Medical Image Recognition for Arthroscopic Images

Final Report

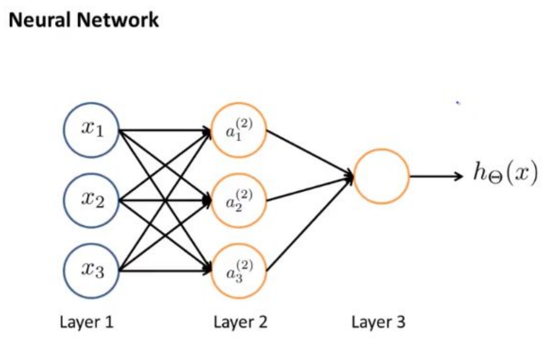
Jacob Kurtz, Phanitta Chomsinsap, Alae (Oliver) Amara, Jonathan Huynh

Intro to Neural Networks and Convolutional Neural Networks (CNN)

Neural Network Architecture

A neural network is an algorithm structure that mimics the neurons in the human visual cortex via neuron connectivity and the assignment of receptive fields. [6] This algorithm receives data sets and learn from them to predict future results. A neural network consists of one input layer, one or more hidden layers, and one output layer as shown in Figure 1. The same figure shows the calculations of the structure where g() is the activation function and Θ is a parameter, or trainable variable. [5] Neural networks have the ability to recognize traits and patterns which gives the algorithm advantages in speed and accuracy over other visual analysis algorithms such as template matching.

The activation function g() determines what the neural network does, such as classification or prediction. It also adds non-linearity to the predictions, meaning it adds flexibility to the neural network's predictions. The parameters Θ, or weights, are values that adjust as the algorithm trains or learns through a back-propagation algorithm. Back-propagation minimizes the loss function output, or error value, which is calculated by comparing the prediction with the true result.



$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

Figure 1. A simple neural network and its calculations [5]

Forward Propagation

Forward propagation is feeding a neural network input data to calculate the output. The calculations are shown in Figure 1. Layer 2's neurons a^n_m is calculated by matrix multiplication of input data x_m and connected parameters Θ, and then feeding the product into activation function g(). A parameter Θ is present in every connection. The Layer 3 neuron hΘ(x) uses Layer 2 neurons as input and is calculated the same way as the previous layer.

Back Propagation

Back propagation is how the neural network trains by adjusting its parameters Θ to minimize error. In Python's Tensorflow library, back propagation is a large black box that can be executed with just one or two lines of code.

Gradient descent is the most common form of back propagation. For back propagation by gradient descent, the output is fed into a chosen loss function. Then each parameter's Θ gradient descent is calculated by taking the derivative of the loss function L(). For parameters Θ in the earlier layers, chain rule is applied to calculate their gradient descent. To update the parameter values, subtract the current parameter values with their respective gradient descent values $dL/d\Theta$ multiplied by a learning rate constant lr.

$$\Theta_{new} = \Theta_{old} - lr * dL/d\Theta$$

The learning rate lr determines how fast the neural network learns. A small learning rate may take too long to train. A large learning rate may overshoot the local minima. Some Tensorflow backpropagation functions, like Adam Optimizer, change the learning rate during training.

Convolutional Neural Network

A convolutional neural network (CNN) is a type of neural network that is used for analyzing images because the architecture mimics a mammalian visual cortex. The difference from ordinary neural networks is that the input layer is an image, which is a 3D RGB dataset flattened into a one-dimensional array [Figure 2]. The hidden layers are 3D volumes, and the output is an array that classifies how much the input fits into each class.

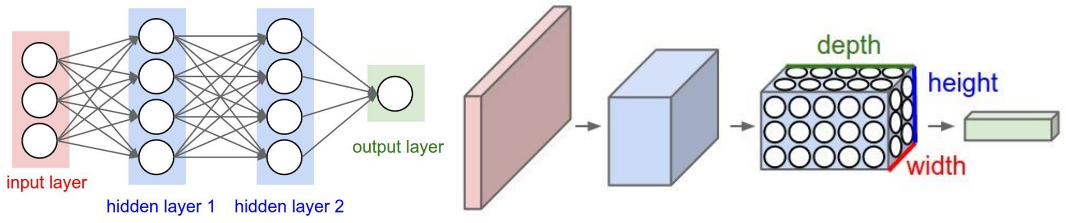


Figure 2. Simple architectures of a normal neural network (left) and CNN (right) [6]

CNNs contains three different types of hidden layers which are the convolutional ReLU layers, pooling layers, and fully connected layers. The ReLU layers are simple input to output operations through the ReLU activation function. Most CNN organize their layers in the following order:

INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC

Where N, M and K are positive integer constants that indicate repetitions.

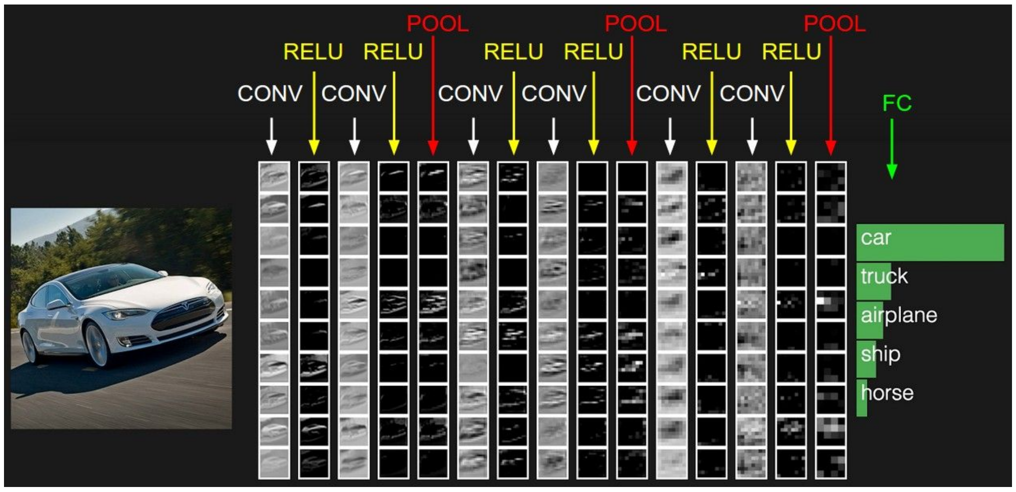


Figure 3. Example of a CNN architecture [6]

The convolutional layer filters characteristics from the previous layer by taking its convolution with a kernel. The characteristics this layer filters is determined by the CNN parameters, the elements within the kernels.

