MAJOR PROJECT PART-1
15B19CI791

BREAST CANCER PREDICTION USING DEEP NEURAL NETWORK

MID EVALUATION REPORT

EVALUATION COMMITTEE:

Dr. Anuja Arora
Mr. Prashant Kaushik

MAJOR PROJECT SUPERVISOR:

Dr. Shikha Jain

S.NO.	NAMES	ENROLLMENT NUMBER	BATCH
1	Manya Agrawal	16103313	B9
2	Pranab Sharma	16103055	B9
3	Rohit Singh	16103258	B9

Index

Introduction

Breast cancer has the second highest mortality rate in women next to lung cancer. As per clinical statistics, 1 in every 8 women is diagnosed with breast cancer in their lifetime. However, periodic clinical check-ups and self-tests help in early detection and thereby significantly increase the chances of survival. Invasive detection techniques cause rupture of the tumor, accelerating the spread of cancer to adjoining areas. Hence, there arises the need for a more robust, fast, accurate, and efficient non-invasive cancer detection system.

Early detection can give patients more treatment options. In order to detect signs of cancer, breast tissue from biopsies is stained to enhance the nuclei and cytoplasm for microscopic examination. Then, pathologists evaluate the extent of any abnormal structural variation to determine whether there are tumors.

Architectural Distortion (AD) is a very subtle contraction of the breast tissue and may represent the earliest sign of cancer. Since it is very likely to be unnoticed by radiologists, several approaches have been proposed over the years but none using deep learning techniques.

AI will become a transformational force in healthcare and soon, computer vision models will be able to get a higher accuracy when researchers have the access to more medical imaging datasets.

The application of machine learning models for prediction and prognosis of disease development has become an irrevocable part of cancer studies aimed at improving the subsequent therapy and management of patients. The application of machine learning models for accurate prediction of survival time in breast cancer on the basis of clinical data is the main objective.

We have developed a computer vision model to detect breast cancer in histopathological images. Two classes will be used in this project: **Benign and Malignant.**

Problem Statement

Tumor classification in a clinical setting is traditionally assessed by pathologists. The most common method is to count mitotic figures (dividing cell nuclei) on hematoxylin & eosin (H&E) histological slides under a microscope. The pathologists will assign a mitotic score of 1, 2 or 3, where a score of 3 represents high tumor proliferation. Two other more objective methods that assess tumor proliferation in the breast include immunohistochemical staining for Ki67 protein. The lack of standardized procedures, debates about clinical utility, issues with Ki67 assay interpretation, and the complex molecular workflow to obtain gene expression has impeded the translation of Ki67 and PAM50 proliferation score for clinical use. Ki67 and PAM50 proliferation score are significantly associated with mitotic counts, but their agreement is not perfect. There are limited studies investigating the relationship between molecular and morphological data, specifically, none has explored the potential of predicting PAM50 proliferation scores from H&E WSIs. Although mitosis counting is routinely performed in most pathology practices, this highly subjective and labor-intensive task suffers from reproducibility problems.

One solution is to develop automated computational pathology systems to efficiently, accurately and reliably classify whether breast tumor is Malignant or Benign, on histopathological images. A large limitation of a lot of previous researches is that they focused solely on mitosis detection in predetermined tumor regions of interest (ROIs). However, in a real-world scenario, automatic mitosis detection should be performed in WSIs and an automatic method should ideally be able to produce a breast tumor proliferation score given a WSI as input.

Our primary goal is to develop algorithms to automatically predict whether breast cancer slide has Malignant or Benign tumor.

Understanding the architecture used

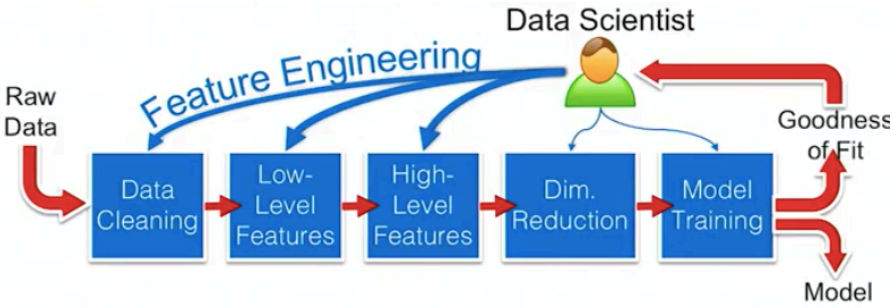
- ## DEEP LEARNING:

A good way to think about deep learning making this analogy- If traditional machine learning is a hammer, then deep learning is a nail gun. It's machine learning with power tools. And just like any other power tool, deep learning gives us the potential to be much more productive and to work much faster and dive deeper into a data science problem. Additionally, like any power tool it opens up new and exciting ways to explore, while avoiding the pitfalls.

Image Classification with traditional ML:

If we use traditional machine learning to do the image classification problem and remember we've formulated our problem as an image classification problem. The process would go roughly like:

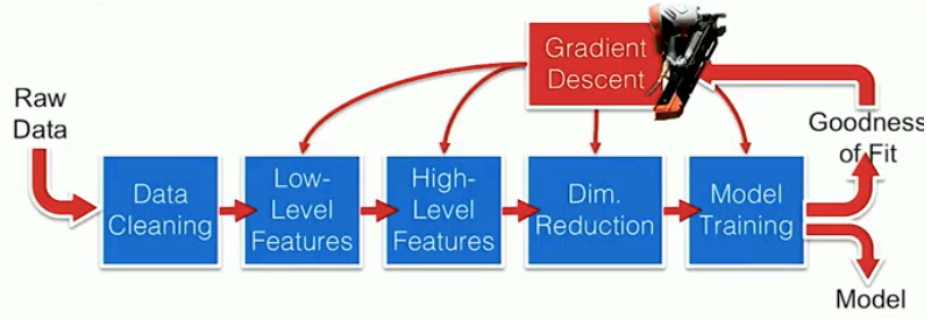
We start with a lot of raw data (in form of images) and we start by massaging that data, cleaning it up into a format that is meaningful for further analysis. Typically, after data cleaning we don't have images as files but rather arrays of integers. Then from this reformatted data, we extract low level features. Things such as edges, and then from those low-level features we can build a pile of features like shapes. At this point one typically has way too many features rooted to the number of labelled points, so we include some dimensionality reduction at that point in the pipeline and then we can start training the model. At the end of the process, we have the model and goodness of fit metrics. Usually at this point we realise that goodness of fit metric is not large enough for real time application.



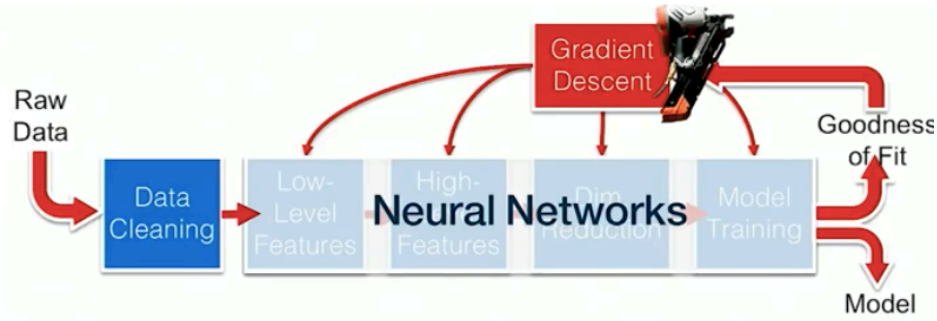
Therefore, we go back and tweak the individual stages, maybe add new code to the individual stages and the work is as concentrated on the feature related feature extraction part of this pipeline. Thus, we call this feature engineering. It's a time consuming and manual process.

Image Classification with traditional Deep Learning:

Now, with deep learning the picture changes. We take this manual feature engineering out of the picture and do that feature engineering task with power tools. And in this particular case the power tool we have used is gradient descent, otherwise known as hill climbing.



We've used gradient descent in this case not to do data cleaning, but to do the other tasks in the machine learning pipeline; feature extraction, dimensionality management and modeling itself. In order to be able to express these steps in such a way that we can apply gradient descent we need to be able to compute a gradient of this goodness of fit metric with respect to all the parameters that define those four processing steps and in practice what that means is these four processing steps become a chain of neural networks which taken together become a single deep neural network.

