Blood Cell Classification

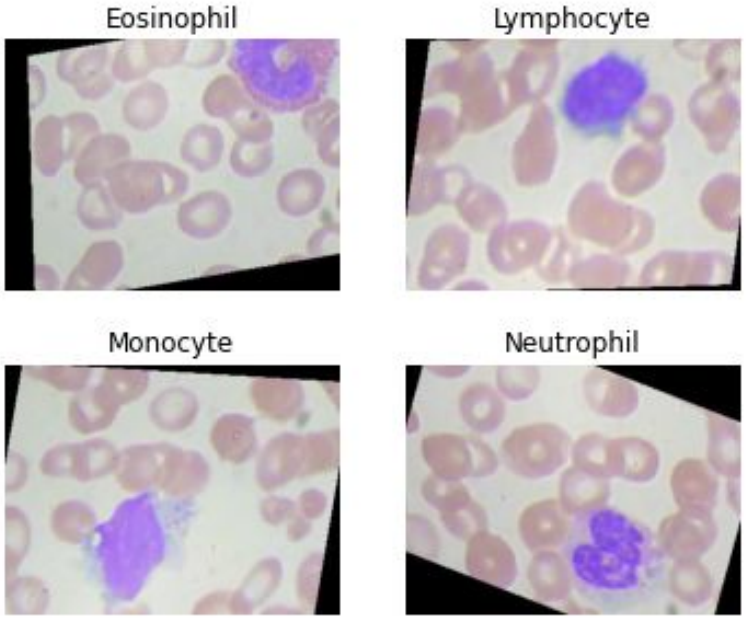
Alba Garcia Vazquez, Miren Lamaison, Edwin Vargas, Fynn Wolf

All group members contributed evenly to this project

UNIVERSITY OF COPENHAGEN

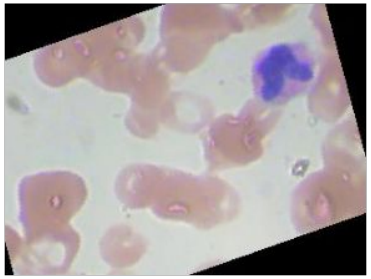
Blood Cell Data Introduction

- Kaggle dataset: 12,500 (320 x 240 pixels) augmented labeled images of blood cells
- The diagnosis of blood-based diseases often involves identifying and characterizing patient blood samples.
- Automated methods to detect and classify blood cell subtypes have important medical applications.
- **Goal: Classify images into four classes**

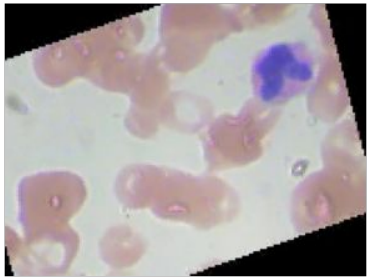


Data Preparation: Overview

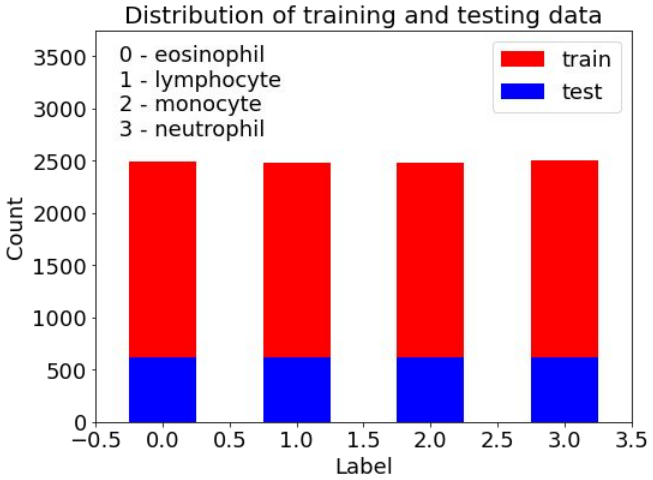
- Compression: 240x320 → 120x160 pixels (using Pillow).
- Data was already pre-augmented to be evenly distributed among the four classes.
- Scaling: four different methods.
- Segmentation: Otsu's thresholding.
- U-Net: fully convolutional network to produce masks.



