ACIT5900

MASTER THESIS

in

Applied Computer and Information Technology (ACIT)

May 2022

Applied Artificial Intelligence

Exploring the Hyperparameter Space of U-Net using Genetic Algorithms

Jon-Olav Holland

Department of Computer Science

Faculty of Technology, Art and Design

Abstract

U-Net based architecture has become the de-facto standard approach for medical image segmentation in recent years. Many researchers have used the original U-Net as a skeleton for suggesting more advanced models such as UNet++ and UNet 3+. For our project, we also seek to optimize the original U-Net. Rather than changing the architecture itself, we optimize hyperparameters which does not affect the architecture, but affects the performance of the model. To optimize the hyperparameters, we use genetic algorithms. After the genetic algorithms have converged, we analyze the results and try to understand why the key factors behind explaining the performance.

Acknowledgments

I am very thankful to my supervisors, Youcef Djenouri and Anis Yazidi for their guidance and support throughput this journey.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Image segmentation models have been gaining traction over the last years. The segmentation models are used in a variety of important fields, some fields which are now being applied to real world applications. One of the most notable fields is medical imaging. In fact, these networks have become so useful that a hospital in Bergen has begun using them for tumor detection (E-Helse, 2019). The models are used as an assistance tool for doctors, yielding the probability of the patient having a tumor. As well as medical assistance, image segmentation is seeing the real world application use in self-driving vehicles. Our goal is to optimize the U-Net model, which is wildly used segmentation model. We seek to optimize the hyperparameters of the model using genetic algorithms, further increasing the performance of the model.

1.1 Motivation

When creating an artificial neural network many choices have to be made about the hyperparameter of the model. Deciding the value of these hyperparameters may seem arbitrary, as it is extremely difficult to assess the optimal value. If the goal is to create an acceptable model, then setting the hyperparameters to a common value would yield such a model. However, finding the optimal value for these hyperparameters is how one can push the model to its very limit. Increasing the performance of a model by even a few percentages, or a fragment of a percentage, can make a large difference in the long run. Whenever image segmentation models are applied to real world cases such as determining the probability of cancerous cells, one percentage increase in performance can be paramount.

A trivial approach to finding the optimal hyperparameters considers testing all possible configurations generated from the domains of the hyperparameters. The number of possible configurations is exponential regarding the number of hyperparameters, while the overall problem is combinatorial. To reduce the computational runtime of the hyperparameter optimization process, we apply genetic algorithms. This allows us to go through the configurations methodically and derive the best or near-best values for the hyperparameters. This method is especially useful for beginners working with deep learning, as they have not yet built up an intuition for appropriate hyperparameter values. Though, the method is not wasted on experts within the field. Even with great intuition and experience, the first proposed hyperparameter value is rarely the optimal one.

The work reported in Ronneberger et al. (2015) introduced U-Net in 2015, and since then the model has been applied to several domains within deep learning computer vision. U-Net is a successful model with many successors. The successors use U-Net as a skeleton, but seek to further improve the model by making minor changes to the architecture. However, none of those successors seeks to improve hyperparameters of the U-Net model itself. We propose hyperparameter optimization to enhance and improve the U-Net model by assessing the optimal hyperparameter values.

1.2 Ethical Implications

The project in itself does not introduce any new technology, but rather seeks to enhance current existing technology. Therefore, the project does not bring in any new ethical dilemmas.

Chapter 2

Theory and Background

This section contains deep learning, and machine learning methods relevant to the project.

2.1 Image Segmentation

The process of image segmentation is to “segment” or “partition” an image into different categories. For example, the functionality in a Zoom call which allows you to change your background, uses image segmentation to differentiate you from the background. This is just one practical example of where image segmentation can be useful. Image segmentation also has applications in face recognition, video surveillance, object detection, medical imaging, and more. Some of these applications work with two-dimensional data, while others work with three-dimensional data.

There are two types of image segmentation: *semantic segmentation*, and *instance segmentation*.

- **Semantic segmentation,** where the goal is to classify each pixel in an image to a category. If you have an image of a forest, the goal of semantic segmentation is to divide each tree under the tree category.

- **Instance segmentation,** which does exactly what semantic segmentation does but takes it one step further. Instance segmentation seeks to divide objects of the same category into a sequence, an instance segmentation of the forest image would then divide the trees into tree 1, tree 2, and so forth.

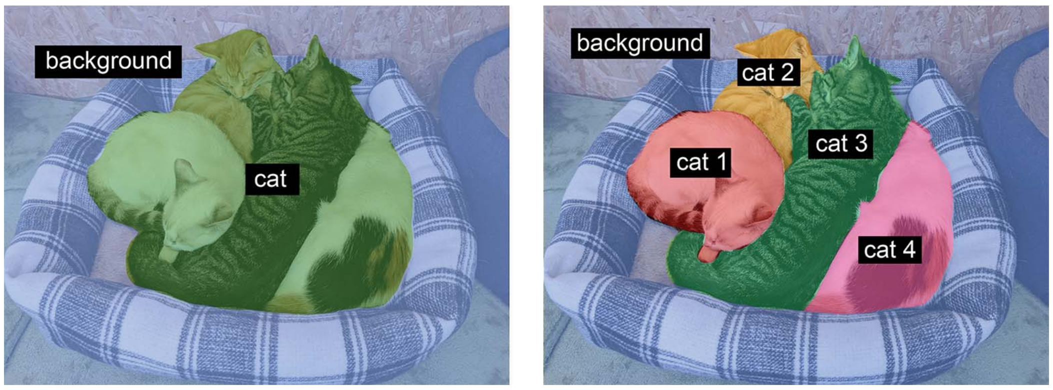


Figure 2.1: Semantic segmentation vs. instance segmentation (Chollet, 2021).

Figure 2.1 visually displays the difference between semantic and instance segmentation. The work of this thesis will focus on semantic segmentation and the phrase will be used interchangeably with image segmentation.

Object detection and segmentation is similar. The goal of object detection is to find the different classes of objects in a given image. Object detection marks the detected object with a square frame. Object detection does not describe the shape of the object, it only shows the location. For some tasks, object detection does not satisfy the requirement. For example, when trying to detect cancerous cells, the shape of the cancerous cell is instrumental when determining the severity of the cancer.

2.2 Convolutional Neural Network

Convolutional Neural Networks (CNN) have proven to be extremely successful when applied to computer vision tasks. Compared to regular densely connected neural networks, CNN have proven to be superior for computer vision. A CNN usually contains two core operations: *convolution* and *pooling*.

2.2.1 Convolution

The origin of the convolutional operation is directly inspired by the biological visual cortex. In short, it attempts to replicate the receptive field by creating artificial neurons which slightly overlap to cover the entire visual field. This yields a set of properties for the CNN that a dense artificial neural network (ANN) is unable to replicate.

Firstly, a CNN is translation-invariant. After a CNN has learned a certain feature or pattern, it is able to recognize this pattern in different locations. Regardless of whether this pattern is at the top left corner of an image, the center, or at the very edge, the CNN should be able to recognize it. A dense ANN would have to be retrained to recognize the pattern in a different location. For a dense ANN, there is no spatial representation for the input, it's a linear sequence of inputs. As a result of the translation-invariance, CNN require fewer training samples to learn representation and it makes CNN better for generalization.

Secondly, the convolutional operation makes the CNN able to learn spatial information in hierarchies. The very first convolutional operation will seek to learn the small features of the image. The next convolutional operation will operate on a slightly smaller sized representation and will seek to learn slightly larger features, and so forth. Therefore, CNN are able to learn larger and more complex features. Figure 2.2 represents the LeNet-5 CNN architecture which stems from one of the most cited papers in deep learning (Lecun et al., 1998).

The convolutional operation works by creating a set of **kernels** (also known as filters). The kernel is a rectangle which is given a size, commonly *3x3* or *5x5*. If the input is a two-dimensional image, the kernels slide across the input image, if the size of the kernel is *3x3*, this corresponds to *3* pixels. The **stride** defines the rate

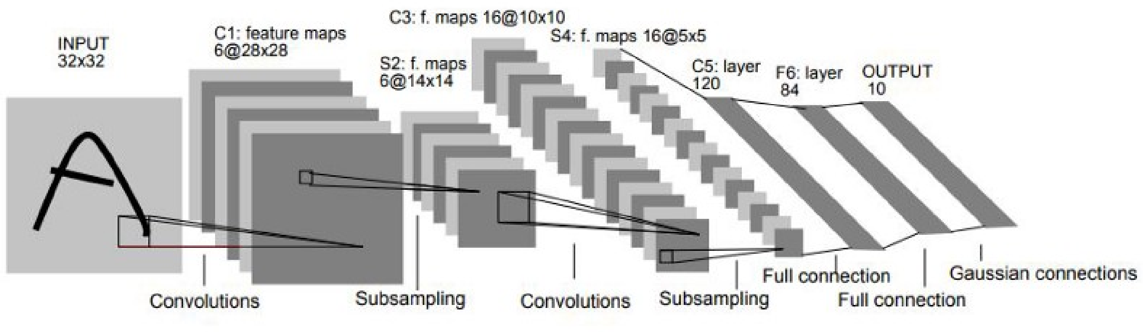
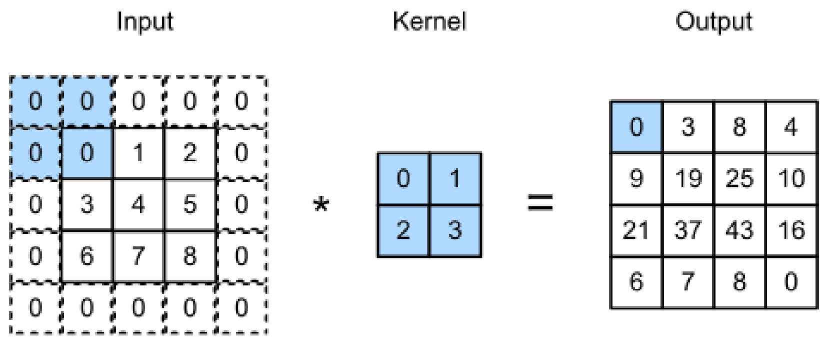


Figure 2.2: LetNet-5 (Lecun et al., 1998).

at which the kernel is moving, if the stride is set to be *1*, the kernel stops at every possible location until it has slid through the entire input. If the stride is set to *2*, the kernel will move *2* units at the time.



$$\begin{array}{|c|c|c|c|c|}\hline 0&0&0&0&0\\\hline 0&0&1&2&0\\\hline 0&3&4&5&0\\\hline 0&6&7&8&0\\\hline 0&0&0&0&0\\\hline\end{array} * \begin{array}{|c|c|}\hline 0&1\\\hline 2&3\\\hline\end{array} = \begin{array}{|c|c|c|c|}\hline 0&3&8&4\\\hline 9&19&25&10\\\hline 21&37&43&16\\\hline 6&7&8&0\\\hline\end{array}$$

Figure 2.3: Visualizing the kernel convolution with kernel size 2. The input also contains padding as can be seen by the border of 0s around the input.

Figure 2.3 shows a sample of how the kernel operates on the given input. In this figure, the kernel size is *2x2*, while the stride is unknown. The values of the kernel are the trainable weights for a CNN. Here we can also see the output of the kernel for one iteration of the process. The sum of the output is defined by equation 2.1.

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x-dx, y-dy) \tag{2.1}$$

Here, $f(x,y)$ is the input which the kernel is applied to. While $g(x,y)$ is the output which is created by the kernel sliding across the input. The ω represents the kernel. And dx, dy is the stride, or the rate of change in position. Lastly, a, b is the size of the input.