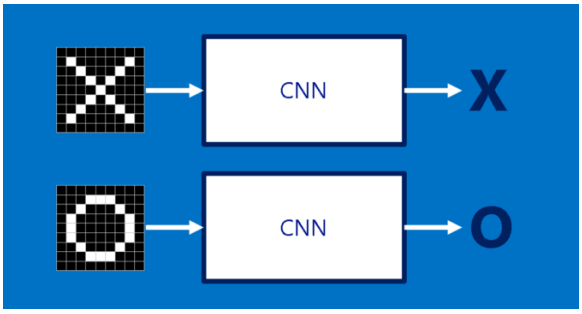


In principle, one we have reformulated the processing steps as a neural network the automated machinery comes into picture. We must remember that gradient descent is a power tool and will apply a great of deal of brute force in exactly the direction we point it at. It's critical to point it in right direction and there is always someone behind the scenes, constructing the neural network, data cleaning pipeline, so that when gradient descent is applied, we end up with a good model. If we don't properly construct everything, we may end up with a major risk of deep learning- **overfitting**. We've got this big neural network in four stages of processing and in effect what that has done is instead of having a small constrained modeling phase, we have a gigantic model which is going to have millions of parameters and the classifications are going to be very few. This is a guaranteed recipe for overfitting. Overfitting means that the **machine learning moments are going to discover correlations that just occurred because of pure random chance** because there are so many variables. The way that we avoid overfitting is with a process called **regularization** which means smoothing search space for the model restricting that search space, so that the areas of that search space

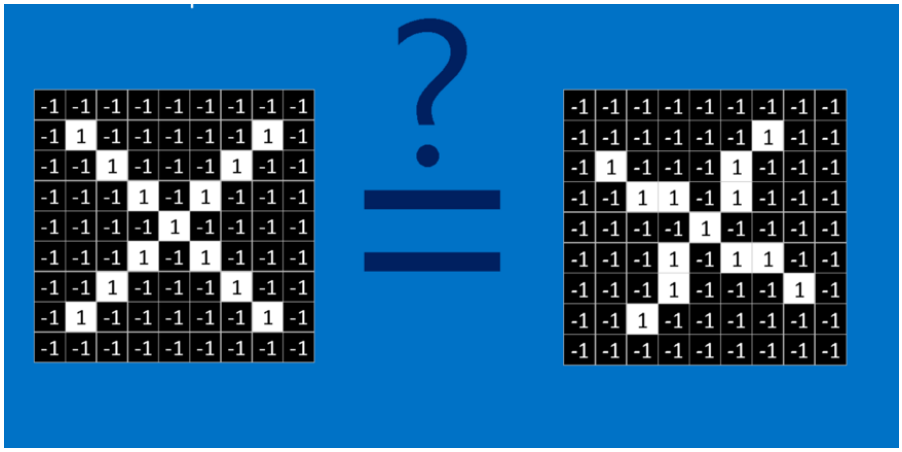
that lead that contain a lot of these false correlations are cut off. Traditionally we would do regularization with a regularization parameter that governs how high some of the model coefficients would be, or the sum of the squares. In case of deep learning we have to be much more aggressive and creative about performing regularization, to the point of using techniques that we wouldn't normally think of as regularization to regularize that search space. For example, **we might change the topology of the neural network or freeze parts of that topology so as to prevent them from producing and finding bad correlations**. If we find that there are certain classes of correlations and our input data is just leading to problems in prediction, we might add noise to the input data randomly perturb images so that we cancel out those problematic correlations. Lots of tricks that we need to apply here all about doing really creative regularization to avoid overfitting because that's the part where this power tool can shoot us on the foot, but it's a powerful technique that has allowed a lot of practitioners to get extremely positive results.

- # CONVOLUTIONAL NEURAL NETWORK (CNN):

:

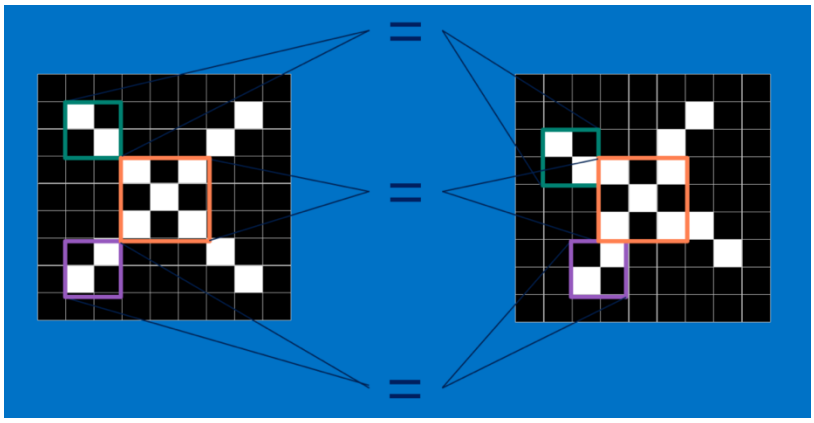


To explain Convolutional Neural Network, we'll stick with a very simplified example: determining whether an image is of an X or an O. Our CNN has one job. Each time we hand it a picture, it has to decide whether it has an X or an O. It assumes there is always one or the other.

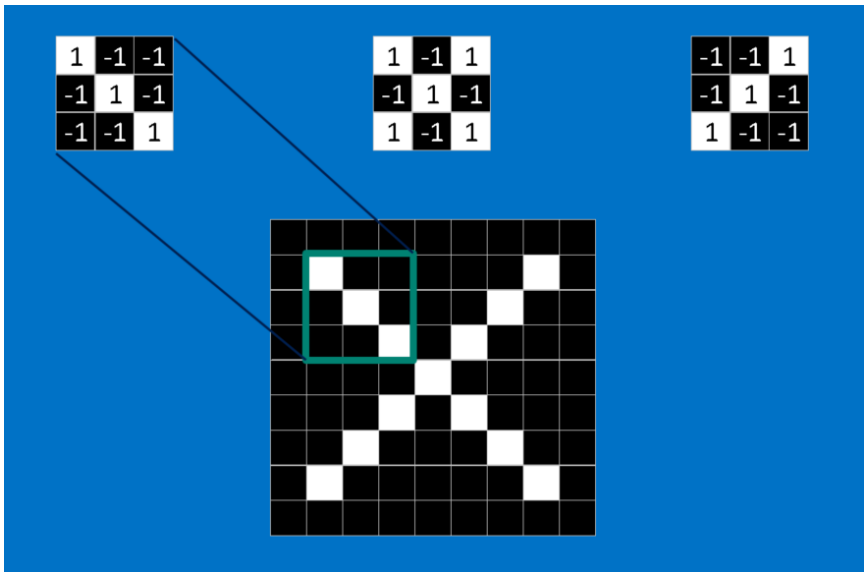


A naïve approach to solving this problem is to save an image of an X and an O and compare every new image to our exemplars to see which is the better match. What makes this task tricky is that computers are extremely literal. To a computer, an image looks like a two-dimensional array of pixels (think giant checkerboard) with a number in each position. In our example a pixel value of 1 is white, and -1 is black. When comparing two images, if any pixel values don't match, then the images don't match, at least to the computer. Ideally, we would like to be able to see X's and O's even if they're shifted, shrunken, rotated or deformed. This is where CNNs come in.

Features:

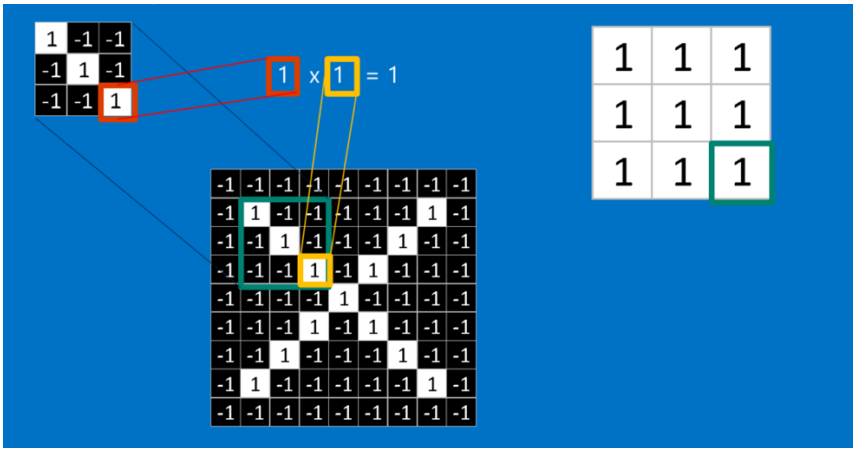


CNNs compare images piece by piece. The pieces that it looks for are called features. By finding rough feature matches in roughly the same positions in two images, CNNs get a lot better at seeing similarity than whole-image matching schemes.