Original (240x320)



Compressed (120x160)



Scaling

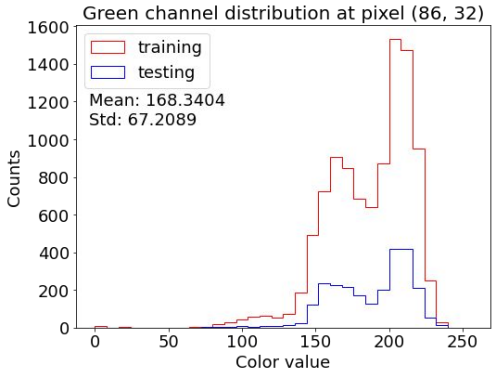
- Four different ways of scaling the train data:

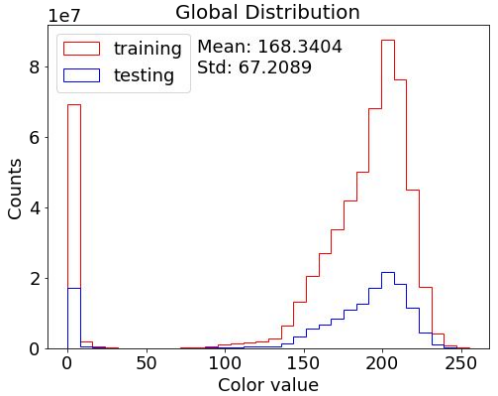
1) Pixel/channel scaling- Standardize each pixel and channel across all images

2) Global scaling- Standardize all pixels and channels using one distribution.

3) Channel scaling- Standardize each channel.

4) Normalization: divide everything by 255 to obtain values between 0 and 1





Scaling

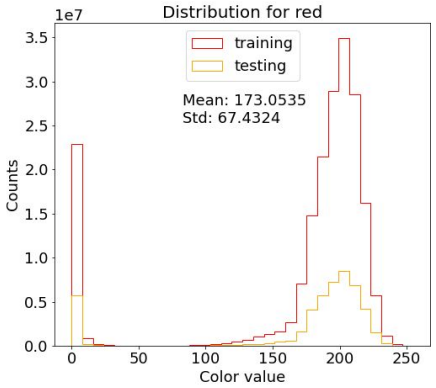
- Four different ways of scaling the train data:

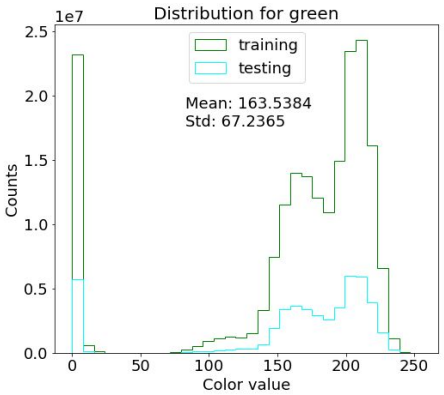
1) Pixel/channel scaling- Standardize each pixel and channel across all images

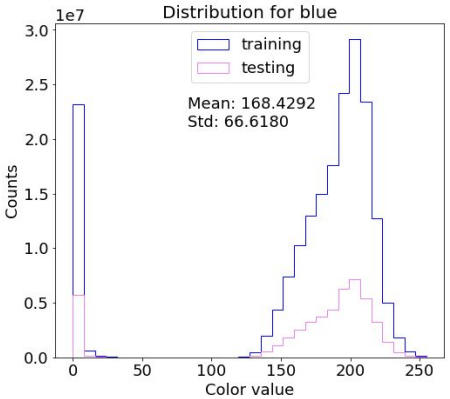
2) Global scaling- Standardize all pixels and channels using one distribution.

3) Channel scaling- Standardize each channel.

