Worcester State University

MA 470: Capstone Experience

Fall 2021

# Optimizing a Convolutional Neural Network to Detect Melanoma in Images

*Author*: Christian Shadis

**Abstract**

Convolutional neural networks are the Machine Learning industry standard for computer vision, and they generally excel at image recognition. The operations performed in a convolutional neural network are mathematic in nature, yet the inner layers of the neural networks are often referred to as a 'black box' due to its complex inner workings. Melanoma, a rare but deadly form of skin cancer, can be seen in images, implying the possibility of image recognition using a convolutional neural network. This project aims to develop said network and use its mathematical properties to optimize its performance.

# 1 Introduction

Machine Learning at its core is an iterative process to find a function to minimize error. A machine learning model accepts inputs and outputs and tries to learn the relationship between them, culminating in the ability to predict an output given some input. The model then compares its prediction to the true outcome (**loss**) and makes adjustments as necessary to make a better prediction in the next iteration. The model does this many times in a process called **training**, in which the model's task is to minimize the value of the loss. Once the model is trained, it is tested using previously unused inputs and compared to the true outcomes, and the model's **validation accuracy** is evaluated.

The goal of this project is to learn about the function of convolutional neural networks, and use this newfound information to improve and, hopefully, optimize a convolutional neural network to detect melanoma, a type of skin cancer, in images of skin lesions. The original target accuracy rate for the fully optimized model was 70%.

## 1.1 Background

Melanoma is the most deadly form of skin cancer in the world. It is very rare, affecting only approximately 0.022% of the United States population in 2019. Though it is rare, even among cases of skin cancer, it accounts for 75% of all skin cancer deaths. [7] It appears in the form of a skin lesion very similar to a nevus (a nevus is a benign colored marking on the skin) and Seborrheic Keratosis, which is a benign growth on the skin. The goal of this project is to improve and optimize a previously-built convolutional neural network to detect melanoma in images of those three skin lesions. As early detection has been labelled paramount in fighting this disease, creating software capable of detecting it could be helpful to streamline the diagnosis process.

## 1.2 The Problem of Machine Learning

On the simplest level, the goal of machine learning is to find a mapping between inputs and outputs that minimizes the loss between the predicted and actual outputs. A program is given a task, and is said to **learn** if its performance improves with experience [2]. The

computer predicts output $p \in \mathbb{R}$ by taking an input $i \in \mathbb{R}$ and multiplying it by a weight $w \in \mathbb{R}$.

$$p = i * w \tag{1}$$

When dealing with multiple inputs and weights, Equation 1 can be extended into a weighted sum to predict $p \in \mathbb{R}$ by multiplying each element of an input vector $i \in \mathbb{R}^1$ by each element of a weight vector $w \in \mathbb{R}^1$ and calculating the sum of each product.

$$p = (i_1 * w_1) + (i_2 * w_2) + ... + (i_n * w_n) \tag{2}$$

In practice, it is common for a machine learning model to output multiple predictions in the form of a vector.

Consider some $m \times n$ matrix $\mathbf{W}$, where $m$ is the number of different predictions to make, and $n$ is the number of features (elements of the input vector) with weights attached to them. $\mathbf{W}_{1,1}$ would be the first feature's weight associated with the first prediction we are making. $\mathbf{W}_{2,1}$, while examining the same feature, will have a weight associated with a different prediction. Now consider some input vector $\vec{v}$, where each $v_n$ is the input value associated with feature $n$. Now let $\vec{p}$ be a vector of predictions, where $p_m$ is the value of prediction $m$. We can now define a relationship between $\mathbf{W}$, $\vec{v}$, and $\vec{p}$. For each $p_m$ in $p$,

$$p_m = \mathbf{W}_{n*} * v_n \tag{3}$$

where $\mathbf{W}_{n*}$ is the $n$th row of the matrix $\mathbf{W}$, and the $*$ operator denotes elementwise multiplication. Simply put, a prediction vector is produced by computing the dot product of the input vector with the weight vector.

Once these predictions are made, a loss function is used to calculate how erroneously the model performed. The model then often uses an algorithm called gradient descent to search for a minimum in the relationship between the feature's weight and the loss calculated. In general, loss can be described as a function describing the distance between a prediction and an expected value. This may come in the form of simple residuals on a regression line, or more complicated functions like cross-entropy.

In practice, inputs often appear to be random. The values across the input may be distributed according to a specific probability distribution, but there is no apparent pattern to the data – each data point is a different random value. Thus we often describe an input as a 'Random Variable' (denoted X), which represents the various possible states the input values may take. The realized values of X are denoted $x_c$ for some $c \in N$.

Thus for some random input $\mathrm{X} \in \mathbb{R}^n$, some random output $\mathrm{Y} \in \mathbb{R}$, some unknown $f$ such that $f(\mathrm{X}) = \mathrm{Y}$, and some loss function $L$ examining the difference between Y and the model's predictions, the problem of machine learning can be summarized as the search for the optimal $\hat{g}$ such that

$$\hat{\mathbf{g}} = \mathbf{arg\ min}\ \mathbf{E}[\mathbf{L}(\mathrm{Y}, \mathbf{g}(\mathrm{X})] \qquad [1] \tag{4}$$

Before clarifying this formula, consider the difference between *min* and *arg min*. The min function returns the minimum value in the range of the function, whereas the arg min function returns the value of the domain for which that minimum is attained. Consider the following example:

**Example 1.** *Let* $k(x) := (x+1)^2 + 1$. *The minimum value of this function occurs at the point (-1, 1).* $min(k(x)) = 1$, *but* $argmin(k(x)) = -1$. *The min function returns the output value 1, whereas the arg min function returns the argument value -1.*

Note that $E[L(\mathrm{Y}, g(\mathrm{X})]$ refers to the expected value of the loss function between two random variables – the true output Y and the output of our proposed mapping $g$ applied to the random input X.

The goal is to find that minimum expectation, but its expected value is inconsequential. The mapping itself, the arg min, is the desired information. Thus we achieve our optimal mapping $\hat{g}$ such that the expected difference between $\hat{g}(x)$ and Y is minimized.

## 1.3 Overview of Gradient Descent

Gradient descent is the fundamental algorithm for machine learning. In oversimplified terms, it can be thought of as the search for a weight configuration such that the loss function is at its minimum.

Consider for some generalized binary classification neural network, we have some desired output $y \in \{0, 1\}$. We also have N features for each input $x \in \mathbb{R}$, some weight $w \in \mathbb{R}$, and some bias (or error) $b \in \mathbb{R}$. We can define a $z := wx + b$. $z$ can be thought of as the 'result' of passing the input through a layer of the model. Lastly, define some $\sigma$ representing an activation function, and let $a := \sigma(z)$.

Binary cross-entropy is a measure of distance between our desired outcome $y$ and our 'true' outcome $a$. It is defined as follows:

$$C = -\frac{1}{N} \sum_{x} [y \ln a + (1 - y) \ln(1 - a)] \tag{5}$$

For each iteration of the learning process, there will be a value for this loss function, which is the target for minimization. In order to minimize this multidimensional surface, one must find the gradient of the surface and negate it. The gradient is used to determine the direction of fastest increase in the original function, so negating the gradient provides a directional vector pointing to the sharpest decrease in loss.

**Definition 1.** *A* ***gradient*** *of a multidimensional function is the vector of all partial derivatives of the function.*

The gradient can be thought of as a large vector containing each adjustment to be made to the layer preceding it, represented by partial derivatives. To adjust the weights of previous layers (a well-known process known as backpropagation), the derivative of the cost function with respect to the weights of the previous current layer $j$ is calculated, and the resulting adjustments are made to the weights in the previous layer, and the process continues recursively through the remainder of the model.

