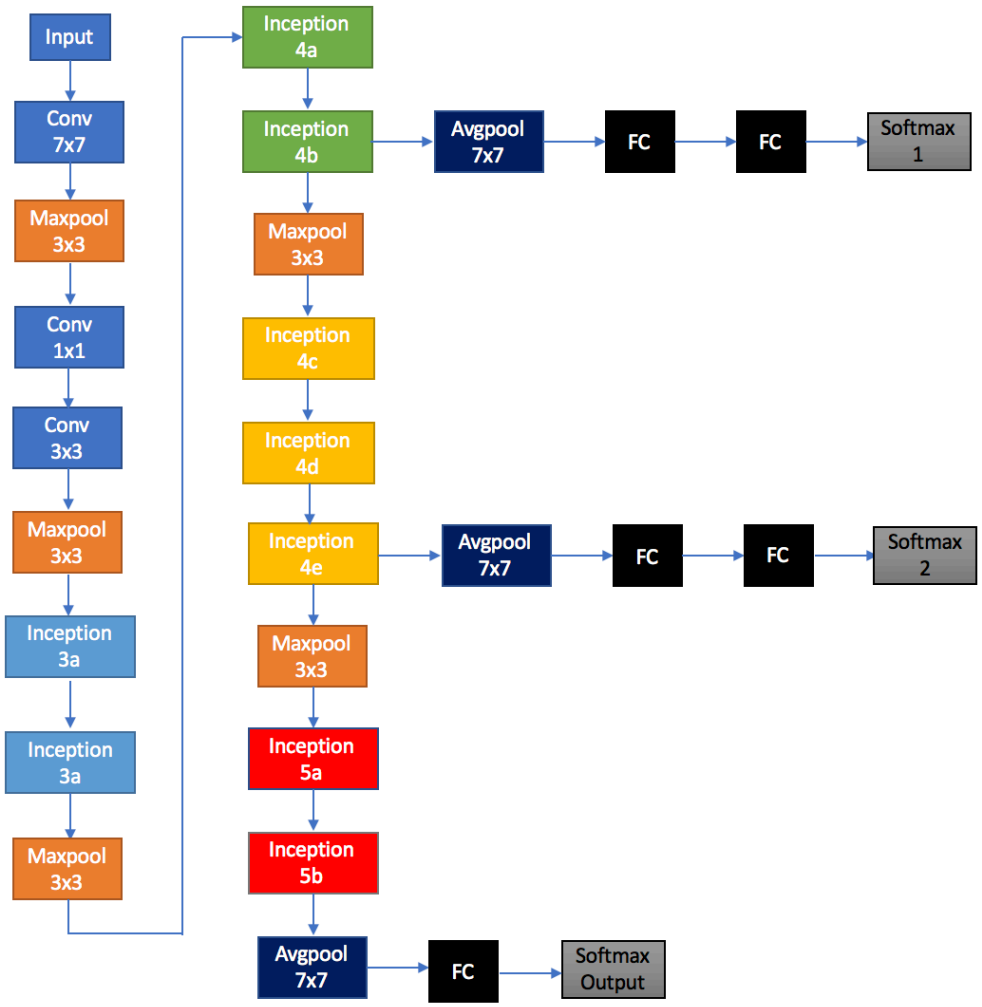


Figure 9: Diagram of Inception architecture from [16].

Table 3: Summary of Inception layers from [16].

Layer/Module	Filter Size	Stride	Output Size
Input Image			224 x 224 x 3
Convolutional	7 x 7	2	112 x 112 x 64
Max pooling	3 x 3	2	56 x 56 x 64
Convolutional	3 x 3	1	56 x 56 x 192
Max pooling	3 x 3	2	28 x 28 x 192
Inception (3a)	3 x 3	2	28 x 28 x 256
Inception (3b)			28 x 28 x 480
Max pooling	3 x 3	2	14 x 14 x 480
Inception (4a)			14 x 14 x 512
Inception (4b)			14 x 14 x 512
Inception (4c)			14 x 14 x 512
Inception (4d)			14 x 14 x 512
Inception (4e)			14 x 14 x 528
Max pooling	3 x 3	2	14 x 14 x 832
Inception (5a)			7 x 7 x 832
Inception (5b)			7 x 7 x 1024
Average pooling	7 x 7	1	1 x 1 x 1024
40 % dropout			1 x 1 x 1024
Linear layer/Softmax			1 x 1 x 1000

ResNet

The ResNet architecture [17], proposed in 2015 by researchers at Microsoft, addressed the problem of degrading performance in deeper networks and introduced a method for achieving better results with substantially deeper networks. The ResNet architecture makes use of residual blocks where shortcut connections are used to add the result of an earlier layer to the next layer.

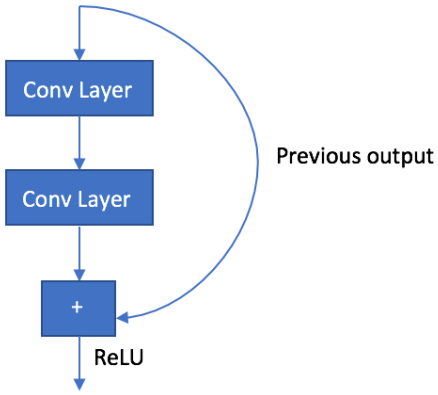


Figure 8: Residual block from original ResNet paper [17].

The ResNet architectures proposed in the original work are VGG-style networks with stacks of residual blocks from Figure 8 that are stacked together and contain convolutional layers with an increasing number of filters as the depth of the network increases. The authors of the ResNet paper proposed several versions of the ResNet architecture, such as ResNet34 containing 30 layers and ResNet50 containing 50 layers. The largest version of ResNet was ResNet152, which contained 152 layers [17]. The authors combined the predictions of three ResNets to achieve a 3.57% error on the ImageNet dataset, winning first place in 2015 ILSVRC [17].

Other CNN Architectures

Since 2015, researchers have designed even more CNN architectures that outperform the previously described networks. We made use of some of these CNN architectures such as SENet and EfficientNet when testing the ensemble framework outlined in this work.

For example, in 2016, researchers introduced a modified version of the ResNet architecture known as ResNeXt [18]. ResNext is a multi-branch version of ResNext that aggregates the outputs of multiple branches or paths in each residual block rather than the outputs of a single path [18]. The ResNeXt team placed second in the 2016 ILSVRC [18], improving upon the results of ResNet.

The DenseNet [19] architecture, which takes the design principle behind ResNet to another extreme, was introduced in 2017. While ResNet included shortcut connections over pairs of consecutive layers, DenseNet includes *dense blocks* where each layer in the block has incoming shortcut connections from all preceding layers within the block. Each layer basically receives the feature maps of the previous layers, which makes DenseNet what the authors of the original paper referred to as a *densely connected convolutional network*.

Another interesting architecture introduced in 2017 was the squeeze-excitation network [20], also referred to as SENet. The motivation behind SENet was to develop an effective way to model the relationships between the channels of convolutional feature maps. SENet contains a series of squeeze-and-excitation blocks with a squeeze operation and an excitation operation.

The squeeze operation takes a volume as input and aggregates the feature maps produce an embedding vector with a component for each channel in the volume. For an $H \times W \times C$ volume, the squeeze operation will produce a vector with C components. The excitation operation takes the output of the squeeze operation as input and produces a set of weights for each channel that are then applied to the original volume to produce an output volume. A squeeze-excitation network can then be constructed by chaining together a series of squeeze-and-excitation blocks. The authors of the SENet paper presented variants of popular architectures such as ResNet that included squeeze-and-excitation blocks. SE-ResNet50, for example, is a ResNet50 CNN with squeeze-and-excitation blocks.

Finally, a major innovation that not only provided improvements in accuracy but also in efficiency in terms of computational speed and memory usage, was EfficientNet [21] introduced in 2019. The main problem investigated in the EfficientNet paper was the problem of developing an effective method for scaling CNNs. The method explored in this work addresses the problem of taking a baseline CNN and scaling the network depth, width, and/or resolution to meet certain resource requirements without changing the function that computes an output from the input image as defined in the original baseline CNN. The authors proposed the following set of constraints [21] for scaling the depth, width, and resolution of the network uniformly based on a user-defined coefficient ϕ:

$$\text{depth: } d = \alpha^{\phi}$$

$$\text{width: } w = \beta^{\phi}$$

$$\text{resolution: } r = \gamma^{\phi}$$

$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$

$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1 \text{ [19]}$$

In the listed constraints, α, β, and γ are constants that are to be optimized using a simple grid search [21]. The ϕ parameter is used to scale the model according to the number of available resources [21].

The baseline CNN in the EfficientNet paper, known as EfficientNetB0, was inspired by the MnasNet [22] architecture search algorithm for MobileNet [23], an efficient class of CNNs designed to be used for mobile vision applications. The EfficientNetB0 baseline network was then scaled using larger values of ϕ to produce EfficientNetB1 through EfficientNetB7 [21]. We specifically used EfficientNetB0, EfficientNetB2 and EfficientNetB5 in this work.

2.1.11 Advanced Optimization Algorithms

While stochastic gradient descent (SGD) is a fundamental and simple optimization algorithm, SGD by itself is not always sophisticated enough to train complex networks. The reason is that SGD computes weight updates based on the first-order derivatives of each parameter with respect to the loss function, and it is possible for the optimization process to get stuck in a local minimum instead of converging to the optimal minimum. For this reason, several more sophisticated optimization algorithms were introduced to produce better convergence when training neural networks.

SGD with Momentum

It is possible to improve SGD by introducing the concept of momentum [24] from accumulated gradients, allowing the optimizer to move through local minima instead. For a given loss function, $L(w)$, we can define a momentum term m that gets updated with each weight update so that the new weight w_{t+1} is defined in terms of the old weight w_t as follows:

$$m_{t+1} = \rho m_t + \frac{\partial L}{\partial w_t}$$

$$w_{t+1} = w_t - \ell m_{t+1}$$

The accumulation of momentum from past updates is similar to concept of momentum in physics and allows the simple SGD algorithm to climb out of local minima if enough momentum has been accumulated.

Adaptive Optimization Algorithms

Simple algorithms such as SGD maintain a constant learning rate that never changes. However, in practice, a constant learning rate may not be the best strategy for optimizing the parameters of a CNN. Adaptive optimization algorithms adapt the learning rate to change over time based on factors such as the number of gradient updates or the accumulated values of the past gradient updates. Some popular adaptive algorithms include Adagrad [25], Adadelta [26], and Adam [27]. These algorithms have the advantage of removing the need to perform a significant amount of hyper-parameter tuning. The Adam algorithm in particular, is an optimization algorithm that is generally effective for a wide variety of problems.