The ReLU layer always follows after the convolutional layer. The ReLU layer is an output from the ReLU activation function f = max(0,x) where x is the elements of the convolutional layer. This classifies the characteristics in the convolutional layer. The max pooling layer simply down-samples the previous layer usually by taking the highest value within a kernel. This expands the receptive field of the later convolutional layers' kernels on the original image. Both the ReLU and pooling layers don't contain parameters because they are input to output functions.

The fully connected layer calculates in the same way as the simple neural network shown in Figure 1. The output of the fully connected layer is an array of values for each class, where the highest value is the prediction. [6]

Training Neural Networks

When training a neural network, 3 datasets are used, and they are the training set, validation set, and test set. The training set is fed into the network, and the network's output is compared with the ground truth to calculate the loss function. The loss function measures the accuracy of the neural network. Common loss functions include the sum of squared differences and cross-entropy. The network trains by minimizing the loss function's output by adjusting weights in the hidden layers with back-propagation of gradient descent.

Accuracy and Splitting Datasets

A dataset is often split into 3 sets: training set, validation set, and testing set.

The training set is used to train the neural network and consists of 60% of the total dataset. The epoch of a neural network is the number of times the entire training set is trained. The training set is tested for accuracy after each epoch for training accuracy which is expected to converge to nearly 100% after all training is finished.

The validation set consists of 20% of the total dataset. The validation set is fed into the network after each epoch to test the network's generalization accuracy. The validation accuracy converges to a value that reflects how well the network predicts untrained data. The validation accuracy is used to indicate when the training is done and whether the network is overfitting or not. Once training is done, the parameters from the epoch with the highest validation accuracy is used to find test accuracy.

The test set is also 20% of the total dataset. The test set is used to find the test accuracy which is done after all training is finished. The test accuracy reflects the accuracy of the network for future predictions. It is often less than the validation accuracy.

Overfitting

Overfitting is a common problem when training neural networks. Overfitting is when the network fits too well with the training set that it fails to generalize predictions for new inputs. This occurs when the neural network is too flexible for the given dataset. Signs of over-fitting include high or 100% training accuracy and low validation accuracy. Methods to prevent overfitting include decreasing the number of parameters, or weights, in the network and increasing the complexity or amount data. In the CNN case, the fully-connected layers at the end of the network tend to have the most parameters, so pooling the image size more is an effective way to reduce the total number of parameters. A biased network is the opposite of an overfit network. A biased network is where the training accuracy is low due to the network being too simple and inflexible.

Stochastic Gradient Descent

Stochastic gradient descent is training with small batches of the training set at a time. Multiple input data can be trained at a single time by shaping the input as a matrix where each column is an input data. Compared to training all the training data at once, stochastic gradient descent prevents bias, avoids CPU memory overflow, and speeds up training. Randomizing the training set order and parameter initialization also makes training more effective.

Dataset

Classes

We decided to classify the 5 most common tools we observed from the videos provided by Arthrex. In addition, we included the “No Tool” class for images with no tools present. Thus, our CNN classifies 6 classes as shown in Figure 4. Our dataset did not include images of tools with their tips hidden, in which identifying the tool was difficult or impossible. The dataset also did not include images of other tools that is not one of the 6 classes. If an image is not one of the 6 classes, the CNN will still attempt to identify the images as one of the 6 classes.

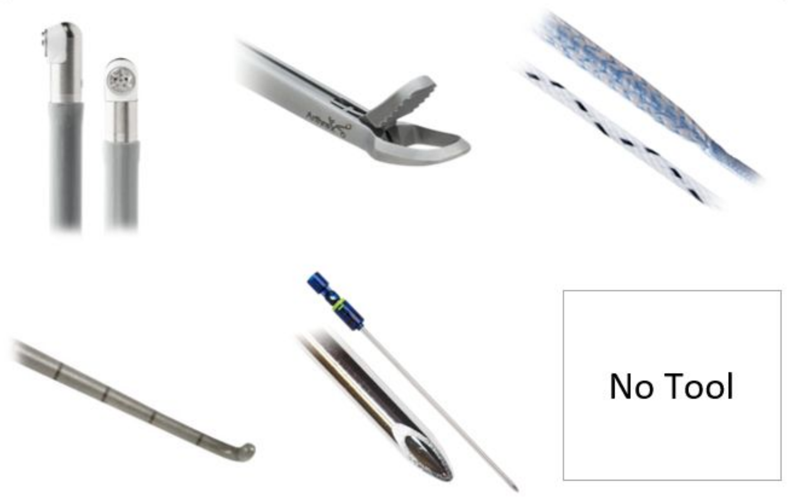


Figure 4. Classes from left to right: Heat Wand, Basket Biter, Suture (top)
Probe, Shaver, and No Tool (bottom)

Training CNNs required a large dataset with their labels, but Arthrex only provided us with unlabeled videos and images. Thus, we have to label the images manually, and our goal was to obtain at least 50,000 labeled images. A tool detection algorithm in Matlab was used to speed up the labeling process. An even distribution of the classes is needed to avoid making the CNN bias towards some classes. Later on, We also augmented the images to increase the size and diversity of the dataset.

Splitting the Dataset

When training neural networks, splitting the dataset into 3 sets is a common practice. [5] The 3 sets are the training set, validation set, and test set. The training set is the largest set, about 60 - 70% of the total dataset. The validation and test set are both 15 - 20% of the total dataset. The training set is used to train the CNN. The number of times the entire training set is learned by the CNN is called an epoch. The training accuracy is how well the CNN can predict the training set, which is the same data it learned from. Thus, the training accuracy should be close to 100% when training is done.