For the binary cross-entropy function, the gradient can be represented explicitly:

$$\frac{\delta C}{\delta w_j} = \frac{1}{N} \sum_x x_j(\sigma(z) - y) \tag{6}$$

In short, the essential goal of gradient descent is to calculate the loss function between desired and resulting outcomes, calculate the negative gradient of that function yielding a vector of weight adjustments, and then backpropagate those weight adjustments back through the model.

### 1.4 Past Work

The neural network used for this classifier was adapted from a basic implementation of the Keras Sequential model (See 2.1) in Python that I had previously built. The model was built with little to no understanding of convolutional neural networks or the Machine Learning process in general. The purpose of the model was to distinguish between melanoma, seborrheic keratosis, and nevi. The model's accuracy was only 38.7% and thus a perfect candidate for improvement and optimization.

In the process of optimizing the network, the problem was refactored into binary classification. In other words, the model classifies an image as 'melanoma' or 'not melanoma' instead of distinguishing between three categories.

## 2 Convolutional Neural Networks

A convolutional neural network is a type of multilayer machine learning model often used in image recognition or classification. The process usually consists of passing input images through a series of mathematical operations called **Convolutions** that extract the most important features of an image. In this model, a 3-dimensional tensor represents the image, one axis of which corresponds to the Red Green and Blue color channels, and two of which correspond to the grid of pixels in an image.

**Definition 2.** *A tensor* $\mathbf{A}$ *is an array with a variable number of axes. We identify the element of a three-dimensional tensor* $\boldsymbol{A}$ *at coordinates* $(i, j, k)$ *by writing* $\boldsymbol{A}_{i,j,k}$.

**Example 2.** *If A is a tensor representing a 256 x 256 x 3 image, we may refer to the Red color channel of the first pixel as* $A_{0,0,0}$ *or in Python as A[0][0][0].*

The images are passed through several transformations, reducing our 3-dimensional tensors to a single output vector containing the image's predicted classification. These output vectors typically contain one entry for every possible classification category. Each index of the vector will contain a 0 except for the index corresponding with the model's classification. This is called one-hot encoding.

**Definition 3.** ***One-hot encoding*** *is the process of converting a categorical label into a vector containing a boolean value for each possible category.*

**Example 3.** *Consider a dataset of locations defined by latitude and longitude. We wish to classify into one of four categories: North, South, East, and West. A single data point would have a label (i.e. 'North'). To one-hot encode this variable, create a vector of four values, and assign a boolean (true/false) flag to each index (i.e. each category). A datapoint labelled 'North' might be one-hot encoded in the form [1, 0, 0, 0], where the the vector can be thought of intuitively as the following: [is_North?, is_South?, is_East?, is_West?].*

**Example 4.** *Consider we are attempting to classify an image into one of three categories: car, motorcycle, or airplane. If our model deemed an input image to be a motorcycle, the output vector would be* $[0, 1, 0]$

## 2.1 The Keras Sequential Model

Keras is a high-level, accessible interface for TensorFlow, a widely-known machine learning library. The Keras library in Python allows a programmer to easily construct complex deep learning models in a relatively small, understandable chunk of code. The Sequential model in Keras is a framework for creating deep learning models in which the output of one layer is input directly into the next. Amongst these layers are convolutional layers, max-pooling layers, and dense layers. The programmer builds the model by adding layers individually, setting parameters for each layer, compiling the model with a loss function and an optimizer, and fitting the model to training data.

For the purposes of this project, I have selected a very standard optimizer (the Adam Optimizer).

## 2.2 The Kernel

The goal of the convolutional layer is to extract important features of the image. This is accomplished by initializing an $n$ x $n$ **kernel**, or weight matrix $\mathbf{W}$.

Common practice before 2010 was to initialize the kernel randomly with the uniform distribution over an interval determined by the size of the current layer's input.

$$W_{i,j} \sim U[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}] \tag{7}$$

where $\mathbf{W}_{i,j}$ is the entry in the $i$th row and $j$th column of the kernel. Xavier Glorot, a machine learning research scientist, found this random initialization to be problematic especially when

working with the sigmoid function, where certain adjustment values for different layers of the network became oversaturated at specific values, crippling the network's performance. [4]

Through experimentation, Glorot and his colleagues developed an initialization which solved the saturation problem, but also increased the stability of the variance of adjustments made at each layer. The initialization became known as the Glorot Initialization (or the Xavier Initialization), and it remains widely in practice today, despite the decrease in use of sigmoid activations (See 8.1.

$$W_{i,j} \sim U[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}] \qquad [4] \tag{8}$$

where $n_j$ represents the size of the $j$th layer of the neural network. The Glorot initialization for a kernel in one layer of the neural network then depends on other layers of the network.

Each time the model creates (or initializes) a kernel, it produces an $n$ x $n$ **kernel**, or weight matrix $\mathbf{W}$ (where $n$ is set by the programmer), with each entry randomly generated with respect to the above uniform distribution.

## 2.3 Convolutional Layers

A convolutional layer receives an input image and iterates a series of kernels (see 2.2) over an image in an effort to extract abstract, high-level details of the image.

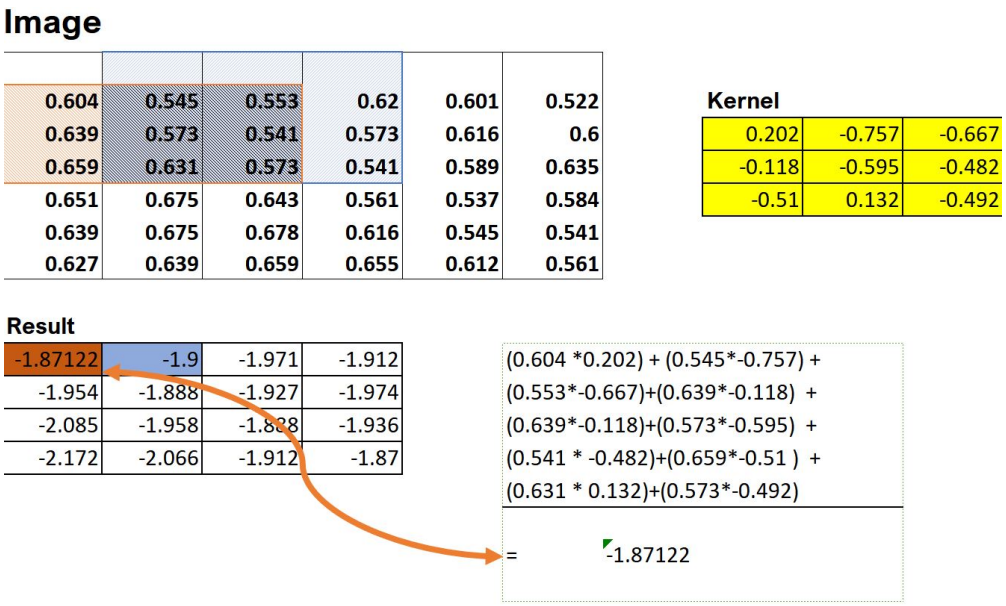


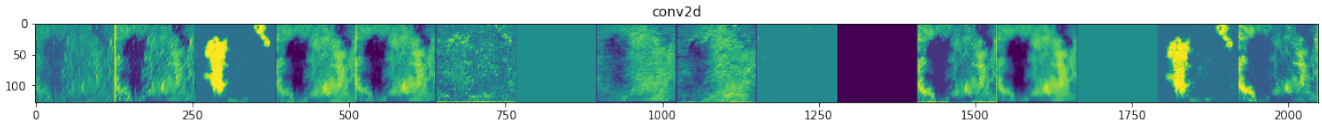

Figure 1: Kernel applied to two regions (orange and blue), result calculated

The kernel, consisting of real numbers between -1 and 1, is first 'applied' to the top-left corner of the image. The kernel will overlap the first $n$ rows and $n$ columns of the grid of pixels

in the image. Each weight is multiplied by the pixel value it overlaps, and all subsequent products are added together. One can think of this process as taking a 'dot product' of the kernel with the region it covers. The resulting sum of products becomes the top-left entry of the convolution layer's output matrix $\mathbf{O}$. The kernel then shifts one pixel to the right (or some other number of steps, defined by **stride**), the process repeats, and the resulting sum of products becomes the entry in $\mathbf{O}_{1,2}$. When the rightmost border of the kernel has reached the edge of the image, the kernel returns to the leftmost edge of the image, and shifts one pixel down (or some other stride). The process repeats until the kernel has been applied to every possible region of the input image.