A normal convolutional operation consists of many kernels being applied to the input, which yields a higher dimension output. It is not uncommon to have hundreds of kernels applied, which will lead to hundreds of outputs. Equation 2.1 defines the output for only one kernel, to get the complete output of the convolutional layer, all the outputs generated by all the kernels have to be concatenated.

One can also use convolutional layers to downsample the data. Downsampling is beneficial for learning the features in a hierarchy as previously discussed. It is also beneficial to decrease the overhead, as the number of dimensions increases. Downsampling with convolutional layers is dependent on the size of the stride. For example, when using stride *2*, instead of stride *1*, the kernel moves two units at the time. Looking back at Equation 2.1, this will make the output half the size of the input. However, more commonly, pooling is used to downsample the data.

2.2.2 Pooling

Pooling can be used to downsample or upsample the data. Pooling is often preferred over using *stride>1* for downsampling. The pooling operation is similar to the convolutional kernel operation as they both take a window and slide it across the input. Both kernel and pooling choose the size of the filter, and the size of the stride. However, the pooling window does not have any trainable weights like the kernel. Instead of the mathematical relationship between the weights of the kernel and the current values the kernel window is sliding over, the pooling operation simply takes the maximum, minimum, or average value in the window. The operations are respectively called max pooling, minimum pooling, and average pooling. The former, max pooling, being the most commonly used of the three.

Due to having no trainable weights, the computational overhead of the pooling layer is very low compared to the convolutional layer. Though it is still crucial for a CNN to downsample the data to learn the features in hierarchy.

2.2.3 Batch Normalization

Ioffe & Szegedy (2015) introduced batch normalization and it has quickly become one of the staple operations within deep learning. Many different neural network architectures utilize batch normalization, including CNN such as U-Net. Batch normalization was originally crafted as a *normalization* technique within neural networks. As a data preprocessing technique, it is common to normalize the data so that the input is centered around 0 and has a unit standard deviation. Normalization leads to stability in the model, while large values might trigger large gradients which can cause the model to diverge instead of converge. Batch normalization takes the normalization technique a step further, and seeks to normalize the data within the network. This is to adjust the internal covariate shift, which refers to the change of the distribution in data between the layers. Layers constantly have to adjust themselves to a different distribution for each training step. As a result, this slows down the convergence. Equation 2.2, 2.3, 2.4, and 2.5 shows the math behind batch normalization.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.2}$$

Equation 2.2 is used to calculate the mean of the current batch.

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{2.3}$$

Equation 2.3 is used to calculate the variance of the current batch.

$$x_i = \frac{x_i - u_B}{\sqrt{o_B^2 + \epsilon}} \tag{2.4}$$

Using equation 2.2 and 2.3, the data is further normalized.

$$y_i = \sqrt{x_i} + \beta \tag{2.5}$$

Lastly as a final input, the data is shifted and scaled, this yields the output of the batch normalization operation.

Though batch normalization was created as a normalization technique to help speed up the training process and increase stability, it delivers additional unexpected positive effects in practice. Batch normalization, through testing, has proven to be a regularization technique, decreasing overfitting and improving generalization. In fact, it has proven to be a suitable replacement for widely used regularization techniques such as dropout. To date, there is no proof nor sound science to why batch normalization works well as a regularization technique. Regularization is further discussed in Section 2.5.5.

2.3 Skip Connections

Most feed-forward networks only contain connections between *layer n* and *layer (n + 1)*. Networks such as these present a hierarchical view of feature engineering. The networks learn the small features in the first set of layers, and as the model progresses and down samples the data, it seeks to learn the larger features of the input.

Skip connections introduce an iterative method of feature engineering, as opposed to the hierarchical method. Unlike most feed-forward networks, the networks using skip connection have an additional connection between *layer n* and *layer (n + r)*, where *r > 1*. Meaning that some layers output is not only sent to the next layer in the network, but also to another layer further ahead in the network.

Although hierarchical feature engineering has been very successful, it does suffer from some problems, especially when the network becomes deep. For example, learning the features of a complex image may require a deep network, while learning the features of a primitive image may not require a deep network. Regardless of the

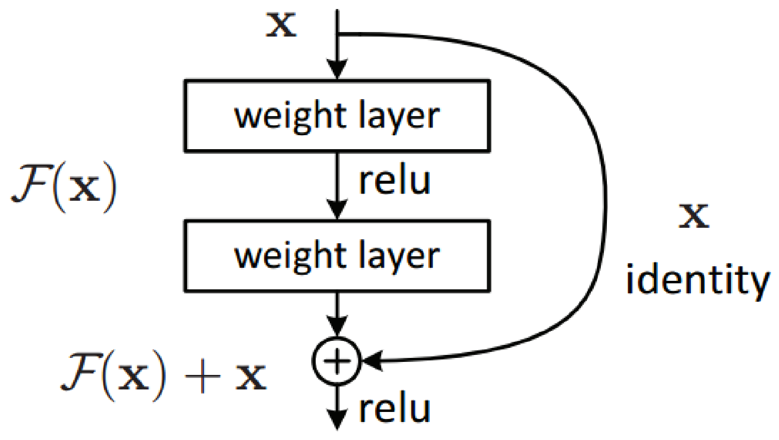


Figure 2.4: Skip connections used in ResNet (He et al., 2015).

input, the network is forced to apply the same level of abstraction to every input. By using skip connections, the network itself can decide how many layers are required to learn the features of the input. It may heavily abuse the skip connections if the input is simple, or utilize every regular layer for a complex input. This does not only grant the network far more flexibility, it also helps speed up the learning process.

There are different types of skip connections, and different architectures use skip connections to solve different problems. Notable successful networks that use skip connections are DenseNet (G. Huang et al., 2016), ResNet (He et al., 2015), and U-Net (Ronneberger et al., 2015).

U-Net utilizes long skip connections. Each block in the encoder has a reflective skip connection to the decoder as Figure 2.5 displays. Although U-Net has been successful when applied to image segmentation, the theory behind these skip connections is not entirely justified. The results are there, but the explanation is not. Thus, the skip connections is often what successors of U-Net seek to improve. This will be further explored in section 3.1.

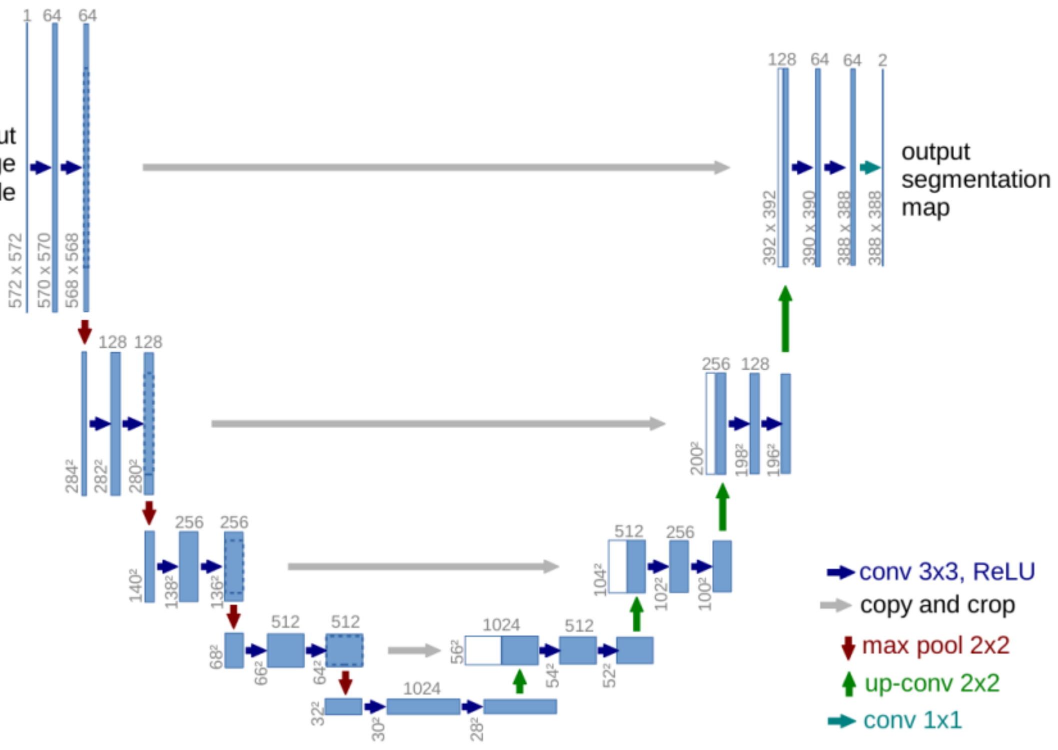


Figure 2.5: U-Net architecture using long skip connections (Ronneberger et al., 2015).

2.4 Dataset

There are two common methods of training a neural network. The model can interact with an environment and learn from the interactions, this is called reinforcement learning. Or the model can learn by getting fed data from a dataset. The latter is the method for this thesis.

The dataset is a set of samples. One sample can be one image, one record in a csv-file, one frame from a video, and so forth. Similarly to how humans learn, the model's abilities often scale with the amount of data it is given. To train the model and to properly evaluate the model, the dataset is split into three different parts: the training set, validation set, and testing set. Each part of the dataset has different responsibilities.

- Though the size of the **training set** may vary, it is always the largest of the three sets. Usually the training set contains 60-80% of the total dataset. The training set is the set which the model adjusts its parameters to, the data that the model is actually learning from.
- The model does not train on the **validation set**. The validation set is used by the developer to see how well the model is generalizing to data it has not *seen*. The developer may adjust hyperparameters to further increase the model's performance on the validation set. The validation set usually contains 10-20% of the the total data from the dataset.
- The **test set** is the final test for the model. When the model is measuring its performance on the test set, neither the model nor the developer should tune the model to further increase the performance. The performance on the test set should be completely unbiased. If the model performs poorly on the test set compared to the validation set, the developer should rather look to change the architecture of the model and reevaluate the current solution, rather than adjusting hyperparameters. Like the validation set, the test set usually contains 10-20% of the total data from the dataset.

2.4.1 K-fold Cross-Validation

To further determine how well the model performs, cross-validation can be applied. When using cross-validation the data is split into two sets; the training set and the test set. The validation set for cross-validation is part of the training set. For example, when using 5-fold cross-validation, we split the training set into 5 equally large sets. Then, we train the model on 4 of these sets, and perform validation on the last set. We then evaluate how well the model performed on the training set and how well it generalized to the temporary validation set. We then repeat these steps, but exchange the current validation set with a different chunk of the training set. It is important that the model does not continue to train from the previously learned data, as the model is currently trained on the new validation set. This would cause severe overfitting and not display how well the model generalizes. Therefore, the model has to be completely reset for the next training split.

2.4.2 Batch Size

The batch size decides how many training samples are used each time the model updates its parameters. For example, using batch size 32, the model will propagate 32 training samples, then update the parameters from the given loss.

2.5 Training

To train a neural network we first need the input data and the target data. Feeding the data to the neural network through forward pass, yields the output data. The loss of the model can then be calculated by comparing target data and the output data. Lastly, all the weights of the model are then updated given the loss. The last step is where the training process becomes difficult. To update all the parameters given the loss value, and finding out how "responsible" each parameter is for the loss. This can be answered by applying gradient descent together with backpropagation. Although there are several methods of training a neural network, gradient descent is by far most widely used.

2.5.1 Gradient Descent

"Suppose you are lost in the mountain in a dense fog; you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient of the error function with regards to the parameter vector 0, and it goes in the direction of the descending gradient. Once the gradient is zero, you have reached a minimum!" - Chollet (2021).