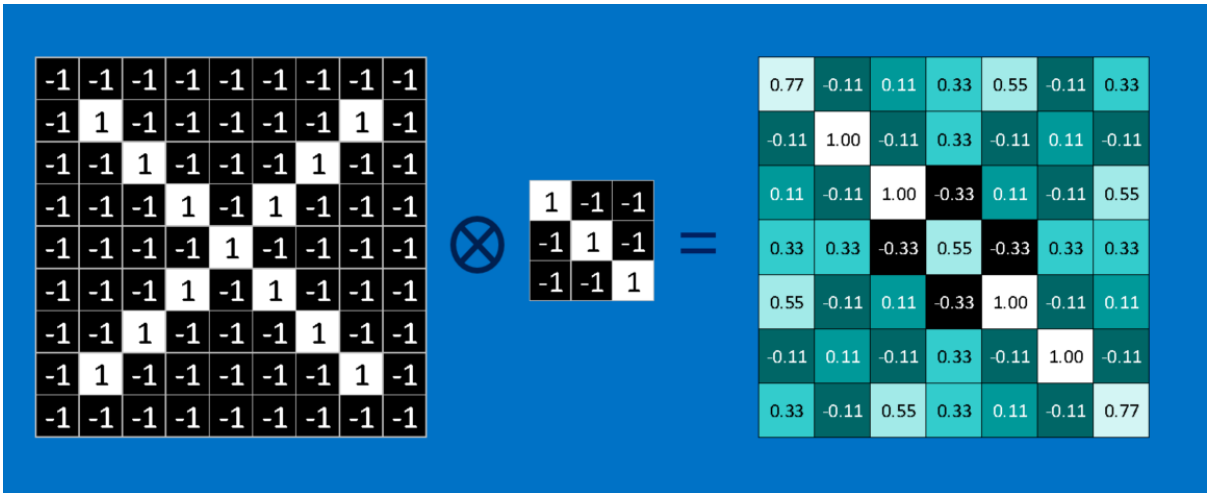


Each feature is like a mini-image—a small two-dimensional array of values. Features match common aspects of the images. In the case of X images, features consisting of diagonal lines and a crossing capture all the important characteristics of most X's. These features will probably match up to the arms and center of any image of an X.

Convolution:

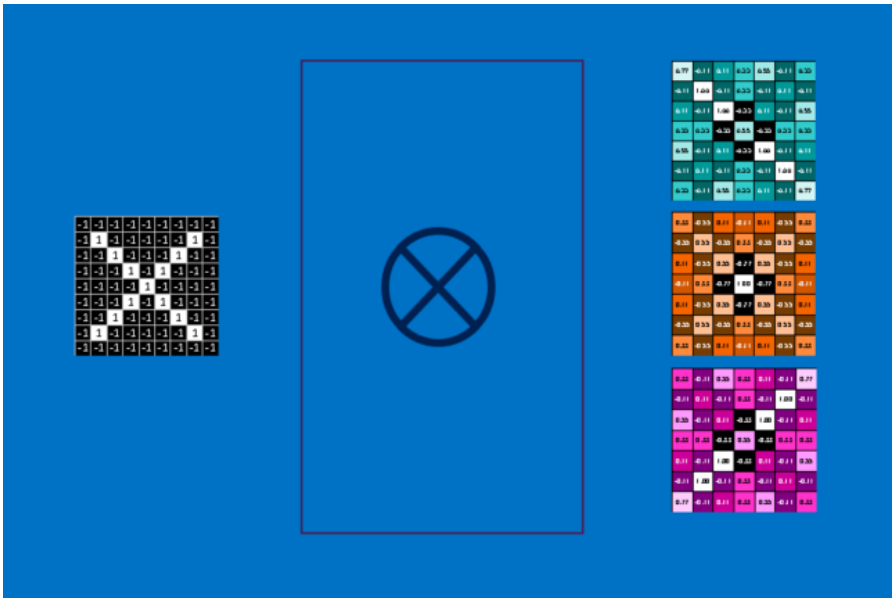


When presented with a new image, the CNN doesn't know exactly where these features will match so it tries them everywhere, in every possible position. In calculating the match to a feature across the whole image, we make it a filter. The math we use to do this is called convolution, from which Convolutional Neural Networks take their name.
The math behind convolution is nothing that would make a sixth-grader uncomfortable. To calculate the match of a feature to a patch of the image, simply multiply each pixel in the feature by the value of the corresponding pixel in the image. Then add up the answers and divide by the total number of pixels in the feature. If both pixels are white (a value of 1) then 1 * 1 = 1. If both are black, then (-1) * (-1) = 1. Either way, every matching pixel results in a 1. Similarly, any mismatch is a -1. If all the pixels in a feature match, then adding them up and dividing by the total number of pixels gives a 1. Similarly, if none of the pixels in a feature match the image patch, then the answer is a -1.



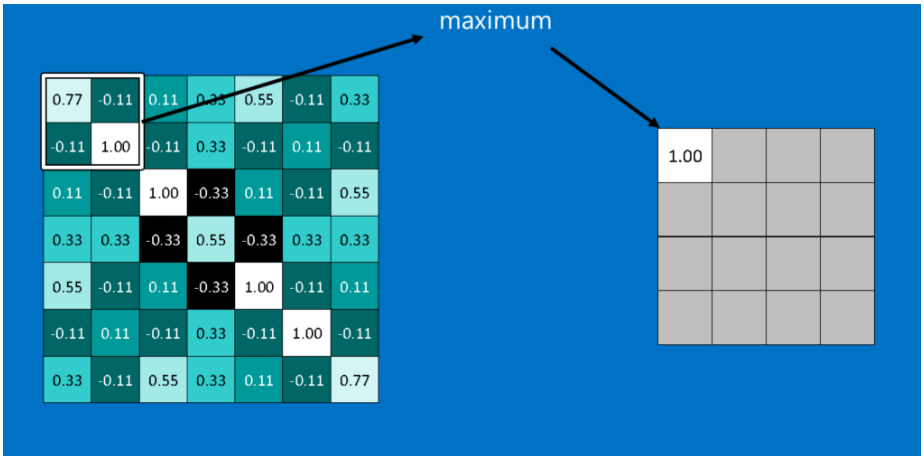
To complete our convolution, we repeat this process, lining up the feature with every possible image patch. We can take the answer from each convolution and make a new two-dimensional array from it, based on where in the image each patch is located. This map of matches is also a filtered version of our original image. It's a map of where in the image the feature is found. Values close to 1 show strong matches, values close to -1 show strong

matches for the photographic negative of our feature, and values near zero show no match of any sort.

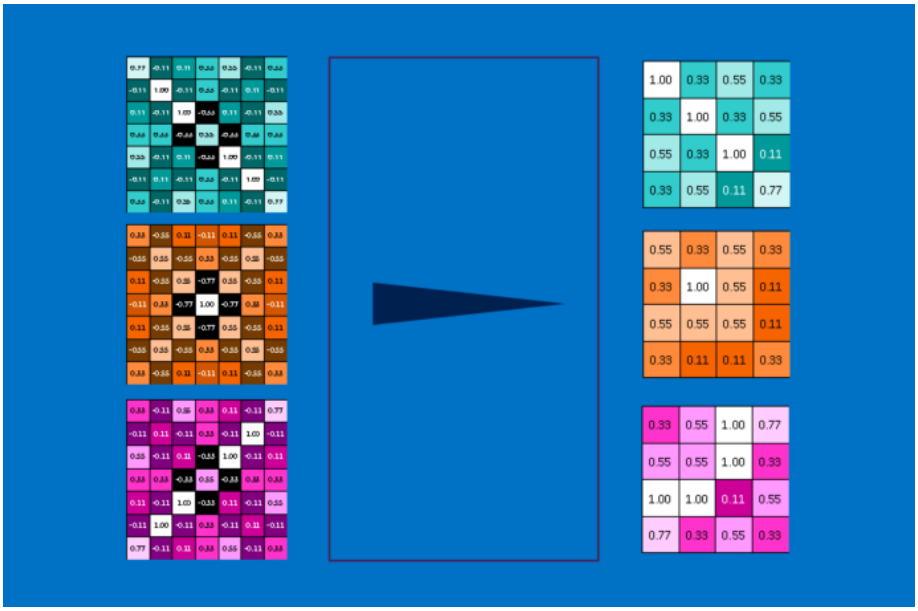


The next step is to repeat the convolution process in its entirety for each of the other features. The result is a set of filtered images, one for each of our filters. It's convenient to think of this whole collection of convolution operations as a single processing step. In CNNs this is referred to as a convolution layer, hinting at the fact that it will soon have other layers added to it.
It's easy to see how CNNs get their reputation as computation hogs. Although we can sketch our CNN on the back of a napkin, the number of additions, multiplications and divisions can add up fast. In math speak, they scale linearly with the number of pixels in the image, with the number of pixels in each feature and with the number of features. With so many factors, it's easy to make this problem many millions of times larger without breaking a sweat. Small wonder that microchip manufacturers are now making specialized chips in an effort to keep up with the demands of CNNs.