The validation set is tested at every epoch to see how well the CNN predicts data it has never seen before. A plot of the validation and training accuracies per epoch can show when the training is done or if the CNN is over-fitting. For example, Figure 5 shows our first CNN training results in Fall quarter December 2017. The CNN is done training when the validation accuracy stopped changing significantly and the training accuracy reached 100%. The CNN is over-fitting because the training accuracy is 100% and the validation accuracy is low at about 65%. This means that the CNN has learned to recognize the training set but fails to predict data it has never seen before.

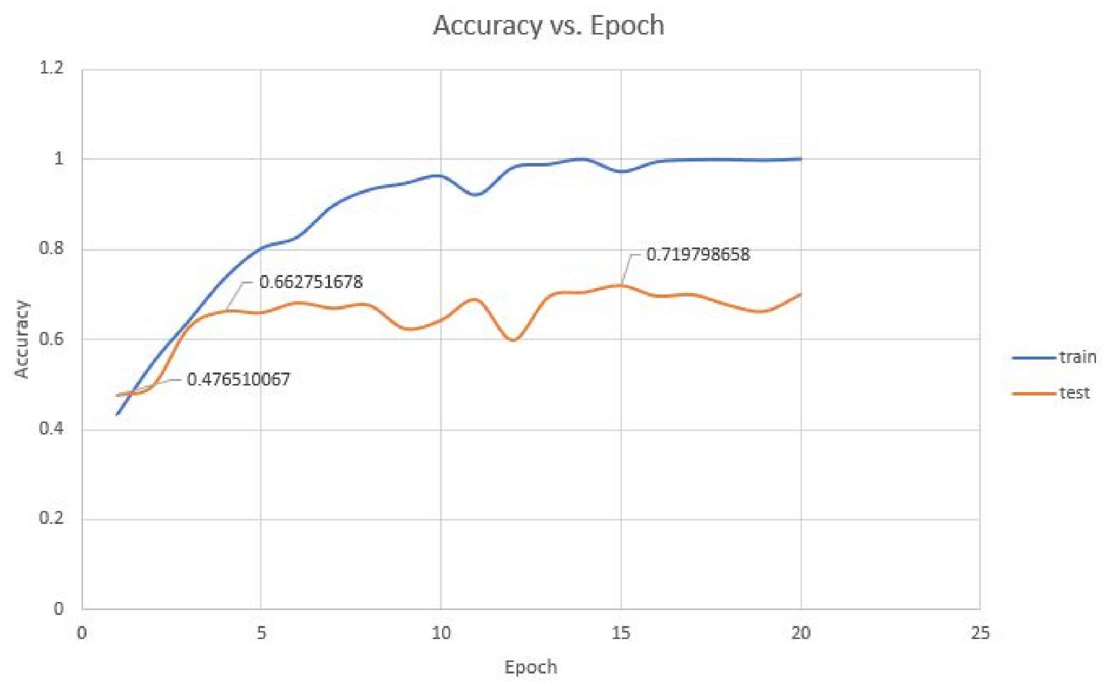


Figure 5. First CNN training results in December 2017

The test set is tested when all training is done, and the test accuracy is the true accuracy value of the CNN. The test set should be diverse and evenly distributed among classes. We collected the test set by picking them from individual images provided by Arthrex, instead of taking frames from the videos. This decreases the chance of having several test images from the same video. Both the validation and test set do not have augmented images because they are used for testing.

Collecting Dataset (Tool Detection Algorithm)

The methodology we have attempted was to utilize range filtering in MATLAB in order to segment the scene into a “foreground” and “background”. Objects in the foreground are separated out as outlines, and then dilated to create closed shapes. These closed outlines are then filled in and converted into a logical mask. The mask can then be applied over the original image to “highlight” tools in the frame. Other avenues to explore include histogram analysis, however the uneven illumination of the film and topological layout of the scene renders this a fairly unreliable method without delving into more computationally-taxing algorithms.

Utilizing the MATLAB function “rangefilt”, we recover an outline of an individual scene from the input video. Rangefilt is a simple function that attempts to denote the distinction between foreground and background objects in a scene by computing the “range value” within an n x n area around each pixel. The range value is simply the smallest magnitude point in the neighborhood, subtracted from the largest Due to this, we get a very high output around the edge of the scene, as the highest pixel values are subtracted from the lowest in an nxn neighborhood, and the edges of the arthroscopy scenes are black (corresponding to a value of 0) returning simply the highest value around the edges. To account for this, we parameterize a circular mask with radius slightly smaller than the boundaries of the arthroscopy footage and apply it over the range-filtered image so the the noise around the edges is simply cropped out, and we retain a scene with most of the relevant information preserved. As mentioned earlier, these “outlines” separate the background and the foreground, and we dilate the lines and fill in any resulting complete, closed structures. These segmented pieces are then passed through a thresholder, deleting each area that is less than 1000 pixels. If any nonzero patches are retained after this thresholding then the scene is considered positive for the presence of a tool. Additionally, this binary-valued segmented image can be applied to the original scene to approximate the location of any detected tools. This is also useful for troubleshooting/mitigating false positives by altering threshold values.

Tools are isolated due to their difference in color at the edges in the image. Compared to the homogeneity of the tissue, the tools differ visually quite drastically, and so tissue is usually not detected as a foreground object. The primary drawback with our algorithm is the fact that if the tool is fairly obscured, poorly illuminated or moving too fast it may not be recognized as a distinct object (as the edges become less differentiable from the background).

The following summarizes the workings of the tool detector [Figure 6]:

1. Input image is range-filtered; Similar to edge-filtering. Intended to separate foreground/visually distinct objects from the background (Top middle).
2. Circular cropping mask applied to filtered image to remove unwanted noise around edges due to circular bound of arthroscopy camera (Top right).
3. Remaining outlines of “foreground” objects are dilated to create closed structures that are then filled in to create a binary segmentation mask (same size as original image) (Bottom left).
4. This mask is then thresholded (about 1000 pixels works well) so as to eliminate minor objects/noise (the tool, if present, will often be the largest object in the foreground) (Bottom middle).
5. If any nonzero clusters remain after thresholding, then we consider the image “positive”; Additionally, the mask can be overlayed over the original image to give an approximation

of where the tool is located in the scene (Bottom right).