Gradient descent is an optimization technique used when training neural networks. The goal of gradient descent is to minimize the loss function by iteratively adjusting the parameters. This is done by taking the partial derivative of the loss function with respect to each parameter, while treating the other parameters as constants. Each partial derivative produces the gradient of the parameter, and all the gradients together yields the gradient vector. The gradient shows which way the parameter should be adjusted to decrease the loss function. Figure 2.6 shows a convex loss function in which the function only has one minimum.

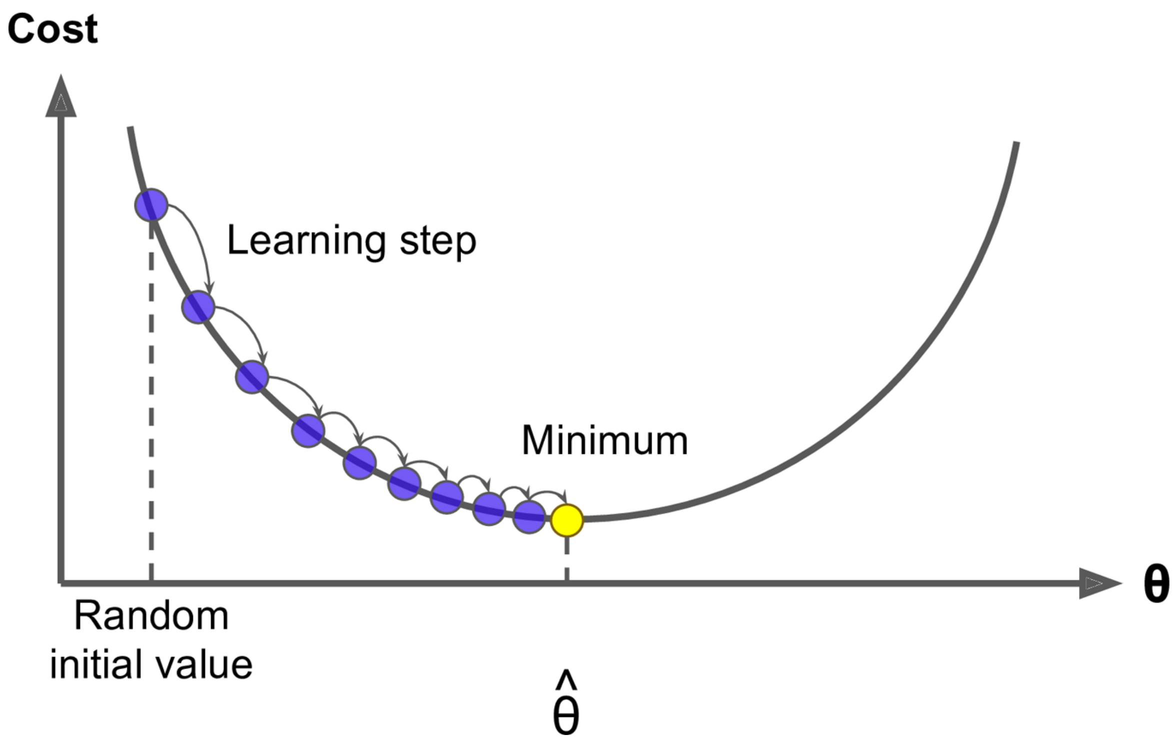


Figure 2.6: Gradient descent valley (Ronneberger et al., 2015).

If we are on the left side of the minimum, we should move to the right by increasing the parameter. If we are on the right of the minimum, we should move to the left by decreasing the parameter.

There are three different gradient descent methods; batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.

Batch gradient descent updates the parameters of the model by measuring the gradients for each sample in the training set. This assures that the optimization is always moving in the downwards (Figure 2.6). However, this also makes batch gradient descent prone to getting stuck in a local minima when the loss function is not convex. Also, it is also time consuming to calculate the gradient for each sample.

Stochastic gradient descent (SGD) takes one batch at random for each training step and calculates the gradients. This makes the training process much more unstable, making the learning step very unpredictable; but overall the model's performance should increase. Due to the nature of SGD, the model will never fully converge, but rather diverge and bounce around the bottom of the loss function. SGD is naturally much faster than batch gradient descent, and is better at escaping local minimas. Therefore, SGD is often the number one choice of gradient descent methods.

Mini-batch gradient descent is the middle ground between batch gradient descent and SGD. It takes part of the training data and calculates the gradients. When it comes to gradient descent methods, mini-batch gradient descent is the jack of all trades, master of none.

Calculating the gradient vector is done by utilizing the chain rule; this is **backpropagation**.

2.5.2 Learning Rate

The learning rate hyperparameter decides the length of each step when adjusting the gradient. Adjusting the learning rate has several implications on the gradient descent algorithm. For example, if the step size is too steep, the algorithm may jump over the minima repeatedly back and forth, causing the optimization to diverge rather than converge at the minimum. But if the step size is too small, convergence might be very slow. Deciding the value of the learning rate can be challenging and dependent on the architecture of the model, the loss function, and the task at hand.

Unlike the convex loss function in Figure 2.6, other more complicated loss functions may contain local minima in addition to the global minimum. This introduces additional challenges, as the gradient descent optimization may get stuck in a local minima and never find the global minimum. To counter this, momentum is often added to the learning rate. Meaning that the learning rate decreases for each iteration of the training process. The earliest epochs of the training process often have the largest impact on the parameter values. By having a relatively large learning rate early in the training process, we can approach the valley of the global minimum and escape the local minima. And by gradually reducing the learning rate, the global minimum will be reached and not diverged by jumping over the valley due to a high learning rate.

2.5.3 Adam

Kingma & Ba (2014) introduces Adam (Adaptive Moment Estimation) as an algorithm for first-order gradient-based optimization of stochastic object functions, based on adaptive estimates of lower-order moments. Adam is a widely used gradient descent optimization technique which is inspired by RMSprop and AdaGrade. Adam takes the best from both RMSprop and AdaGrade and combines it into one of the most used gradient based optimization algorithms.

AdaGrade adds momentum to the learning rate based on the output of the gradients.

RMSprop builds further on AdaGrade seeking to improve the optimization. RMSprop updates the learning rate based on the average of recent magnitudes of the gradients. This makes RMSprop robust against noisy problems.

Adam calculates the exponential moving average of the gradient and the squared gradient, while applying beta hyperparameters to control the decay rates of the moving averages.

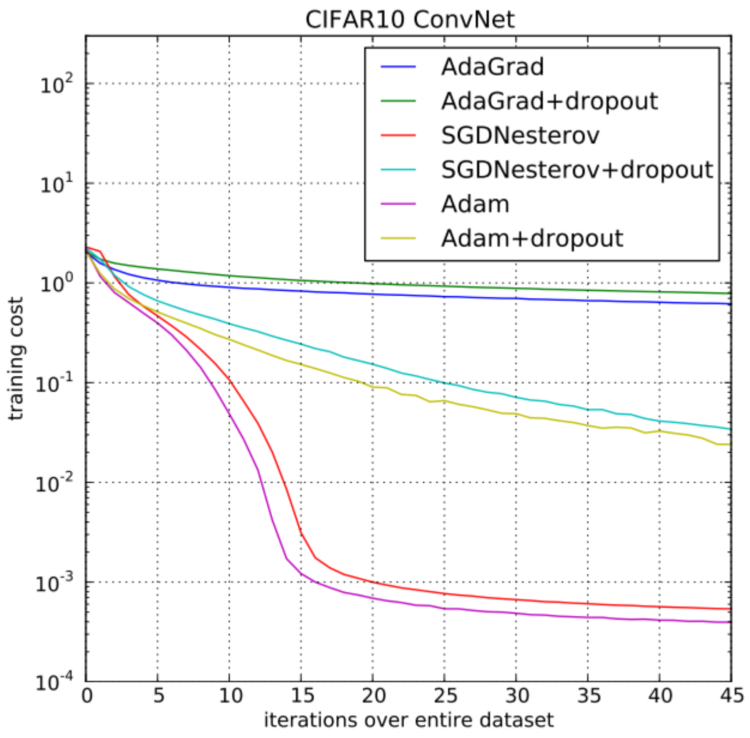


Figure 2.7: Training cost on CIFAR10 CNN comparison between AdaGrade, SGDNestrov, and Adam (Kingma & Ba, 2014).

Kingma & Ba (2014) compares Adam, SGDNesterov, and AdaGrade in Figure 2.7. They apply the optimization techniques to a CNN model, both with and without dropout (Section 2.5.5). The figure displays the loss on the training set, which does not properly display the generalization given the optimization technique.

2.5.4 Epoch

One epoch corresponds to the model being trained on the entire training set once. Unless the goal is to create a zero-shot learning model, the model should be trained on the training set for many epochs. When using a low number of epochs, the model may not be trained well enough to learn the training set, let alone generalize to the validation set. If both the training and validation error is high, the model is *underfitting*. Therefore, the model should be trained until the training error begins to stabilize. Stabilization in the training error indicates that the model has reached its limit given the training dataset, and to train it further may only increase *overfitting*. Overfitting occurs when the model works well on the training set, but struggles to generalize to the validation set. This can be seen if the training error is low, while the validation error is high.

2.5.5 Regularization

To further reduce overfitting, regularization techniques are applied to the model. The goal of regularization techniques is to decrease overfitting and increase generalization. The training error may be higher, but as a reprisal the validation error decreases. U-Net uses dropout as one of their major regularization techniques. Dropout, as the name implies, drops randomly selected neurons during forward pass. In other words, the neurons are deactivated and do not further send any data to the neuron in the next layer. The amount of neurons which are dropped is dependent on the dropout rate. A dropout rate of 0.5 would result in half the neurons getting dropped for the layer. Dropout is usually applied to several layers in the network. In theory, dropout should result in the model being less dependent on certain neurons, and rather seek to make every neuron important. Each neuron should be responsible for achieving a good result. Dropout is only enabled during training, and is disabled during validation and testing. Meaning that the model uses all the neurons during validation and testing.

Chapter 3

Related Work

The related work chapter contains relevant models and methods to the project. As well as state of the part models for image segmentation.

3.1 U-Net Architectures

This section covers the initial U-Net architecture and architectures which further expands on U-Net. For our project we use the standard U-Net architecture, but we do derive some relevant hyperparameter values from the predecessors of U-Net. As well as some minor design choices they made to further improve U-Net.

3.1.1 U-Net

Ronneberger et al. (2015) proposed U-Net as a solution for segmentation which required fewer annotated samples. The architecture is based on CNN, but follows an encoder-decoder structure with skip connections as seen in Figure 2.5. The encoder uses pooling as part of the downsampling, while upsampling in the decoder utilizes transpose convolution in favor of pooling. Each convolutional layer in both the encoder and decoder is followed by a ReLU activation function, while the final output layer applies the sigmoid activation function yielding each pixel a value between zero (0) and one (1). In their original paper, they experiment with their model on two-dimensional data and achieve state-of-the-art results while requiring fewer samples for training. However, the authors note that they are certain their model will be suitable for many other tasks. Indeed, U-Net has been a great inspiration for many models which are tailor-made for 3D segmentation.

3.1.2 UNet++

Using the original U-Net as a baseline, UNet++ seeks to improve some of the weaknesses and limitations in U-Net for image segmentation (Zhou et al., 2020). Their ultimate goal is to achieve higher accuracy for image segmentation. The authors state their findings which is that a deeper U-Net network is not necessarily better, and that the optimal architecture depends on the size and difficulty of the dataset. Therefore, UNet++ seeks to create one architecture which yields the optimal architecture regardless of the task at hand.