4) Normalization: divide everything by 255 to obtain values between 0 and 1

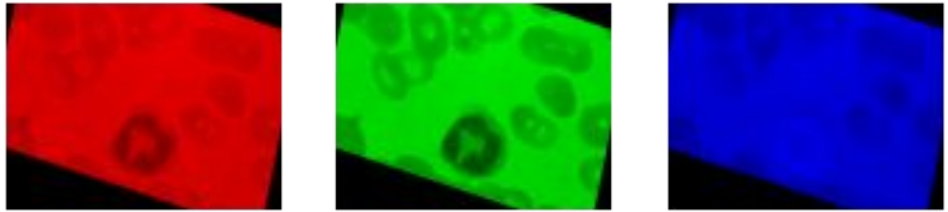


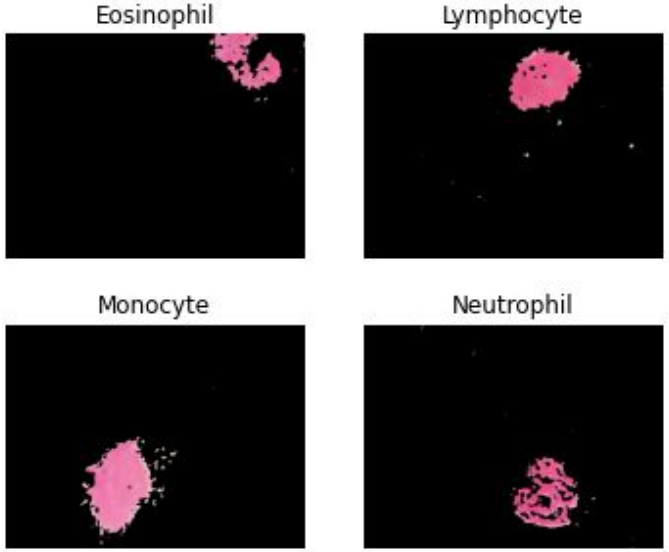




Segmentation

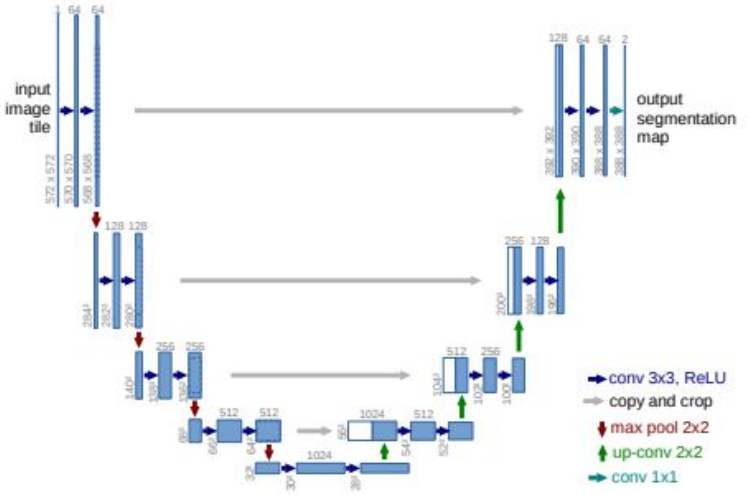
- Implemented based on Otsu's threshold values across the rgb channels.
- The threshold minimizes intra-class variance and maximizes inter-class variance.
- We find the threshold value between the two distributions for each channel.
- This mask works well for most images.