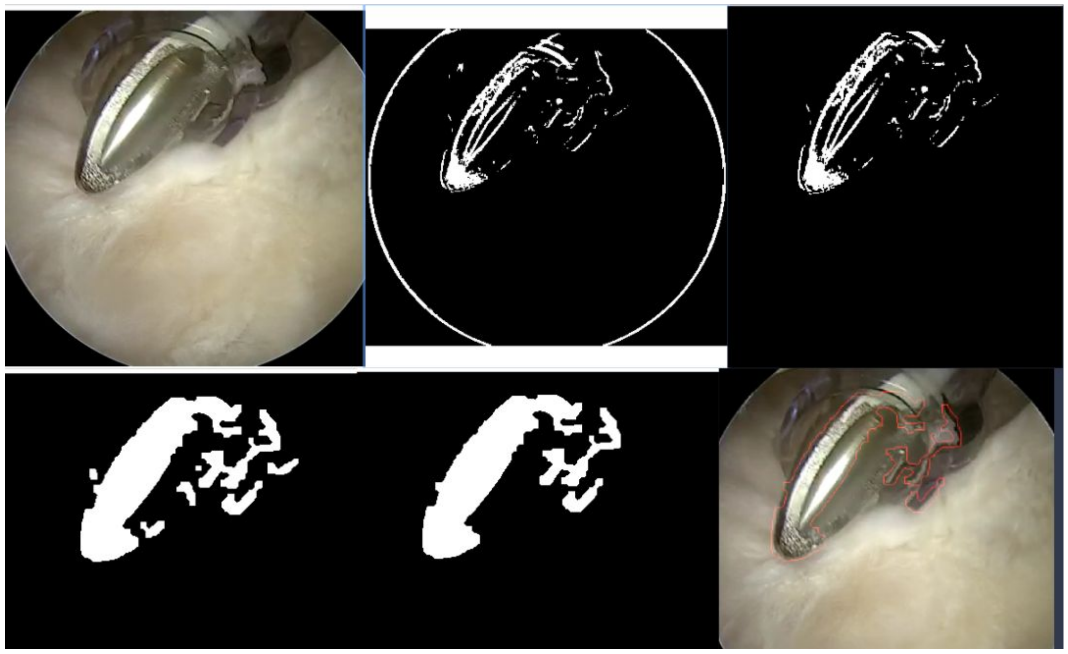


Figure 6. The figures show the workings of the tool detector

While originally proposed for use in the overall classification pipeline, this algorithm saw more use as tool for image labeling. Due to the fact that the entire dataset had to be extracted from hundreds of hours of video, and that we needed tens of thousands of labeled images for adequate results we utilized this as a soft filter, intended to separate out frames in a video where a tool's presence has been detected. This implementation took the continuity of the videos into account, and interpolated presence of tools for false negatives, and segmented out entire sets of multiple images where the same tool is presumed to have been visible. Representative images from this set were then displayed to a human labeler, who would then merely need to enter a single number to label the entire segmented set appropriately. This immensely helped mitigate the workload of labeling our own dataset, but had limitations, as false segmentations/erroneous interpolations would increase the need for vigilant pruning of these labeled datasets afterwards.

Data Augmentation and Final Dataset

After 3 months of labeling images, we collected over 100,000 labeled images. However, the classes were uneven, so we took 10,000 labeled images per class for the training set. This was not enough to effectively train the CNN, so 5 image augmentations were applied to every image to multiply the training set from 10,000 to 60,000 images per class. With 6 classes, the training set has a total of 360,000 images. The 6,000 images in the validation set and 1,200 images in the test set were not augmented. All sets have evenly distributed classes.

Examples and description of augmentations:

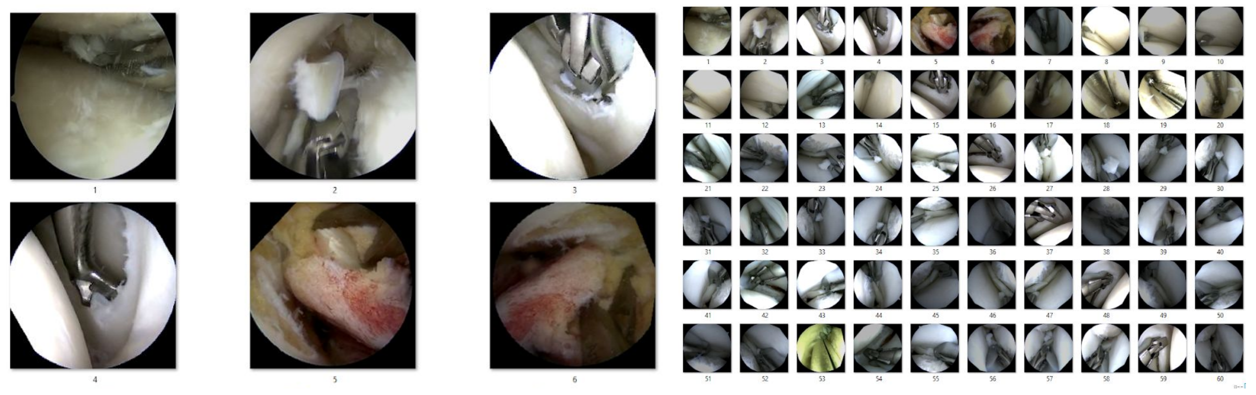


Figure 7. Brightness: The pixel value of the original train images are scaled by 0.7 and 1.6 at random.