The authors arrive at the UNet++ architecture through an iterative process. First, they create an architecture named U-Net^e. U-Net^e is an ensemble architecture which combines U-Nets of varying depths into one unified structure. Each U-Net within this architecture partially shares the encoder, but has its own decoder. Further they introduce UNet+, which removes the original skip connections and instead connects every two adjacent nodes in the ensemble. Each node represents one convolutional block, which consists of upsampling/downsampling, an activation function, and batch normalization. Lastly, building on the success of DenseNet (G. Huang et al., 2016), they introduce dense connections in UNet+. This change yields their final architecture, UNet++.

As a result, UNet++ is more flexible than U-Net, while also outperforming its predecessor for both 2D and 3D segmentation. However, UNet++ does have an increased amount of parameters compared to U-Net. UNet++ requires 9.0M while U-Net requires 7.8M.

3.1.3 UNet 3+

As UNet++ is built upon U-Net, UNet 3+ is built upon UNet++ (H. Huang et al., 2020). The authors state that UNet++ does not explore sufficient information from full scales and there is still a large room for improvement. Exploring the full scale of the input involves explicitly learning the position and boundary of the object. This can be especially useful for medical imaging, where the images can contain organs of varying sizes. In addition to increasing the performance and accuracy, they seek to reduce the required amount of parameters.

Yet again, the change lies within the skip connections. UNet 3+ uses full-scale skip connections compared to the densely and nested skip connections used in UNet++. Figure 3.1 illustrates the comparison between the different architectures and their skip connections.

As a result, UNet 3+ is able to achieve higher accuracy using the dice metric while simuntionaly requiring fewer parameters. However, the accuracy is measured on a dataset which is more fit for the UNet 3+ architecture. The dataset contains livers and spleens of varying sizes given the image.

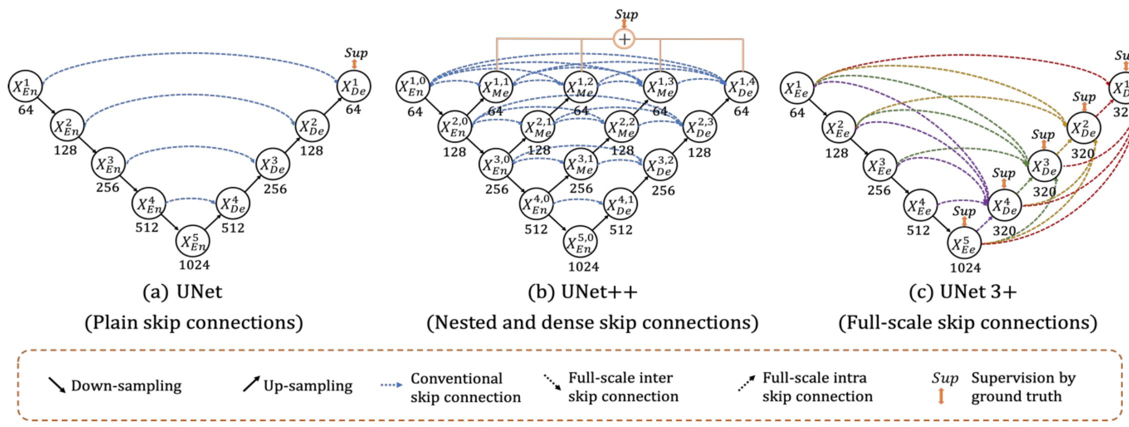

(a) UNet
(Plain skip connections)
(b) UNet++
(Nested and dense skip connections)
(c) UNet 3+
(Full-scale skip connections)
Down-sampling
Up-sampling
Conventional skip connection
Full-scale inter skip connection
Full-scale intra skip connection
Sup Supervision by ground truth

Figure 3.1: U-Net, UNet++, and UNet 3+ comparison

3.2 Neural Architecture Search

NAS (Neural Architecture Search) is a technique used for automatically creating artificial neural networks. Similiar to evolutionary algorithms, NAS can be used for hyperparameter optimization. NAS consists of a search space, search strategy, and a performance estimation strategy. NAS algorithms broadly falls under three different categories: Reinforcement learning (RL) based NAS algorithms, gradient-based NAS algorithms, and Evolutionary computation (EC) based NAS algorithms (ENAS) (Y. Liu et al., 2020).

3.2.1 NAS-Unet

Weng et al. (2019) is the first attempt at applying NAS to medical image segmentation. NAS-Unet uses the UNet architecture as the backbone of the model, while applying NAS to find an optimized architecture which outperforms other variants of UNet for 3D semantic segmentation. In addition, they drastically reduce the amount of parameters compared to UNet.

The core idea behind NAS-Unet is to create two cell architectures, DownSC and UpSC, then apply NAS to find the optimized versions of these cells. DownSC is a block used to downscale the data in the encoder part of the UNet, while UpSC is a block used to upscale the data in the decoder part of the UNet. They select a set of primitive operations to the cells which requires their expertise. The authors state that the operations are chosen by the most popular and successful CNN architecture for image classification. In addition, they value no redundancy and less parameters in the operations. The former referring to each operation having some unique properties, while the latters goal is to reduce the amount of paramters compared to original UNet. Their search strategy is inspired by DARTS (H. Liu et al., 2018), which is a gradient-based NAS algorithm. They do adjust DARTS to accelerate the search process by using Binary Gate, which updates only one architecture parameter by gradient descent at each step. Unlike DARTS, which updates all architecture parameters at each step.

NAS-Unet is applied to Promise12, Chaos, and NERVE datasets. The datasets consists of medical images taken with Magnetic Resonance Imaging (MRI), Computed Tomography (CT), and ultrasound (Weng et al., 2019). The authors conclude that NAS-Unet is able to outperform basline methods like U-Net and FC-Densenet, while simultaneously requiring less parameters.

Chapter 4

Methodology

The goal is to find the optimal learning rate, number of epochs, and batch size for the U-Net architecture. All these hyperparameters have a direct impact on the performance of the model without changing the architecture. To optimize the hyperparameters, we use genetic algorithms. This chapter contains the methodology use to optimize the hyperparameters, and the justification behind the algorithms and methods we use.

4.1 Genetic Algorithm

Genetic algorithms (GA) are based on the theory of evolution. Though evolution in real life is complex with many different parts, GA is still able to draw out the essence of evolution and produce good results. GA consists of a population and a population consists of a number of individuals (also called solutions). Each individual consists of genetics. For our task, the genetics are the hyperparameters; learning rate, number of epochs, and batch size. The individuals are then tested in the environment. Their genetic, or hyperparameters, are applied to the model and the model produces a loss. The individuals are then granted fitness based on the loss, higher loss yields lower fitness, and vice versa. After each individual in the population is tested and has received their fitness, this generation is complete. The last step is for the current population of the next generation. To do so, parents are selected based on their fitness to reproduce, higher fitness yields higher chance of becoming a parent; survival of the fittest. The next generation's population is then repopulated by the set of parents, and the process begins anew. Optimally, the GA will converge toward a global optimum where most individuals consist of the optimal hyperparameter values for the model.

4.1.1 Defining the Search Space

We have defined the search space of the GA to consist of the learning rate, the number of epochs, and the batch size. However, we still have to define and justify the range of the hyperparameters. The range could be somewhat arbitrary following the constraints of the hyperparameter itself. For example, the learning rate could be limited between 0 and 1. Though such a high learning rate is never used, it is somewhat pointless to include it. Rather, we should clearly define a productive range for all the hyperparameters so that the GA may converge faster.

The **learning rate** is constrained between 0 and 1. Ronneberger et al. (2015) does not state which exact learning rate they use. However, looking back at related work that have based their models on the U-Net architecture, the learning rate is frequently low. For example, Zhou et al. (2020) uses a learning rate of 3e-4 for the UNet++ model. H. Liu et al. (2018) dives deeper into learning rate as a standalone hyperparameter. They test different learning rate optimizers and different initial learning rates. In all their experiments, the learning rate is <0.1. For our experiment we want to test out the viable learning rates, but we also want to reiterate exactly why higher learning rates (>0.1) are not seen as frequently. We want to use the GA to prove that the learning rate will converge towards a lower value, and in addition find the optimal value. Therefore, we set the limit for the learning rate to be between 0 and 0.35.

The learning rate also has a close relationship with the loss function. When the loss function is convex, we don't have to worry about getting stuck in a local minima. When there is only one minimum, which is the global minimum, the optimization algorithm may favor lower learning rates as it converges towards the global minimum regardless. However, if the loss function is multimodal, the opposite applies. Multimodal loss functions contain one or more local minimas, which may encourage initial high learning rates to escape the valleys containing the local minima. For our tests, we will use a multimodal loss function as will be discussed in Section 4.2.

Batch Size Deciding the range of the batch size is somewhat trivial. For the range of batch size, we are more interested in seeing the result which the GA converges to for the batch size rather than defining the range. The norm in deep learning is to define the batch size to be a number which is the power of two. We also account for high batch size requiring more memory. Therefore, we set the range to be between 1-32.

Number of Epochs If evaluating from the training error, the GA would likely converge towards the highest possible epochs. However, we evaluate the loss, or fitness, from the validation error. Due to evaluating on the validation set, there does not have to be an upper limit on the number of epochs. But the limitation of testing each model for every individual in the population over multiple generations requires the upper limit to be relatively low compared to how many epochs a model is commonly trained for. To reiterate, for each individual in the population the model has to be trained for n epochs. As we use a population size of 100, this boils down to training 100 models for n epochs. And after all that is done, only 1 generation is complete. To reach convergence, far more generations are required. However, as will be discussed later, convergence can be reached relatively early compared to other GA tasks due to the nature of our task. Therefore, we limit the epoch range between 1 and 30.

Reiterating the hyperparameter search space:

- **Learning rate:** $0 < lr < 0.35$

- **Batch size:** $1 < B < 32$
- **Epochs:** $1 < e < 30$

4.1.2 Selection

Further we have to select the individuals which will be used to repopulate the next generation. A naive approach to selection, is to take the top performing individuals, and iterate through them to generate the next generation. However, an approach like this excludes the lesser performing individuals. This is not necessarily always a good thing. Lesser performing individuals can contain genetics that would be a great fit when combined with the better performing individuals. Therefore, want a selection method that favors the best performing individuals, but does not entirely exclude the lesser performing individuals. Two common methods, which both fulfill this goal, is roulette wheel selection and tournament selection.

Roulette wheel selection operates as a spin-the-wheel approach. Each individual is granted a chunk of the wheel based on their fitness. Better fitness yields a larger chunk of the wheel. A fixed location is appointed to the wheel, then the wheel is spun. The fixed point when choosing the first parent as the wheel stops spinning. The process is repeated until enough parents are chosen. An individual can be a parent multiple times in a row. The roulette wheel selection favors the individuals with the greatest fitness, while not excluding the individuals with less fitness.

Tournament selection selects a set of individuals from the population at random. The set then competes in a "tournament" and the winner of the tournament is selected to be a parent. The outcome of the tournament is in a sense predetermined, as the winner of the tournament is the individual with the best fitness. There are no new tests applied to the tournament participants. Tournament selection has some built in exclusion for the very worst performing individuals. Say the population size is 100, and the tournament size is 5. With these given values, the 4 lowest performing individuals can never be selected as parents. The population does not contain a tournament set where these 4 are able to win. Figure 4.1 shows an example of tournament selection.