We typically, but not necessarily, choose the kernel to have odd dimensions to ensure a pixel is the exact center of the kernel. This makes mapping feature maps to the next layer easier for the computer since spatial symmetry is preserved. Thus for this project, we will choose kernel sizes with square, odd dimensions. Additionally, we typically choose stride to be 1 to ensure the kernel considers each pixel equally. A designer may choose another value of stride for any number of reasons, such as ensuring kernels do not visit the same pixel multiple times. For the purpose of this project, setting stride = 1 is practical.

For example, passing an image of a skin lesion through a convolutional layer with 16 different 3 x 3 filters (kernels), created with the Glorot initialization, will produce something similar to the following set of 'feature maps'. This process is typically repeated through several layers of convolution.




## 2.4 Activation Functions

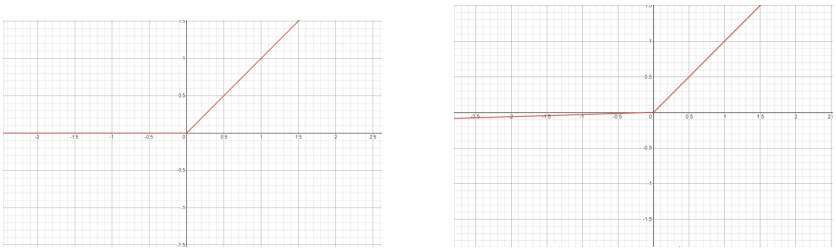
An activation is some sort of nonlinearity applied at one layer of the neural network to the output of the preceding convolution. One of the most commonly used activation functions, which was originally implemented in the CNN, is the Rectified Linear Unit (ReLU) function. Given some input $x$, the ReLU function calculates $\hat{x}$ such that:

$$\hat{x} = \max(0, x) \tag{9}$$

Simply put, the ReLU function 'deactivates' any neurons with a negative value. ReLU is among the most commonly-used activation functions for problems involving computer vision, partly because ReLU does not suffer from a vanishing gradient. In other words, the derivative of the function at a large x-value does not approach zero. In an applied context, a vanishing gradient can cause the updates to the weight matrix to be become negligibly small and the model may stop learning. ReLU suffers its own drawbacks, as once any neurons are set to zero, they remain 'dead' throughout the rest of the model, halting their learning.

This problem is addressed using the slightly modified LeakyReLU function. The LeakyReLU function accepts a small parameter, $\alpha$, which is multiplied by the x-input instead of setting to 0. This allows the network to process that negligible weight without completely killing its functionality. LeakyReLU addresses the main downside of ReLU function – the complete deactivation of neurons – while retaining its positive characteristics. The LeakyReLU function is be defined by:

$$\textbf{leakyReLU: } \mathrm{f}(x) = \max(\alpha x, x) \tag{10}$$



(a) ReLU (b) LeakyReLU with $\alpha = 0.03$

Figure 2: ReLU vs. LeakyReLU

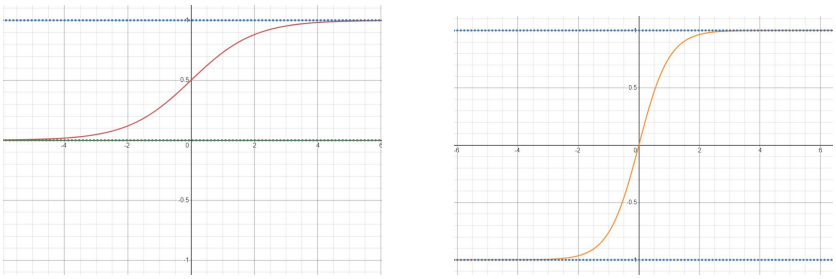
Activation functions have several important qualities. First, the function must be differentiable everywhere, since backpropagation (adjusting weights backward throughout layers) in CNNs is based on calculating derivatives. Second, activation functions should typically be zero-centered so the most 'average' neurons have the smallest effect on the model. The role of activation functions is to extract non-linear patterns in the data, which is extremely unlikely to be linear. For example, there is no linear relationship between a pixel's value and the adjacent pixel, but there could be a nonlinear relationship between groups of pixels. Activation functions serve to highlight those relationships, determining the values to pass on to the next layer of the neural network.

There are many other activation functions, such as the sigmoid and tanh.

$$\textbf{Sigmoid: } \sigma(x) = \frac{1}{(1 + e^{-x})} \tag{11}$$

$$\textbf{Tanh: } \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{12}$$

The sigmoid activation function was once used commonly on convolutional layers, but now is largely out of use, as it suffers the vanishing gradient problem, is not centered around zero, and is computationally expensive. However, it remains in use for only the final layer of binary classification models such as this due to its output range being saturated with values near 0 or 1. For more on the sigmoid function, see 8.1.



(a) Sigmoid (b) Tanh

Figure 3: Sigmoid vs. Tanh

The tanh function solves the sigmoid's problem of being zero-centered, but also suffers the vanishing gradient problem and is also computationally expensive.

For implementation, see 4.2.

## 2.5 Max Pooling Layers

A powerful feature of convolutional neural networks are their ability to detect patterns in the image. A convolutional layer provides a set of 'feature maps' - transformed versions of the input image - as input to an intermediary *max pooling layer*, which condenses the feature map by iterating a small 2 x 2 'window' over the image, populating a new image containing the maximum value from each region the window covered.

**Input**

| | | | |
|---|---|---|---|
| -1.89122 | -1.9 | -1.971 | -1.912 |
| -1.954 | -1.888 | -1.927 | -1.974 |
| -2.085 | -1.958 | -1.888 | -1.936 |
| -2.172 | -2.066 | -1.912 | -1.87 |

**Max-Pooled**

| | |
|---|---|
| -1.888 | -1.912 |
| -1.958 | -1.87 |

Figure 4: Figure 1 result passed through a max pool layer with a 2x2 window and stride=2

Max Pool Layers allow the neural network to 'select' the most important features to pass on to the next convolution layer or a dense layer. A 2x2 window with stride=2 for an image with even dimensions will halve the image's size, resulting in only the most pronounced weights being fed forward to the next layer, which improves pattern detection and reduces computational cost.

## 2.6 Dense Layers

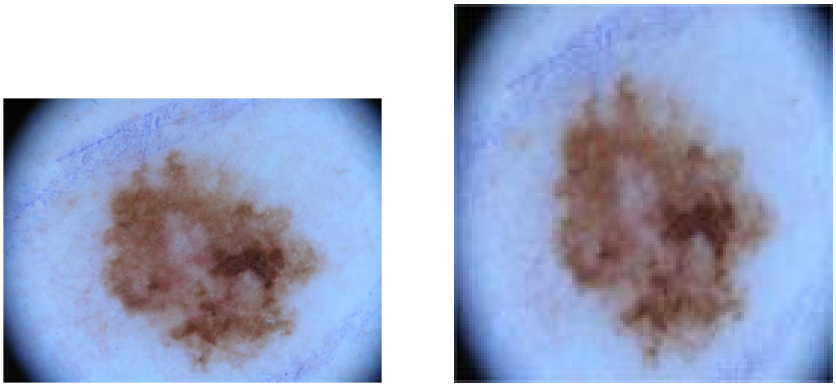
Dense Layers are the foundation of modern deep learning. They behave similarly to convolution layers in that they use a set of weights to compute a 'dot product' with a set of inputs and then pass that through an activation function. The essential difference between the two types of layers is the fact that in a convolutional layer, the model seeks patterns

in a certain neighborhood surrounding a central pixel, whereas a dense layer seeks patterns between a pixel and all other pixels. Thus each pixel is connected to every other pixel. This is from where the terms 'dense' and 'densely-connected' were derived. The inner workings of densely connected layers are quite complex. For this project, it suffices to understand that a densely connected layer seeks image-wide patterns while a convolutional layer seeks region-wise patterns.