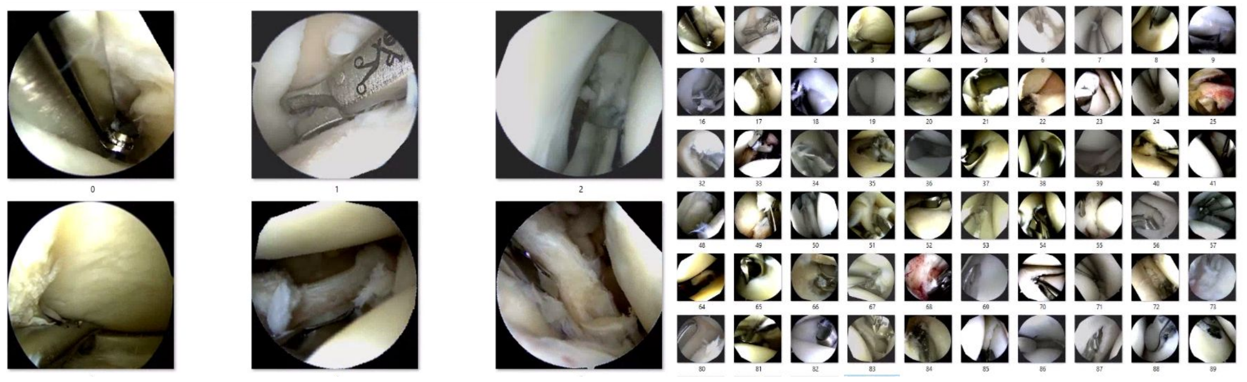


Figure 8. Contrast: Each image's contrast is randomly scaled by 0.7, 1.5, or 2.0 by using Python's PIL library ImageEnhance module

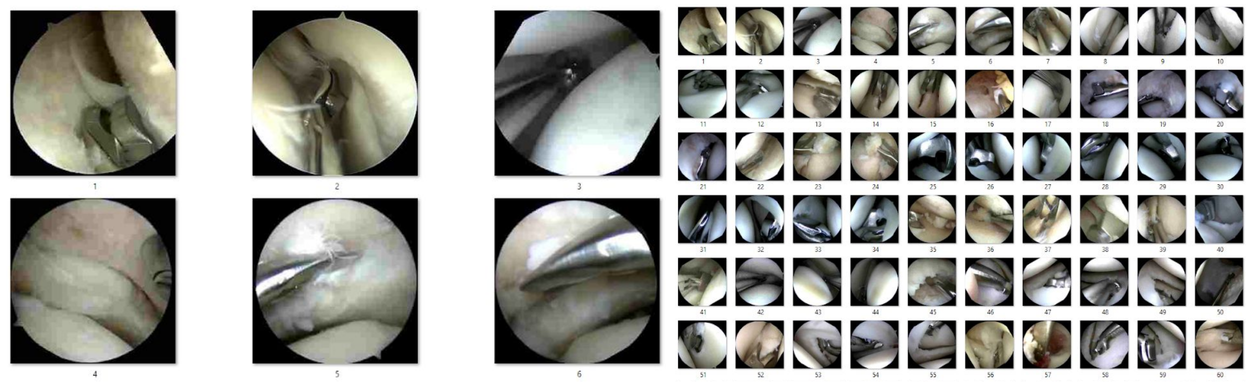


Figure 9. Jpeg Compression: Matlab's imwrite function allows for the specification of compression quality when selecting the JPEG file type. We applied jpeg compression randomly to the images with the quality parameter set from 10 to 30. Quality parameter is a scalar from 0

to 100 with 0 being the worst quality with highest compression and 100 being the best quality with lowest compression.

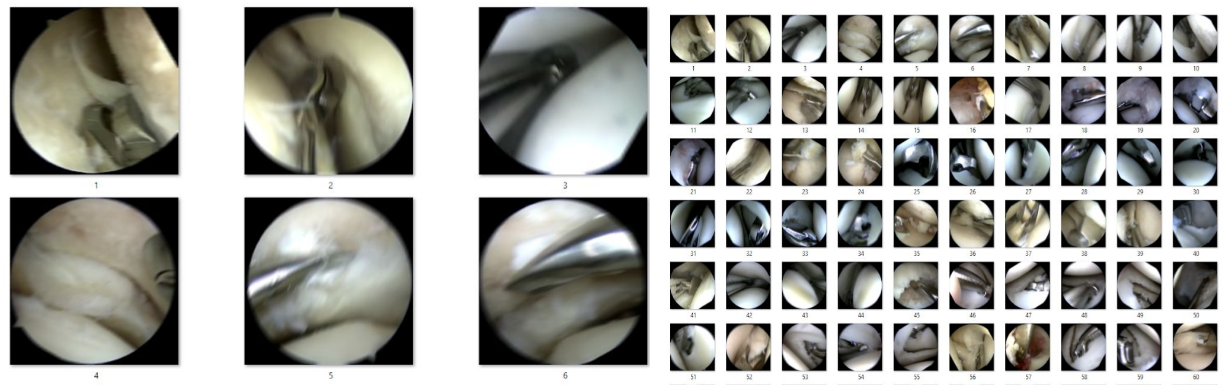


Figure 10. Motion Blur: Utilizing built in matlab functions, we applied motion blur to augment our dataset. The motion blur was applied at random with anywhere from 5 to 15 pixels of motion blur at a random angle from 0 to 360 degrees.

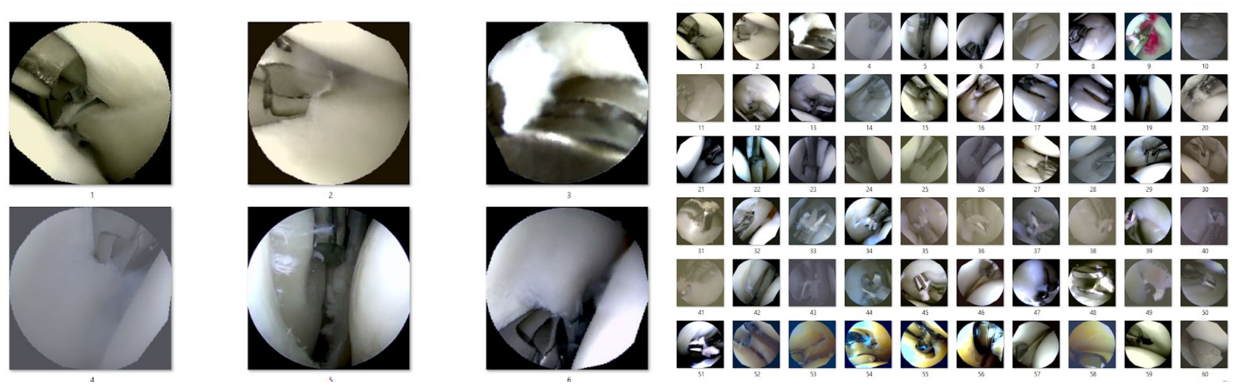


Figure 11. Color Transfer: Implementation of the “Color Transfer Between Images” proposed by Reinhard et al., 2001 [9]. Mapped histogram distributions of training images to other randomly sampled arthroscopic images to induce variations in color/dynamic range.