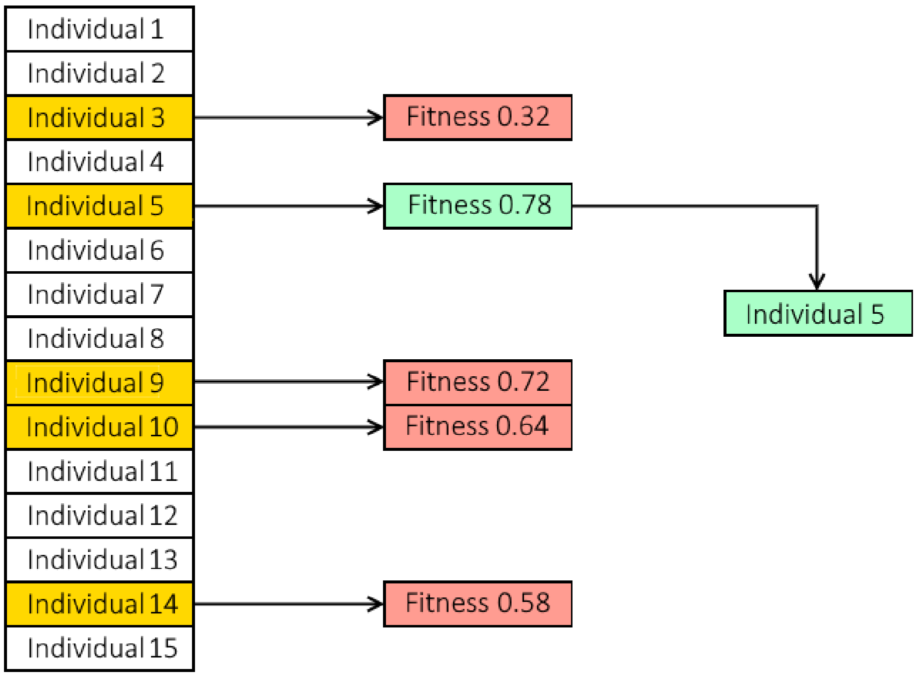
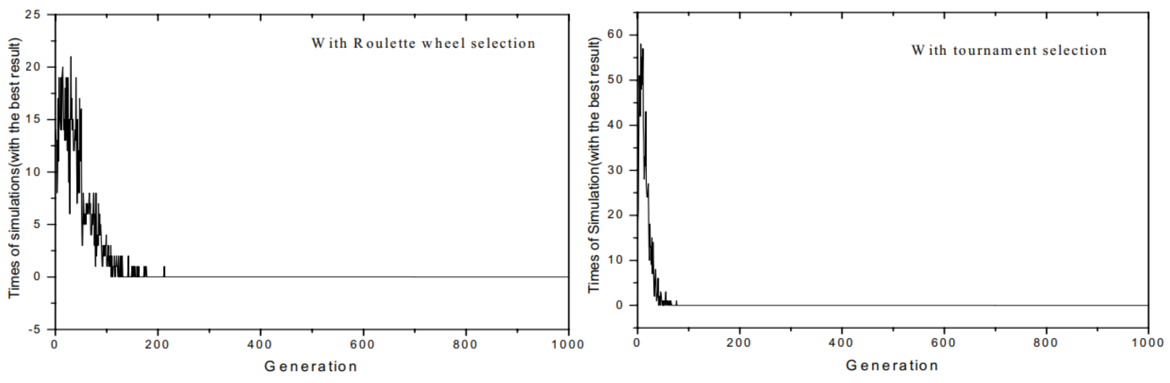


Figure 4.1: A population containing 15 individuals where tournament selection is applied. At random, 5 individuals are randomly chosen to compete in the tournament. The winner of the tournament is based on the predetermined fitness of the individual. In this case, individual 5 wins the tournament and is selected to be a parent.

Zhong et al. (2005) compares the performance between roulette wheel selection and tournament selection. They test the selection methods on simple mathematical functions with different hyperparameters (mutation rate, population size).

In their experiment using function $f(x1, x2) = x^2 + x^2$, population size 100, mutation rate 15%, tournament selection is able to slightly outperform roulette wheel selection. After 100 generations, the algorithm with tournament selection has all worked out the satisfied solution in the 1000 trials, but the one using roulette wheel selection only hits 952 times on the satisfied solution (Figure 4.2). Further experiments with different functions yields the same result. Tournament selection is able to converge to the global optima for all individuals in fewer generations than roulette wheel selection.



(a) Roulette wheel selection. (b) Tournament selection.

Figure 4.2: Comparison between roulette wheel selection and tournament selection (Zhong et al., 2005).

Both roulette and wheel selection and tournament selection fulfill our need, we use tournament selection due to the results derived from Zhong et al. experiments.

As well as selection, we apply *elitism* to our GA. Elitism is a method used to ensure that the very best individuals pass their genetics to the next generation. Instead of only being able creating offspring, the selected elites automatically pass to the next generation. The individuals can still become parents by the selection algorithm. Applying elitism slightly reduces the randomness and helps the GA converge. In some shorts tests, we try different elite sizes, but end up using 5 as the elite value. Meaning the 5 best individuals in the population are part of the next generations population as well.

4.1.3 Crossover

After applying for tournament selection and finding the set of parents, the selected parents have to reproduce and create new individuals. The new individual consists of a mixture of the parents genes. There are different methods of crossover, most commonly one-point crossover and uniform crossover. Figure 4.3 shows one point crossover, where a random point in the genes are chosen, the offspring then consists of the first half from one parent, and the second half from the other parent. Figure 4.4 shows uniform crossover, where each gene is chosen 50-50% between the parents. For our task we use uniform crossover.

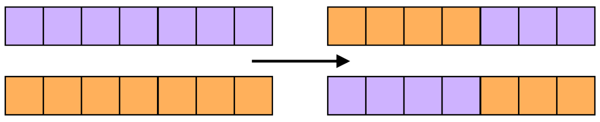


Figure 4.3: One-point crossover.

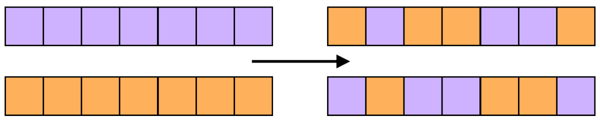


Figure 4.4: Uniform crossover.

When performing crossover, we can either take the genes as their values, or cast the values to binary representations. For example, the batch size gene can either be one number between 1 and 32, or it can be the binary representation between 1 and 32. When expressing the gene as only one digit, the crossover makes the offspring inherit either the complete gene from one parent or the other. However, while representing the gene as a binary value, the offspring can inherit the mixture of the gene from both parents. This mixture could represent an entirely new value, which would increase the *exploration* in the genetic algorithm.

Exploration is a tradeoff between *exploitation*. Exploration seeks to explore new possible solutions in the search space, while exploitation seeks to exploit current successful solutions. As with any optimization technique, there has to be a balance between exploration and exploitation. Too much exploration and the GA diverges and keeps exploring a set of seemingly infinite values. Too much exploitation and the GA converges to a local minima, unable to find the best possible individuals and converge to the global optima.

The selection process is largely responsible for the exploitation of the GA. Where the selection process, with some randomness, picks out the best individuals in the population and seeks to further exploit their genetics. The mutation process is largely responsible for the exploration part of the GA, as it seeks to add new genes to the population. Crossover, however, is between the middle of exploration and exploitation. The crossover process is given to the successful parents, but it has the possibility to change the sequence of the different genes. Every combination of the genes are part of the search space, and the crossover process helps explore the different combinations systematically. When using a binary representation, even more exploration is added to the crossover process, as it now has the ability to alter the values of the individual genes.

For our project, we mainly focus on using decimal values for the genes. However, we do experiment with binary values to see the difference between the results in Section 5.3.

4.1.4 Mutation

When a new individual is created, there is a chance that the individual is mutated. Mutation alters the genes of the individual. Mutation can happen to one gene or multiple genes by altering the value of the gene. Alternation causes the gene to be randomized within the constraints of the gene. By applying mutation new genes are introduced to the population. If these genes are not good, meaning that they score a low fitness, the mutated individual carrying the new genes will quickly be eliminated from the population. However, if the genes are good, the GA will further exploit the newfound genes and reproduce the individuals

Since mutation is responsible for the exploitation, choosing the chance of an individual being mutated, or the mutation rate, is the value which is most crucial for the balance.

Doerr et al. (2017) explores different mutation rates for optimizing multimodal functions. In their research they state that when using uniform crossover with a bit-string representation of size n, the recommended mutation rate is $p_n = 1/n$. For the bit-string representation of our hyperparameters this would resolve in $n = \{x, y, z\}$. Doerr et al. (2017) argues that the $1/n$ recommendation could result in overfitting and convergence to the local optima. In their experiments they observe that a mutation rate of $2/n$ or higher leads to higher exploration and a faster convergence. However, their results are heavily reliant on their task as well and may not transfer well to our task. Therefore, we experiment further with the mutation rate ourselves to find the best mutation rate for our project. Acknowledging the results from Doerr et al. (2017), we set the potential mutation rates to be $p_n = \{15, 20, 25, 30, 35, 40\}$. To test the different mutation rates, we initialize the GA as usual, and observe how the GA converges given the different mutation rates. However, due to these tests being time-consuming, we use half the data in this experiment. All other variables which are not part of the genetic code are kept static and the training progress is made equal for all the tests. Each of the tests are given the exact same training data and validation data.

4.2 Evaluating the Model

This section goes through the dataset being used and how each model is trained. The model is evaluated by a loss function, and the loss function is the fitness of each individual in the population.

4.2.1 Dataset

We apply our solution to the ultrasound nerve dataset. As with most medical imaging datasets, the data is imbalanced. Figure 4.5 displays a record from the dataset, one for the ultrasound image of the neck, and one for the manually annotated segmentation of the nerve. As seen, the nerve annotation is small when compared to the full image. Some images do not contain a nerve at all. The dataset contains 5600 images where all the images are labeled. We use 560 images to train a new model with each individuals hyperparameters. And we use 100 images for the validation. We don't need to hold out any specific data for the test dataset, as we don't use the entire dataset.

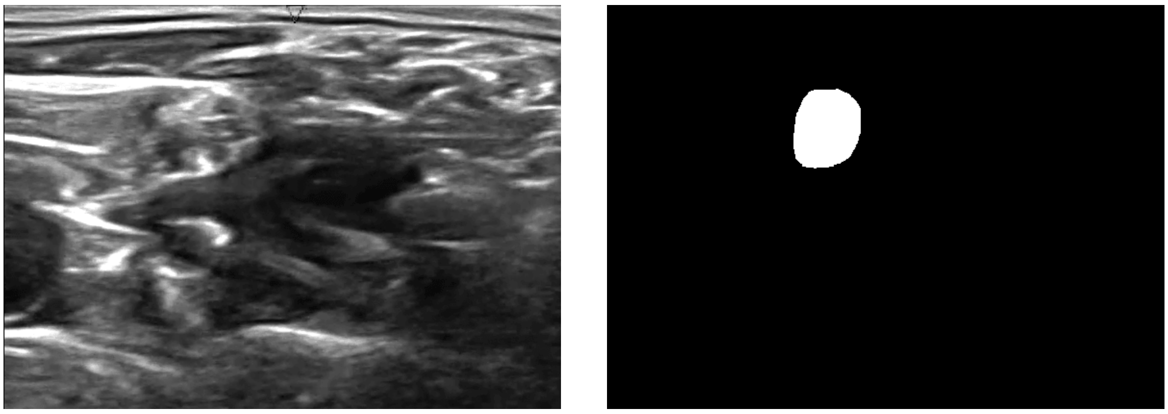

(a) Image. (b) Ground truth mask.