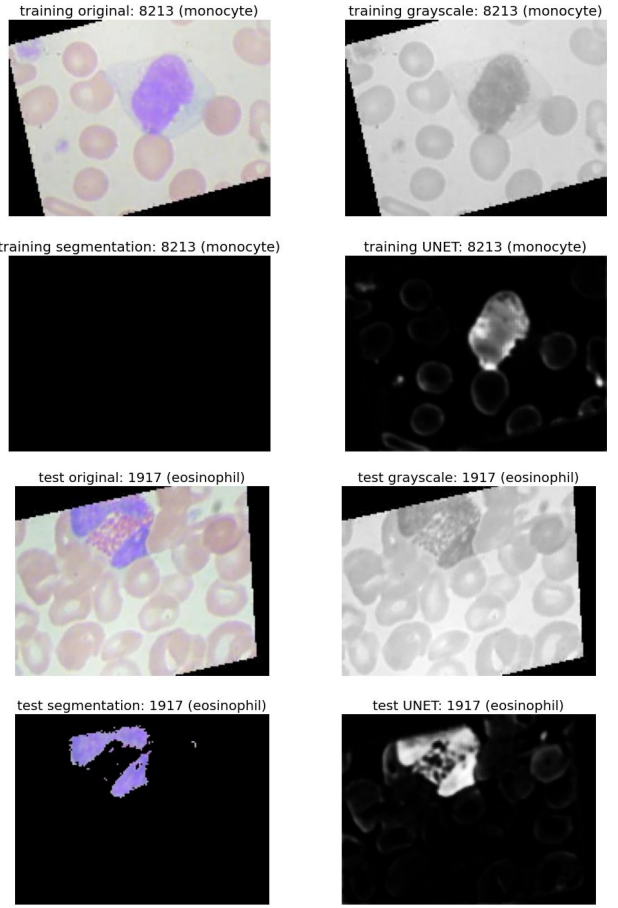
U-Net

- Fully convolutional network: classifies every pixel within an image as part of some prespecified class (semantic segmentation).
- Takes the image as an input and a mask as a label. The trained network can predict appropriate masks for other unseen images.
- Structure consists of two paths: encoding the image and decoding it, outputting a classification mask
- Used the masks obtained from segmentation and grayscaled images to implement the U-Net.



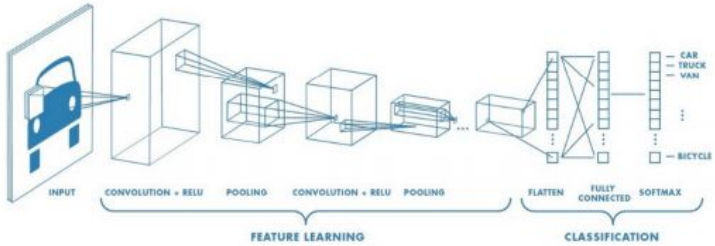
U-Net

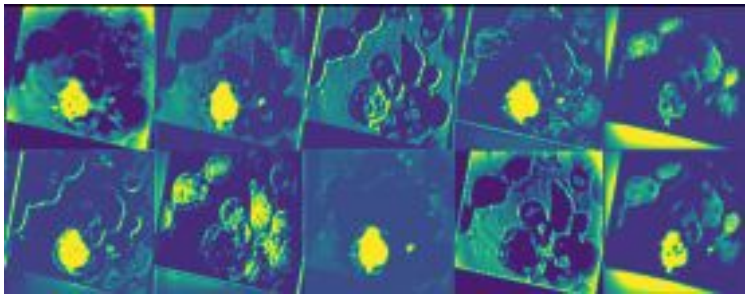
- Fully convolutional network: classifies every pixel within an image as part of some prespecified class (semantic segmentation).
- Takes the image as an input and a mask as a label. The trained network can predict appropriate masks for other unseen images.
- Structure consists of two paths: encoding the image and decoding it, outputting a classification mask
- Used the masks obtained from segmentation and grayscaled images to implement the U-Net.



Convolutional Neural/Fully Convolutional Networks (CNN/FCN)

- CNNs use filters to assign importances to various aspects of the input image.
- Neural layers are then used to output the target parameter.
- FCNs: same as CNNs but with convolutional layers instead of neural layers at the end.
- Both use dropout and maxpooling layers.





