Computer Vision, Convolutional Neural Networks, and Breast Cancer Classification

Pro Kumar

University of Delaware

June 18, 2020

Contents

Abstract

The goal of this project is two-fold. The first goal is to explore the field of computer vision from its history to the usefulness of convolutional neural networks (CNNs). The second goal is to study the design of CNN architectures and to implement these ideas for the purpose of detecting the most common type of breast cancer, invasive ductal carcinoma (IDC). We use image data from "Deep Learning for Digital Pathology Image Analysis: A Comprehensive Tutorial With Selected Use Cases", a survey on digital pathology techniques conducted by Andrew Janowczyk and Anant Madabhushi in 2016. For this second goal, we start with a simple densely connected neural network for a baseline comparison. Much of the work of this project went into the design of ProNet, a CNN architecture made scratch using common design principles. Finally, we explore a scaled version of AlexNet implemented by Janowczyk and Madabhushi. The hope of this project is to inspire readers to pursue further study in the field on computer vision and deep learning.

1 Introduction

1.1 Computer Vision, a brief history

The following primarily comes from the lectures of Stanford University's course CS231n: Convolutional Neural Networks for Visual Recognition during the Spring 2017 semester [10].

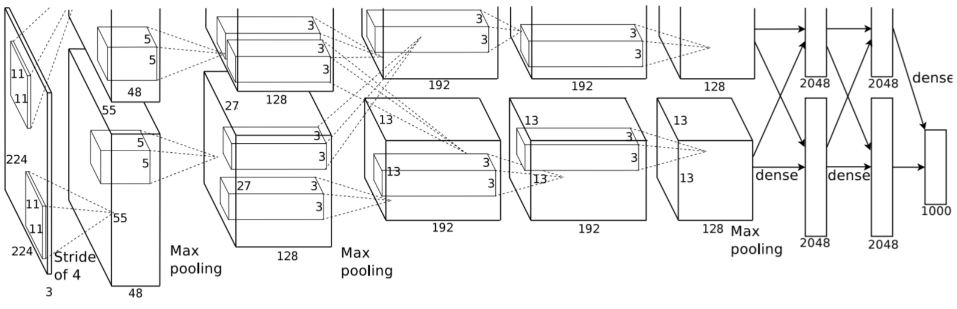


Figure 1: The AlexNet architecture which won the 2012 ImageNet object recognition competition. (Credit: Krizhevsky et al in [8].)

The course is taught by Fei-Fei Li, who is the "inaugural Sequoia Professor in the Computer Science Department at Stanford University, and Co-Director of Stanford's Human-Centered AI Institute" [14].

According to Fei-Fei Li, the history of vision begins about 543 million years ago (mya.) The Earth was mostly water and not so devoid of life, but the species were not so plenty, and not too exciting as vision had yet to arrive on the scene. Most animals consumed if food happened to be swimming by. Then roughly 540 mya, in a period of about 10 million years, there was an explosion of species; going from numbers of just a few to hundreds of thousands. This event is sometimes referred to as "Evolution's Big Bang". Dr. Andrew Parker, a zoologist at Oxford University, wrote a book titled *In the Blink of an Eye* in which he proposes the most viable theory for this event: animals began to develop a visual system, which began a sort of arms race for survival. Today, about 50% of the neurons in the human cortex are involved in visual processing and vision remains the primary sensory system for many species. So, how have humans used these neurons to develop artificial visual systems?

Mechanical visual systems probably began around the time the camera obscura was invented in 1545. The camera obscura uses an idea that is very similar to what biologists believe were primitive visual systems: basically a box with a pinhole that allows light through and "embed" on the screen at the back. Fast-forward to 1959, where two scientists, David Hubel and Torsten Wiesel, collaborated to determine the many of the most basic facts about mammalian visual systems; the most influential of these findings to modern deep learning came from their studies in recording neural activity in cats [4]. In 1963, Larry Roberts wrote what is considered to be the world's first PhD dissertation in the field of computer vision [11]. Through the course of the next 50 or so years, many works helped lay the bricks of computer vision and deep learning, from Fischler and Elschlager in 1973 who studied how to decompose complex structures in images [3], to Shi and Malik in 1997 who used graph theoretic ideas for image segmentation [12]. In 2009, Fei-Fei Li and her collaborators formulated ImageNet, one of the world's largest repositories of images which was created in hopes of inspiring innovation in the field through an open competition, the ImageNet Competition in object recognition. A major turning point came in 2012 when Alex Krizhevsky and collaborators won the challenge that year with what is considered to be the first practically viable convolutional neural network (CNN) called AlexNet [8]. See Figure 1 for the AlexNet architecture. And in each of the following years, deeper and deeper architectures involving CNNs have won the challenge. While CNNs were around for a while (see LeCun 1998 [9]), their usage only became prominent in the last year or so probably due to factors such as increased processing speeds and efforts towards data curation.

Why are CNNs so efficient and accurate? To explore this, let's first look at their mathematical structure.

1.2 Mathematics for CNNs

In this section we examine convolutions briefly as mathematical objects, and then develop bit of physical intuition to guide us as to how they are applied on a computer. Much of the following is based on exposition from Goodfellow [4] with insight taken from Bracewell [1], Chollet [2], Weisstein [17], and Wu [19].