Design Choices for Best Experiment

24-Convolutional Layer Network Architecture:

- Batch normalization after each convolutional layer
- Average pooling at every 3 convolutional layers
- Xavier Initialization
- 1.97M trainable parameters
- Residual blocks
- Leaky relu
- Keep prob 0.25
- Adam Optimizer
- Cross-entropy
- Mean subtraction

At the end of the project, our CNN model has a test accuracy of 85.3% and the Titan V GPU can output the results in 29.73 frames per second. The accuracy per class is displayed in Figure 13. The architecture is shown in Figure 12 where 3 convolutional layers and a pooling layer are repeated 8 times followed by a fully-connected layer, dropout layer, and output fully-connected. There is a total of 24 convolutional and pooling layers.

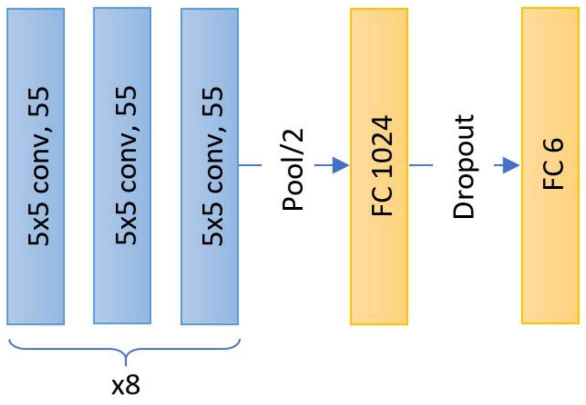


Figure 12.: Final CNN Architecture

Class	**Accuracy**
Basket Biter	0.780
Heat Wand	0.810
Probe	0.940
Shaver	0.755

Suture	0.875
No Tool	0.955
Average Accuracy: 0.853	
Timing Analysis: ● 29.73 frames per second ● Near Real-Time	

Figure 12. Accuracy per class and timing analysis for final CNN model

All convolutional layers are followed with a leaky ReLU activation function and have 55, 5x5 kernels. The pooling layers uses average pooling. There is about 1.97 million parameters. The model uses cross-entropy as loss function, and Adam Optimizer for back-propagation. The model includes the features described in the following paragraphs.

Drop-out Layer

During training, drop-out layers ignores or cuts off neurons from the previous layer with an arbitrary probability. This decreases the chance of overfitting during back-propagation training. For our model, the keep-probability, or the probability each neuron is not dropped, is set at 0.25 (or 25%), because our results were constantly overfitting. The drop-out layer is the second to last layer which is right before the output fully-connected layer. No neurons are dropped during forward propagation, or testing.

Leaky ReLu

Leaky ReLU is a variant of the ReLU activation function as shown in Figure 14. The advantage of the normal ReLU function over the Sigmoid or hyperbolic functions is that the gradient does not saturate, or become zero, when given a high order input. Leaky ReLU further strengthens this advantage by removing the possibility of zeros gradients when given a negative input. Our CNN model replaces the ReLU functions with leaky ReLU functions which follow after every convolutional layer. The purpose of activation functions is to add non-linearity to the predictions.

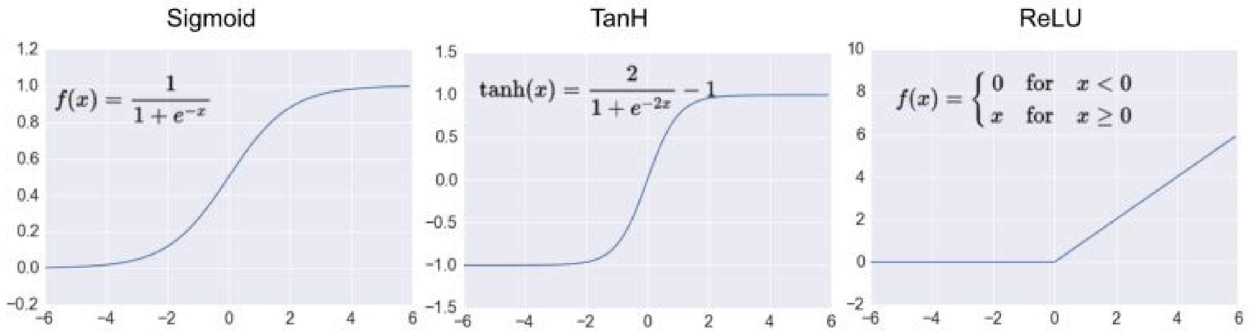


Figure 13. Sigmoid, Hyperbolic, and ReLU Function

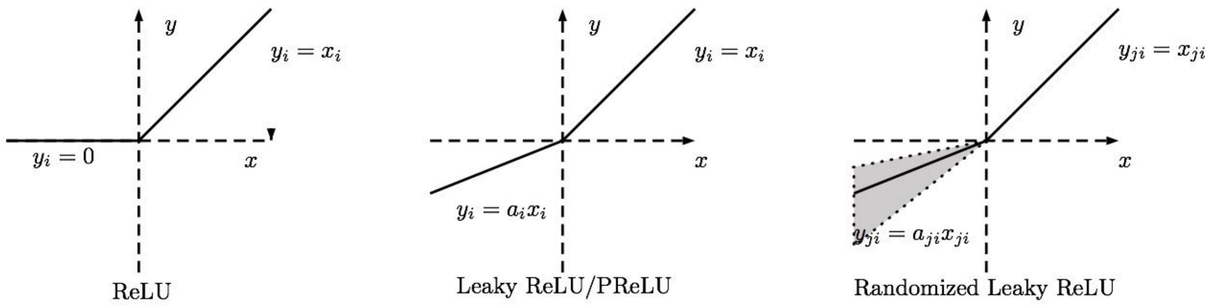


Figure 14. ReLU and Leaky ReLU [1]

Cross-Entropy Loss Function

The cross-entropy loss function evaluates performance for classification models that return a probabilistic output between 0 and 1 (just as the softmax classifier does). Cross-entropy loss is also called log loss, due to the fact that it utilizes the log function to derive the loss metric. A perfect classifier would have a loss of 0, and grows in accordance with the function depicted below. This loss function especially penalizes erroneous, confident predictions generated by the classifier.