# 3 Data Preprocessing

## 3.1 Image Resizing

The images included in this classifier were taken from the Melanoma Detection Dataset on Kaggle [5]. There are 2750 images total, split into three categories: Melanoma ($n$ =521), Nevus ($n$ =1843), and Seborrheic Keratosis ($n$ =386). They were non-uniform in size, and high-definition. In order to create a neural network to classify these images, they were resized. For the network, all images were resized using the Python Image Library (PILLOW) to the low-resolution size of 256x256 to improve normality among all images in the training set and to enable the model to handle each picture's file size. This allows each image to be analyzed in the same way and much more efficiently, but reduces the resolution and accuracy (hence, detail) of the image, ultimately harming our model's performance.



(a) Original High-Definition Image

(b) Resized Image

Figure 5: Image of melanoma before and after resizing

## 3.2 Numerical Representation

Each image is originally represented as a 256 x 256 x 3 tensor, representing the three color channels on the 256x256 grid of pixels. Each entry in the array is an integer in the interval $[0, 255]$, representing the strength of the representation of that particular color channel on that pixel. We then transform the values in the tensor by scaling each entry by a factor of $1/255$ so all values lie between 0 and 1. In Python, this was implemented as a 'numpy'

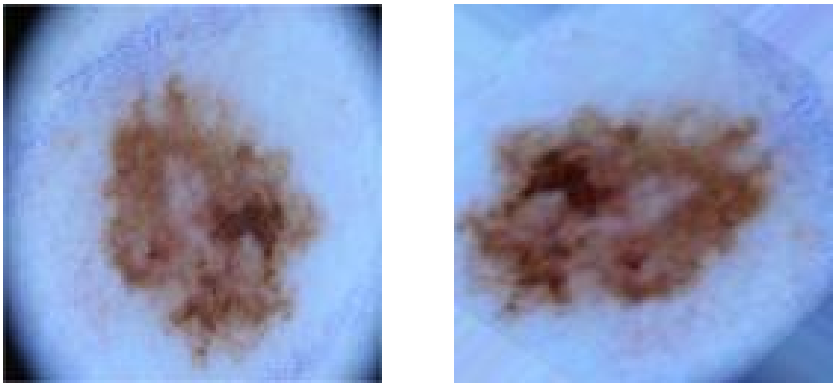
(numeric library of tools for Python) $n$-dimensional array, which can essentially be thought of as a list of arrays, which are themselves just lists of lists.

```
[[0.21960784 0.17254902 0.11372549 ... 0.7372549  0.74901961 0.76078431]
 [0.4627451  0.43921569 0.4        ... 0.7372549  0.74901961 0.76862745]
 [0.72941176 0.73333333 0.70588235 ... 0.74117647 0.75686275 0.76862745]
 ...
 [0.85098039 0.86666667 0.89019608 ... 0.67058824 0.66666667 0.66666667]
 [0.85882353 0.87058824 0.89019608 ... 0.66666667 0.66666667 0.6627451 ]
 [0.8627451  0.8745098  0.89411765 ... 0.6745098  0.68627451 0.68627451]]
```

Figure 6: Numerical representation of first color channel of Figure 5b

## 3.3 Data Augmentation

Once a training set of 256x256 images was produced, the ImageDataGenerator class from the 'keras.preprocessing.image' package was used to generate slightly modified, 'augmented' versions of the images to increase the possible sample size. Transformations applied to images include shear, zoom, and horizontal reflection. Each image is used to generate approximately 35 additional images. After augmentation, there are over 18000 images of melanoma, over 72000 images of nevi, and over 12000 images of seborrheic keratosis.



(a) Original 128x128 image Image

(b) Augmented Image

Figure 7: Image of melanoma before and after augmenting

Augmenting extra images increased the size of the training set by tenfold, while also providing variations of images - this improves the model's ability to generalize by exposing it to different 'angles' of the image, and making it more difficult for the model to memorize the training images.

### 3.4 Data Selection

Images used for the validation set (n = 200) were selected first and removed from the training pool. Then, the remaining training images were augmented. A random sample of melanoma images (n = 1000) and non-melanoma images (nevus n = 500 , seborrheic keratosis n = 500) were then chosen and pooled into an overall training set. The shuffled training pool was then split into an 80-20% train-test split. The image-label pairs in each the train set, test set, and validation set were then shuffled in tandem to retain their accuracy and prevent the model from learning patterns from the order in which the data was entered. There now existed a shuffled training and testing set for each epoch (learning iteration) of the deep learning model, along with a previously-unseen shuffled validation set to evaluate the model's accuracy and generalization ability after training.

## 4 Final Model Architecture

There are a plethora of standard structures for convolutional neural networks. Over the course of several months, I experimented with hundreds of configurations of the Keras model, and while none were 'optimal', there was a configuration that outperformed the rest. This section aims to dissect the structure of this 'Final Model'.

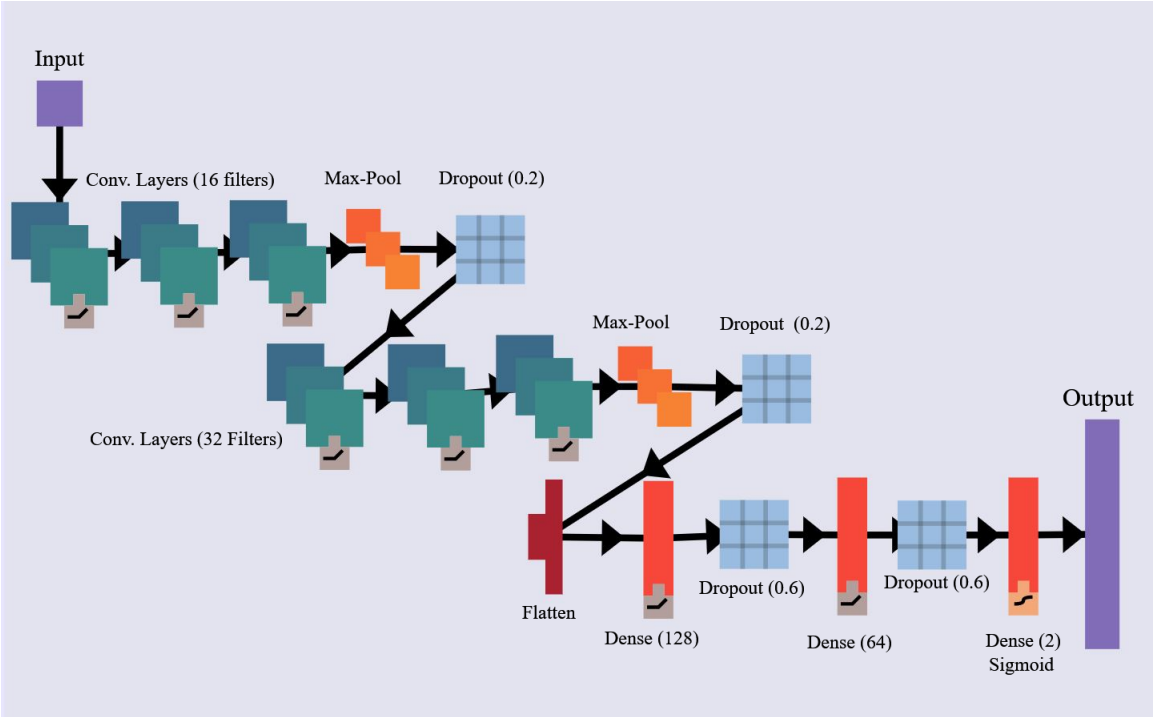



Figure 8: Architecture of final model
(Built with https://math.mit.edu/ennui/)

### 4.1 Convolutional Layers

The final model constructed consists of **6** convolutional layers in two blocks. In the first block, each layer has 16 filters. In the second, each layer has 32 filters. Kernels applied to each of those layers were 3 x 3.