When speaking of a convolution we usually mean a convolution of two functions.

Definition 1 *Let $\mathbb{R}$ represent the space of all real numbers. Suppose we have two (Riemann) integrable functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$. The* ***convolution*** *of f and g is a function*

$$(f * g)(x) := \int_{\mathbb{R}} f(t)g(x-t)dt. \tag{1}$$

How can we interpret a convolution of two functions and besides applications to CNNs, how are they used? From Bracewell [1] and Weisstein [17], physically, we can think of convolutions determining how much overlap exists between two functions as one function (g, in our definition) is "swept" over f. The "overlap" is determined by the product of f and g and the "sweeping" is determined by varying the integral with respect to t over all of $\mathbb{R}$.

In general, f will represent the input to the convolutional neural network, and g will represent the **kernel**, or **filter**, of the convolutions used. Each kernel is used to detect one main feature about the input data, and many kernels may be used in a different layers of a CNN.

Now, Definition 1 cannot be implemented as is on the computer since it is an integral over all space; we must discretize.

Definition 2 *Given two functions $f : \mathbb{Z} \to \mathbb{R}$ and $g : \mathbb{Z} \to \mathbb{R}$. The* ***discrete convolution*** *of f and g is the function*

$$(f \bullet g)(x) := \sum_{n=-\infty}^{\infty} f(n)g(x-n) \tag{2}$$

We still have infinities! So, suppose our function f is defined to be zero everywhere except a finite number of points. Then this definition yields a finite sum for the convolution of f with g. This is a reasonable supposition because, usually, in the context of machine learning, f represents input data, which of course is finite, and g represents some weights attached to the input instances.

The input and basic data structures of machine learning algorithms are tensors. Scalars are order-0 tensors (we shall say, 0-tensors), vectors (as represented by a single column/row) are order-1 tensors (1-tensors), matrices are order-2 (2-tensors), and this idea can be generalized to n dimensional arrays. In short, general $n-$dimensional tensors represent multilinear maps and their theory stretches every which way from research as purely mathematical objects to applications in general relativity and the ever-growing field of quantum computation. For the purposes of this project, we deal primarily with at most order-3 tensors.

A color image can be represented as a 3D numerical arrays given three axes (height, width, channels). The height and width are vectors in $\mathbb{R}^n$ and $\mathbb{R}^m$ respectively (each given by number of pixels n and m) and channels are vectors in $\mathbb{R}^3$. The images we deal with in this project are 3-tensors of size (50,50,3); that is, 50 pixels by 50 pixels, and each pixel is described by 3 RGB numbers giving its color. See Figure 2 for an example of a color image tensor.

Using these ideas, we can extend the definitions of convolution above to multidimensional case. For example, suppose we have a discrete 2-dimensional input $f(x, y)$ and a discrete 2-dimensional

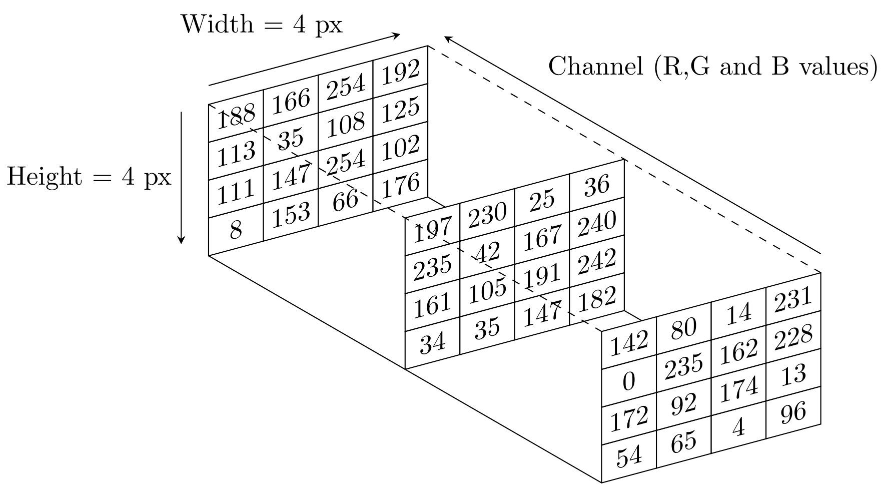


Figure 2: Example of an order-3 tensor representing a 4 × 4 color image. (Code to generate image adapted from [18].)

kernel $g(x, y)$, then the convolution of f and g is

$$(f * g) = \sum_{n,m=-\infty}^{\infty} f(x, y)g(x - n, y - m). \tag{3}$$

So how does convolution work with these tensors in practice? For simplicity, we consider a grayscale image and a convolutional kernel each given by a matrix (a 2-tensor). Typically, the kernel has a shape that is smaller than the image. So, consider the following image $A \in \mathbb{R}^{m\times n}$ and kernal $K \in \mathbb{R}^{p\times q}$ with $0 < p < m$ and $0 < q < n$.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \qquad K = \begin{bmatrix} k_{11} & \cdots & k_{1q} \\ \vdots & \vdots & \vdots \\ k_{p1} & \cdots & k_{pq} \end{bmatrix} \tag{4}$$