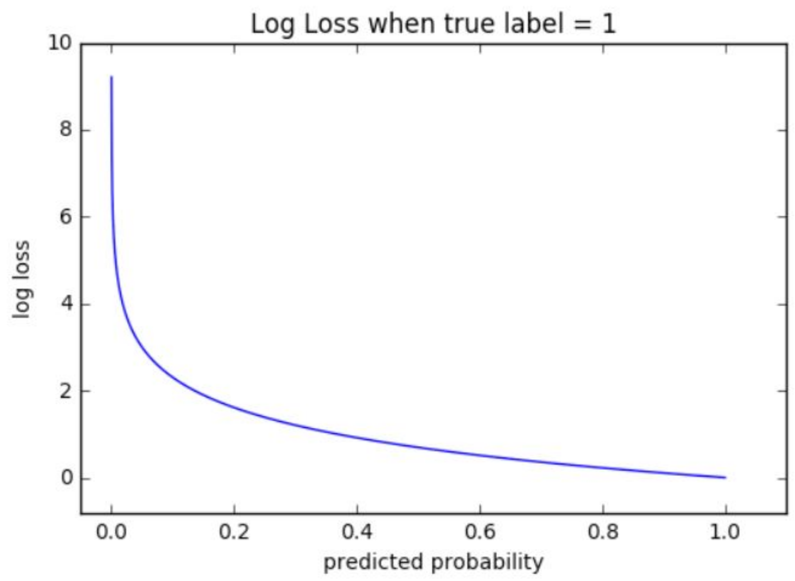


Figure 15. Plot of cross-entropy function [2]

Adam Optimizer

The Adam Optimizer is an iteration of conventional Stochastic Gradient Descent (SGD) that utilizes the concept of “momentum” to adapt the learning rate automatically as training progresses. In vanilla SGD the backpropagation is performed over batches of the training data, as opposed to evaluating gradients independently for every piece of training data and as in non-stochastic GD algorithms. This helps to avoid bias, stabilizes/speeds up the learning and eases computational constraints. While in classical SGD the learning rate can be adaptive as well (though not a necessity), Adam optimization learns the learning rate itself on a per-parameter basis. In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients, similar to momentum

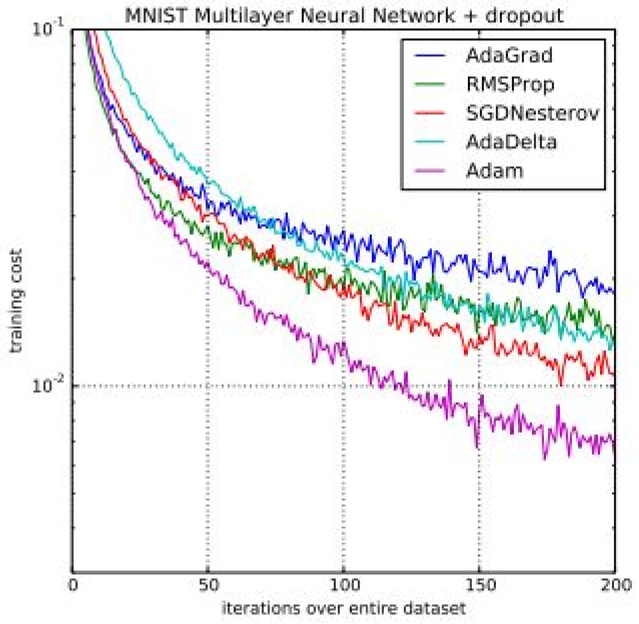


Figure 16. Example of convergence rates for different optimizers [3]

Batch Normalization

Batch normalization is the normalization of “mini-batches” of activations from previous hidden layers. This is accomplished via mean subtraction and unitization of the variance. This is done to reduce internal covariate shift, and so that large deviations in activation values don’t bias the network weight updates and cause unwanted oscillations in the weights as the network attempts to converge on loss minimas.

Residual Blocks (Skip Connections)

Residual blocks, or skip connections, adds an output from an earlier layer to the output of a later layer as shown in Figure 17. This mitigates the problem of vanishing gradients. Vanishing gradients is when trainable parameters in the earlier layers update slowly or don’t update at all because the gradient calculated through gradient descent is close to zero. Normally, when calculating the gradient, chain rule is applied for every layer from the end of the model to the layer where the gradient is calculated. The values that come out of the chain rule are typically small and often between 0 and 1, so multiple products of these small numbers lead to a gradient near zero.

Residual blocks, or skip connections mitigates this vanishing gradient problem by skipping layer connections. This reduces the number of times chain rule is applied which prevents gradients from falling to zero and trains the CNN faster.

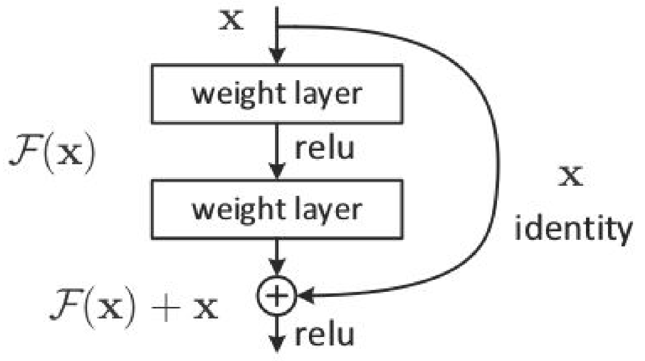


Figure 17. Residual Block Example[8]

Mean Subtraction

Per-pixel mean (calculated across entire training dataset) is subtracted from every input into the CNN. This is a method of input data normalization, and is done to center the data around zero so that gradients don't go out of control during backpropagation due to uneven scaling of parameters.

Pooling Methods

The pooling layer down-samples the previous layer in the 2D plane to increase the receptive field size of later convolutional layers and to decrease processing power needed to calculate the outputs. We experimented with 3 of the following pooling methods: max pooling, average pooling, and stride 2 convolution. Our final CNN architecture uses average pooling, but most of the other experiments used max pooling.