### 4.2 Activation Functions

The LeakyReLU activation function ($\alpha = 0.3$) was applied to the output of each convolutional layer. This weakened the strength of weights associated with negative values (without killing it completely), and then passed this transformed data to the next layer.

### 4.3 Max-Pool Layers

After each block of convolutional layers followed a max-pooling layer. The window size for each layer was 2 x 2 and the window moved with stride = 2, effectively halving the size of feature maps.

### 4.4 Dropout Layers

Dropout layers are helpful for improving the model's ability to generalize patterns to data previously unseen. The layer is passed a parameter representing the proportion of active neurons to deactivate. The purpose of dropout is to keep the model from overfitting and memorizing the dataset, forcing it to make more abstract connections. In the final model, there were **4** dropout layers, placed **after each convolution block** with a lighter rate of 0.2, and **after each of the first two dense layers** with a more intense rate of 0.6.

### 4.5 Dense Layers

Dense layers, as described in 2.6, are useful for detecting global patterns in the data. In the final model, there were **three** dense layers, with **128**, **64**, and **2** hidden units, respectively. The final dense layer of the model is outfitted with the sigmoid activation function, which effectively transforms the data to be close to 0 or to 1, determining which prediction is made. The sigmoid function transforms a vector of values into a one-hot encoded vector containing the prediction.

## 5 Tuning Hyperparameters

A hyperparameter is a vague descriptor, but for the scope of this project, we will define it as so:

**Definition 4.** *A **hyperparameter** is any element of the model that can be pre-configured*

For the purpose of comparing effects independently, we will construct a 'control' model identical to the Final Model. This model, being trained on a sample size of 2000, has **6**

convolutional layers (with **32**, **32**, **32**, **64**, **64**, **64** filters, respectively), the leakyReLU Activation Function ($\alpha = 0.3$), a max-pool layer following every convolutional block (with a 2x2 window and stride = 2), a flatten layer, **3** dense layers (with **128**, **64**, and **2** units), and finally a sigmoid activation to classify. There will be **4** dropouts, after **each convolutional block (0.2)** and **after the first two dense layers (0.6)**. The model will use the Adam Optimizer with learning rate of **0.001**, will train with batch size **16** for some variable number of epochs (see 5.4). This base model distinguishes between melanoma and non-melanoma images with 52.50% accuracy

In optimizing the model, I focused on several specific hyperparameters: the learning rate for the Adam optimizer, the Batch Size for each epoch of the model, the type and parameters of the activation function, the number of training iterations (epochs), and the sample size. It is essential to emphasize that there is no mathematical formula for the 'correct' or 'ideal' setting for these parameters - their performances depend heavily on the exact data being analyzed. Thus it is commonplace to think of hyperparameters as small 'tuning knobs' on our machine that we tweak, observe, and tweak again.

There are methods such as random search and grid search which test a wide array of hyperparameter values and return the best performer – in this way, one can think of hyperparameter tuning as a sort of optimization problem as well; we optimize the configuration of our model which optimizes our mapping from input to output [8]. Grid search and random search (along with other hyperparameter tuning systems) were not used in this project due to time limitations. However, many different values for each hyperparameter were chosen and tested.

## 5.1 Effect of Adam Optimizer Learning Rate

The Adam optimizer is one of the standard, 'out-of-the-box' optimizers in Keras for convolutional neural networks, but it has an important parameter, learning rate, which can have a profound effect on the performance of the model.

Keras supports a number of 'callbacks', which are simply tests that can run at the end of each epoch to determine whether to continue or terminate training. One such callback is the 'Reduce Learning Rate on Plateau' callback. Its function is to detect when the validation loss has stopped improving (specifically, when four epochs have passed without a 0.001 decrease in val_loss). At this point, it reduces the learning rate (specifically by a factor of 0.2) in an attempt to circumvent the relative minimum or saddle point, if such a point is the cause for the decrease in learning. The learning rate may be reduced at any point, and any number of times until the learning rate is at a pre-defined minimum (in this case, 0.000001).

To determine the optimal value for learning rate, a number of different values were tested (both with and without the Reduce Learning Rate On Plateau callback) and the results were recorded in the table below.

| learning rate | callback? | train_loss | train_acc | val_loss | val_accuracy |
|---|---|---|---|---|---|
| 0.00001 | yes | .6278 | 66.19% | .6550 | 64.65% |
| 0.00001 | no | .6509 | 64.25% | .6852 | **68.69%** |
| 0.00005 | yes | .5558 | 72.50% | .6826 | 56.06% |
| 0.00005 | no | .4200 | 83.38% | .6802 | 59.60% |
| 0.0001 | yes | .5819 | 71.94% | .6799 | 59.09% |
| 0.0001 | no | .4272 | 81.44% | .6577 | 63.64% |
| 0.0005 | yes | .6177 | 68.00% | .6764 | 61.11% |
| 0.0005 | no | .1857 | 92.87% | .6553 | 67.68% |
| 0.001 | yes | .6345 | 65.19% | .6792 | 67.17% |
| 0.001 | no | .6846 | 56.38% | .6552 | 60.10% |
| 0.005 | yes | .6933 | 50.56% | .6932 | 54.55% |
| 0.005 | no | 772.3 | 49.44% | .6882 | 50.51% |
| 0.01 | yes | .6671 | 61.44% | .6806 | 64.65% |
| 0.01 | no | .7137 | 55.62% | .6750 | 62.12% |

## 5.2 Effect of Batch Size

Convolutional layers can detect high-level patterns in an image, but the model should be able to work with multiple images. This is where the concept of **batching** is introduced. Generally, batching is the process by which a set of images is broken down into smaller subsets (**batches**) which are then passed into the model.

Recall that a layer's output is then passed through other layers in the network, which adjust the weight matrices used to make predictions about the identity of an image. The size of each batch passed through the convolutional layer essentially determines how many images are considered by the model before weights are updated back throughout the model.

Weight updates are typically determined in a common algorithm known as gradient descent. There are three types of gradient descent, directly dependent on batch size. Gradient descent can refer to either stochastic gradient descent, batch gradient descent, or mini-batch gradient descent.

**Definition 5.** ***Stochastic Gradient Descent*** *entails updating weight matrices repeatedly after every image is passed through the model (batch_size = 1)*

**Definition 6.** ***Batch Gradient Descent*** *entails updating weight matrices after all images have been passed through the model (batch_size = sample_size)*

**Definition 7.** ***Mini-batch Gradient Descent*** *entails updating weight matrices repeatedly after batches of images (batch_size pre-determined) are passed through the image.*

Batch size is an important hyperparameter of the model. The batch size determines how many samples (images) are passed through the model at a time before adjusting weights based on the error. Batch size can be set as low as one, or as high as the number of images

in the training set, but it is typically set to some number in between. Common practice is to set it to a power of two, but this is non-essential. The model was run with various batch sizes (including those that fulfill the requirements for stochastic, batch, and mini-batch gradient descent) and its performance evaluated with each. The results were recorded below.

| Batch Size | train_loss | train_acc | val_loss | val_accuracy |
|---|---|---|---|---|
| 1 | .6600 | 62.25% | .6699 | 59.60% |
| 2 | .6583 | 62.06% | .6289 | **68.18%** |
| 8 | .6406 | 65.19% | .6642 | 61.11% |
| 16 | .6291 | 67.62% | .6724 | 65.15% |
| 32 | .6461 | 64.69% | .6540 | **68.18%** |
| 64 | .6337 | 67.37% | .6778 | 64.14% |
| 128 | .6373 | 66.85% | .6807 | 60.10% |
| 256 | .6905 | 53.27% | .6941 | 50.51% |