The first element (in the first row) of the resulting convolution matrix Z is formed by taking overlapping the kernel such that the top left corners of each matrix align. We take the product of the corresponding matrix elements and sum them. That is, the first element of the convolution is

$$Z_{1,1} = a_{1,1}k_{1,1} + \cdots a_{1,q}k_{1,q} + a_{2,1}k_{2,1} + \cdots + a_{2,q}k_{2,q} + \cdots + a_{p,1}k_{p,1} + \cdots + a_{p,q}k_{p,q} \tag{5}$$

The second element (in the first row) of the convolution matrix is taken by shifting the overlapped kernel one pixel to the right, so that we get

$$Z_{1,2} = a_{1,2}k_{1,1} + \cdots a_{1,(q+1)}k_{1,q} + a_{2,2}k_{2,1} + \cdots + a_{2,(q+1)}k_{2,q} + \cdots + a_{p,2}k_{p,1} + \cdots + a_{p,(q+1)}k_{p,q} \tag{6}$$

We can continue shifting ("sweeping") and computing in this fashion until the rightmost elements of the kernel align with the rightmost elements of the image. After calculating the last resulting convolution element of the first row, we can move the kernel one pixel down on the image and repeat the process of the calculating the convolution elements as we shift the kernel to the left. We do this until the the bottom right element k_{pq} of K aligns with the bottom right element a_{mn} of A. (Note: there is nothing special about moving the kernel in this fashion. We could have just as easily started at the bottom right corner of the image and made our way to the top left. This is a result of the algebraic properties of the convolution, which will not be of consequence for us.) So, in general, to compute the the $(i, j)-$th element of the convolution Z of A with K,

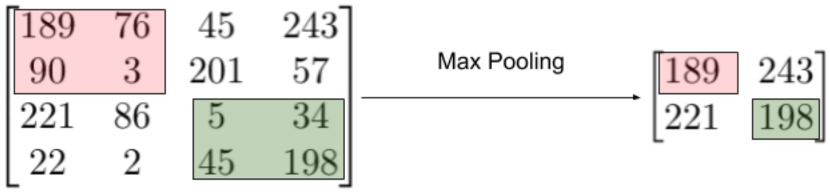


Figure 3: Example of the max pooling operation on one color channel of an image tensor. We divide the original 4×4 matrix into four 2×2 sub matrices, and take the maximum element of each to form the resulting "pooled" matrix.

we get

$$Z_{i,j} = \sum_{m=1}^{p} \sum_{n=1}^{q} a_{(i+m-1),(j+n-1)} \cdot k_{m,n} \tag{7}$$

At this point, we see the resemblance between equation (7) and equation (3). Using equation (7), we may code the convolution operation in the 2D case. This idea is easily extended to the 3D case, for example, when we're dealing with color images. We just add in a "sweep" over the third axis, which produces a third summation index in equation (7), and the convolution Z becomes three-dimensional.

While the above describes the action of convolution in terms of matrices, this action is usually the first of three main "sub-layers" of a CNN. The other two, in order of input propagation, are the **Pooling** layers and the **Fully Connected** layer. The latter is a network of densely connected units; that is, just basic NNs. For the former, the function of pooling "is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting" [4]. Let's see how it works.

Suppose we have our color image 3-tensor described by the axes (height, width, channel), where the height is h pixels, width is w pixels, and the three channels are (r, g, b). Consider only one of the channels, say r. Now, we can imagine dividing this matrix into n sub-matrices of the same size say (p, q), where p divides h, $p|h$, and $q|w$. Finally, we can perform our "pooling" operations. There are two commonly forms of pooling: max pooling and average pooling. The max pooling function simply takes the maximum value of entries in a sub-matrix to populate an output matrix. The average pooling function takes the average of the values of the sub-matrix. In each case, the output is a matrix of size $\frac{h}{p} \times \frac{w}{q}$. The pooling is then done for each channel so the final output 3-tensor has shape $\frac{h}{p} \times \frac{w}{q} \times 3$. See Figure 3 for an illustration of the max pooling function.

Based on the lectures from Stanford's CS231n course, in terms of how a CNN "learns", the beginning layers are usually meant to detect very small/fine features of our image like lines and dots. The next layers may detect slightly higher level features like corners, and as the image representation passes through deeper and deeper layers of convolutions/poolings, the networks takes in higher order concepts, like a face or hair, etc.

1.3 Benefits of CNNs

There are three main benefits of using CNNs: sparse connectivity, parameter sharing, and equivariance each of which are related ideas [4].

Sparse connectivity: Consider a simple neural network (NN) which has just an input layer and an output layer each containing n units and m units, respectively. Each unit in one layer is connected the each unit in the other layer, giving us a total of $n \times m$ connections/parameters to store/train. As we mentioned in the previous section, a convolution typically uses a kernel that is smaller than the input, let's suppose of size (k, k) where $k < n$ by at least one order of magnitude, then the total number of connections/parameters to store/train would be $n \times k$

(also, at least one order of magnitude smaller than previously.) And this is the benefit of sparse connectivity. If we want to detect more exciting features of our data set, we can include hidden layers between the input and output, which clearly depicts how quickly a plain neural network's computations/storage may get out of hand.