2.1.12 Image Segmentation

The CNN architectures described in section 2.1.10 were originally designed for image classification problems, which involve assigning labels to entire images. Image segmentation is a more complex task that involves assigning a label to each pixel in an image, essentially dividing the image into segments or regions. An example of an image segmentation task is detecting an object such as a car in an image and separating it from the background. In this case, the segmentation task involves labeling the pixels belonging to the car to the pixels belonging to the background in the image. In a sense, image segmentation is a pixel-level classification problem. Image segmentation problems frequently occur in medical imaging. For example, one common problem in cancer research is performing cancer segmentation on images of tissue samples in order to identify regions of normal and cancer tissue.

2.1.13 Transpose Convolutions

Architectures for image segmentation differ from architectures for image classification in that they lack fully connected layers at the end of the network. CNNs for image segmentation are often referred to as fully convolutional networks because their final outputs are produced by convolutional, rather than fully connected layers. In order to make some of these fully convolutional architectures feasible, a new kind of convolution operation, known as a transpose convolution [28] was introduced. While convolutional layers often down-sample images or feature maps by making them smaller in the first two dimensions, transpose convolutions do the opposite. A transpose convolution can be viewed as the opposite of a standard convolution and

each pixel in the input is mapped to multiple pixels in the output, thus producing a larger output as demonstrated in the example in Figure 10.

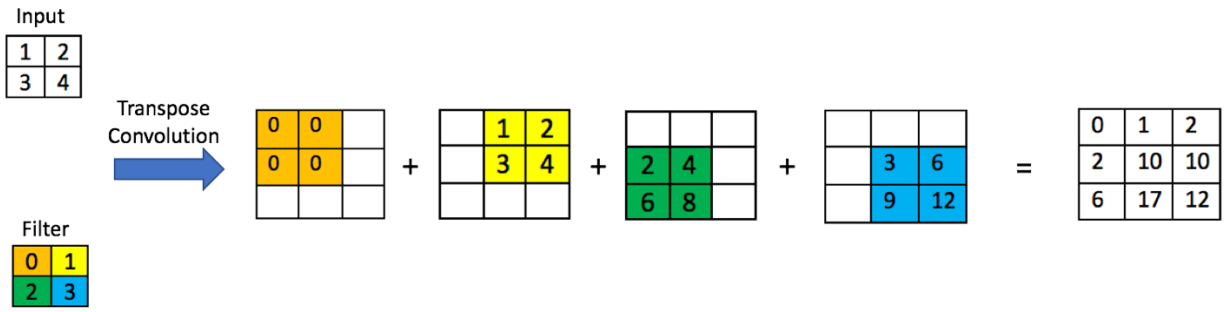


Figure 10: Example of a transpose convolution.

Several fully convolutional architectures for image segmentation use transpose convolutions to up-sample smaller feature maps into larger feature maps, eventually generating a segmentation map that is the same size as the original input image.

2.1.14 Fully-Convolutional Architectures

The state-of-the art segmentation architectures usually follow an encoder-decoder pattern where the encoder consists of standard convolutional and pooling layers are used to down-sample an image into a smaller feature map and the decoder consists of transpose convolutions and up-sampling layers that transform the feature map into a segmentation output that is the same size as the original image.

FCN

FCN [29] was the first proposed fully convolutional architecture and provides a general framework for converting architectures with fully-connected layers at the end to fully-

convolutional networks. A simple method proposed in the FCN paper for converting fully connected architectures to fully convolutional models is to remove the fully-connected layers, replace them with convolutional layers, add a 1 x 1 convolution and use bilinear up-sampling to increase the size of the output to match the dimensions of the input image [28]. While this configuration can sometimes produce good results, the output is often coarse due to the use of heavy up-sampling at the end of the network.

U-Net

U-Net [30] is a fully convolutional CNN architecture that was originally designed for medical imaging tasks and was first proposed in 2015. The first portion of the architecture is an encoder with standard convolutional and max pooling layers that generate features maps with lower resolutions than the original image. The second part of the architecture is a decoder that takes the feature maps at smaller resolutions and up-samples them using transpose convolutions and concatenates these feature maps with the corresponding feature maps from the encoder. The final output of the architecture is a segmentation map indicating the pixel-level segmentation class probabilities. The authors of the original U-Net paper asserted that some of the advantages of the architecture include its reasonable training time and ability to achieve good results with small datasets [30]. The architecture's success even on such little training data is partly dependent on the use of heavy data augmentation [30]. We used the U-Net architecture successfully in a previous work focused on cancer segmentation using head and neck histological slides [31]. Our version of the U-Net architecture that we adapted for our previous work is presented in Figure 11.

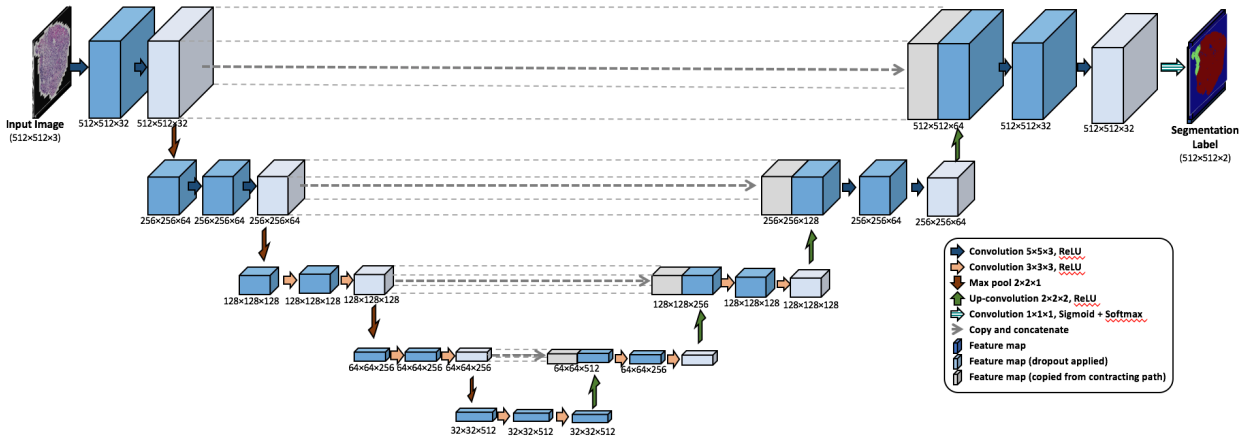


Figure 11: Diagram of the U-Net architecture we adapted in [31].

FPN

Feature pyramid networks (FPN) [32] were first proposed in 2017 and take advantage of the pyramidal structure of deep convolutional neural networks. The FPN architecture follows a similar encoder-decoder pattern to U-Net, except it features bottom-up and top-down pathways along with lateral connections that combine lower resolution and higher resolution feature maps. The bottom-up pathway contains standard convolutional and pooling layers along with the intermediate feature maps at a predefined set of *stages*. We did not use the FPN architecture in this paper, but it is a viable alternative to U-Net that we may consider using in the future.

2.1.15 Image Segmentation Metrics and Loss Functions

Segmentation Accuracy

The segmentation accuracy is a simple metric that measures the number of pixels that were classified correctly in the entire image as defined in the equation that follows.

$$accuracy = \frac{\#\ of\ correctly\ classified\ pixels}{\#\ of\ pixels}$$

The advantage of this metric is that it is simple to understand and easily interpretable. The output of a segmentation model will be a set of class probabilities assigned to each pixel. For binary classification problems, the class assigned to each pixel is determined by the predicted probability and a predetermined probability threshold. This probability threshold is usually 0.5 by default, but can be adjusted based on the behavior of the model on the validation data. The accuracy is then calculated based on the class labels according to the probability threshold.

Cross-Entropy

Cross-entropy is a loss function used for classification problems and is a concept that originated in probability and information theory. For a set of predicted class probabilities $\hat{p} = [\hat{p}_0, \dots, \hat{p}_n]$ and a ground-truth one-hot vector $c = [c_0, \dots, c_n]$ with a value of one in the position of the correct class and zeros elsewhere, the cross-entropy is defined as:

$$CE(\hat{p}, c) = -\sum_{i=0}^{n} c_i \log(\hat{p}_i)$$

For segmentation problems, the cross-entropy is usually calculated for each pixel and averaged across the entire set of pixels in each image in order to produce the final loss value [30].

Dice Coefficient

The dice coefficient [33] is a segmentation metric that can also be adapted to be used as a loss function. The dice coefficient measures the overlap between the predicted segmentation map and

the correct segmentation labels. For a predicted segmentation map $\hat{Y}$and the correct segmentation labels Y, the dice coefficient is defined as:

$$Dice(\hat{Y}, Y) = \frac{2|Y \cap \hat{Y}|}{|Y| + |\hat{Y}|}$$

The dice coefficient can also be adapted to produce a loss function known as the soft dice loss [33] as demonstrated in the following equation.

$$DL(\hat{Y}, Y) = 1 - \frac{2|Y \cap \hat{Y}|}{|Y| + |\hat{Y}|}$$

IOU (Intersection Over Union)

The intersection over union (IOU) [33] is a segmentation metric that is quite similar to the dice coefficient. For a predicted segmentation map $\hat{Y}$and the correct segmentation labels Y, the IOU is defined in the following equation.

$$IOU(\hat{Y}, Y) = \frac{|Y \cap \hat{Y}|}{|Y \cup \hat{Y}|}$$

For the purpose of evaluating a CNN's segmentation performance on multiple images, the mean IOU across all of the images in the testing or validation set is usually reported. Like the dice coefficient, the IOU can also be adapted to serve as a loss function [34] known as the Jaccard loss, which is defined as follows:

$$JL(\hat{Y}, Y) = 1 - \frac{|Y \cap \hat{Y}|}{|Y \cup \hat{Y}|}$$

2.1.16 Transfer Learning

Transfer learning is the practice of taking patterns learned from one machine learning task and reusing them for training models to perform a different task. Typically, transfer learning is applied to segmentation models by replacing the encoder section with the convolutional layers of a CNN such as Inception or ResNet that has been pre-trained on a dataset such as ImageNet. Figure 12 illustrates this approach. This is a technique that we use in this work and it can produce better results by allowing the network to make use of the features that have already been learned from the ImageNet classification task.

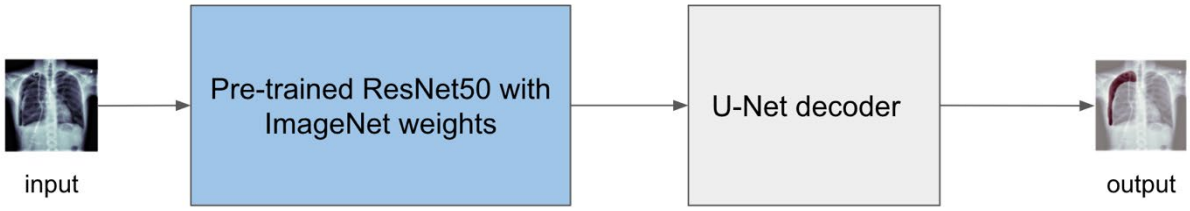


Figure 12: Illustration of transfer learning for image segmentation.