Pooling:

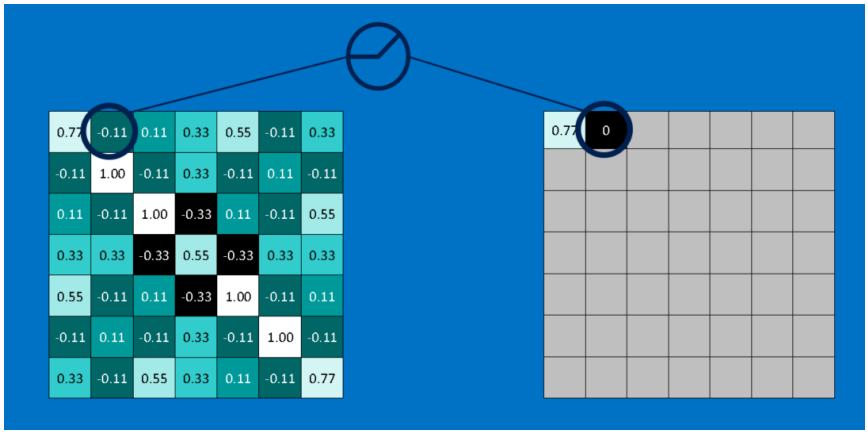


Another power tool that CNNs use is called pooling. Pooling is a way to take large images and shrink them down while preserving the most important information in them. The math behind pooling is second-grade level at most. It consists of stepping a small window across an image and taking the maximum value from the window at each step. In practice, a window 2 or 3 pixels on a side and steps of 2 pixels work well.
After pooling, an image has about a quarter as many pixels as it started with. Because it keeps the maximum value from each window, it preserves the best fits of each feature within the window. This means that it doesn't care so much exactly where the feature fit as long as it fit somewhere within the window. The result of this is that CNNs can find whether a feature is in an image without worrying about where it is. This helps solve the problem of computers being hyper-literal.

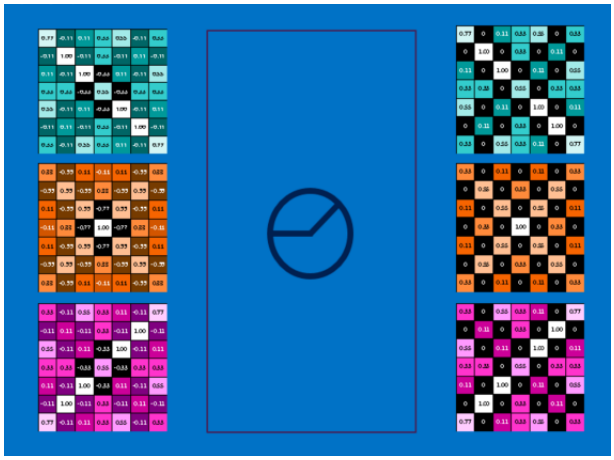


A pooling layer is just the operation of performing pooling on an image or a collection of images. The output will have the same number of images, but they will each have fewer pixels. This is also helpful in managing the computational load. Taking an 8 megapixel image down to a 2 megapixel image makes life a lot easier for everything downstream.

Rectified Linear Units (ReLU):

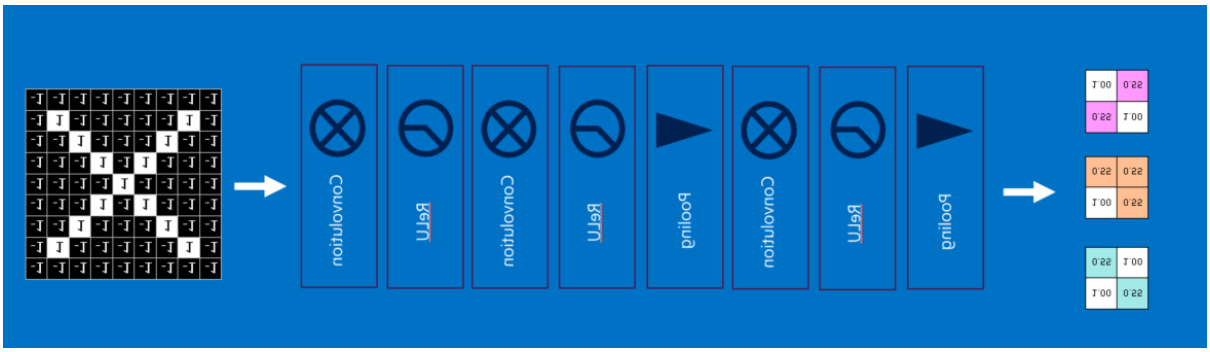


A small but important player in this process is the Rectified Linear Unit or ReLU. It's math is also very simple—wherever a negative number occurs, swap it out for a 0. This helps the CNN stay mathematically healthy by keeping learned values from getting stuck near 0 or blowing up toward infinity. It's the axle grease of CNNs—not particularly glamorous, but without it they don't get very far.

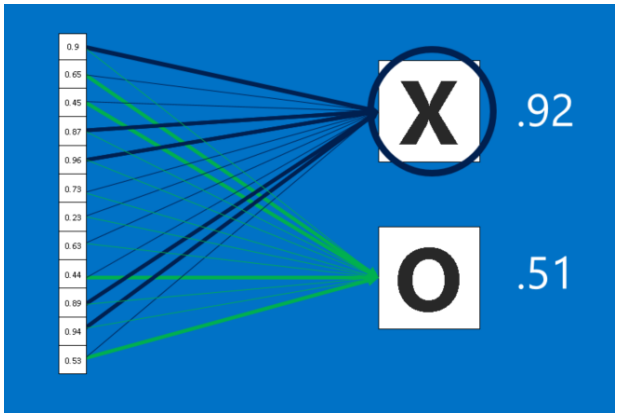


The output of a ReLU layer is the same size as whatever is put into it, just with all the negative values removed.

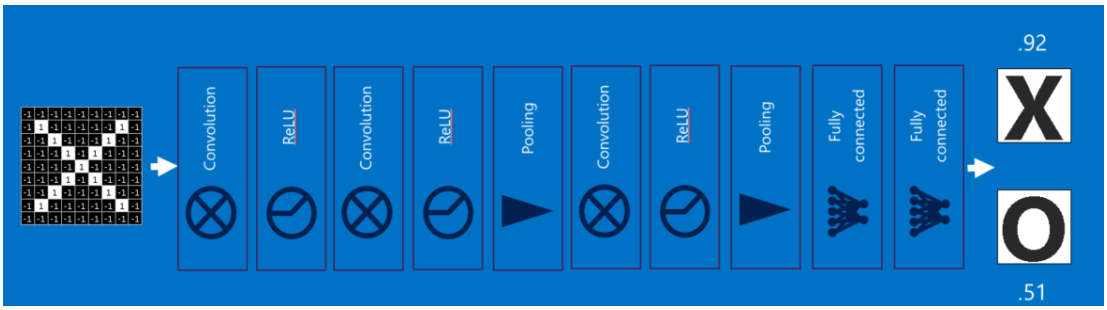
Adding up → Deep learning:



Fully connected layers:



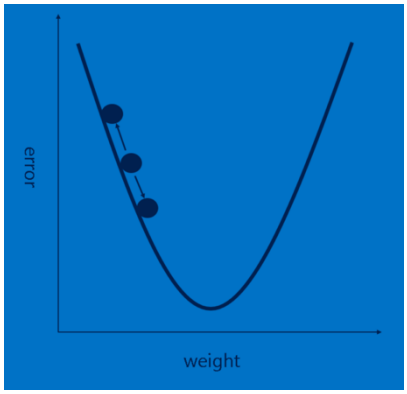
CNNs have one more arrow in their quiver. Fully connected layers take the high-level filtered images and translate them into votes. In our case, we only have to decide between two categories, X and O. Fully connected layers are the primary building block of traditional neural networks. Instead of treating inputs as a two-dimensional array, they are treated as a single list and all treated identically. Every value gets its own vote on whether the current image is an X or and O. However, the process isn't entirely democratic. Some values are much better than others at knowing when the image is an X, and some are particularly good at knowing when the image is an O. These get larger votes than the others. These votes are expressed as weights, or connection strengths, between each value and each category.
When a new image is presented to the CNN, it percolates through the lower layers until it reaches the fully connected layer at the end. Then an election is held. The answer with the most votes wins and is declared the category of the input.



Fully connected layers, like the rest, can be stacked because their outputs (a list of votes) look a whole lot like their inputs (a list of values). In practice, several fully connected layers are often stacked together, with each intermediate layer voting on phantom "hidden" categories. In effect, each additional layer lets the network learn ever more sophisticated combinations of features that help it make better decisions.

Backpropagation:

Our story is filling in nicely, but it still has a huge hole—Where do features come from? and How do we find the weights in our fully connected layers? If these all had to be chosen by hand, CNNs would be a good deal less popular than they are. Luckily, a bit of machine learning magic called backpropagation does this work for us.
To make use of backpropagation, we need a collection of images that we already know the answer for. This means that some patient soul flipped through thousands of images and assigned them a label of X or O. We use these with an untrained CNN, which means that every pixel of every feature and every weight in every fully connected layer is set to a random value. Then we start feeding images through it, one after other.