Parameter sharing: In a simple densely connected NN, we learn each of the parameters corresponding to each connection in the network, which using the same values from above comes out to $n \times m$ parameters. From the previous section, we notice that to compute the convolution of an image with some kernel, we simply shift the kernel tensor across the image tensor. That is, we are using the same matrix over the whole input; we only learn one set of parameters for all locations. In other words, we share the parameters across the image, which further benefits the storage benefits of the algorithm.

Equivariance: Because of parameter sharing, convolution is **equivariant** to translation. Mathematically, a function $f(x)$ is equivariant to another function g when $f(g(x)) = g(f(x))$. So, convolving and then translating an image will produce the same representation as first translating and then convolving it. This may be useful when we want to detect one feature across the entire image.

2 The breast cancer classification problem and dataset

In this project we use convolutional neural networks on histologic images of breast tissue to classify whether the tissue contains traces of invasive ductal carcinomas (IDCs), a type of breast cancer "growing in a milk duct and [that] has invaded the fibrous or fatty tissue of the breast outside of the duct. IDC is the most common form of breast cancer, representing 80 percent of all breast cancer diagnoses" [7].

The initial study of this project is based on Janowczyk and Madabhushi in 2016, who surveyed a few techniques for digital pathology [6].

The breast cancer data set was downloaded from Kaggle, but is originally part of a study by Janowczyk and Madabhushi [6] surveying machine learning techniques for digital pathology. According to the dataset page on Kaggle, "the original dataset consisted of 162 whole mount slide images of Breast Cancer (BCa) specimens scanned at 40x. From that, 277,524 patches of size 50 × 50 were extracted (198,738 IDC[(-)] negative and 78,786 IDC positive[(+)])" [16]. About 0.8% of the image patches in the entire data set are not of size 50 × 50, and are so discarded. Based on how the data set was annotated with patient ID numbers, there are 280 unique patient IDs. From this, we select a patient at random and selected their data patches as well as those of the next 7 patients. The reason for this is to prevent computational overload. Once the number of IDC(+) and IDC(-) cases were equalized from this data set, we worked with a total of 3198 image patches for this project. From this, we computed a train/test split ratio of 75:25 and performed validation on 25% of the train set.

Examples of the image patches with their respective labels are presented in Figure 4.

3 CNN Architectures: Process & Results

CNN architectures usually consist of an input layer, passing through a convolutional layer, an activation function, followed by a pooling layer, and finally through a fully connected layer. Common implementations use alternating convolutional and pooling layers followed by a fully connected layer. We apply three different NN architectures to the data set described. Namely, we use (1) the simple densely connected NN architecture from the introductory TensorFlow Tutorial, see [15]. We also use (2) a neural network made from scratch, called "ProNet" using several design principles from Goodfellow [4], Chollet [2], and CS231n [10]. Finally, we use

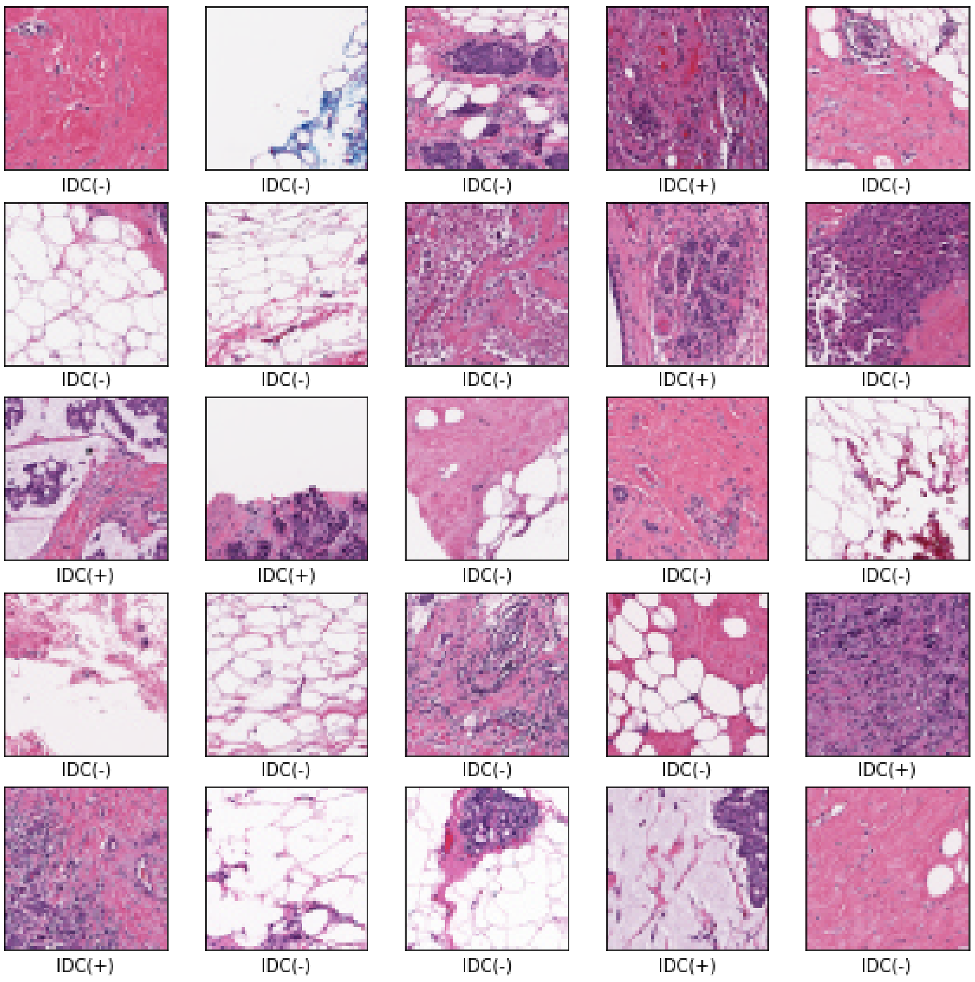


Figure 4: Examples of image patches from the breast cancer data set. Each image is labelled IDC(+) or IDC(-) depending on whether it contains cancer tissue or not, respectively.