2.2 Kaggle SIIM ACR Pneumothorax Segmentation Challenge Dataset

2.2.1 Pneumothorax – Medical Background

Pneumothorax, often referred to as a collapsed lung, is a condition where air escapes from one or both of the lungs, and accumulates in the space outside the lung [35]. This condition can escalate to become life threatening, and must be treated immediately when diagnosed [35]. Pneumothorax can occur without a clearly connected cause (primary pneumothorax) or due to underlying lung disease (secondary pneumothorax) [35]. The diagnosis process usually requires a chest X-ray or

CT scan [35]. Because physicians viewing chest X-rays or CT scans can make interpretation errors, building accurate deep learning models for pneumothorax detection can potentially save lives.

2.2.2 Competition Background

The SIIM ACR Pneumothorax Segmentation Challenge, was a recent competition hosted by the Society for Imaging Informatics in Medicine (SIIM) on Kaggle, a platform for data science competitions [36]. The goal of the competition was to train models to detect pneumothorax, in chest X-rays [36]. The task is essentially a medical image segmentation problem because it involves identifying pneumothorax regions within chest x-rays. The competition data is publicly available on Kaggle and includes a training set and a test set that was only completely revealed at the end of the competition. The competition submissions were evaluated using the dice coefficient for image segmentation [36]. The competition was conducted in two stages. In the first stage, a set of labeled training images and unlabeled testing images were provided. For stage 2 of the competition, a larger training set consisting of both the training and testing images from stage 1 was compiled and a separate set of testing images was used for evaluation.

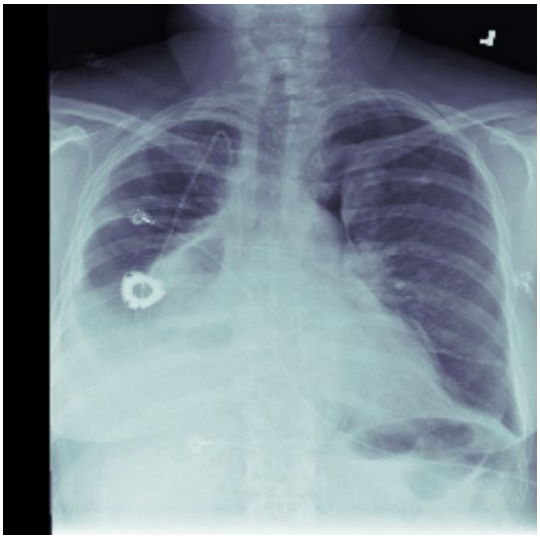


Figure 13: Chest X-ray image from the SIIM ACR Pneumothorax Segmentation Challenge.

2.2.3 Dataset Details

For the purpose of this work, we used the stage 2 training and testing images. The training data consists of 12,089 images in DICOM (Digital Imaging and Communications in Medicine) format and annotations containing image IDs and run-length-encoded (RLE) masks to provide the correct segmentation maps that should be used for training models [36]. Only some and not all of the images actually contained instances of pneumothorax and these images were labeled by binary segmentation masks in the image annotations. The testing set consisted of 3,205 images in DICOM format without accompanying annotations [36]. When preprocessing the data, we extracted the RGB image tensors for each image from the DICOM data.

2.3 Ensemble Machine Learning

Ensemble machine learning refers to a class of machine learning techniques that involve using multiple different machine learning models to obtain better predictive performance. In machine

learning competitions, the top performing submissions often involve ensemble machine learning techniques. The predictions of multiple models are typically combined to produce an ensemble model that performs better than each of the individual models.

2.3.1 Majority-Voting and Ensemble Averaging

Majority-voting and ensemble averaging are perhaps the simplest ensemble techniques and can be applied to a wide variety of machine learning algorithms. Majority-voting is a procedure where several different machine learning models for classification are trained on the same dataset, and at inference time, the class predictions of each model are aggregated using a voting scheme. Each model contributes a "vote" for the class that it predicts and the final class prediction corresponds to the class that received the most votes, or was predicted by the most models. Several papers that reported results on the ILSRVC claimed to achieve better results by creating ensembles of multiple CNNs using majority-voting.

Ensemble averaging is similar to majority voting, except that it is applied to problems that involve predicting continuous values as opposed to classification problems. Like majority-voting, ensemble averaging involves training multiple different models. However, since the predictions generated by the models are continuous, the final output is generated by averaging the individual predictions.

2.3.2 Stacked Generalization

Stacked generalization [2], also referred to as "stacking", is a more sophisticated form of ensemble machine learning. Rather than using a simple averaging or majority-voting scheme,

stacked generalization involves using a meta-learner that learns to combine the predictions of multiple machine learning models in order to produce the most accurate results [2]. The meta-learner predicts the final output based on the original input instance and the outputs produced by the sub-models. Stacked generalization generally works well when combining diverse models that have different types of advantages. This concept is powerful because the meta-learner or aggregating model, can learn to correct the mistakes of the sub-models and identify which sub-models perform better on certain types of input instances.

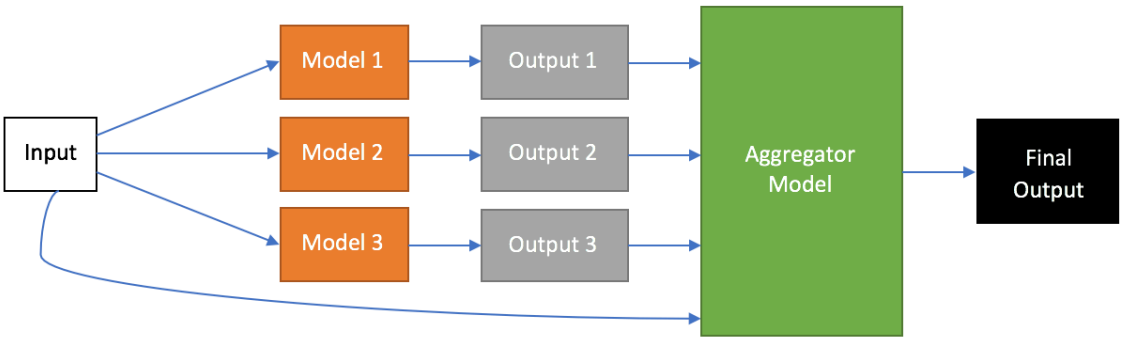


Figure 14: Visualization of stacked generalization.

Stacked generalization is not limited to the configuration in Figure 14 and can be expanded to include multiple layers of sub models and aggregator models. In practice, stacked generalization achieves the best results when a diverse range of models are included, because this allows the combined model to make use of the advantages of a wide variety of models.

CHAPTER 3

STACKED GENERALIZATION FRAMEWORK

3.1 Overview of Proposed Stacked Generalization Framework

The framework that we propose in this work is a general method for combining the predictions of multiple CNNs for segmentation using stacked generalization. Previously, ensemble learning techniques for CNNs were applied in an offline manner. Teams that participated in the ImageNet challenges often achieved their best results by combining the predictions of multiple CNNs, that were trained separately, using simple majority voting or averaging schemes. However, the proposed framework allows us to create a larger CNN consisting of multiple CNNs and an aggregator model that combines the predictions of these CNNs to produce the final output. This entire model is trained as a single unit, with computations on same image occurring in parallel with each CNN. The aggregator model receives the predicted segmentation maps of each CNN along with the input image and uses this information to produce the final output segmentation map. We propose several variations of this framework with different aggregator models. Figure 15 presents a visualization of the general flow from input to output in this framework.

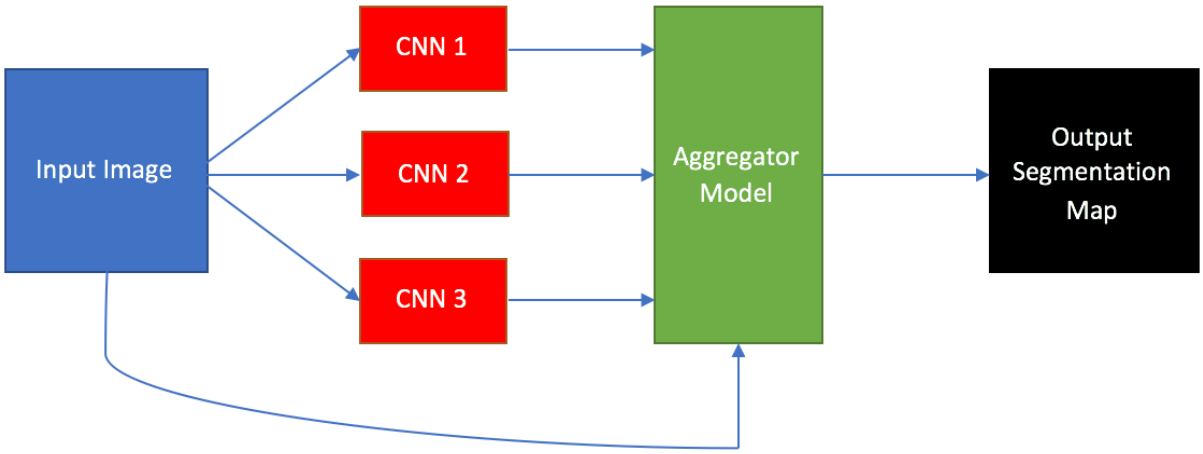


Figure 15: Stacked generalization framework with multiple CNNs.

We refer to this framework as StackingNet and propose two variations of the framework, StackingNetV1 and StackingNetV2. The main difference between each variation is the type of aggregator model used. In addition, since we have designed this framework for medical image segmentation, the CNNs that are included will be encoder-decoder architectures with pre-trained encoders that can be used for transfer learning.

3.1.1 StackingNetV1

StackingNetV1 is the version of this framework with the simplest aggregator model. The framework combines the predictions of multiple segmentation models using a series of 1 x 1 convolutions with a different number of filters. Each segmentation output is concatenated with the input image and passed into a separate layer with 1 x 1 convolutional filters. The results of

each convolutional layer are then concatenated and passed through a final 1 x 1 convolutional layer to produce the final segmentation map.