## 5.3 Effect of LeakyReLU Alpha Level

As discussed in section 4.1, the LeakyReLU activation function was selected for the final model. LeakyReLU has an adjustable parameter denoted $\alpha$ which represents the magnitude of the slope for the portion of the graph on the subdomain $(-\infty, 0)$. The model's performance for several different values $\alpha$ was evaluated and the results recorded below.

| $\alpha$ | train_loss | train_acc | val_loss | val_accuracy |
|---|---|---|---|---|
| 0.0001 | .6938 | 48.75% | .6932 | 49.49% |
| 0.0005 | .6893 | 54.31% | .6925 | 53.54% |
| 0.001 | .6929 | 51.31% | .6932 | 35.35% |
| 0.005 | .6922 | 50.88% | .6930 | 50.51% |
| 0.01 | .6952 | 45.31% | .6932 | 49.49% |
| 0.05 | .6937 | 50.81% | .6933 | 50.51% |
| 0.1 | .6913 | 52.94% | .6932 | 53.03% |
| 0.3 | .6501 | 61.69% | .6771 | **63.13%** |

## 5.4 Number of Epochs

An epoch in a neural network is a training iteration in which the entire dataset is passed through the neural network once. During each epoch, the model is given data and makes predictions against a test set, before updating the weights throughout the model. The number of epochs would seem to have a strong positive correlation with our model's testing accuracy, but the number of epochs is subject to diminishing returns. The more the model learns the training data, the more likely it is to overfit - to memorize the training data without learning the abstract patterns behind it and fail to generalize its findings to unseen data. Even if overfitting is avoided, the accuracy of the network tends to plateau at certain points – this could be where the network found a relative minimum in the loss function (or, worse, a saddle point), when the absolute goal of the model is to find the global minimum.

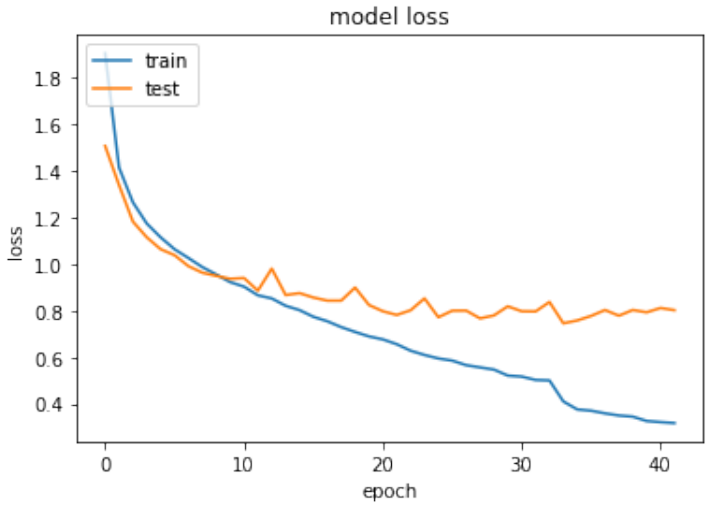



Figure 9: Example of overfitted model. Overfitting begins around epoch ten.

Overfitting is detected by analyzing the changes in the training accuracy of the model versus the testing accuracy of the model – if training accuracy increases while testing accuracy halts, then the model has begun to 'memorize' the dataset. The model has learned what each training image looks like but has not learned the underlying patterns in the data needed to accurately make predictions on newly-introduced images.

In the process of developing this model, I observed overfitting many times. In earlier epochs, training and testing accuracy improve at similar rates, but as the number of epochs increases, the training loss continues to improve while the testing loss plateaus. Thus the model is beginning to overfit (memorize the training data) when that divergence occurs. In Keras, the earlyStopping callback is used to cease training early once validation loss (or some other specified metric) stops improving for some length of time.

Instead of defining the number of epochs manually, I configured an earlyStopping callback to halt training when the testing accuracy has remained stagnant for 15 epochs. In other words, when the model has stopped learning generalized patterns, it will be stopped from learning.

## 5.5 Effect of Activation Function

The control model was modified to run with four different activation functions. The resulting measurements are recorded below.

| Function | train_loss | train_acc | val_loss | val_accuracy |
|---|---|---|---|---|
| ReLU | .6931 | 51.00% | .6931 | 50.00% |
| LeakyReLU($\alpha = 0.3$) | .6501 | 61.69% | .6771 | **63.13%** |
| sigmoid | .6966 | 46.44% | .6938 | 50.00% |
| tanh | .1179 | 97.81% | .6512 | 60.50% |

### 5.6 Effect of Sample Size

Since validation images were selected before augmentation, there will be no more than 100 images in the validation sets for each skin lesion. For small sample sizes, the validation set was chosen to be 10% of the data. This percentage is smaller for larger sample sizes, but taking 500 or 1000 non-augmented images is not possible with this data set.

| Sample Size | train_loss | train_acc | val_loss | val_accuracy |
|---|---|---|---|---|
| n = 40<br>v = 4 | .7216 | 56.25% | .8708 | 50.00% |
| n = 200<br>v = 20 | .6940 | 56.25% | .6963 | 50.00% |
| n = 1000<br>v = 100 | .6425 | 64.38% | .6910 | 56.00% |
| n = 2000<br>v = 200 | .2291 | 92.69% | .7031 | **57.00%** |

Table 1: Model's performance by training set sample size (n) and validation set size (v)

### 5.7 Optimized Model

For each previous section, the best-performing value for each parameter was re-applied to our 'final' model. Thus the Adam optimizer was set to a learning rate of 0.00001 without the learning rate reduction callback, LeakyReLU was chosen as the activation with $\alpha = 0.3$, batch size was set to 32, and the sample size was set to $n = 2000$. The resulting model distinguished melanoma from non-melanoma images with 62.50% accuracy, compared to the base model's 52.50% accuracy – an improvement of 10%.

## 6 Conclusion

Throughout this project I learned about the function of convolutional neural networks: How each layer works, how data is represented manipulated and output, and how to change model hyperparameters to improve performance.

At this point, the convolutional neural network still suffers from poor performance. There are a number of limiting factors that could be contributing to this poor performance. First, the images were resized from high-resolution images to small 256 x 256 versions. Running the model with higher resolutions could make the model more capable of detecting subtle patterns in the data that are only evident with a higher resolution. Second, the workstation the model was developed on was limited to 26GB of RAM. With more computing power, higher sample sizes and resolutions could be considered.

While the original project goal of 70% accuracy was unmet, the model improved from distinguishing between the three skin lesions at a rate under 40% to classifying the existence (or nonexistence) of melanoma at an accuracy rate between 60 - 65%. This does not mean that, given an image of melanoma, there is a 65% chance of correct classification. Instead, it means that given a large dataset of images of skin lesions with and without melanoma, the model will correctly classify 65% of the images.

It is unclear whether it is possible to develop a convolutional neural network to accurately predict the type of skin lesion using the images in this dataset. With more computing power and higher resolution and sample sizes, it could be achievable.

# 7 Bibliography

# References


[1] J. Koushik (2016). Understanding Convolutional Neural Networks. *ArXiv*. abs/1605.09081.

[2] I. Goodfellow and Y. Bengio, and A. Courville (2016). Deep Learning. *MIT Press*.

[3] A. Trask (2019). Grokking Deep Learning. *Manning Publications Co.*.

[4] Glorot, X., Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (pp. 249–256). *PMLR*.

[5] Santori, Pablo Lopez. (May 2020). Melanoma Detection Dataset, Version 1. Retrieved 10/14/2021 from https://www.kaggle.com/wanderdust/skin-lesion-analysis-toward-melanoma-detection/.