CNN/FCN Structures

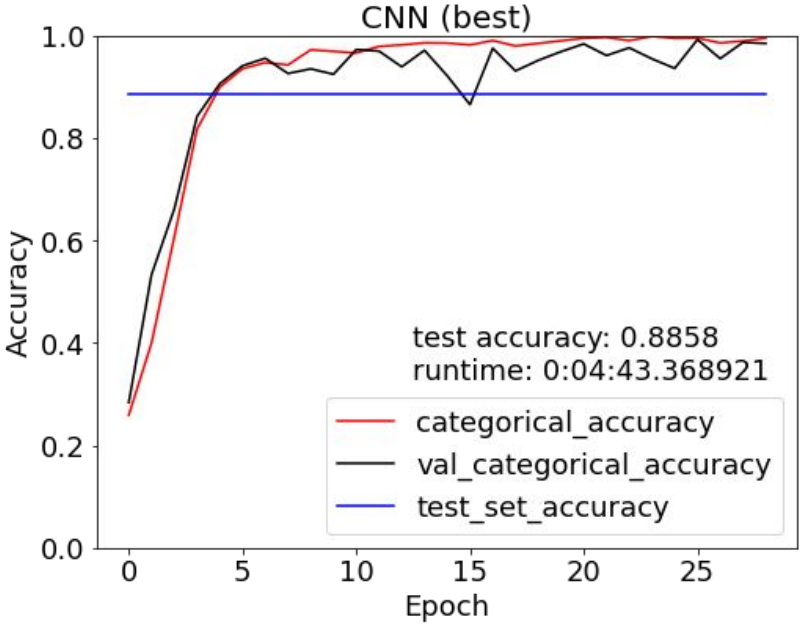
- Both CNN and FCN use five sections each with a convolution, maxpool, and dropout layer. First maxpool layer pools as (3, 4) to transform 120 x 160 to 40 x 40.
- CNN ends with a dropout layer and two dense layers.
- FCN ends with two more sections without maxpooling. The image size has been reduced to (1, 1)
- Hidden layers: ReLu as activation.
  Output layer: softmax as activation.
- Loss function: categorical crossentropy.
- Best CNN: 3,312,964 trainable parameters.
  Kernel size: 2x2. Dropout rate: 0.3.
- Best FCN: 2,853,892 trainable parameters.
  Kernel size: 2x2. Dropout rate: 0.1.

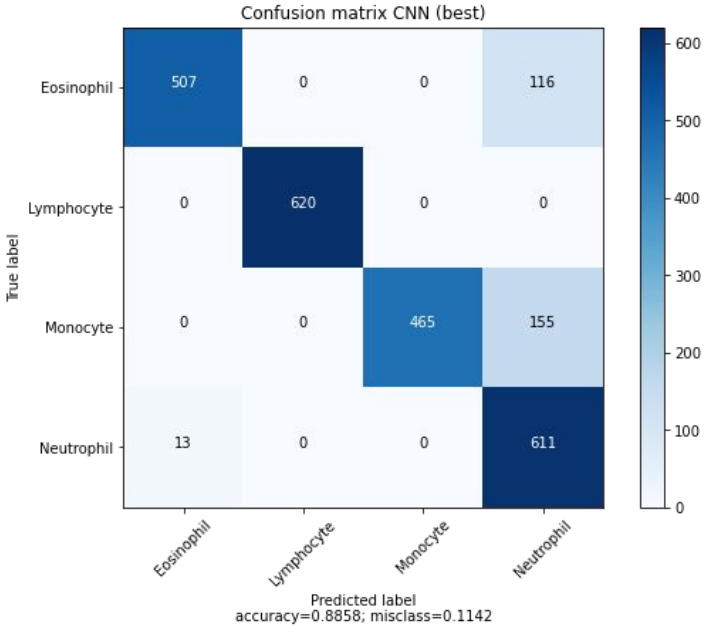
Aiming for Success/Hyperparameterization

- Scalings: channel scaling proved the best, followed closely by global scaling. Channel/pixel scaling and normalization had limited accuracies; the latter had stability issues.
- Masks: segmentation and U-Net masking yielded unstable and inaccurate results.
- Optimizers: Adam proved to be the most stable and best performing optimizer using a learning rate of 0.001.
- Kernel size: 2 for the convolutional layers resulted in the highest accuracies
- Batch size: 128 was both reliable and fast.
- Unable to properly hyperparameterize (K-Fold, Talos would crash due to RAM limitations).