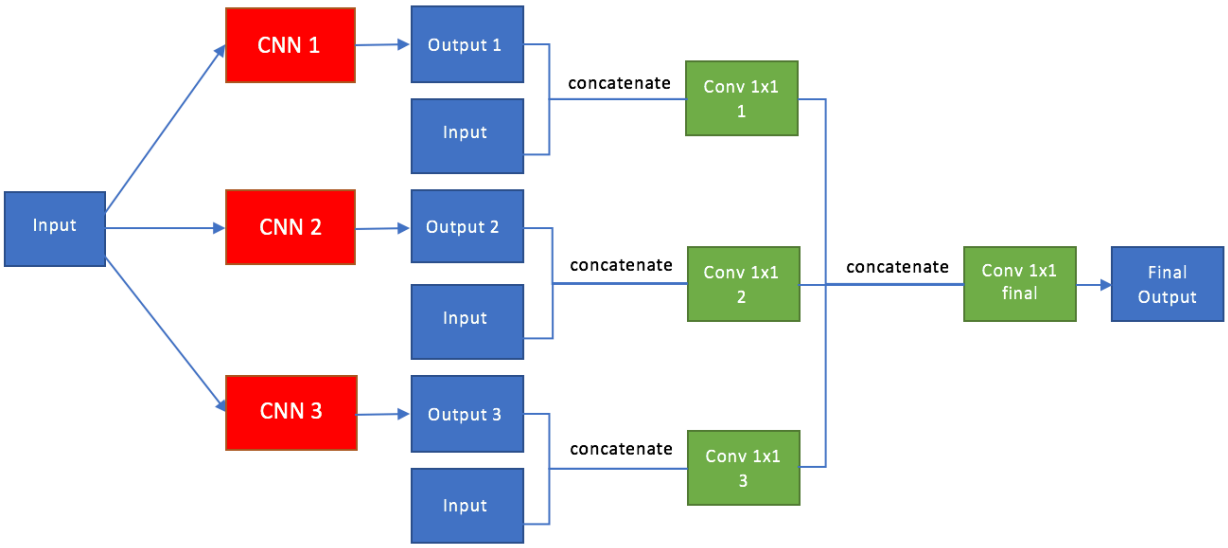


Figure 16: StackingNetV1 architecture with an ensemble of three CNNs.

A 1 x 1 convolutional layer multiplies a separate weight for each filter to each pixel value in each channel in the input and sums the values to produce each feature map in the output. Assuming the segmentation problem involves C classes, this framework was originally designed so that each 1 x 1 convolutional layer has C filters. This parameter must hold true for the final convolutional layer because it produces a segmentation map with C class probabilities assigned to each pixel. However, this parameter can be adjusted for the previous 1 x 1 convolutional layers.

One potential issue with this architecture occurs when encoder-decoder architectures with large pre-trained encoder networks are used. In these situations, the inclusion of multiple CNNs with large encoders drastically increases the amount of random access memory (RAM) required to train this architecture on a GPU. There are two options for addressing this issue. The first option is to distribute the model across multiple GPUs if it exhausts the RAM on one GPU. This technique was used in the AlexNet paper and deep learning libraries such as Keras make this configuration easy to implement given that multiple GPUs are already available on the machine being used. The second option is to use CNN architectures that are more memory efficient as sub-models.

3.1.2 StackingNetV2

StackingNetV2 involves an ensemble of multiple CNNs, but allows for more complex aggregator models than the set of 1 x 1 convolutions used in StackingNetV1. The outputs of all the sub-models are concatenated together along with the original input image and passed to an aggregator model that intelligently combines the intermediate outputs to produce the final output. Like the sub-models, the aggregator model can also be a fully-convolutional encoder-decoder architecture, but the difference is the encoder should not be pre-trained because the input to the aggregator model is not a standard image. The input to the aggregator model is a volume consisting of the original input image and the result of concatenating the predictions from each sub-model. The StackingNetV2 architecture described in this section is shown in Figure 17.

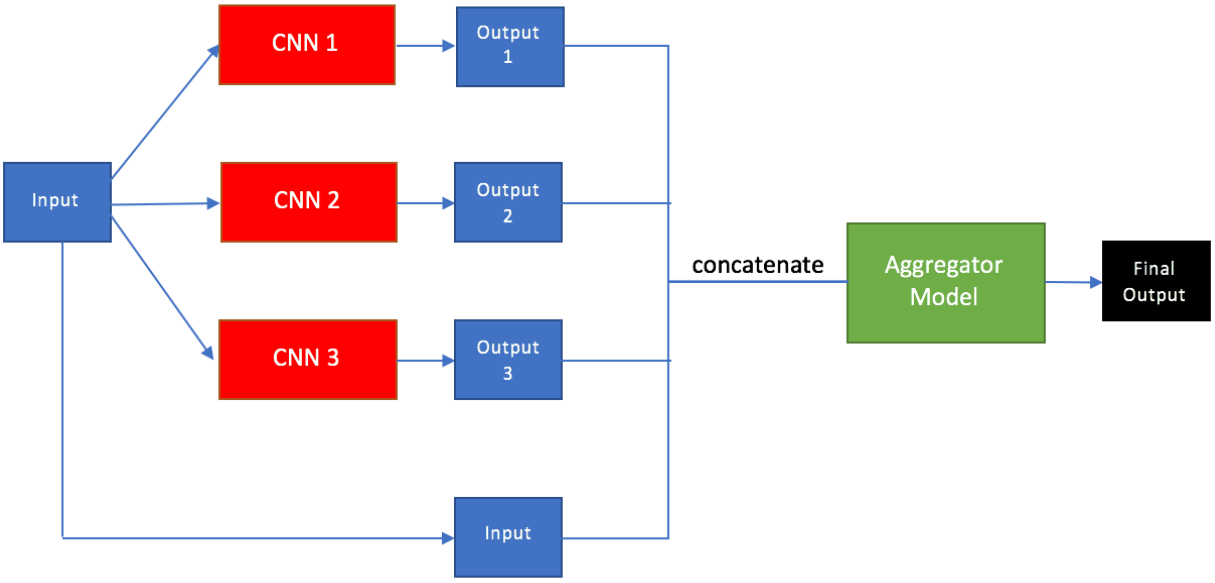


Figure 17: StackingNetV2 architecture.

3.1.3 StackingNet Training Algorithm

In order to train StackingNet effectively it is important to make use of transfer learning to avoid training the entire network with all of the sub-models and the aggregator model from scratch. For a StackingNet architecture with sub-models $S_1, \ldots S_n$ and an aggregator model A, we propose the following algorithm for training the network:

1. Initialize the weights of sub-models.
2. Train each of the sub-models on the given dataset separately and compute the performance of each sub-model on the validation data.
3. Assemble the StackingNet architecture with the aggregator model and freeze the weights in the sub-models.

4. Train the StackingNet architecture, keeping the sub-model weights frozen and only adjusting the weights of the aggregator model.
5. Optional step: repeat steps 1-4 with different StackingNet architectures and use these architectures as sub-models for an even larger StackingNet architecture.

Step 4 is important because failing to freeze the weights of the sub-models will allow the optimization algorithm to adjust both the weights of the aggregator model and the sub-models at the same time, potentially damaging the already optimized weights of the sub-models.

Note that step 5 allows us to produce extremely large architectures with multiple layers of stacking with different levels of sub-models. Figure 18 demonstrates the type of model that step 5 would produce. This option thus requires a significant amount of memory as the combined network will be extremely deep and will contain multiple separate paths for the input images. For the purpose of this work, we only experimented with a StackingNetV2 architecture containing a single set of sub-models and one aggregator model.

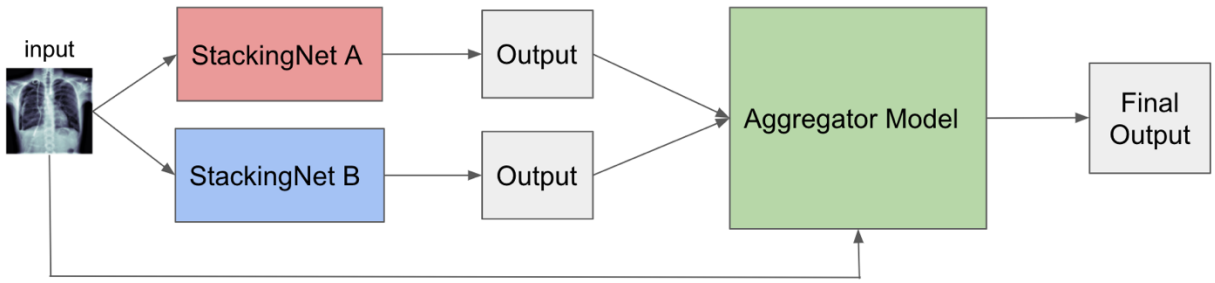


Figure 18: StackingNet with an additional level of stacked generalization.

3.2 Implementation Details

Each of the StackingNet variations were implemented in the Python programming language using Keras [37], a high-level deep learning library developed by Francois Chollet. We also made use of an open-source library called Segmentation-Models [38] to simplify the process of implementing the encoder-decoder architectures with pre-trained encoders. Most of the models for the Kaggle Pneumothorax Segmentation Dataset were trained on Kaggle's cloud platform with an available GPU, while the models for the other datasets were trained on an NVIDIA 1080Ti GPU.

3.3 Selecting Sub-Models for StackingNet

When selecting the sub-models that are to be used for each version of StackingNet, it is important to make sure that the sub-models can achieve similar levels of performance on the validation data. If one sub-model used in the framework performs significantly worse than the others, it can potentially saturate the predictions and lower the performance of the final model. In addition, ideally, the architectures of the sub-models should be different from each other. The variability in sub-models enables the aggregator model to recognize the types of instances that each sub-model performs well on and thus combine the predictions intelligently.

CHAPTER 4

METHODS

4.1 Methods for Kaggle Pneumothorax Segmentation Dataset

The dataset for the Kaggle SIIM-ACR Pneumothorax Segmentation Challenge consisted of images in DICOM format with annotations in the form of image IDs and run-length-encoded (RLE) masks. The goal of the competition was to train models to predict the existence of pneumothorax in the test images and the location and extent of the condition with the segmentation masks.

4.1.1 Training, Validation, and Testing Split

The competition involved two stages. The first stage included a separate training and testing set. The annotations for the training images for stage 1 were made accessible via the Cloud Healthcare API and the only the images were provided for the testing set. The results on the testing set were calculated using automated scoring software hosted on the Kaggle platform. The stage 2 training data combined the stage 1 testing and training data and used an additional separate set of images for testing. We evaluated our model's performance on the stage 2 testing data and split the stage 2 training data into separate sets for training and validation. The training and validation sets we used consisted of 10,842 images and 1,205 respectively. The private testing set consisted of 3,205 images.