(3) a slightly modified AlexNet architecture from Janowczyk and Madabhushi which has been optimized for smaller image patches [6] (please note, this AlexNet architecture is not the same as the one described into the introduction, but is based on Alex Krizhevsky's work.)

3.1 TensorFlow Tutorial (TFT) Network

The TFT network consists of just a fully connected NN, which we apply for baseline comparison with our other architectures. The TFT network consists of the following (which is summarized in Figure 5):

- One Flatten layer which takes our input tensor and flattens it into a 1-dimensional array by simple concatenation.
- One Dense layer of 128 units, with ReLU activation function.
- An output layer of 10 units, which was meant for the 10 classes this network was originally meant to classify certain pictures into. This layer was adjusted to simply have 1 unit based on the coding requirements for the labels which were just scalars.

This network was originally designed to take in grayscale images (2-tensors), however, since it was only a matter of adjusting the input shape, we also modified it to accept color images to

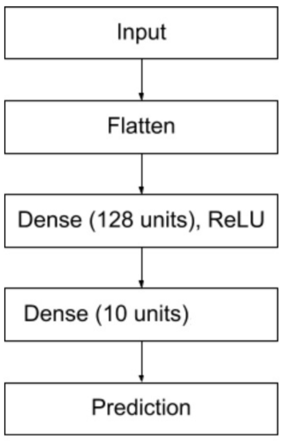


Figure 5: TFT network diagram. Note the last layer was adjusted to just have one unit based on how the labels were output as simple scalar.

determine how much the network learns from the color information of the image patches.

The networks were each compiled with an "Adam" optimizer (with the default learning rate of 10^{-4}), and a SparseCategoricalCrossentropy loss function. The models were training for 10 epochs without regard to which of the epochs yielded the best model (either in terms of maximum validation accuracy or minimum validation loss.) These were just the settings that came along with the TensorFlow Tutorial.

Testing our models, we achieve test accuracies of 57% and 77% after applying the TFT network on the grayscale and color test sets respectively.

No other work was done to improve this TFT architecture.

3.2 ProNet

The bulk of the work for this project went into designing ProNet, a CNN made from scratch using principles found in Goodfellow [4], Chollet [2], and CS231n [10]. The main process of designing this network involved tuning hyperparameters to create a network which overfit the data in a reasonable training period and then regularizing the model. Another goal was to create the smallest possible architecture (in terms of the number of the trainable parameters) that would reasonably fit our data.

Overfitting in this case was measured comparing the validation loss to the training loss; if there was a minimum in the plot of the validation loss followed by a leveling off/increase, then we achieved overfitting. Creating an overfitted model involved adding layers, adding more units, and training for many epochs. From this overfitted model, we applied regularization in the form of L1/L2 regularizations, removing layers and Dropout layers. (Dropout layers randomly set neurons to zero in the forward pass of the image representation. This in turn allows the model to not learn redundancy, as subsequent neurons would be forced to not rely on those "zeroed" units.) Another design consideration is zero-padding, where we padded the image edges with zero valued pixels for the convolutional kernels to take edge effects more into account.

For some of the compiling specifications, we used ReLU activations after each convolutional layer, up until the last fully connected layer which uses a sigmoid activation. We used an "Adam" optimizer with a binary crossentropy loss function as recommended by Chollet for binary classification problems. The most time-consuming part of this process and considered to be the most important by Goodfellow and CS231n was tuning the learning rate. This was done by observing the training and validation loss curves and comparing them with Figure 6, which comes from CS231n.