Each image the CNN processes results in a vote. The amount of wrongness in the vote, the error, tells us how good our features and weights are. The features and weights can then be adjusted to make the error less. Each value is adjusted a little higher and a little lower, and the new error computed each time. Whichever adjustment

makes the error less is kept. After doing this for every feature pixel in every convolutional layer and every weight in every fully connected layer, the new weights give an answer that works slightly better for that image. This is then repeated with each subsequent image in the set of labeled images. Quirks that occur in a single image are quickly forgotten, but patterns that occur in lots of images get baked into the features and connection weights. If you have enough labeled images, these values stabilize to a set that works pretty well across a wide variety of cases.
As is probably apparent, backpropagation is another expensive computing step, and another motivator for specialized computing hardware.

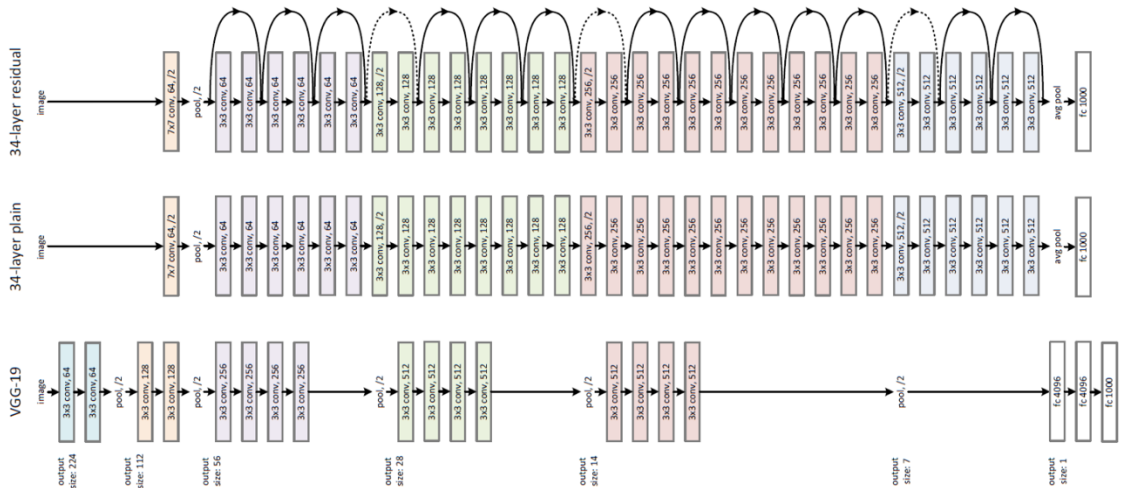
- # TRANSFER LEARNING & ResNet152 MODEL: **(Pertaining to its application)**

We decided to use the pre-trained **Resnet-152** to extract features for classification.

Resnet-152 is a type of specialized neural network that helps to handle more sophisticated deep learning tasks and models. It has received quite a bit of attention at recent IT conventions, and is being considered for helping with the training of deep networks. Resnet introduces a structure called residual learning unit to alleviate the degradation of deep neural networks. This unit's structure is a feedforward network with a shortcut connection which adds new inputs into the network and generates new outputs. The main merit of this unit is that it produces better classification accuracy without increasing the complexity of the model

Once we have our dataset, we're ready to start training. For an image classification problem, CNNs work well. A CNN as compared to other neural networks or models is that it's a model that takes into account spatial structure, therefore it works well for images. While we can build these up by scratch, if we plan to apply it to a certain problem, it can be helpful to start with a model that's been built already, and trained on a benchmark dataset for which there exists the maths, a model implementation and even the weights that were trained in the experiments for which the research was done. If we start with that, we can manipulate that model and then massage the weights for our particular problem. This lets you take a model that's quite large and it's very powerful but start it at a good initial starting point and by point and by only training certain portions of it, we can minimise overfitting but still manage to get the benefits.

In our case we have used a ResNet152 model which is a residual neural network with 152 layers. The diagram shown below is at the 34 layer variant, but they're quite similar:



If we grab this network with the pretrained weights, we find that the task for which it was trained on was ImageNet which has a thousand classes. We have two classes and they're not cats, dogs and buildings. We start with the pretrained model, and we rip off the end with the thousand class classifier and replace it with our own two class classifier. After this replacement, we're starting with a random and final layer. We don't want to destroy the rest of the pretrained weights, while we're trying to get that search for a good parameter so that final layer. Therefore we will freeze the existing model and we train just the new classifier we've placed on top of the model. We're technically doing a surgery on the pre-made model. This lets us get a final layer into a good search base. The final classifier outputs will tell us total probability of Benign and Malignant tumors upon each individual image validated on to the network.

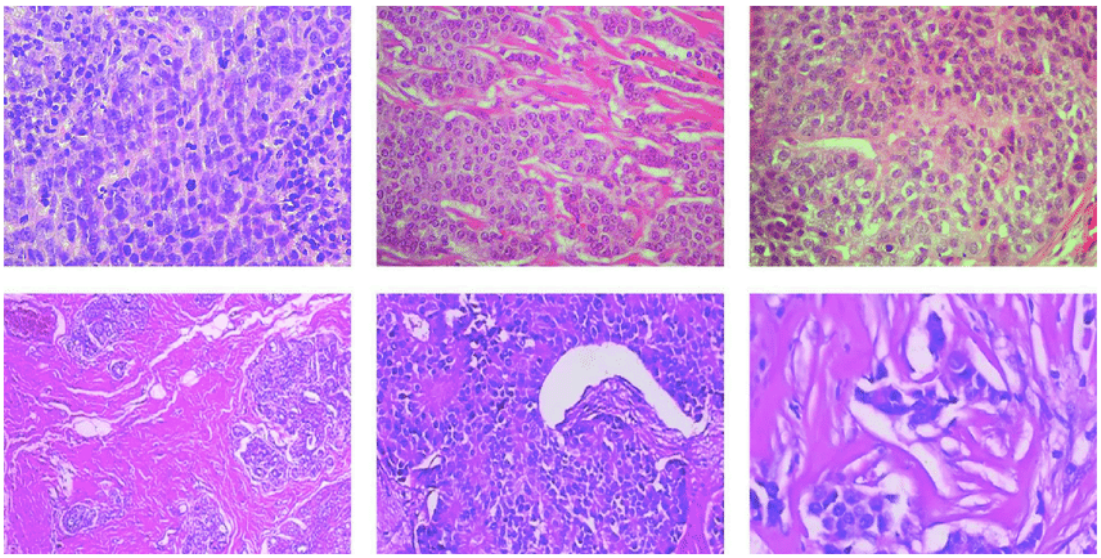
Work plan & flow
(till current project development)

- DATASET USED:

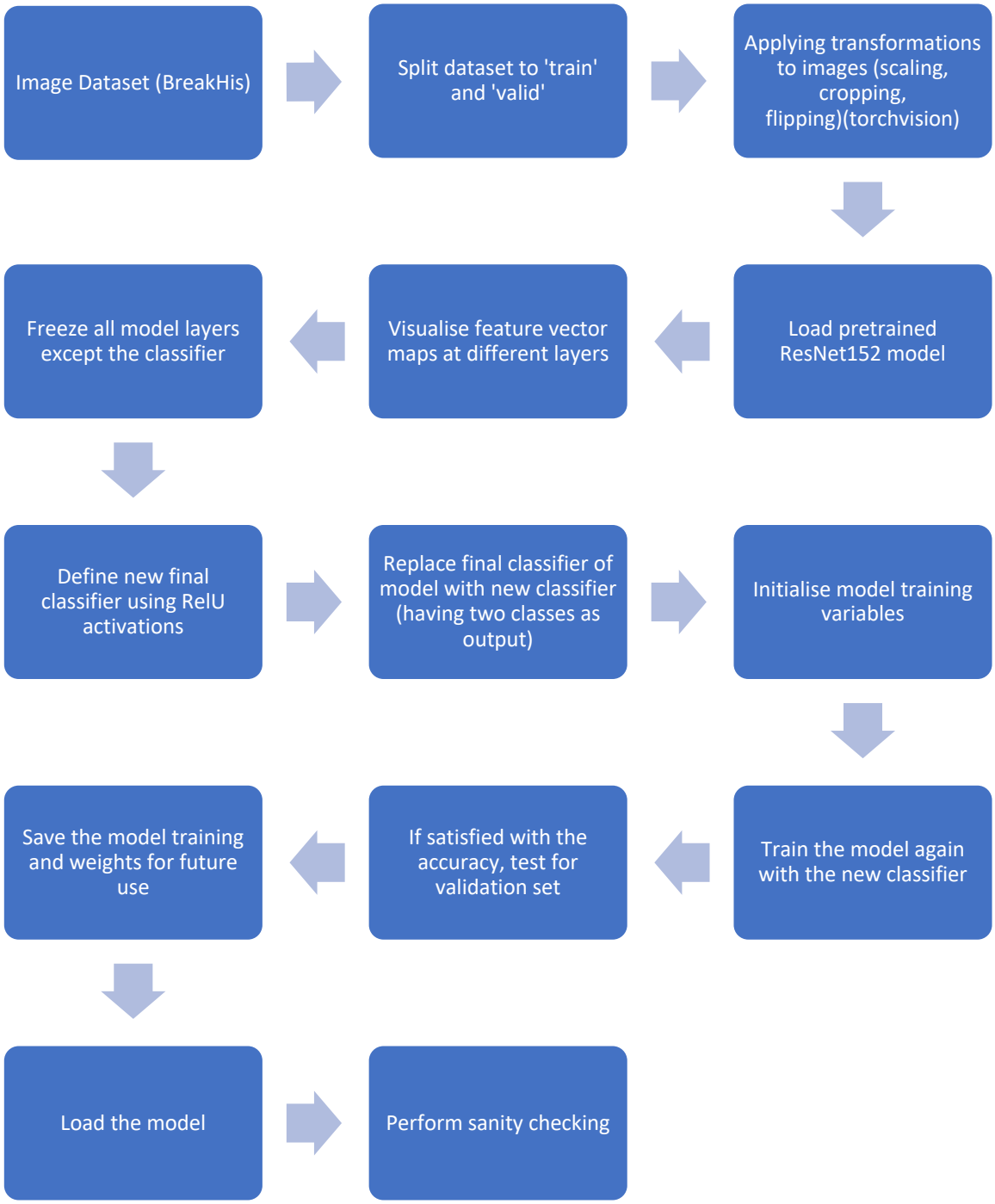
The Breast Cancer Histopathological Image Classification (BreakHis) is composed of 7,909 microscopic images of breast tumor tissue collected from 82 patients using different magnifying factors (40X, 100X, 200X, and 400X). To date, it contains 2,480 benign and 5,429 malignant samples (700X460 pixels, 3-channel RGB, 8-bit depth in each channel, PNG format). This database has been built in collaboration with the P&D Laboratory - Pathological Anatomy and Cytopathology, Parana, Brazil.