[6] Mishkin, D., Sergievskiy, N., Matas, J. (2016). Systematic evaluation of convolution neural network advances on the Imagenet. Computer Vision and Image Understanding, 161. https://doi.org/10.1016/j.cviu.2017.05.007

[7] Davis, L. E., Shalin, S. C., Tackett, A. J. (2019). Current state of melanoma diagnosis and treatment. Cancer biology therapy, 20(11), 1366–1379. https://doi.org/10.1080/15384047.2019.1640032

[8] Zheng, Alice (2015). Evaluating Machine Learning Models. *O'Reilly Media, Inc.*.

[9] Yedlin, M., Jafari, M. (2020, February 4). Binary cross-entropy. YouTube. Retrieved December 15, 2021, from https://www.youtube.com/watch?v=wpPkDSMzdKo

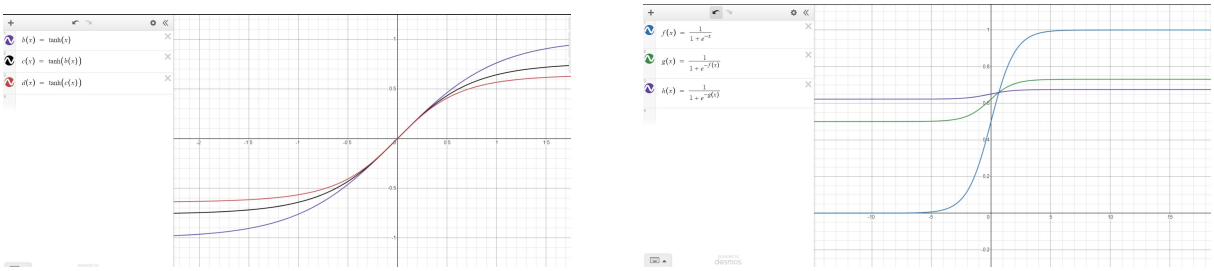
# 8 Appendix: Python Code and More

## 8.1 Why Not Sigmoid?

As referenced in 2.4, the sigmoid function was once widely in use for inner layers of a convolutional neural network. There are multiple reasons why that is no longer the case. First, computational expense is paramount when building a model with millions of parameters and calculations occurring. In my own experiments, I found the sigmoid function to take over twice as long as the LeakyReLU function when performed on a large input.

The second problem with the sigmoid function is its center being nonzero. This becomes a problem when stacking multiple sigmoid calculations on top of each other (or with convolutions in between). Consider a similar function, the tanh. The primary difference between the two is the fact that the tanh is centered about 0. Taking the tanh of the output of the tanh function ($\tanh(\tanh(x))$), or even the composition of tanh with that composition, will result in a zero-centered range of values. In contrast, taking the composition of two sigmoid functions results in a messier outcome.

The final problem with using the sigmoid function in the inner layers of a convolutional neural network is its vanishing gradient problem. In this context, gradient refers to the derivative of the function. As evident by figure 10, the derivative of the function is near-zero at almost all values of x. Therefore, computing derivatives as a learning measure (which is the foundation of gradient descent) becomes ineffective.



(a) Tanh composed with itself

(b) Sigmoid composed with itself

Figure 10: Composition of similar activation functions

## 8.2 Code: Imports and Function Definitions

```
import tensorflow as tf
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Conv2D, MaxPooling2D, Flatten,
    Dense, Dropout, BatchNormalization
from keras.models import Sequential
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.layers import LeakyReLU
```

```
from keras.regularizers import l2
from keras.callbacks import EarlyStopping,
    ReduceLROnPlateau, ModelCheckpoint
import os
import keras
import numpy as np
from skimage.io import imread
from PIL import Image
from datetime import datetime
import random
from matplotlib import pyplot as plt
from tqdm import tqdm
from keras.preprocessing.image import img_to_array, load_img
from mpl_toolkits.axes_grid1 import ImageGrid
from google.colab import files
from keras.utils import np_utils
import pandas as pd
import seaborn as sns

def pixels_from_path(file_path):
    im = Image.open(file_path)
    im = im.resize(IMG_SIZE)
    np_im = np.array(im)
    #matrix of pixel RGB values
    np_im = np.array((1./255) * np_im)
    return np_im
```