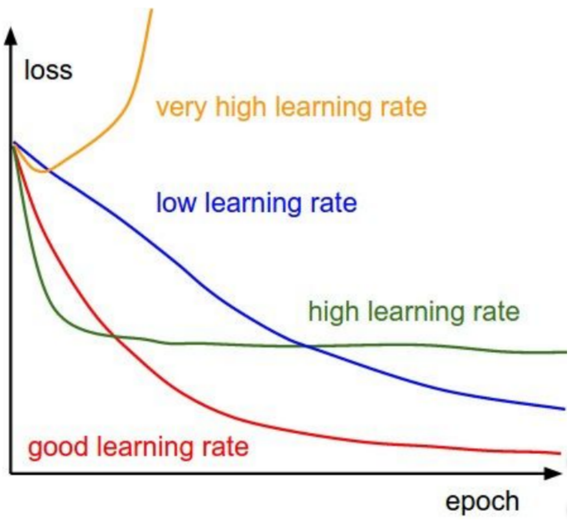
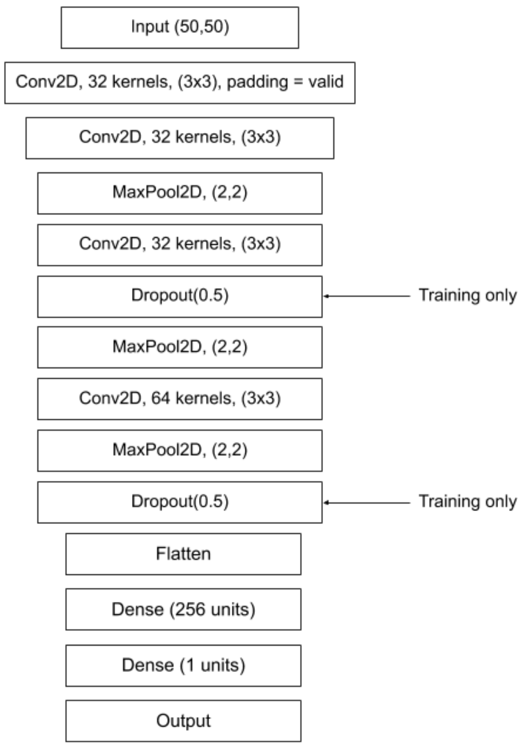


Figure 6: Plot of the effects of learning rate on the loss function during training epochs. Graph credit: Lecture 6 of CS231n in Spring 2017 [10].

We note that if we observed a somewhat linear, but decreasing learning rate, then the learning rate was too low. But if it there was a sudden drop followed by a flattening effect (or if the loss began increasing), then the learning rate was too high. Tuning this parameter, it was found that a value on the order of 10^{-5} was optimal for training.

The final architecture consists of the following (which is summarized in Figure 7):

- An input layer taking the shape (50,50,3) of the color image patches.
- A convolutional layer consisting of 32 kernels of size (3,3), with "same" padding (enough padding to maintain the height and width of the image representation), and ReLU activations.
- A convolutional layer of 32 kernels of size (3,3), ReLU activations.
- A Max Pooling layer with kernel of size (2,2) (i.e we essentially reduced our spatial representation, excluding depth, by half.)
- A convolutional layer of 32 kernels of size (3,3), with ReLU activation.
- Dropout layer (50%).
- MaxPooling with kernel size (2,2).
- Convolutional layer of 32 kernels of size (3,3), with ReLU activation. We included an L1-regularizer at this point.
- MaxPooling with kernel size (2,2).
- Dropout layer (50%).
- Dense (256 units), ReLU activation.
- Dense (1 unit), sigmoid activation.



(a) Summary of ProNet architecture.

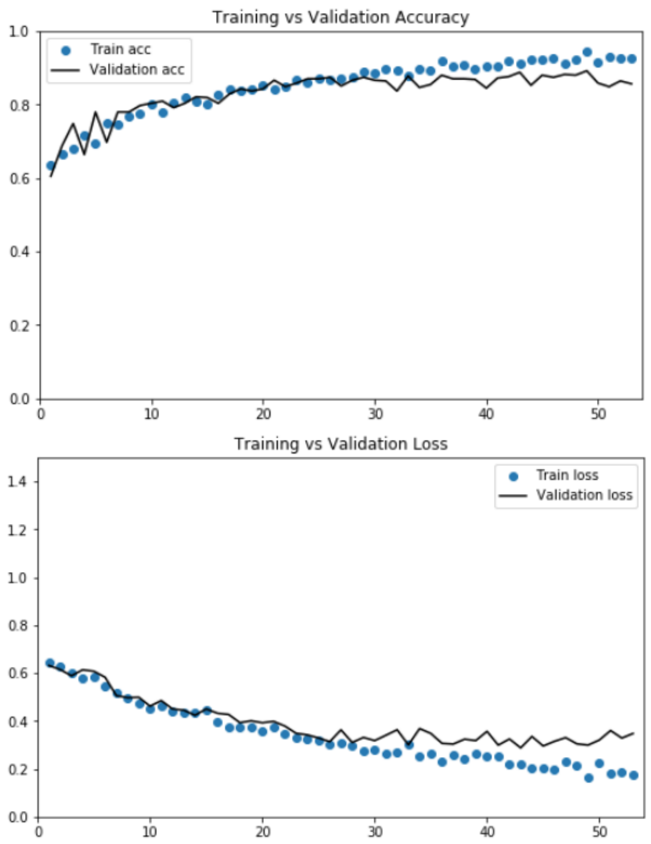


Figure 7: *(Left)* Summary of ProNet architecture. *(Right)* Training and Validation accuracy and loss curves.

This final architecture has a total of 64,033 trainable parameters.
The reason we have what seems like so many of the same convolutional layers, is because initially our numbers of kernels started at 16 and layer after layer we would double them, in accordance with the idea of learning higher order concepts as the image representation progressed. Of course, this helped us create models that overfit, however, the number of trainable parameters was becoming a bit too computationally expensive. So, in accordance with regularization and our goal of creating the smallest possible working model, we reduced the number of layers and the number of kernels in convolutional layers. One hypothesis as to why this architecture may work better is that the shapes of the cancerous regions are in essence “blobs”, which have no “higher order abstraction.” (At least none that we had previous knowledge of for the purposes of this project.) For example, if it was shown that cancerous regions form particular shapes or gather around tissue containing certain associated biological features, then the use of increasing the number of kernels further and further would be justified.