The dataset BreaKHis is divided into two main groups: benign tumors and malignant tumors. Histologically benign is a term referring to a lesion that does not match any criteria of malignancy - e.g., marked cellular atypia, mitosis, disruption of basement membranes, metastasize, etc. Normally, benign tumors are relatively "innocents", presents slow growing and remains localized. Malignant tumor is a synonym for cancer: lesion can invade and destroy adjacent structures (locally invasive) and spread to distant sites (metastasize) to cause death.

The dataset currently contains four histological distinct types of benign breast tumors: adenosis (A), fibroadenoma (F), phyllodes tumor (PT), and tubular adenona (TA); and four malignant tumors (breast cancer): carcinoma (DC), lobular carcinoma (LC), mucinous carcinoma (MC) and papillary carcinoma (PC).



- # DESIGN (FLOWCHART):



- # FAMILIARITY WITH TOOLS AND EQUIPMENTS USED:

Pytorch:

PyTorch is a Python package that provides two high-level features which include Tensor computation (like NumPy) with strong GPU acceleration and Deep neural networks built on a tape-based autograd system. Tensors are nothing but multidimensional arrays. Tensors in PyTorch are similar to numpy's ndarrays, with the addition being that Tensors can also be used on a GPU.

What exactly are tensors ?

- A tensor is a **vector** or **matrix** of n-dimensions that represents all types of data. All values in a tensor hold identical data type with a known (or partially known) **shape**. The shape of the data is the dimensionality of the matrix or array.

PyTorch uses an imperative / eager paradigm. That is, each line of code required to build a graph defines a component of that graph. We can independently perform computations on these components itself, even before your graph is built completely. **This is called "define-by-run" methodology.**
It has been well documented that PyTorch outperforms its competition in having lowest median time per epoch while training LSTMs. It is built to be deeply integrated into Python.

TensorFlow:

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. TensorFlow is essentially a library developed by the Google Brain Team to accelerate machine learning and deep neural network research.

Tensorflow architecture works in three parts:

- Preprocessing the data
- Build the model
- Train and estimate the model

It is called Tensorflow because it takes input as a multi-dimensional array, also known as tensors as discussed above. TensorFlow offers multiple levels of abstraction. Tensorflow makes use of graphs framework as well. Tensorflow also offers options to load data dynamically which have really large sizes, The pipeline will load the data in batches. TensorFlow is essentially used by us to scale the models in close conjunction with Keras.

OpenSlide:

OpenSlide is a C library that provides a simple interface to read whole-slide images. OpenSlide has been supported by the National Institutes of Health and the Clinical and Translational Science Institute at the University of Pittsburgh. The library can read Aperio, Hamamatsu, Leica, MIRAX, Sakura, Trestle, and Ventana formats, as well as TIFF files that conform to a simple convention. This library reads whole slide image files (also known as virtual slides). It provides a consistent and simple API for reading files from multiple vendors.In our application OpenSlide is used to read the WSI Images into the program.

Sci-Kit Learn:

Scikit-learn is a free software machine learning library for the Python programming language. It is essentially built on The library is built upon the SciPy (Scientific Python) that must be installed before you can use scikit-learn. This stack that includes:

- NumPy: Base n-dimensional array package
- SciPy: Fundamental library for scientific computing
- Matplotlib: Comprehensive 2D/3D plotting
- IPython: Enhanced interactive console
- Sympy: Symbolic mathematics
- Pandas: Data structures and analysis

The library is focused on modeling data. It is not focused on loading, manipulating and summarizing data. In our application we make use of scikit-image library, scikit-image is an open-source image processing library for the Python programming language. It includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

Keras:

Keras is an open-source neural-network library written in Python. It is capable of running on top of TensorFlow and others. It is essentially used for high level front end for the deep learning modesl.It is a fantastic API for Neural Nets.

Apache Spark and Apache System ML:

Apache Spark is an open-source distributed general-purpose cluster-computing framework. It helps the data science applications more business ready. *Apache Spark* is the open standard for flexible in-memory data processing that enables batch, real-time, and advanced analytics. It helps us have benefits over traditional machine learning methods. It is used for data cleaning and preparation in our application and ease the entire process.

Apache SystemML is a flexible machine learning system that automatically scales to Spark and Hadoop clusters. It is highly customizable with python. It is essentially focused on linear algebra that means one can write their own equations with ease. All these systems allows us to tile the images and prepare the images and process them using ResNets. By training only specific regions of it, we can minimize overfitting and still able to get the results that we require.

- # UNDERSTANDING THE CODE FLOW (+screenshots of outputs):

The code is developed on python language, with its functioning and testing being done on Jupyter notebook via Anaconda Navigator.

We begin with loading the python packages on the notebook. It's checked whether we're running the code on GPU or CPU, since we plan to make heavy computations on the system and the speed of processing of the code relies on this information.

```
CUDA is not available.  Training on CPU ...
cpu
```

Label Mapping: We need to load in a mapping from category label to category name. We can find this in the file `cate.json`. This will give us a dictionary mapping the integer encoded categories to the actual names.

Loading the data: Here we'll use 'torchvision' to load the data. The dataset is split into two parts, training and validation. For the training, we want to apply transformations such as random scaling, cropping, and flipping. This helps the network generalize leading to better performance. When using a pre-trained network like ResNet152, we also need to make sure the input data is resized to 224x224 pixels as required by the networks.

The validation set is used to measure the model's performance on data it hasn't seen yet. For this we don't want any scaling or rotation transformations, but we need to resize then crop the images to the appropriate size.

The pre-trained networks available from 'torchvision' were trained on the ImageNet dataset where each color channel was normalized separately. For both sets we need to normalize the means and standard deviations of the images to what the network expects. For the means, it's ' [0.485, 0.456, 0.406]' and for the standard deviations ''[0.229, 0.224, 0.225]' , calculated from the ImageNet images. These values will shift each color channel to be centered at 0 and range from -1 to 1.

```
IMAGE_DATASETS: {'train': Dataset ImageFolder
    Number of datapoints: 159
    Root location: b_cancer_data2/train
    StandardTransform
Transform: Compose(
               RandomRotation(degrees=(-30, 30), resample=False, expand=False)
               RandomResizedCrop(size=(224, 224), scale=(0.08, 1.0), ratio=(0.75, 1.3333), interpolation=PIL.Image.BI
LINEAR)
               RandomHorizontalFlip(p=0.5)
               ToTensor()
               Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
           ), 'valid': Dataset ImageFolder
    Number of datapoints: 35
    Root location: b_cancer_data2/valid
    StandardTransform
Transform: Compose(
               Resize(size=256, interpolation=PIL.Image.BILINEAR)
               CenterCrop(size=(224, 224))
               ToTensor()
               Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
           )}
DATALOADERS: {'train': <torch.utils.data.dataloader.DataLoader object at 0x126f60438>, 'valid': <torch.utils.data.dat
aloader.DataLoader object at 0x126f608d0>}
```

Now that the data is ready to be fed to the network, we shall build and train the classifier. We are using one of the pretrained models from `torchvision.models` to get the image features. Further, we are building and training a new feed-forward classifier using those features. Resnet-152 pretrained model is used for this image classifier.