4.1.2 Training Procedure

For the Kaggle Pneumothorax Segmentation Challenge dataset, we trained eight different CNN architectures, which are listed below:

1. U-Net w/ ResNet18 encoder (U-ResNet18)
2. U-Net w/ SEResNet18 encoder (U-SEResNet18)
3. U-Net w/ SEResNeXt50 encoder (U-SEResNeXt50)
4. U-Net w/ EfficientNetB0 encoder (U-EfficientnetB0)
5. U-Net w/ EfficientNetB2 encoder (U-EfficientnetB2)
6. U-Net w/ EfficientNetB5 encoder (U-EfficientNetB5)
7. StackingNetV1
8. StackingNetV2

Referring to the list above, StackingNetV1 was constructed as an ensemble of U-ResNet18, U-SEResNeXt50 and U-EfficientNetB0. StackingNetV2 was constructed with the previously trained U-SEResNeXt50 and U-EfficientNetB2 models as sub-models and with U-EfficientNetB0 as the aggregator model. For CNNs 1-6, a batch size of 16 images was used, whereas for the StackingNet variations, a smaller batch size of 8 images was used to avoid exhausting the RAM on the GPU due to the size of the networks. The loss function used for all networks was a combination of the binary cross-entropy and dice loss function. Each network was trained using the Adam optimization algorithm with an initial learning rate of 10^{-3} that was adjusted after each epoch using a procedure known as cosine annealing. Each network was

trained for a maximum of 30 epochs, with an early stopping scheme where training would be stopped early if the network did not improve its performance on the validation set for five consecutive epochs. During each training run, the performance of the model after each epoch was monitored and the weights that yielded the best performing model were saved to disk.
In order to prevent overfitting, the following batch-based data augmentation techniques were applied randomly using the Python albumentations library [39]:

- Horizontal Flips
- Randomly modifying either the contrast, gamma, or brightness of the images.
- Randomly performing either an elastic transformation, grid distortion, or optical distortion of the images.
- Extracting a random-sized crop from each of the original images.

4.1.3 Evaluation Metrics

We used the dice coefficient described in section 2.1.15 to evaluate the model's performance. This metric was calculated on the validation dataset using our own Python code, and calculated on the hidden test dataset using Kaggle's scoring software.

CHAPTER 5

RESULTS

5.1 Segmentation Metrics

We used the dice coefficient to evaluate the accuracy of each segmentation model. The dice coefficients across the validation set were calculated using our own Python code, while the dice coefficients across the testing set were calculated using Kaggle's submission portal and scoring software for the competition. We have verified that two computations are functionally equivalent based on the definition provided in our code and the mathematical definition outlined in the competition page. The difference in performance between the validation and testing sets can be explained by the fact that the testing set is much larger than the validation set.

Table 4: Segmentation results for each model.

Model	**Validation Dice**	**Testing Dice**
U-ResNet18	0.8458	0.7445
U-SEResNet18	0.8391	0.7768
U-SEResNeXt50	0.8614	0.7914
U-EfficientNetB0	0.8746	0.8002
U-EfficientNetB2	0.8527	0.8032
U-EfficientNetB5	0.8614	0.8062
StackingNetV1	0.8672	**0.8078**
StackingNetV2	0.8819	**0.8111**

Based on the results, we can clearly see that StackingNetV1 and StackingNetV2 outperformed the individual CNN models on the testing set, with StackingNetV2 achieving the best results out of all the models on both the validation and testing sets.

5.2 Training Graphs

In this section, we present graphs of the training and validation loss values for both the StackingNet variations as well as some of the individual CNN models listed in the previous section. The volatility of the loss curves for the validation set can be explained by the small size of this set compared to the training set. For each network, the best weight configuration was saved to disk if performance improvements on the validation data occurred after each epoch. We measured performance on the validation data based on the dice coefficient value and not the validation loss. For this reason, we also included the graphs showing the dice coefficient values of both the StackingNet variations on the training and validation during training.

5.2.1 StackingNetV1 Training Graphs

The loss graph for StackingNetV1 in Figure 18 features a rapid decrease in the training loss that plateaus over time. The network was trained for a total of 20 epochs, with the best model being obtained at the fourth epoch as demonstrated by the dice coefficient graph in Figure 19.

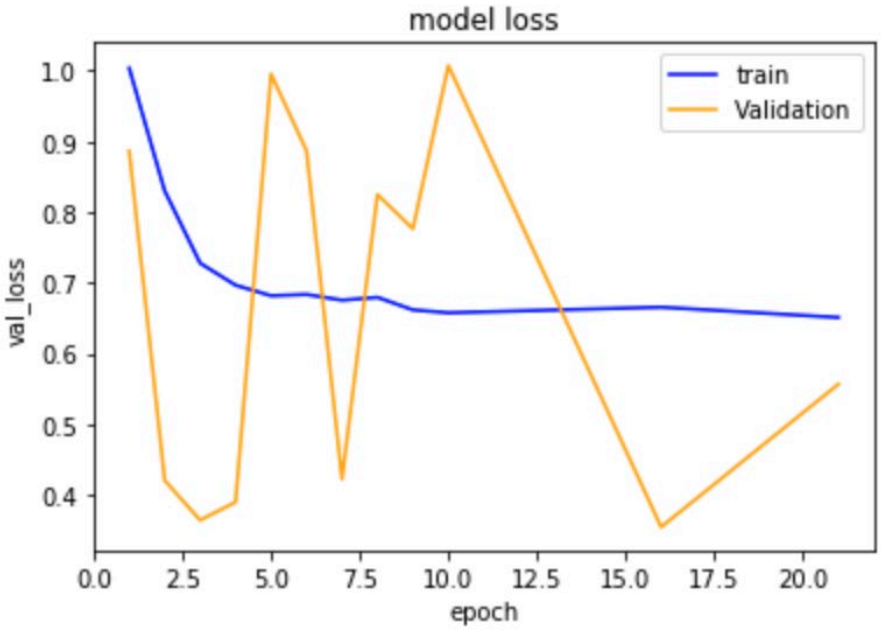


Figure 19: Loss graph for StackingNetV1

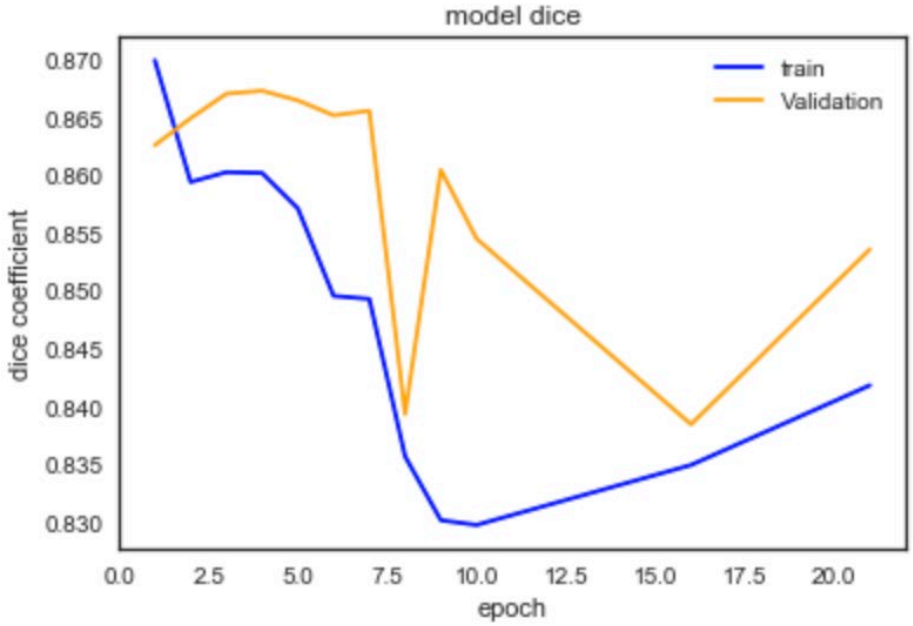


Figure 20: Dice coefficient graph for StackingNetV1

5.2.2 StackingNetV2 Training Graphs

The loss graph for StackingNetV2 in Figure 20 presents a much more gradual decrease in the training loss. The network was trained for a total of 20 epochs, with the best model being obtained at epoch 15 as demonstrated in the dice coefficient graph in Figure 21. The much more gradual decrease in the training loss and longer convergence time can be explained by the fact that the StackingNetV2 aggregator model is much more complex than that of StackingNetV2 and thus takes a longer time to train.

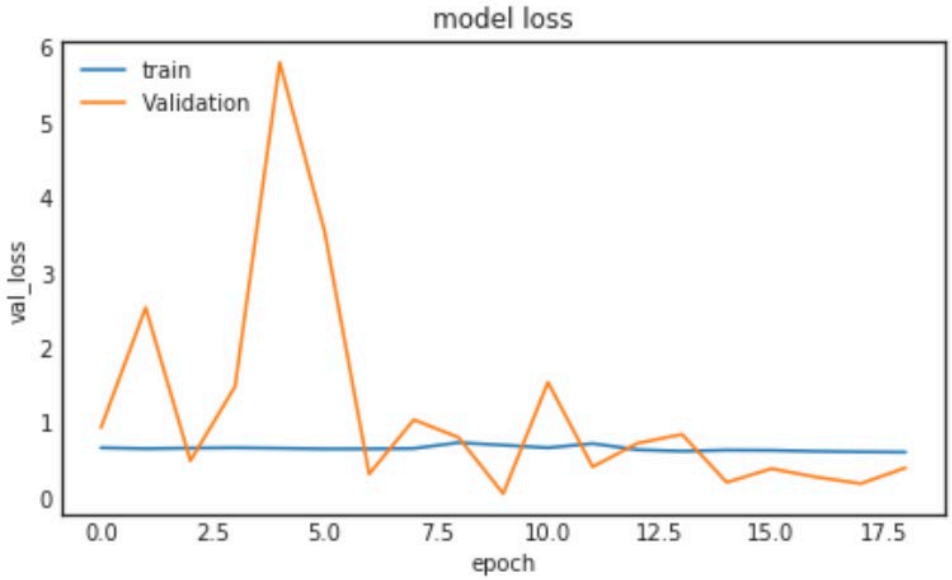


Figure 21: Loss graph for StackingNetV2

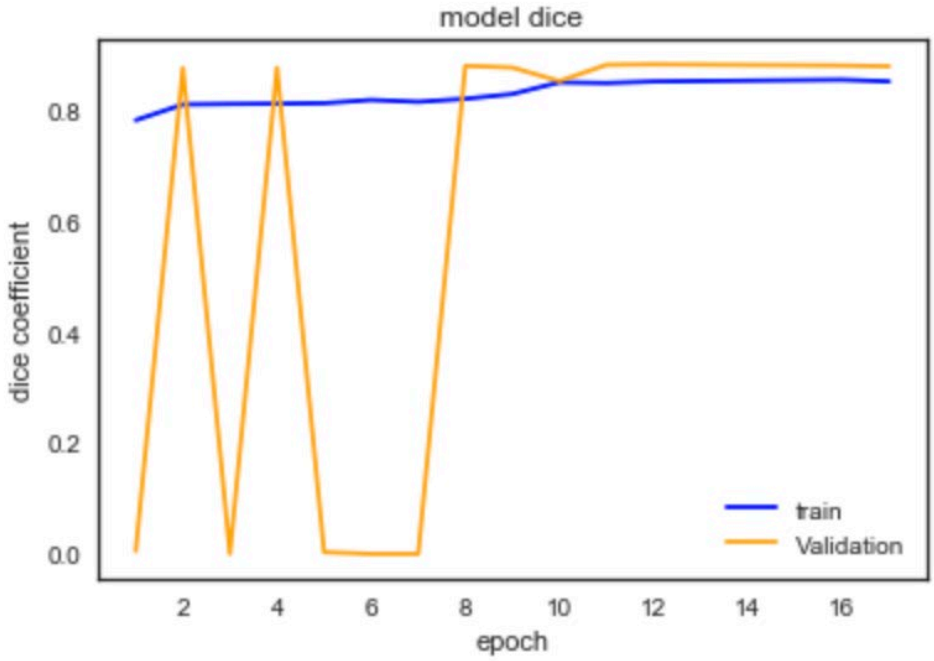


Figure 22: Dice coefficient graph for StackingNetV2.