Figure 4.5: Ultrasound nerve image (Nerve, 2016)

4.2.2 Loss Function

When evaluating the fitness of an individual, we use the loss function. The loss function is used to determine the performance of the model. Higher loss grants lower fitness, and lower loss grants higher fitness.

Loss functions and metrics are used to evaluate the performance of a model. Loss functions are directly applied to the model through backpropagation, while metric is more of a human measure for the developer to see how well the model is doing. The choice of which loss function to use depends on a set of many different variables: what is the goal of the model, what is the architecture of the model, what type of data is the model operating with, how imbalanced the dataset is, and more.

When deciding which loss function to use, we have to account for the dataset. Given that we use the ultrasound nerve dataset, we have to pick a loss functions better fit for the imbalance of the data.

Ma (2020) explores different loss functions for segmentation by creating a taxonomy. In their research, they sort the loss functions into meaningful categories, while also distinguishing between the different types of loss functions for segmentation. Ma states in their research that there has not yet been a comprehensive empirical comparison of all the different segmentation loss functions. And therefore it is hard to identify the best loss function. However, they do give some recommendations based on his research. Ma distinguishes the different loss functions into 4 different types.

1. **Distribution-based loss** measures the distance between the ground truth and the prediction. Most commonly used for this category is cross-entropy. For a balanced dataset, distribution-based loss is recommended.

2. **Region-based loss** measures the overlapping regions between the ground truth and the prediction. For a mild imbalance in the dataset, region-based loss is recommended.

3. **Boundary-based loss** is a relatively new type of loss that seeks to minimize the distance between the ground truth and the prediction. As the name implies, the loss function is calculated as a function of the differentiated boundary. Boundary-loss is recommended in addition with other loss functions.

4. **Compound loss** is a combination of previous categories, for example the weighted sum between a distribution-based loss and region-based loss. A mixture of distribution-loss and region-loss, or region-loss and boundary-loss, is recommended for a highly imbalanced dataset.

Similarly to the taxonomy created by Ma, Jadon (2020) creates a survey of the different loss functions for segmentation. In their survey, they go through the different loss functions used for segmentation, and weighs their strengths and weaknesses. Jadon also concludes that there is no *“one fits all”* when it comes to loss functions for segmentation.

Going back to the ultrasound nerve dataset, a distribution-based loss function is not recommended. A distribution-based loss function, such as cross-entropy, would be heavily rewarded if it only predicted “not nerve” for all pixels in the image. The small loss it would receive when it encounters a nerve by using this method, would be insignificant when compared to the otherwise success.

Region-based loss is a good fit given that most medical imaging is imbalanced, while not always heavily imbalanced. We look closer at the dice loss, which is derived from intersection-over-union (IoU), also known as the Jaccard index (equation 4.1). According to both Ma and Jadon, this loss function is widely used for segmentation and works well with imbalanced data.

$$Jaccard(U, V) = \frac{|U \cap V|}{|U \cup V|} \tag{4.1}$$

IoU is a method of measuring the overlapping labels. It measures the overlapping true and false labels, then divides it by the union of the labels. As expected, this heavily punishes the model by falsely predicting a nerve in the wrong spot. But in addition, it also punishes the model if it were to only predict false labels (no nerve) for every input. Encouraging the model to actually find the nerves and not stall the learning as a distribution-based loss function would. However, this is one problem with IoU; it is not differentiable. For backpropagation to work, the loss function has to be differentiable. As a solution, the dice loss is a derivable loss function which is based on IoU.

$$DL(y, \hat{p}) = 1 - \frac{2y\hat{p} + 1}{y + \hat{p} + 1} \tag{4.2}$$

Here, y is the ground truth, and $\hat{p}$ is the predicted segmentation. The 1 is added to the numerator and denominator for edge case scenarios where $y = \hat{p} = 0$.

When deriving the results for this thesis, the dice loss is used when measuring the performance of one model. Dice loss function is multimodal, which may encourage higher learning rates so that it may escape local minimas. It's difficult to estimate whether or not a convex loss function would be better or not. However, as most segmentation loss functions are multimodal, it's good to consider non-convexity.

4.2.3 Optimizer

Each individual in the population will be evaluated with the same configurations. Zhou et al. (2020) to optimize their UNet++ model. Likewise, we will use Adam to optimize our model. Adam (Section 2.5.3) adds momentum to the learning rate, which results in the learning rate hyperparameter not being static.

4.3 Weight Initialization Method

Initialization of a neural network's weights is very important to assure the stability of the network. Neural networks often exhibit the quality of successively weaker layers, or successively stronger layers, especially when the depth of the network increases. This effect is also known as the vanishing and exploding gradient problem. When vanishing gradients occur, the gradients which are used to update the weights, become increasingly small, which stalls the learning process. When exploding gradients occur, the gradients become increasingly large, which makes the model unstable and the gradients can not make any reasonable updates.

There are many methods used to combat the vanishing and exploding gradients. Such as regularization techniques, using a non saturating activation function like ReLU, clipping gradients, weight initialization methods, and more. Historically for weight initialization, the weights were given a random value between a certain small interval such as $[-1, 1]$. Several different methods have now emerged which have proven to be effective for creating a stable model.

4.3.1 Choosing Weight Initialization Method

Boulila et al. (2021) goes through the different weight initialization methods and the current trends in deep learning. They look into the strengths and weaknesses of the different methods, and for which type of model they may be best suited. From their research they look into 5 different initialization methods; all-zeroes/constant, random, LeCun, Xavier, and He. The all-zeroes/constant and random initialization are somewhat primitive when compared to the latters. They are both simple and fall short when compared to LeCun, Xavier, and He. Further, they do an application case study, looking at which weight initialization method is used within the different domains. For segmentation, He initialization is often used. For other domains, Xavier is often used. Interestingly, LeCun is not often used within any domains.

Kumar (2017) goes deeper into the difference between He initialization and Xavier initialization. Given the experiments and the theoretical insight they propose, they show that a network with ReLU activation function does not converge with the Xavier initialization, but does converge with the He initialization. Therefore, recommending He initialization when using ReLU as opposed to Xavier initialization. We follow this recommendation, and choose Xavier initialization to be our weight initialization method.

4.3.2 Reducing Randomness

When going through the search space with the genetic algorithms, it is difficult to exactly assess how well an individual performed when accounting for the randomness. One individual may be given a set of initial weights which instantly yields a set of better weights than the other individuals in the generation. This does not point to the fact that the individual has a set of better hyperparameters, but rather that the individual got lucky with its initial weights. The individual is then given a high reward due to luck, and will likely pass on to the next generation, either through elitism or as a parent. However, the individual or predecessors are not able to replicate the same success from the previous generation, as the luck may have run out. This will cause stalling in the genetic algorithm, rewarding individuals for luck instead of the performance. It is costly for the genetic algorithm to get rid of a falsified individual, and the time would be better spent exploring new individuals, or expanding on successful individuals.

Therefore, to rid the algorithm with the element of luck, each time a new generation starts, a new set of initial weights is created. This set of weights is then applied to each individual in the population. This gives each individual an even playing field, assuring that one individual is not granted a randomly generated set of good weights, while one is given a randomly generated set of difficult weights. Though, this does not entirely ensure that there is no randomness to evaluating the performance of a individual. The set of weights may be more suited for one individual's hyperparameters compared to others. But by creating a new set of initial weights for each generation, the robustness of the individuals are tested across multiple generations, eventually yielding the best individuals across multiple different sets of initial weights.

4.4 Summary

We initialize the genetic algorithm with a population size of 100 individuals. Each individual holds the potential values for the hyperparameters; learning rate, batch size, epochs. Each individual is then tested on the nerve dataset, and then tested on the validation dataset. For every individual in the population, the individuals are given the same initial weights and the same data. The validation test yields a fitness based on the dice loss function. The fitness is used to determine the success of the individual, and the best individuals are selected to reproduce the next generation. The process for a number of generations, or until the GA converges, meaning that the performance of the population as whole stagnates.

Chapter 5

Results

This chapter goes through the results we derived from following our methodology. All results are evaluated on the models performance for segmentation on the nerve cell dataset.

5.1 Experimenting with Different Mutation Rates

To evaluate the optimal mutation rate, we run a set of short tests on a minor dataset. These tests consist of a training set of 100 images from the nerve dataset, and the epoch range is set between 0 and 1. Due to the low amount of data used, and the low number of epochs, we evaluate from the training loss, and not the validation loss. We run these tests to find the mutation rate we want to use and to see if there are any discrepancies between the different mutation rates. Table 5.1 shows the different range of mutation rates and the performance. The performance is calculated by the dice loss functions. Regardless of the mutation rate, there is no large different in the performance. The 15% mutation rate is slightly above the rest, this may be random. Regardless, we will use 15% mutation rate for our initial experiment.

Mutation Rate	Performance
5%	0.7274
10%	0.7273
15%	0.7465
20%	0.7278
25%	0.7289
30%	0.7366
35%	0.7274
40%	0.7276
45%	0.7272

Table 5.1: Table contains the best found individual after 50 generations given the different mutation rates.

5.2 Experiment with Decimal Gene Representation

For our initial main experiment we run the GA on part of the nerve cell dataset. The never cell dataset contains 5600 images, we train the model with each individual's hyperparameters on 560 images. Each individual is granted a fitness based on the dice loss function. We run the GA for a total of 50 generations. Depending on the task, running GA for only 50 generations can be low, but we show that the GA converges for 50 generations. We have also ensured that the dataset is somewhat balanced – our partial dataset does not contain only non-never images.

Table 5.2 contains the information regarding the generation's best individual and their hyperparameters. Note that the table can be somewhat misleading, as there is often not *one* best individual, but often a set of individuals that achieve the same fitness of *0.7858*. The very first epoch is not very insightful, as the hyperparameters are completely random at this point. Regardless of the hyperparameters, the model is still tailored for the task at hand, and the best

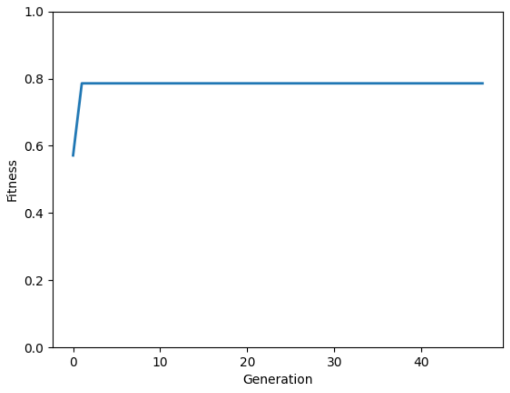
Hyperparameter	Generation					
	1	10	20	30	40	50
Learning rate	0.0697	0.2238	0.2036	0.1563	0.2036	0.2036
Batch size	2	1	1	1	1	1
Epochs	9	27	15	21	11	11
Fitness	0.5714	0.7858	0.7858	0.7858	0.7858	0.7858

Table 5.2: The hyperparameter values of the best performing individual in the population across different generations. There may be multiple individuals which have reached the same highest fitness.