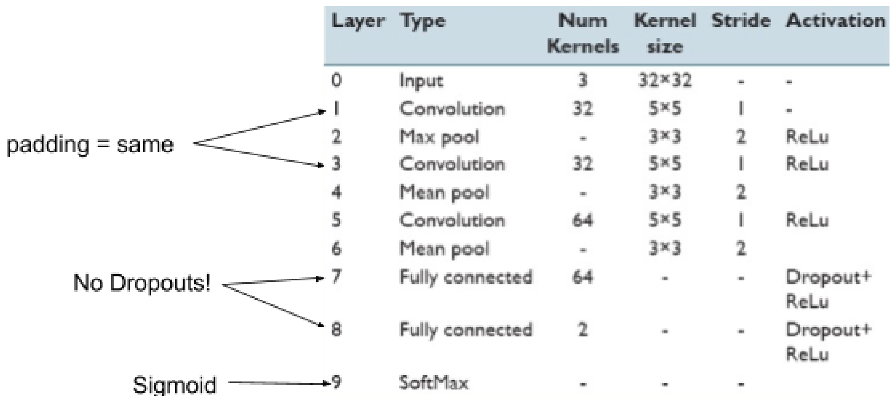
The training process involved two extremely useful Callback functions built into Keras: EarlyStopping and ModelCheckpoint. The former stops the training process automatically after a certain point, for example, we chose to stop the training after about 20 epochs passed the point where validation loss achieved a minimum. The latter was used to save the best trained model, in this case, based on the lowest validation loss.

The training curves for this architecture on our dataset are also presented in 7. We note these curves indeed still show over-fitting. So we could have included a bit more regularization to make the training more robust, however, as we progressed with the project, these training instances began to slow down the machine, so we needed to limit the number of epochs of training. Nonetheless, these curves show the potential to improve the models.

Testing our best ProNet model, we achieve a test accuracy of 89% on our color image test set

3.3 AlexNet

Janowczyck and Madabhushi outlined a simple AlexNet architecture which is optimized for smaller image sizes (32 × 32). The architecture and training curve are presented in Figure 8.


padding = same
No Dropouts!
Sigmoid

Layer	Type	Num Kernels	Kernel size	Stride	Activation
0	Input	3	32×32	-	-
1	Convolution	32	5×5	1	-
2	Max pool	-	3×3	2	ReLu
3	Convolution	32	5×5	1	ReLu
4	Mean pool	-	3×3	2	
5	Convolution	64	5×5	1	ReLu
6	Mean pool	-	3×3	2	
7	Fully connected	64	-	-	Dropout+ ReLu
8	Fully connected	2	-	-	Dropout+ ReLu
9	SoftMax	-	-	-	

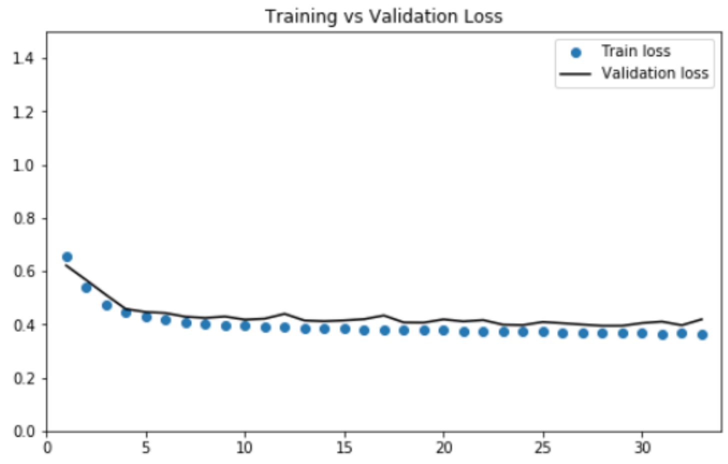


Figure 8: *(Top*: AlexNet Architecture from Janowczyck and Madabhushi. The labels to the left of the chart indicate the changes we made for implementation and overfitting. *(Bottom):* Validation curve for AlexNet. Note the number of epochs of training, 35.

The model was slightly adjusted for implementation. Namely, "same" padding was used in layers 1 and 3 which are the first two convolutional layers. We removed the Dropouts in layers 7 and 8, as we trying to overfit the data. Finally the Softmax layer at the end was replaced with a Sigmoid instead in accordance with the ProNet model above.

What are the benefits of this architecture? AlexNet has been shown to work for small image patches, as noted by Janowczyck and Madabhushi. This means we can train on more image patches for the same computational cost. It also only has about 4,000 more trainable parameters than the ProNet architecture, which does not significantly affect the computational capacity.

Testing our best AlexNet model, we achieve a test accuracy of 85% on our test color images.