5.2.3 U-EfficientNet Training Graphs

The loss graphs for EfficientNetB0, EfficientNetB2, and EfficientNetB5 are shown in Figures 23-25 respectively. All of the graphs seem to follow a similar trend with the training loss decreasing steadily throughout training. For U-EfficientNetB0 the best model was obtained after seven epochs and for EfficientNetB2 and EfficientNetB5, the best models were obtained after four epochs.

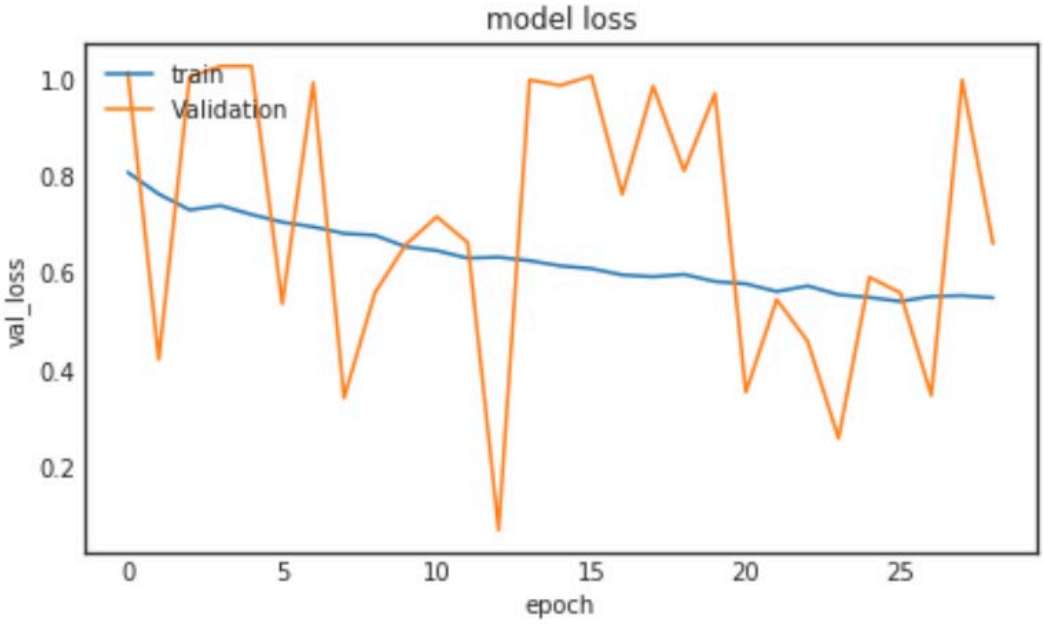


Figure 23: Loss graph for EfficientNetB0

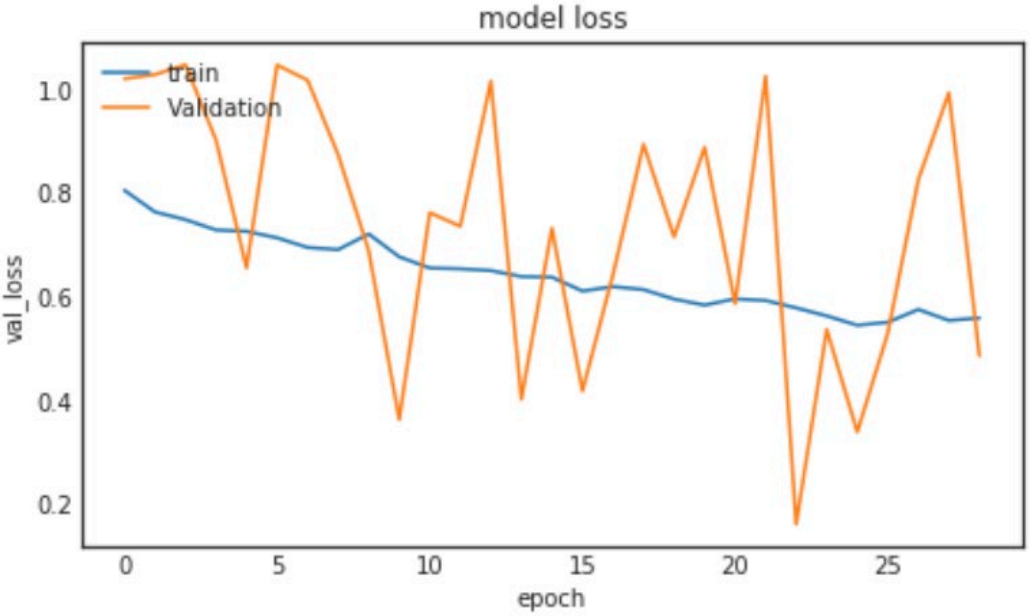


Figure 24: Loss graph for EfficientNetB2

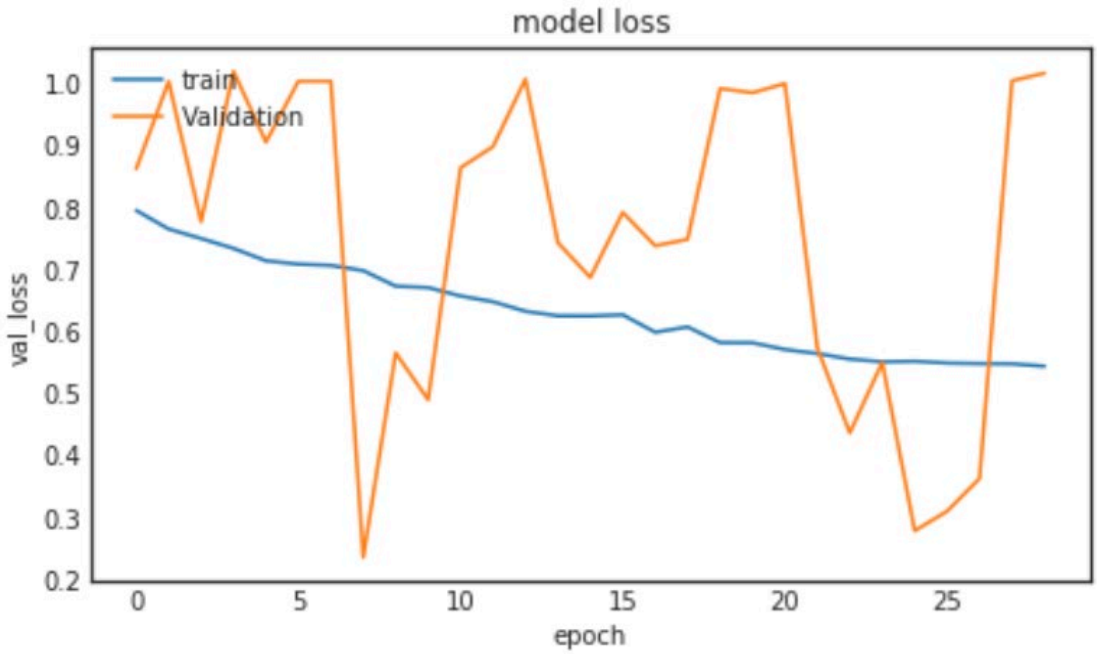


Figure 25: Loss graph for EfficientNetB5

5.2.4 SE-ResNeXt50 Training Graphs

The training loss follows a pattern similar to that of the EfficientNet variations in the SE-ResNeXt50 loss graph shown in Figure 25. The training loss decreases steadily throughout training, while the validation loss remains volatile, but reaches several local minima during the training process. Surprisingly, the best performing model on the validation data, in terms of the dice coefficient, was obtained after only two epochs.

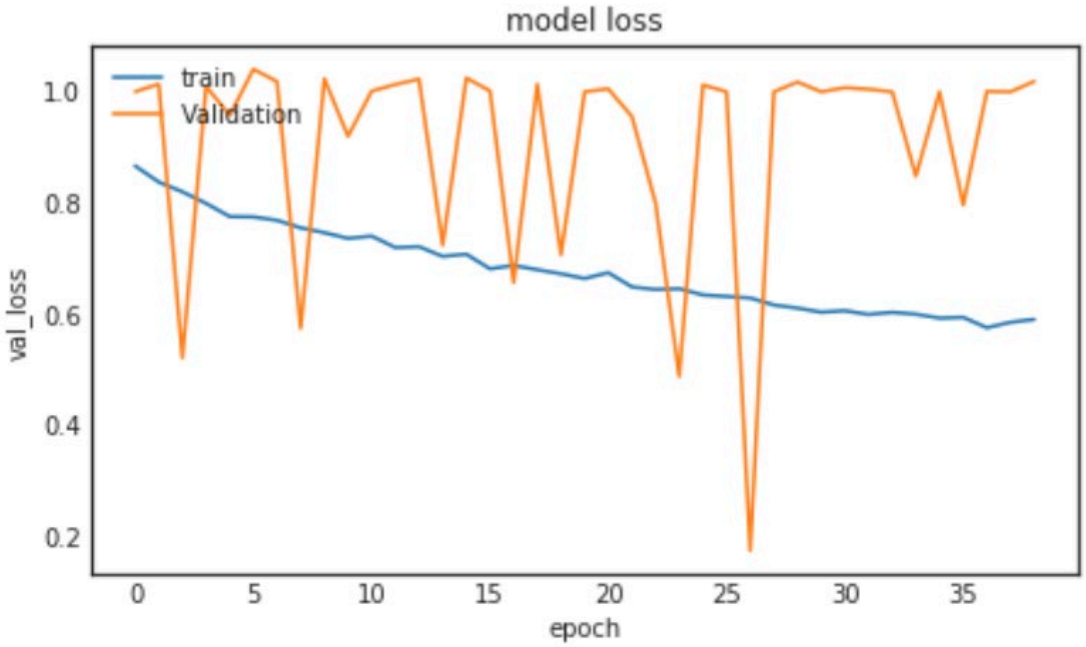


Figure 26: Loss graph for SE-ResNeXt50

5.3 Visualizations

In order to evaluate the performance of the trained models, we have also visualized the predictions of these models on a set of sample images from the validation set. Due on the competition format, we do not have direct access to the correct labels for the testing images because the test scores are computed using Kaggle's submission portal and scoring software. For this reason, visualizations on the validation set are more effective for evaluating the accuracy of the models because there are ground truth segmentation maps that we can visualize and compare to the predicted segmentation maps from each model.