Max pooling down-samples by taking a block of pixels from the previous layer and keeping the largest value. Our experiments with max pooling took 2x2 blocks of pixels. The weakness of max pooling is that the operation is non-differentiable, so when optimizing by gradient descent, the pixels dropped by max pooling are completely ignored.

Average pooling down-samples similarly to max pooling. The difference is that average pooling takes the average of the block of pixels, whereas max pooling simply keeps the largest value. Our final CNN model uses average pooling because it gave a slightly higher test accuracy compared to the other pooling methods. Our average pooling layer also takes a 2x2 block pixels.

Stride 2 convolution is the same 2D convolutional operation that produces the convolutional layers but it is done with the kernel skipping 1 pixel at every step. This produces a layer with half the dimensional lengths in the 2D plane, which is the same down-sampling size as our max and average pooling methods.

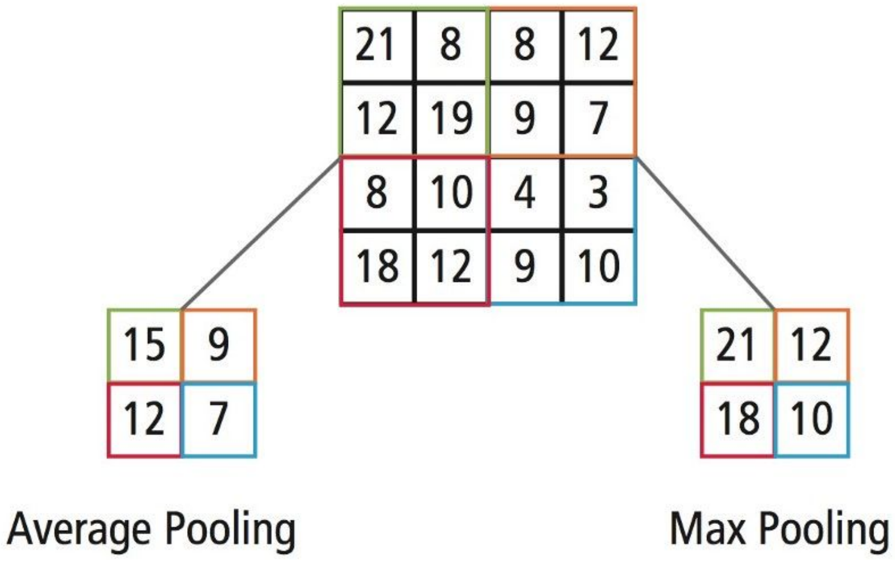


Figure 18. Example of Average and Max Pooling [7]

Increase of Activation Maps in Deeper Layers

Observing common motifs used in popular CNN architectures, we noticed that the VGG networks have a tendency to increase the number of feature maps as the network gets deeper.

Weight Initialization

Another experiment we did is to explore different ways to initialize the weights. The first method is truncated normal distribution, which is a normal distribution with a given standard deviation (in our case 0.01) with finite range by truncating the extreme values (those that are too low/too high) [Figure 20]. Using truncated normal is a good way to overcome saturation that may occur from the activation function. For example, let's consider sigmoid function [Figure 19]. Because the sigmoid function tends to become flat for larger/smaller values, this means that the activation function will become saturated and the gradients will start approaching zero (meaning that the learning will stop). Similarly, xavier initialization is a gaussian distribution of zero mean and a fixed variance. Having a fixed variance at each layer helps prevent the value from exploding or vanishing to zero. These two initialization results gave us comparable test accuracy [Figure 21], but we decided to keep xavier initialization for our best architecture because it takes into account the number of input and output neurons in each layer into calculating the variance, and is also said to perform better.

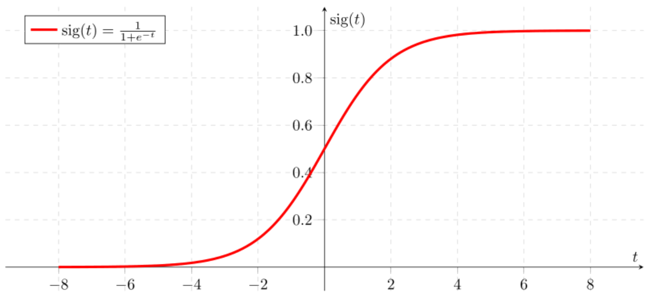


Figure 19. Sigmoid Activation Function (Wikimedia)

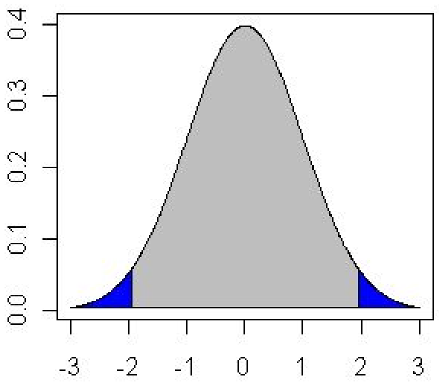


Figure 20. Truncated Normal Distribution (https://www.statsref.com/)

Weight Initialization	Test Accuracy
Truncated Normal	0.789
Xavier	0.781

Figure 21. 16 conv layer network w/o normalization

References

ReLU and variants picture:
[1] Taposh Dutta-Roy. Medical Image Analysis with Deep Learning — II. https://medium.com/@taposhdr/medical-image-analysis-with-deep-learning-ii-166532e964e6. Accessed June 15, 2018.

Cross-entropy picture:
[2] Cross-Entropy. http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html. Accessed June 15, 2018.

ADAM picture:
[3] Jason Brownlee. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. Accessed June 15, 2018.

[4] Matthew Mayo. Neural Network Foundations, Explained: Activation Function https://www.kdnuggets.com/2017/09/neural-network-foundations-explained-activation-function.html. Accessed June 15, 2018.

[5] Andrew Ng, “Machine Learning,” https://www.coursera.org/learn/machine-learning. Accessed June 15, 2018.

[6] CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/convolutional-networks. Accessed June 15, 2018.

[7]Designing machines that see. https://www.embedded-vision.com/. Accessed June 15, 2018.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. https://arxiv.org/pdf/1512.03385.pdf. Accessed June 15, 2018.