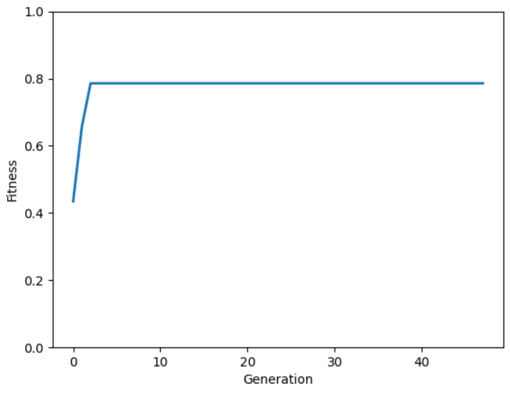
individual is able to reach relatively high fitness. As seen in Figure 5.1, early in the GA the best individual(s) hit a ceiling for their fitness, unable to push past 0.78 fitness. This may be due to the nature of the dataset and the U-Net model – some nerves are difficult for the model to segment.

It may also be a clear indicator that the GA converges to a minima, specifically the valley of 0.78 fitness. Whether or not this is the local minima or global minima is difficult to assess. Given the architecture of the model, does there exist a set hyperparameter that is able to reach above 0.78 fitness? If yes, the GA is stuck in a local minima. If not, the GA is stuck in a 0.78 minima, which is optimal. To further explore the potential, we increase the exploration in Section 5.3.

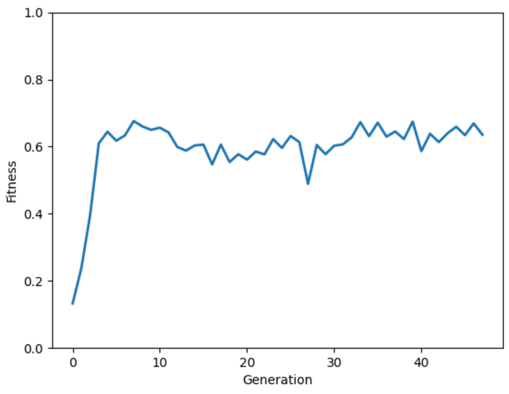
We can further evaluate the model by inspecting the predicted segmentation. For the manual inspection, we use the hyperparameters of the top individual from Table 5.2. Figure 5.2 and 5.3 shows 3 images; the input, the hand annotated nerve cell, and the predicted segmentation. For Figure 5.2, the model is able to assess the locations of many nerve cells by just training for 11 epochs. However, the shape is often not entirely correct. Also, the model seems to be predicting nerve cells in images that contain no nerve cells. Figure 5.3 shows some of the better results by our solution.



(a) The best fitness achieved by an individual.



(b) The average fitness of the elites (top 5 individuals).



(c) Average fitness across the entire population.

Figure 5.1: Displays the fitness of different categories of individuals for each generation.

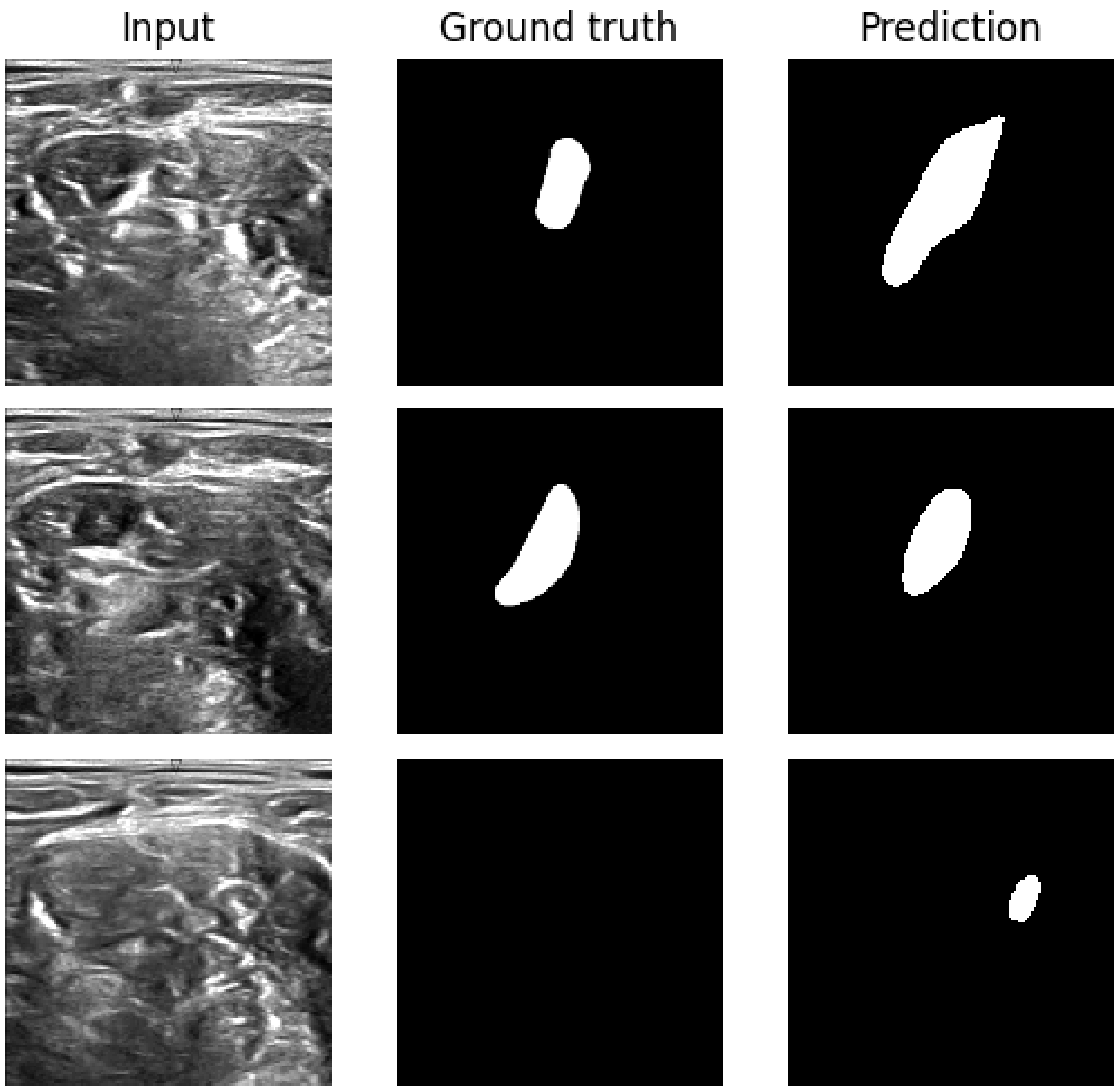


Figure 5.2: Ultrasound images of the neck as input on the left side, center shows the hand annotated nerve, right side shows the prediction by the model. These are some of the worse results.

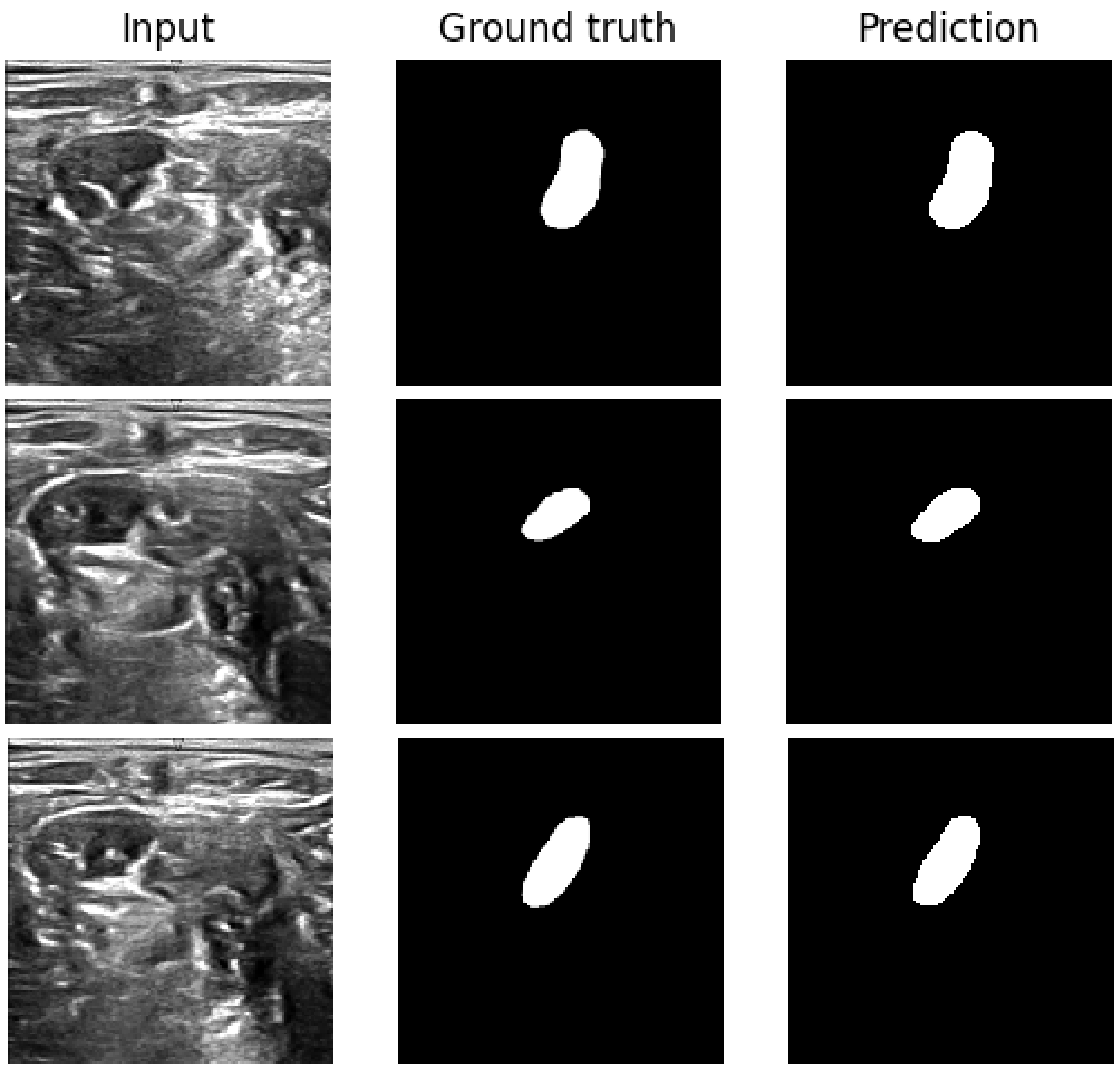


Figure 5.3: Ultrasound images of the neck as input on the left side, center shows the hand annotated nerve, right side shows the prediction by the model. These are some of the best results.

5.3 Experiment with Binary Gene Representation

The decimal gene representation shows signs of the individuals converging to a local minima. There are no individuals who are able to pass the local minima; 0.78 fitness. Also, many individuals are able to reach this fitness per generation towards the final generations. Therefore, we have to increase the exploration of the GA, and lessen the exploitation. Our goal with this experiment is to outperform the decimal gene representation experiment.

To increase the exploration, we slightly increase the mutation rate. A higher mutation rate is a direct addition to exploration. As well as higher mutation rate, the binary gene representation acts as a great exploration technique. Allowing the individuals to receive newfound genes through the crossover part of the GA. To decrease exploitation, we reduce the elite size from 5 to 1. Only 1 individual will automatically carry on to the next generation as opposed to 5. This will add more diversity to the population.

Hyperparameter	Generation					
	1	10	20	30	40	50
Learning rate	0.1815	0.0508	0.1295	0.0478	0.2446	0.2894
Batch size	1	1	1	1	1	1
Epochs	30	28	14	12	9	18
Fitness	0.7858	0.7858	0.7858	0.7858	0.7858	0.7858

Table 5.3: The hyperparameter values of the best performing individual in the population across different generations. There may be multiple individuals which have reached the same highest fitness.