Resnet-152 is a type of specialized neural network that helps to handle more sophisticated deep learning tasks and models. It has received quite a bit of attention at recent IT conventions, and is being considered for helping with the training of deep networks. Resnet introduces a structure called residual learning unit to alleviate the degradation of deep neural networks. This unit's structure is a feedforward network with a shortcut connection which adds new inputs into the network and generates new outputs. The main merit of this unit is that it produces better classification accuracy without increasing the complexity of the model

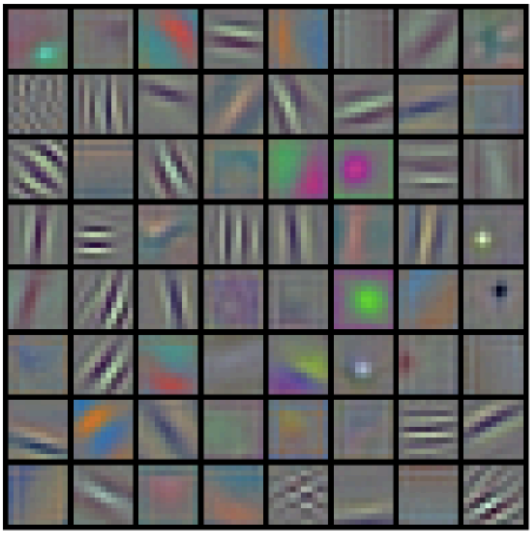
All layers except the final layer are frozen, so we don't destroy the weights. We define a new, untrained feed-forward network as a classifier, using ReLU activations. We need to ensure that our input_size matches the in_features of pretrained model.

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

.
.
.

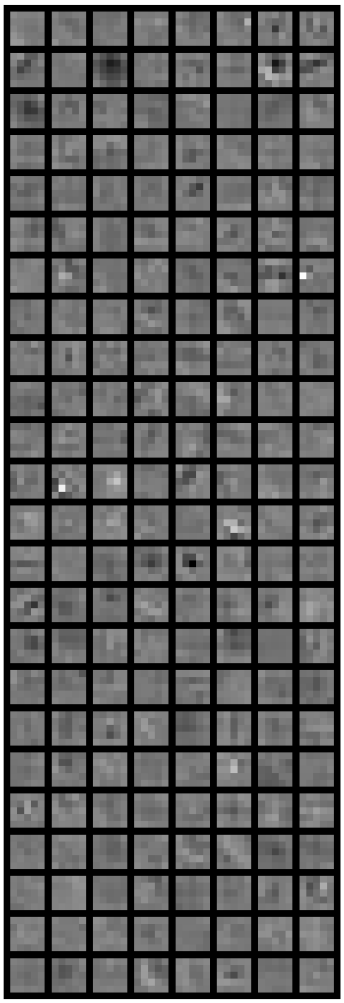
```
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
torch.Size([512, 128, 1, 1])
```

The pre-trained model may seem to be like a black box at first, but it is easily interpretable. How deep learning and CNN concepts are used in Resnet models for utilization in ML is explained in Understanding Architecture section of the report.
In order to make the pretrained models more interpretable, we have visualized the feature vectors, outputs of each layer. The initial layers take care of the big features like edges, shapes, etc:

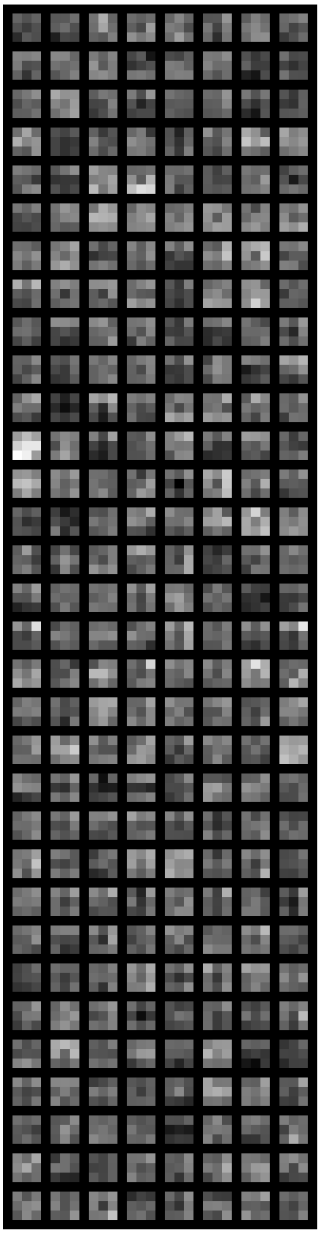


96 convolutional kernels of size 11×11×3 learned by the first convolutional layer on the 224×224×3 input images to the left.

The output of one of the middle Layers is visualised as follows (medium detail):



The output of the final layer is visualised as follows (high detail):



Now the model is ready to be retrained (with the new classifier). Number of epochs, criterion (NLLLoss because output is LogSoftMax), Adam optimiser, learning rate, momentum, decau LR by factor of 0.1 every 5 epochs, and the rest of initialisations are performed and finally model is called with its function signature values which were initialised earlier.

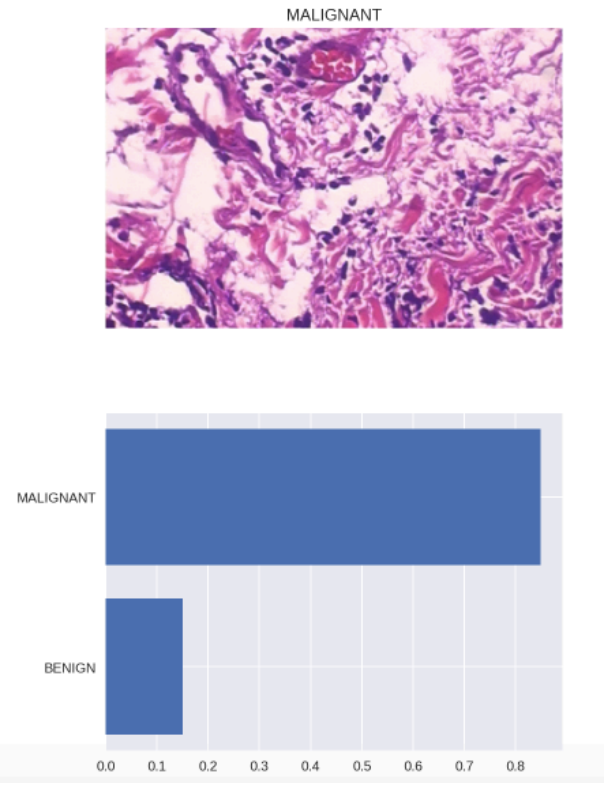
```
NLLLoss()
SGD (
Parameter Group 0
    dampening: 0
    initial_lr: 0.0006
    lr: 0.0006
    momentum: 0.9
    nesterov: False
    weight_decay: 0
)
<torch.optim.lr_scheduler.StepLR object at 0x13cc32ba8>
```

Now that the network is trained, we will save the model so we can load it later for making predictions. We will save the mapping of classes to indices which we get from one of the image datasets: `image_datasets['train'].class_to_idx`. We will attach this to the model as an attribute which makes inference easier later on.

```model.class_to_idx = image_datasets['train'].class_to_idx```
We want to completely rebuild the model later so we can use it for inference. Later, the model is loaded again.

Inference for classification: We are passing an image into the network and predicting the class. Write a function called `predict` that takes an image and a model, then returns BENIGN and MALIGNANT along with the respective probabilities.

Sanity Checking: Now that we can use a trained model for predictions, we check to make sure it makes sense. Even if the validation accuracy is high, it's always good to check that there aren't obvious bugs. Using `matplotlib` to plot the probabilities for the two classes as a bar graph, along with the input image. It should look like this:


```

The sanity check can be done for all the validation files together as well, and the excel output is as follows:

S.no	PATH	BENIGN	MALIGNANT
0	b_cancer_da	0.87087087	0.12912913
1	b_cancer_da	0.94094094	0.05905906
2	b_cancer_da	0.84084084	0.15915916
3	b_cancer_da	0.95095095	0.04904905
4	b_cancer_da	0.9009009	0.0990991
5	b_cancer_da	0.99099099	0.00900901
6	b_cancer_da	0.82082082	0.17917918
7	b_cancer_da	0.87087087	0.12912913
8	b_cancer_da	0.95095095	0.04904905
9	b_cancer_da	0.97097097	0.02902903
10	b_cancer_da	0.84084084	0.15915916
11	b_cancer_da	0.89089089	0.10910911
12	b_cancer_da	0.98098098	0.01901902
13	b_cancer_da	0.93093093	0.06906907
14	b_cancer_da	0.83083083	0.16916917
15	b_cancer_da	0.9009009	0.0990991
16	b_cancer_da	0.87087087	0.12912913
17	b_cancer_da	0.95095095	0.04904905
18	b_cancer_da	0.85085085	0.14914915
19	b_cancer_da	0.97097097	0.02902903
20	b_cancer_da	0.93093093	0.06906907
21	b_cancer_da	0.88088088	0.11911912