4 Conclusion & Future Work

In terms of the train/test split that we used for our dataset, the best performance comes from ProNet (with a test accuracy of 89%), followed closely by AlexNet (test accuracy of 85%), and then the TensorFlow Tutorial fully connected network on our color images (test accuracy of 77%) and on the grayscaled image patches (test accuracy of 57%). However, by no means, do we claim generalizability of these models, for we have only trained on a portion of 9 patients' worth of data (out of a total of 280 patients.) For the curious reader, we performed a test to quantify this generalizability out of curiosity. We selected 30 random groups of 2 to 4 patients (outside of our training set) and evaluated our models on each. The results are summarized as follows:

- TFT (Grayscale) $\rightarrow 66 \pm 13$ test accuracy.
- TFT (RGB) $\rightarrow 77 \pm 11$ test accuracy.
- ProNet $\rightarrow 75 \pm 9$ test accuracy.
- AlexNet $\rightarrow 71 \pm 13$ test accuracy.

These results show the test accuracies are not significantly different. A better method of training would have been to randomly select an equal number of positive and negative cases from the entire patient data set. This would allow our models the learn at least some of the inherent variability between patients.

One of the biggest suggestions from Chollet, Goodfellow and CS231n would be to use pre-trained models or at least more proven architectures. Specifically, models which have been proven to work in the past. This was a little bit of the motivation of testing the version of AlexNet given by Janowczyck and Madabhushi, of course, our model was not pre-trained. The process of creating a CNN by scratch is lengthy; there are more hyperparameters to take into account than those presented in this project, and at times tuning one hyperparameter causes another to lose its tuning. Also, one should be careful of using proven architectures (without pre-training). For example, the AlexNet from the introduction which won the 2012 ImageNet challenge has upwards of 65 million trainable parameters, which would need much more computational power to train than was used in this project. To this end, one other path to explore would be using cloud computing technologies like those of Google [5], however these incur a financial cost.

In this project, we described the history of computer vision and convolutional neural networks. We described CNN's foundations in mathematics as well as the consequent benefits. And while we only applied a few architectures to one specific “simple” binary classification problem of detecting breast cancer from real image patches, we hope this shows, at the very least, the potential of CNNs and will lead the reader to further explore the field of computer vision.

5 Acknowledgements

Thank you Dr. Dominique Guillot on a wonderful semester in MATH 637! I looked forward to every class session, and ever minute of my project, and was deeply saddened towards the end when I knew it was ending. I will most certainly further explore the ideas in this report, as well as delve deeper into deep learning and artificial intelligence. Overall, I will continue bettering myself towards the standards you set in your class.

References

[1] R. Bracewell. The Fourier Transform and its Applications. Third Edition. International Editions. McGraw Hill. 2000.

[2] F. Chollet. Deep Learning with Python. Manning Publications Co. 2018.

[3] M. Fischler, R. Elschlager. The Representation and Matching of Pictorial Structures. IEEE Transactions on Computers. vol. C-22, no. 1, pp. 67-92, Jan. 1973, doi: 10.1109/T-C.1973.223602.

[4] I. Goodfellow, Y. Bengio, A. Courville. Deep Learning. The MIT Press. 2016. `http://www.deeplearningbook.org`.

[5] Google Cloud. GPUs Pricing. Compute Products. 2020. `https://cloud.google.com/compute/gpus-pricing`

[6] A. Janowczyk, A. Madabhushi. Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases. J Pathol Inform. 2016; 7: 29. 2016 Jul 26. doi: 10.4103/2153-3539.186902.

[7] Johns Hopkins Breast Center. Invasive Ductal Carcinoma (IDC) Breast Cancer. Johns Hopkins Medicine. 2017. `https://www.hopkinsmedicine.org/breast_center/breast\_cancers\_other\_conditions/invasive\_ductal\_carcinoma.html`

[8] A. Krizhevsky, I. Sutskever, G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. NIPS Proceedings. Neural Information Processing Systems Foundation. 2012. `https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`

[9] Y. LeCun, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324. `http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf`

[10] F. Li, J. Johnson, S. Yeung. CS231n: Convolutional Neural Networks for Visual Recognition. Stanford University. Spring 2017. `http://cs231n.stanford.edu/2017/`

[11] L. Roberts. Machine perception of 3-d solids. PhD Thesis. Dept. of Electrical Engineering. Massachusetts Institute of Technology. 1963. `http://hdl.handle.net/1721.1/11589`

[12] J. Shi, J. Malik. "Normalized cuts and image segmentation." Pattern Analysis and Machine Intelligence, IEEE Transactions on 22.8 (2000): 888-905.

[13] A. Parker. In the Blink of an Eye. The Free Press, Simon and Schuster. 2003.

[14] Stanford Profiles. Fei-Fei Li. Stanford University. `https://profiles.stanford.edu/fei-fei-li`

[15] TensorFlow. Basic classification: Classify images of clothing. 2020. `https://www.tensorflow.org/tutorials/keras/classification`

[16] P. Timothy. Breast Histopathology Images. Kaggle (Dataset). Updated 2018.

[17] Weisstein, Eric W. "Convolution." From MathWorld–A Wolfram Web Resource. `https://mathworld.wolfram.com/Convolution.html`

[18] M. Wibrow. Drawing Multidimension Array using TikZ. StackExchange (forum). https://tex.stackexchange.com/questions/295006/drawing-multidimensional-array-using-tikz

[19] J. Wu. Introduction to Convolutional Neural Networks. National Key Lab for Novel Software Technology (LAMBDA Group). Nanjing University, China. 2017. `https://pdfs.semanticscholar.org/450c/a19932fcef1ca6d0442cbf52fec38fb9d1e5.pdf`.