5.3.1 Validation Set Visualizations for StackingNetV1

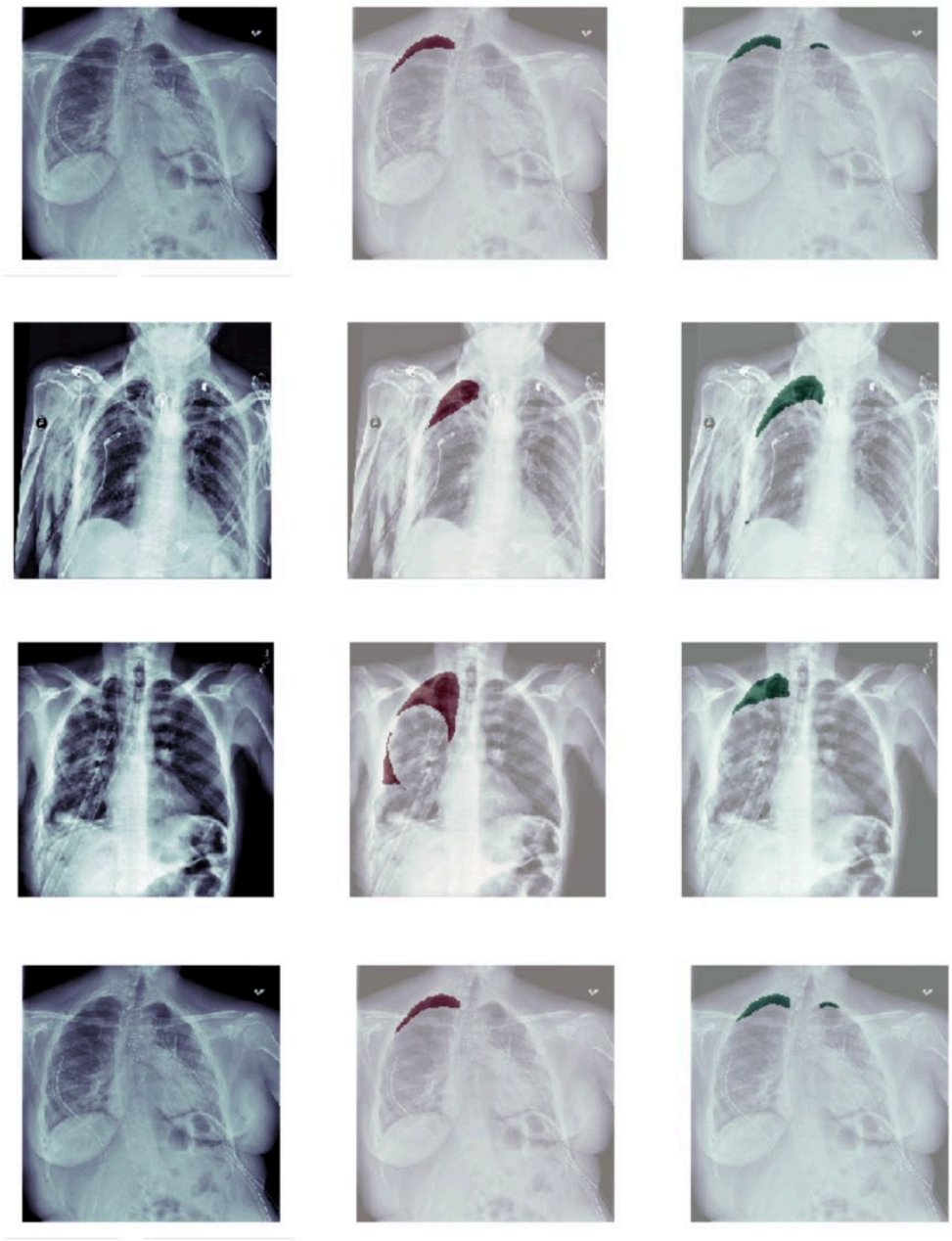


Figure 27: Sample visualizations of StackingNetV1 predictions on the validation set: original images (left), ground truth segmentation maps (center), predicted segmentation maps (right).

5.3.2 Validation Set Visualizations for StackingNetV2

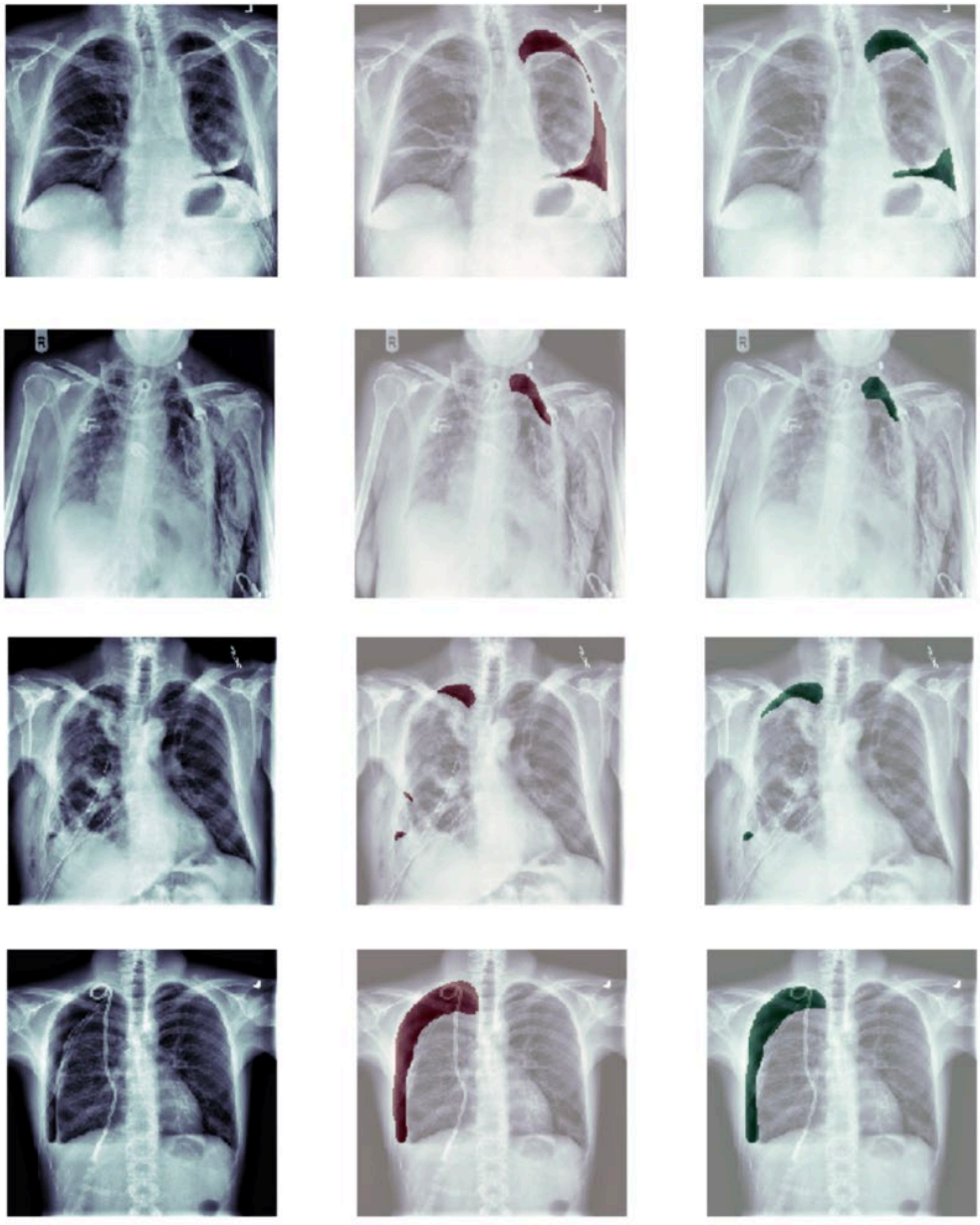


Figure 28: Sample visualizations of StackingNetV2 predictions on the validation set: original images (left), ground truth segmentation maps (center), predicted segmentation maps (right).

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The goal of this research was to design a framework for applying the principles of stacked generalization to training CNNs and to demonstrate that the designed framework can produce improvements in performance when compared to methods involving individual CNNs. Based on the results, it is clear that the StackingNetV2 architecture composed of EfficientNetB0 and SE-ResNeXt50 sub-models and an EfficientNetB2 aggregator model outperformed all of the other models we tested on the validation and testing data. Both StackingNetV1 and StackingNetV2 not only outperformed the individual sub-models, but also outperformed the much larger and more complex EfficientNetB5. We can clearly see that both StackingNet variations have the potential to produce performance improvements by combining the predictions of different sub-models with an aggregator model.

We purposely chose sub-models that achieved similar metrics on the validation data and had fundamentally different architectures. By following this principle and combining SEResNeXt50 and EfficientNetB2 sub-models, our StackingNetV1 and StackingNetV2 achieved the results that we desired. Had we chosen a set of sub-models with large performance gaps or sub-models that were very similar (such as two EfficientNet variations), we may not have achieved the same results. In practice, a variety of CNN architectures should be trained and compared based on their performance on the validation data before selecting the CNNs that will be used in a StackingNet ensemble. In many application studies involving CNNs for image segmentation,

researchers will try different architectures or train different versions of the same architecture with different hyper-parameters and the StackingNet algorithm provides a clear method for combining the results of multiple experiments to produce an even better ensemble model.