## 8.3 Code: Data Preprocessing

```
SAMPLE_SIZE = 1000
valid_size = 100

random.seed(1)
start_time = datetime.now()

melanoma_directory = r"../melanoma/picscopy/melanoma"
nevus_directory = r"../melanoma/picscopy/nevus"
seb_directory = r"../melanoma/picscopy/seborrheic-keratosis"

melanoma_validation_directory = r"../melanoma/picscopy/mel_valid"
nevus_validation_directory = r"../melanoma/picscopy/nev_valid"
seb_validation_directory = r"../melanoma/picscopy/seb_valid"

removalcount = 0
```

```
resizecount = 0
IMG_SIZE = (128, 128)

#Resize images, remove all augmented
melfilenames = os.listdir(melanoma_directory)
for filename in tqdm(melfilenames):
    if filename.startswith("aug"):
        removal = str(melanoma_directory+"/"+ filename)
        removal = removal..replace('/', '\\')
        os.remove(removal)
        removalcount = removalcount + 1
        continue
    if filename.endswith(".jpg"):
        file_open = str(melanoma_directory+"/"+ filename)
        file_open = file_open.replace('/', '\\')
        im = Image.open(file_open)
        width, height = im.size
        if(width != IMG_SIZE[0] or height != IMG_SIZE[1]):
            im = im.resize(IMG_SIZE, Image.LANCZOS)
            resizecount = resizecount+1
            removal = str(melanoma_directory + "/" + filename)
            removal = removal.replace('/', '\\')
            os.remove(removal)
            savename = str(melanoma_directory+"/"+ filename)
            savename = savename.replace('/', '\\')
            im.save(savename)
        continue
    else:
        continue

nevfilenames = os.listdir(nevus_directory)
for filename in tqdm(nevfilenames):
    if filename.startswith("aug"):
        removal = str(nevus_directory+"/"+ filename)
        removal = removal.replace('/', '\\')
        os.remove(removal)
        removalcount = removalcount + 1
        continue
    if filename.endswith(".jpg"):
        file_open = str(nevus_directory+"/"+ filename)
        file_open = file_open.replace('/', '\\')
        im = Image.open(file_open)
        width, height = im.size
        if(width != IMG_SIZE[0] or height != IMG_SIZE[1]):
            resizecount = resizecount+1
```

```
            im = im.resize(IMG_SIZE, Image.LANCZOS)
            removal = str(nevus_directory + "/" + filename)
            removal = removal.replace('/', '\\')
            os.remove(removal)
            savename = str(nevus_directory+"/"+ filename)
            savename = savename.replace('/', '\\')
            im.save(savename)
        continue
    else:
        continue

sebfilenames = os.listdir(seb_directory)
for filename in tqdm(sebfilenames):
    if filename.startswith("aug"):
        removal = str(seb_directory+"/"+ filename)
        removal = removal.replace('/', '\\')
        os.remove(removal)
        removalcount = removalcount + 1
        continue
    if filename.endswith(".jpg"):
        file_open = str(seb_directory+"/"+ filename)
        file_open = file_open.replace('/', '\\')
        im = Image.open(file_open)
        width, height = im.size
        if(width != IMG_SIZE[0] or height != IMG_SIZE[1]):
            resizecount = resizecount+1
            im = im.resize(IMG_SIZE, Image.LANCZOS)
            removal = str(seb_directory + "/" + filename)
            removal = removal.replace('/', '\\')
            os.remove(removal)
            savename = str(seb_directory+"/"+ filename)
            savename = savename.replace('/', '\\')
            im.save(savename)
        continue
    else:
        continue
print('Removed ', removalcount, ' files')
print('Resized ', resizecount, ' files')

# Data generators to augment images
train_datagen = ImageDataGenerator(
        # rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
```

```
        rotation_range=45)

test_datagen = ImageDataGenerator()
i=0
k=0
l=0

# Build validation set
mel_validation_files = os.listdir(melanoma_validation_directory)
nev_validation_files = os.listdir(nevus_validation_directory)
seb_validation_files = os.listdir(seb_validation_directory)

mel_valid_set = np.asarray(
    [pixels_from_path(os.path.join(melanoma_validation_directory, mel))
                            for mel in mel_validation_files]
)
nev_valid_set = np.asarray(
    [pixels_from_path(os.path.join(nevus_validation_directory, nev))
                            for nev in nev_validation_files]
)
seb_valid_set = np.asarray(
    [pixels_from_path(os.path.join(seb_validation_directory, seb))
                            for seb in seb_validation_files]
)

#Augment images
print('Generating some augmented images...')
aug_mel_directory = str('picscopy\\melanoma')
for batch in tqdm(train_datagen.flow(mel_train_pool,
                                     batch_size = 40,
                                     save_to_dir=aug_mel_directory,
                                     save_prefix="aug",
                                     save_format="jpg")):
    i += 1
    if i > len(mel_train_pool):
        print('done with MELANOMA augmentation')
        break

aug_nev_directory = str('picscopy\\nevus')
for batch in tqdm(train_datagen.flow(nev_train_pool,
                                     batch_size = 40,
                                     save_to_dir=aug_nev_directory,
                                     save_prefix="aug",
                                     save_format="jpg")):
    k += 1
```

```
    if k > len(nev_train_pool):
        print('done with NEVUS augmentation')
        break

aug_seb_directory = str('picscopy\\seborrheic-keratosis')
for batch in tqdm(train_datagen.flow(seb_train_pool,
                                     batch_size = 40,
                                     save_to_dir=aug_seb_directory,
                                     save_prefix="aug",
                                     save_format="jpg")):
    l += 1
    if l > len(seb_train_pool):
        print('done with SEBORRHEIC KERATOSIS augmentation')
        break


mel_train_set = []
nev_train_set = []
seb_train_set = []

# Build training sets from random samples
mel_files = random.sample(os.listdir(melanoma_directory), SAMPLE_SIZE)
nev_files = random.sample(os.listdir(nevus_directory), SAMPLE_SIZE)
seb_files = random.sample(os.listdir(seb_directory), SAMPLE_SIZE)
print("loading training melanoma set...")
mel_train_set = np.asarray(
    [pixels_from_path(os.path.join(melanoma_directory, mel))
     for mel in tqdm(mel_files)]
)
print("loading training nevus set...")
nev_train_set = np.asarray(
    [pixels_from_path(os.path.join(nevus_directory, nev))
     for nev in tqdm(nev_files)]
)
print("loading training seb set...")
seb_train_set = np.asarray(
    [pixels_from_path(os.path.join(seb_directory, seb))
     for seb in tqdm(seb_files)]
)

# Split training set into train/test
test_prop = 0.8
mel_test_set = mel_train_set[int(test_prop * len(mel_train_set)):]
mel_train_set= mel_train_set[:int(test_prop * len(mel_train_set))]
nev_test_set = nev_train_set[int(test_prop * len(nev_train_set)):]
```

```
nev_train_set = nev_train_set[:int(test_prop * len(nev_train_set))]
seb_test_set = seb_train_set[int(test_prop * len(seb_train_set)):]
seb_train_set = seb_train_set[:int(test_prop * len(seb_train_set))]

# Generate one-hot encoded labels for training set
x_train = np.concatenate([mel_train_set, nev_train_set, seb_train_set])
labels_train = np.asarray(
    [0 for _ in range(len(mel_train_set))]
    +[1 for _ in range(len(nev_train_set))]
    +[2 for _ in range(len(seb_train_set))]
)
labels_train = keras.utils.to_categorical(labels_train)

x_test = np.concatenate([mel_test_set, nev_test_set, seb_test_set])
    labels_test = np.asarray([0 for _ in range(len(mel_test_set))]
                         +[1 for _ in range(len(nev_test_set))]
                         +[2 for _ in range(len(seb_test_set))])
labels_test = keras.utils.to_categorical(labels_test)

x_valid = np.concatenate([mel_valid_set, nev_valid_set, seb_valid_set])
labels_valid = np.asarray([0 for _ in range(valid_size)]
                      +[1 for _ in range(valid_size)]
                      +[2 for _ in range(valid_size)])
labels_valid = keras.utils.to_categorical(labels_valid)

# Shuffle order of train/test/validation sets while maintaining label int
x_train_and_labels = list(zip(x_train, labels_train))
random.shuffle(x_train_and_labels)
x_train,labels_train = zip(*x_train_and_labels)
x_train = np.array(x_train)
labels_train = np.array(labels_train)

x_test_and_labels = list(zip(x_test, labels_test))
random.shuffle(x_test_and_labels)
x_test,labels_test = zip(*x_test_and_labels)
x_test = np.array(x_test)
labels_test = np.array(labels_test)

x_valid_and_labels = list(zip(x_valid, labels_valid))
random.shuffle(x_valid_and_labels)
x_valid,labels_valid = zip(*x_valid_and_labels)
x_valid = np.array(x_valid)
labels_valid = np.array(labels_valid)
```

## 8.4 Code: Neural Network Construction

```
model=Sequential()
opt = Adam(learning_rate = 0.001)
reg = l2(l=0.1)
alpha = 0.3

model.add(Conv2D(16,(3,3), padding = 'SAME'))
model.add(tf.keras.layers.LeakyReLU(alpha=alpha))
model.add(Conv2D(16,(3,3), padding = 'SAME'))
model.add(tf.keras.layers.LeakyReLU(alpha=alpha))
model.add(Conv2D(16,(3,3), padding = 'SAME'))
model.add(tf.keras.layers.LeakyReLU(alpha=alpha))

model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.2))

model.add(Conv2D(32,(3,3), padding = 'SAME'))
model.add(tf.keras.layers.LeakyReLU(alpha=alpha))
model.add(Conv2D(32,(3,3), padding = 'SAME'))
model.add(tf.keras.layers.LeakyReLU(alpha=alpha))
model.add(Conv2D(32,(3,3), padding = 'SAME'))
model.add(tf.keras.layers.LeakyReLU(alpha=alpha))

model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.2))

model.add(Flatten())

model.add(Dense(128))
model.add(Dropout(0.6))
model.add(Dense(64))
model.add(Dropout(0.6))

model.add(Dense(2, activation='sigmoid'))

model.compile(optimizer=opt,
              loss='binary_crossentropy',
              metrics=['accuracy'])

early_stop = keras.callbacks.EarlyStopping(monitor='val_accuracy', min_de
    patience=15, restore_best_weights = True)
reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=
    min_lr=0.00001,min_delta=0.01)
batch_size = 32
```

```
epochs = 500
steps_per_epoch = (len(x_train) // (batch_size))

history = model.fit(x_train,
                    labels_train,
                    batch_size=batch_size,
                    # shuffle = True,
                    epochs=epochs,
                    steps_per_epoch = steps_per_epoch,
                    callbacks=[early_stop, reduce_lr],
                    validation_data=(x_test, labels_test))

print(model.summary())

preds = model.predict(x_valid)
preds_class = np.argmax(model.predict(x_valid), axis=-1)

scores = model.evaluate(x_valid, labels_valid, verbose=1)
print("\nAccuracy: %.2f%%" % (scores[1]*100))

with open('output.txt', 'a') as output:
    output.write('Accuracy of %.2f %% recorded at %s ... Elapsed: %s \n'
                 % ((scores[1]*100),
                    datetime.now(),
                    str(datetime.now() - start_time)))
print("train_loss: ", history.history["loss"][-1])
print("train_acc: ", history.history["accuracy"][-1])
print("val_loss: ", scores[0])
print("val_accuracy: ", scores[1])
```