Table 5.3 shows the fitness binary gene representation. As with the decimal gene representation, the GA is unable to escape the 0.78 ceiling. The far higher weight on exploration did not help it escape. Though it is not possible to thoroughly conclude without testing every possible configuration of the hyperparameters, there might not exist a configuration which yields a higher fitness than 0.78 with the given architecture of the model.

5.4 Batch Size

Interestingly, batch size seems to be the only hyperparameter which converges. For all individuals with a decent fitness, the batch size is set to 1. With regard to this, we do not have to account for the batch size when analyzing the different combinations of hyperparameters. However, as we defined the range of the batch size in Section 4.1.1, we mostly cared about the convergence. With only 1 in batch size, the gradients are updated for each image in the dataset. Since the gradients are not updated in batches, the runtime is increased. The time saved from having batches is redacted, but the performance increases as a result.

5.5 Epoch

The number of epochs varies. Many individuals are able to reach the minimum of 0.78 fitness, regardless of the number of epochs. One individual is able to reach 0.78 fitness in only 9 epochs, the lowest amount required which has been seen for all individuals. There does not seem to be a close relationship between the number of epochs and the learning rate. Few epochs do not result in a higher learning rate, and many epcosh do not result in a lower learning rate. It would be expected that the number of epochs followed the same path as the batch size hyperparameter. That each individual wants to get as much as possible with no regard to time efficiency. However, the number of epochs seems to be far more irrelevant than the batch size for this task.

5.6 Learning Rate

Unlike the batch size, the learning rate does not fully converge towards a set value. However, compared to the traditional low learning rates seen in recent literature (Zhou et al., 2020), the learning rate converges towards a much higher value. Most individuals with good fitness hover around 0.2 as a learning rate. Though, this may be related to the number of epochs that the models limited to. Fewer epochs means that the model may want to have as much impact for each epoch.

5.7 Comparison

We compare our found best hyperparameters with the “standard” hyperparameters on 250 epochs to see how well our result scales. For our hyperparameters we use one of the top individuals which has a 0.2036 learning rate and 1 batch size. For the standard hyperparameters we use 0.0003 learning rate and batch size 8. We evaluate these results on the same exact data as previously tested. This is to further inspect if our current solutions are stuck in a local minima. Figure 5.4 shows the fitness curve of the same model with the different hyperparameters.

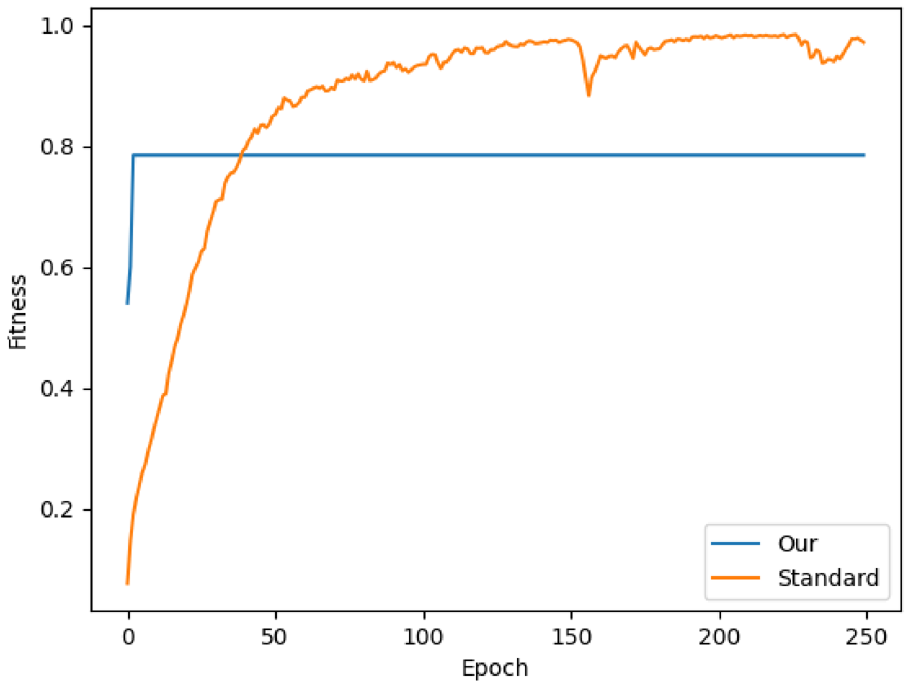


Figure 5.4: Comparison between the model with different hyperparameters.

The model with standard hyperparameter is able to catch up to our solution after approximately 40 epochs. This is out of our epoch range, as we set the maximum number of epochs to be 30. At 30 epochs, the standard version is still below ours in fitness. The comparison shows clearly that our solution is able to reach a good fitness in a small amount of epochs, but it is not able to reach the fitness of the standard version after many epochs. This comparison also reveals that the GA is either stuck in a local minima, or that using the hyperparameters as genes is not enough to reach the global minima with the given number of epochs.

5.8 Summary

We first ran a set of tests on a small part of the dataset to find the optimal mutation rate. However, regardless of the mutation rate, the GA converged to nearly the same fitness for the best individual in the population.

We ran the GA for 50 generations with decimal representation of the genes. The individuals hit the ceiling of 0.78 fitness in an early generation, and could not get past it in any later generation. To increase the exploration of the GA, we tried a binary representation of the genes and reduced the elite size from 5 to 1, but the results were the same. None could get past 0.78 fitness.

Chapter 6

Discussion and Conclusion

In this chapter, we discuss the project and add our conclusion. Discussion is related to challenges around the project.

6.1 Limitations

Running the GA for 50 generations requires approximately 5 GPU days with a NVIDIA Tesla V100 SXM2 GPU . We also only use part of the nerve dataset, and not the entire dataset. Using the entire dataset would require 50 GPU days. In addition to the runtime, any microchange would render the current result irrelevant, meaning we would have to run the GA over again with the new changes. Therefore, most of the final tests were done towards the last month of the thesis.

The original goal of the thesis was to explore both 2D data and 3D data. Towards the beginning of the thesis we did some experiments with 3D data on a brain tumor dataset. However, since 3D data contains even more information than 2D data, the runtime becomes too large. Therefore, we only run the GA on 2D data.

With more resources the code could be run on more data and more epochs, potentially deriving better and more insightful results.

6.2 Reflection

Looking back at the thesis as a whole there are some things I would have done differently. Some minor changes could result in better results. For example, in Section 4.2.2 I go through the different loss functions and justify the choice of using Dice Loss. At this point I was still uncertain which dataset would be primarily used, and whether I would test on one or more datasets. Therefore, I chose using Dice Loss which performs best on imbalanced data, as I knew the data would likely be medical imaging. However, the Nerve Cell Dataset which we run our tests on, is heavily imbalanced. A compound loss function would be a better fit for this dataset, as encouraged by Ma (2020).

As seen in the results, the models quickly converge after a few number of epochs. Perhaps the learning rate should be set to a far smaller range, to encourage the projectory seen by the standard curve in Figure 5.4. This way, the GA would not opt for the quick and potentially premature convergence. It would also force the GA to respect the epoch hyperparameter, as it was mostly irrelevant in the results.

6.3 Conclusion

We used genetic algorithms to optimize the learning rate, batch size, and the number of epochs hyperparameters. Our solution shows that a high learning rate and low batch size can cause good performance in a low number of epochs. Choosing a high learning rate, around 0.2 can be beneficial when training the model for a limited number of epochs. However, as the number of epochs increases, a much lower learning rate has better performance. We also explored different rates of mutation, but there was little difference.

References

Boulila, W., Driss, M., Al-Sarem, M., Saeed, F. & Krichen, M. (2021, 2). Weight initialization techniques for deep learning algorithms in remote sensing: Recent trends and future perspectives.

Chollet, F. (2021). *Deep learning with python, second edition*.

Doerr, B., Le, H. P., Makhmara, R. & Nguyen, T. D. (2017, 3). Fast genetic algorithms.

E-Helse. (2019, 12). Utredning om bruk av kunstig intelligens i helsesektoren.

He, K., Zhang, X., Ren, S. & Sun, J. (2015, 12). Deep residual learning for image recognition.

Huang, G., Liu, Z., van der Maaten, L. & Weinberger, K. Q. (2016, 8). Densely connected convolutional networks.

Huang, H., Lin, L., Tong, R., Hu, H., Zhang, Q., Iwamoto, Y., … Wu, J. (2020). Unet 3+: A full-scale connected unet for medical image segmentation. In (Vol. 2020-May). doi: 10.1109/ICASSP40776.2020.9053405

Ioffe, S. & Szegedy, C. (2015, 2). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Jadon, S. (2020, 6). A survey of loss functions for semantic segmentation. doi: 10.1109/CIBCB48159.2020.9277638

Kingma, D. P. & Ba, J. (2014, 12). Adam: A method for stochastic optimization.

Kumar, S. K. (2017, 4). On weight initialization in deep neural networks.

Lecun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*, 2278-2324. doi: 10.1109/5.726791

Liu, H., Simonyan, K. & Yang, Y. (2018, 6). Darts: Differentiable architecture search.

Liu, Y., Sun, Y., Xue, B., Zhang, M., Yen, G. G. & Tan, K. C. (2020, 8). A survey on evolutionary neural architecture search. doi: 10.1109/TNNLS.2021.3100554

Ma, J. (2020, 5). Segmentation loss odyssey.

Nerve. (2016, 2). https://www.kaggle.com/competitions/ultrasound-nerve-segmentation/data.

Ronneberger, O., Fischer, P. & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In (Vol. 9351). doi: 10.1007/978-3-319-24574-4_28

Weng, Y., Zhou, T., Li, Y. & Qiu, X. (2019). Nas-unet: Neural architecture search for medical image segmentation. *IEEE Access*, *7*, 44247-44257. doi: 10.1109/ACCESS.2019.2908991

Zhong, J., Hu, X., Zhang, J. & Gu, M. (2005). Comparison of performance between different selection strategies on simple genetic algorithms. In (p. 1115-1121). IEEE. doi: 10.1109/CIMCA.2005.1631619

Zhou, Z., Siddiquee, M. M. R., Tajbakhsh, N. & Liang, J. (2020). Unet++: Redesigning skip connections to exploit multiscale features in image segmentation. *IEEE Transactions on Medical Imaging*, *39*. doi: 10.1109/TMI.2019.2959609

Appendix A

Appendix

A.1 Code Snippets

This section contains some of the core code snippets used to execute our solution. The code snippets contains abstract methods, but the purpose of the methods should be evident.

A.1.1 Main Loop

The main loop which iterates through the selected number of generations.

```
def train(x_train, y_train, x_val, y_val, population):

    n_generation = 50

    for i in range(n_generation):
        for solution in population:
            model = u_net()

            train(model, x_train, y_train)
            loss = validate(model, x_test, y_test)

            solution.fitness = 1 - loss

        population = update_population(population)
```

Listing 1: Main loop.

A.1.2 GA

The GA code used to update the population for each generation.

```python
import numpy as np

def update_population(population):

    mutate_rate=15
    tournament_size=15
    elite_size = 1

    new_population = np.empty(shape=(pop_size,), dtype=Solution)
    population = np.array(sorted(population,
                             key=lambda x: x.reward,
                             reverse=True))
    new_population[:elite_size] = population[:elite_size]

    for i in range(elite_size, pop_size):
        parent_1, parent_2 = tournament_selection(population, tournament_size)
        child = uniform_crossover(parent_1, parent_2)
        mutate(child, mutate_rate)
        new_population[i] = child
```

Listing 2: GA code.