Based on the performance of StackingNet on the segmentation task from Kaggle Pneumothorax Segmentation Challenge, we can conclude that the StackingNet algorithm has the potential to yield performance improvements on medical image segmentation tasks. Furthermore, the algorithm is flexible and can be modified to produce ensemble models with several layers of stacking and provides an effective method for using the principles of stacked generalization to improve performance. For medical imaging tasks where performance is critical and researchers may resort to trying a wide variety of CNN architectures and hyper-parameter combinations to achieve the best performance. Rather than reinventing the wheel and designing more sophisticated CNN architectures, researchers can instead use StackingNet to combine different CNN architectures that they have already trained and achieve performance improvements with less effort. While memory issues may occur when using StackingNet to combine larger, more memory-intensive CNN architecture such as VGG on smaller GPUs, techniques such as distributing models across multiple GPUs can often solve these problems. We believe that this framework provides a path for effectively taking advantage of stacked generalization and ensemble learning to produce more robust and more accurate deep learning models for image segmentation.

6.2 Future Work

This study does have several limitations which suggest areas that must be investigated further in order to make a stronger case for the StackingNet algorithm.

The first limitation of this study is that due to time constraints, we were only able to obtain performance metrics for one medical imaging dataset. Ideally, we would like to compare the performance of StackingNet ensembles against individual CNNs across multiple medical imaging datasets. In a previous work that we presented at the 2020 SPIE Medical Imaging conference, we trained a U-Net CNN to perform cancer segmentation using a dataset of histological slides from head and neck squamous cell carcinoma patients [31]. Since this dataset was novel and represented a challenging medical imaging task, it would be interesting to compare StackingNet's performance to the individual CNNs such as the U-Net trained on this dataset.

In addition to evaluating StackingNet's performance on different medical imaging datasets, we would also like to evaluate StackingNet's ability to combine CNNs for benchmark object detection and segmentation tasks. For this purpose, we may consider standard object detection and segmentation datasets such as the Common Objects in Context (COCO) dataset [40]. The COCO dataset is a large object detection, segmentation, and image captioning dataset [40]. The dataset is public and is a project maintained by individuals from several universities and research groups such as Google Brain and Facebook AI Research (FAIR). The entire dataset contains 330,000 images, with over 200 labeled images. Like ImageNet, the COCO dataset has separate challenges each year and these challenges have often been used as benchmarks for evaluating

CNNs for object detection and segmentation [40]. Since COCO is such a widely used dataset, evaluating StackingNet's performance on this dataset would allow us to compare it to a much broader range of methods used in previous works.

The second limitation of this study is that we only considered very simple StackingNetV1 and StackingNetV2 ensembles with only at most three different CNNs due to the limited amount of RAM on our GPUs. In the future, we would like to consider *wider* StackingNet models with a larger number sub-models and *deeper* StackingNet models with multiple layers of stacked generalization. We could make these experiments feasible by either using combinations of smaller, less memory-intensive CNN architectures or by distributing the models across multiple GPUs with more memory.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G.E. Hinton. "ImageNet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 1, pp. 1097–1105. 2012.

[2] D.H. Wolpert, "Stacked Generalization", *Neural Networks*, vol. 5, no. 2 pp. 241 – 259. 1992.

[3] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, vol. 65 no. 6 pp. 387 – 408. 1958.

[4] S. Ronaghan, "Deep Learning: Overview of Neurons and Activation Functions," *Medium*, 26-Jul-2018. [Online]. Available: https://medium.com/@srnghn/deep-learning-overview-of-neurons-and-activation-functions-1d98286cf1e4. [Accessed: 27-Mar-2020].

[5] H. Robbins and S. Monro. "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3. pp. 400-407. Sep. 1951.

[6] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. "Learning Representations by Back-propagating Errors," *Nature*, vol. 323 no. 9. pp. 533 – 536. Oct. 1986.

[7] F.-F. Li, J. Johnson, and S. Yeung, “Convolutional Neural Networks (CNNs/ConvNets),” *CS231n Convolutional Neural Networks for Visual Recognition*. [Online]. Available: https://cs231n.github.io/convolutional-networks/#conv. [Accessed: 09-Apr-2020].

[8] Y. LeCun and P. Haffner, L. Bottou, and Y. Bengio. “Object Recognition with Gradient-Based Learning,” in *Proceedings of Shape, Contour and Grouping in Computer Vision*, 1999, p. 319.

[9] S. Ioffe, C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” in *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[10] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. “Dropout: A Simple Way to Prevent Neural Network Overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929 – 1958. 2014.

[11] A. Krogh, J.A. Hertz. “A Simple Weight Decay Can Improve Generalization,” in *Proceedings of the 4th International Conference on Neural Information Processing Systems*, 1991.

[12] A. Mikołajczyk and M. Grochowski, "Data augmentation for improving deep learning in image classification problem," *2018 International Interdisciplinary PhD Workshop (IIPhDW), Swinoujście*, 2018, pp. 117-122.

[13] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” pp. 32--33, 2009.

[14] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. “Imagenet: A large-scale hierarchical image database,” in *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[15] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” CoRR, vol. abs/1409.1556, 2014.

[16] C. Szegedy et al., “Going Deeper with Convolutions.” 2014.

[17] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770-778.

[18] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He, “Aggregated Residual Transformations for Deep Neural Networks,” in *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[19] G. Huang, Z. Liu, L. v. d. Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261-2269.

[20] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, “Squeeze-and-Excitation Networks,” in *IEEE Transactions of Pattern Analysis and Machine Intelligence,* 2017.

[21] M. Tan and Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” 2019.

[22] M. Tan *et al.*, “MnasNet: Platform-Aware Neural Architecture Search for Mobile.,” in *CVPR*, 2019, pp. 2820–2828.

[23] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.” 2017.

[24] Polyak, B.T., “Some methods of speeding up the convergence of iteration methods”. *USSR Computational Mathematics and Mathematical Physics*, vol. 4 no. 5, pp. 1–17, 1964.

[25] J. C. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.,” J. Mach. Learn. Res., vol. 12, pp. 2121–2159, 2011.

[26] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” CoRR, vol. abs/1212.5701, 2012.

[27] D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization.” 2014.

[28] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning.” 2016.

[29] E. Shelhamer, J. Long, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, 2017.

[30] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional Networks for Biomedical Image Segmentation,” in *Medical Image Computing and Computer-Assisted Intervention -- MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234--241.

[31] A. Mavuduru, M. Halicek, M. Shahedi, J. V. Little, A. Y. Chen, L. L. Myers, and B. Fei "Using a 22-layer U-Net to perform segmentation of squamous cell carcinoma on digitized head and neck histological images", Proc. SPIE 11320, Medical Imaging 2020: Digital Pathology, 113200C (16 March 2020).

[32] T.-Y. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, “Feature Pyramid Networks for Object Detection.,” in *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 936–944.

[33] Jeremy Jordan, “Evaluating image segmentation models.,” *Jeremy Jordan*, 18-Dec-2018. [Online]. Available: https://www.jeremyjordan.me/evaluating-image-segmentation-models/. [Accessed: 27-Mar-2020].

[34] D. Zhou *et al.*, “IoU Loss for 2D/3D Object Detection.,” *CoRR*, vol. abs/1908.03851, 2019.

[35] P. Zarogoulidis, I. Kioumis, G. Pitsiou, K. Porpodis, S. Lampaki, A. Papaiwannou, N. Katsikogiannis, B. Zaric, P. Branislav, N. Secen, G. Dryllis, N. Machairiotis, A. Rapti, and K. Zarogoulidis, “Pneumothorax: from definition to diagnosis and treatment,” *Journal of thoracic disease*, Oct-2014. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4203989/. [Accessed: 29-Mar-2020].

[36] S. G. Langer, G. Shih, Society for Imaging Informatics in Medicine (SIIM), American College of Radiology (ACR), “SIIM-ACR Pneumothorax Segmentation,” *Kaggle*. [Online]. Available: https://www.kaggle.com/c/siim-acr-pneumothorax-segmentation. [Accessed: 29-Mar-2020].

[37] F. Chollet, “Keras”, 2015, https://keras.io.’

[38] P. Yakubovskiy, “Segmentation Models,” *GitHub repository*, GitHub, 2019, https://github.com/qubvel/segmentation_models.

[39] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, “Albumentations: Fast and Flexible Image Augmentations,” in *Information*, vol. 11, no. 2, 2020.

[40] T.-Y. Lin *et al.*, “Microsoft COCO: Common Objects in Context.” 2014.

BIOGRAPHICAL SKETCH

Amol Mavuduru was born in Phoenix, Arizona on September 11, 1997. He joined UT Dallas as an undergraduate computer science student in August 2016. He joined the Quantitative BioImaging Laboratory led by Dr. Baowei Fei in May 2018. The following year, he won a Jonsson School Undergraduate Research Award for his work in Dr. Fei's lab on a year-long research project focused on training convolutional neural networks to perform head and neck cancer segmentation. He graduated Magna Cum Laude from UT Dallas with his bachelor's degree in computer science in May 2019 and matriculated into the computer science master's program at UT Dallas that same year. As a graduate student, he continued working in Dr. Fei's lab and co-authored a paper titled "Using a 22-layer U-Net to perform segmentation of squamous cell carcinoma on digitized head and neck histological images", which he presented at the SPIE 2020 Medical Imaging Conference in Houston, Texas.

CURRICULUM VITAE

Amol Mavuduru
Email: Amol.Mavuduru@utdallas.edu
Website: www.amolmavuduru.me

EDUCATION

University of Texas at Dallas
M.S., Computer Science — August 2018 - May 2020

University of Texas at Dallas
B.S., Computer Science — August 2016 - May 2019

WORK EXPERIENCE

Data Science Intern
Hunt Oil Company — May 2019 – August 2019

- Trained predictive models for optimizing the oil well completion process using time-series data collected in real-time.
- Made models accessible to engineers by embedding them in a web application.

Big Data Analytics and AI Intern
Verizon Communications Inc. — June 2018 – August 2018

- Built solutions for various machine learning use cases ranging from price prediction to email parsing.
- Presented solutions to management, including the CIO of Verizon.

RESEARCH EXPERIENCE

Research Assistant
Quantitative BioImaging Laboratory — May 2018 – present

- Worked under Dr. Baowei Fei on research projects focused on training deep learning models to perform medical imaging tasks such as cancer segmentation.
- Received a Jonsson School Undergraduate Research Award for my work.
- Lab website: fei-lab.org

PUBLICATIONS

A. Mavuduru, M. Halicek, M. Shahedi, J. V. Little, A. Y. Chen, L. L. Myers, and B. Fei "Using a 22-layer U-Net to perform segmentation of squamous cell carcinoma on digitized head and neck histological images", Proc. SPIE 11320, Medical Imaging 2020: Digital Pathology.