Results & Conclusions

- Output of the epochs:

```
Epoch 1/10
----------
train Loss: 0.3126 Acc: 0.8772
valid Loss: 0.3267 Acc: 0.8587

Epoch 2/10
----------
train Loss: 0.3089 Acc: 0.8809
valid Loss: 0.3109 Acc: 0.8753

Epoch 3/10
----------
train Loss: 0.2941 Acc: 0.8861
valid Loss: 0.3213 Acc: 0.8625

Epoch 4/10
----------
train Loss: 0.2899 Acc: 0.8849
valid Loss: 0.3074 Acc: 0.8740

Epoch 5/10
----------
train Loss: 0.2932 Acc: 0.8840
valid Loss: 0.3180 Acc: 0.8740

Epoch 6/10
----------
train Loss: 0.2641 Acc: 0.8960
valid Loss: 0.3076 Acc: 0.8753

Epoch 7/10
----------
train Loss: 0.2666 Acc: 0.8947
valid Loss: 0.3045 Acc: 0.8740

Epoch 8/10
----------
train Loss: 0.2712 Acc: 0.8936
valid Loss: 0.3043 Acc: 0.8740

Epoch 9/10
----------
train Loss: 0.2661 Acc: 0.8985
valid Loss: 0.3061 Acc: 0.8766

Epoch 10/10
----------
train Loss: 0.2721 Acc: 0.8954
valid Loss: 0.3056 Acc: 0.8734

Training complete in 26m 23s
Best valid accuracy: 0.876598
```

- Validation on the training is performed which gives an accuracy of 86%.
- We come to conclude that it was beneficial to use a pre-trained network for this utility because the weights gained by ResNet152 for ImageNet dataset proved to be working for the dataset we provided for training.
- Image preprocessing is essential before we feed the network with the images.
- It is critical to note that in case we had a larger dataset, the model had to be fine-tuned before required to be trained.
- Since models requiring image processing require long processing time, it is beneficial to store the model in memory so it can be loaded again for use in future.

Further plans of action and extensions to current project

Along with the current project development, we plan to further enhance and extend it via three major links.

- ## UTILISING WSI DATASET:

From an independent source from BreakHis dataset, dataset we chose to further develop our work with is Breast Cancer Tumor Proliferation Assessment (TUPAC16). This dataset consists of 500 whole-slide images (WSIs) of breast tissue tumors.

Process that produced these images:

1. Biopsy of breast tumor
2. Slicing of tissue thinly onto slide
3. Staining the slide (with H&E)
4. Examining the slide with microscope
5. Scan at high resolution

The ground truth of this data set has labelling as per tumor proliferation score, and classified as “BENIGN” or “MALIGNANT”.

BreakHis dataset consists of 7,909 magnified images of breast tissue slides.
We plan to divide each WSI into tiles, and process them to make uniform magnified slides just like the ones present in BreakHis dataset. This will help increase the practical use of our system, since in real world scenario we begin with the WSIs itself. Also, the preprocessing steps will help us gain better accuracy since we will tackle with issues such as tissue percentage, high varying staining concentrations, etc.

- ## CONSEQUENT ACTIONS ON ResNet152 MODEL:

Pre-trained ResNet152 model is trained on ImageNet. ImageNet is a dataset of over 15 million labelled high-resolution images with around 22,000 categories. In all, there are roughly 1.2 million training images, 50,000 validation images and 100,000 testing images.
Upon generating samples from TUPAC dataset, we will have potential 4 million samples, which is a comparable number to ImageNet samples. In such a case finetuning of few frozen layers along with replacement of classifier will take place.

- # SURVIVAL RATE PREDICTION (an extension to Breast Cancer Prediction):

The development of breast cancer involves a progression through intermediate stages until we reach the invasive carcinoma and finally evolves into metastatic disease. Given the variability in clinical progression, the identification of markers that could predict the tumor behavior is particularly important in breast cancer.

In our study of the condition we have used images (Whole Slides) to identify the tumor classes, essentially whether its benign or malignant. To extent our study further we propose a way to apply some machine learning models like Support Vector Machine(SVM),Classification and Regression Tree(CART), K-Nearest Neighbours(KNN) and Random Forest to identify specific markers that could be important prognostic markers for the prediction of breast cancer.

We begin with identifying various such markers that can be used as the diagnostic tool to aid us in prediction. The most well-established breast molecular markers with prognostic and/or therapeutic value like hormone receptors, HER-2 and others as well as several proteins that can be used for this application.

References & Bibliography

1. *Veta, M., Heng, Y. J., Stathonikos, N., Bejnordi, B. E., Beca, F., Wollmann, T., ... & Hedlund, M.(2019). Predicting breast tumor proliferation from whole-slide images: the TUPAC16 challenge. Medical image analysis, 54, 111-121*

2. *Couture, H. D., Williams, L. A., Geradts, J., Nyante, S. J., Butler, E. N., Marron, J. S., ... & Niethammer, M. (2018). Image analysis with deep learning to predict breast cancer grade, ER status, histologic subtype, and intrinsic subtype. NPJ breast cancer, 4(1), 30.*

3. *Yu, Y., Lin, H., Meng, J., Wei, X., Guo, H., & Zhao, Z. (2017). Deep transfer learning for modality classification of medical images. Information, 8(3), 91.*

4. *Spanhol, F. A., Oliveira, L. S., Petitjean, C., & Heutte, L. (2016, July). Breast cancer histopathological image classification using convolutional neural networks. In 2016 international joint conference on neural networks (IJCNN) (pp. 2560-2567). IEEE.*

6. *Simonyan, K., Vedaldi, A., & Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034.*

7. *Ranjan, R., Patel, V. M., & Chellappa, R. (2017). Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 41(1), 121-135.*

8. *Suzuki, K. (2017). Overview of deep learning in medical imaging. Radiological physics and technology, 10(3), 257-273.*

9. *Kermany, D. S., Goldbaum, M., Cai, W., Valentim, C. C., Liang, H., Baxter, S. L., ... & Dong, J. (2018). Identifying medical diagnoses and treatable diseases by image-based deep learning. Cell, 172(5), 1122-1131.*

10. *Ramos-Pollán, R., Guevara-López, M. A., Suárez-Ortega, C., Díaz-Herrero, G., Franco-Valiente, J.M., Rubio-Del-Solar, M., ... & Ramos, I. (2012). Discovering mammography-based machine learning classifiers for breast cancer diagnosis. Journal of medical systems, 36(4), 2259-2269.*

11. *Reis, S., Gazinska, P., Hipwell, J. H., Mertzanidou, T., Naidoo, K., Williams, N., ... & Hawkes, D. J.(2017). Automated classification of breast cancer stroma maturity from histological images. IEEE Transactions on Biomedical Engineering, 64(10), 2344-2352.*

12. *Cardoso, J. S., & Domingues, I. (2011, December). Max-coupled learning: Application to breast cancer. In 2011 10th International Conference on Machine Learning and Applications and Workshops (Vol. 1, pp. 13-18). IEEE.*

13. *Dramićanin, T., Lenhardt, L., Zeković, I., & Dramićanin, M. D. (2012). Support Vector Machine on fluorescence landscapes for breast cancer diagnostics. Journal of fluorescence, 22(5), 1281-1289.*

14. *Adabor, E. S., & Acquaah-Mensah, G. K. (2017). Machine learning approaches to decipher hormone and HER2 receptor status phenotypes in breast cancer. Briefings in bioinformatics, 20(2), 504-514.*

15. *Ganggayah, M. D., Taib, N. A., Har, Y. C., Lio, P., & Dhillon, S. K. (2019). Predicting factors for survival of breast cancer patients using machine learning techniques. BMC medical informatics and decision making, 19(1), 48.*

16. *Chan, T.-H., Jia, K., Gao, S., Lu, J., Zeng, Z., & Ma, Y. (2015). PCANet: A Simple Deep Learning Baseline for Image Classification? IEEE Transactions on Image Processing, 24(12), 5017–5032. doi:10.1109/tip.2015.2475625*

17. *Xu, Y., Mo, T., Feng, Q., Zhong, P., Lai, M., & Chang, E. I.-C. (2014). Deep learning of feature representation with multiple instance learning for medical image analysis. 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). doi:10.1109/icassp.2014.68538*