CNN Results

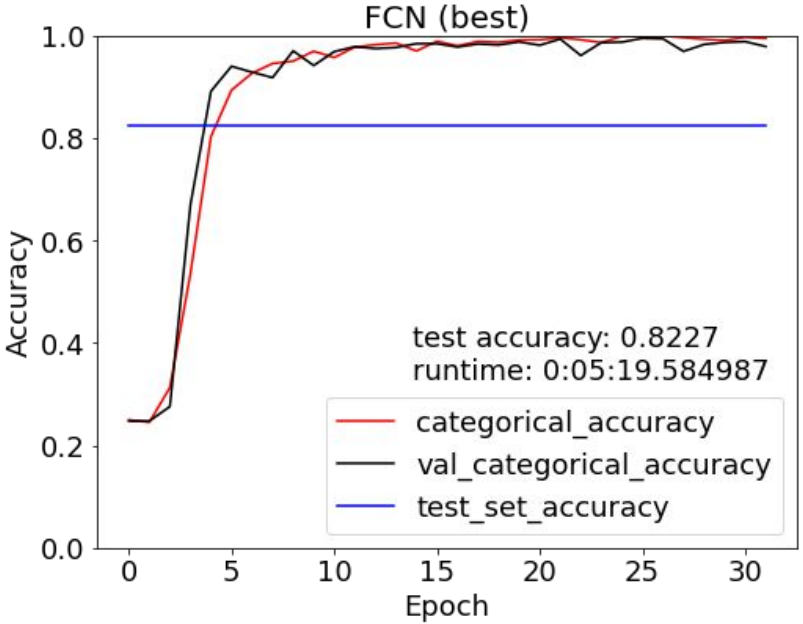
- Best results were obtained using channel scaling, no segmentation, and no u-net.
- Dropout rate: 0.3; Kernel Sizes: (2,2,2,2,2).

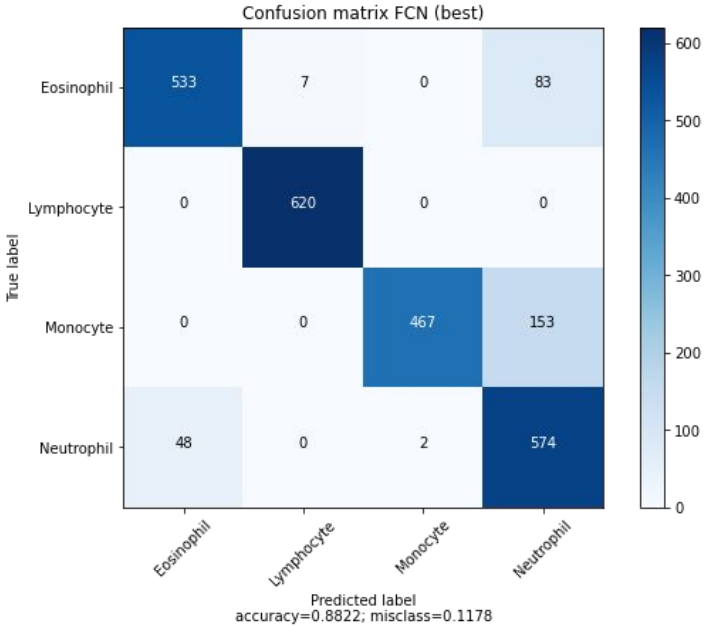




FCN Results

- Best results were obtained using channel scaling, no segmentation, and no u-net.
- Dropout rate: 0.1; Kernel Sizes: (2,2,2,2,2).





Summary of CNN and FCN Results

Method	Accuracy (10 trials)	Model Size (MB)	Epoch Number	Convergence
Best CNN (channel scaled, no mask)	0.87 ± 0.02 [0.826, 0.887]	39.776155	30 ± 5	100%
Best FCN (channel scaled, no mask)	0.882 ± 0.007 [0.870, 0.897]	34.267841	26 ± 8	90%

Kaggle Contributor (Top 4 most popular)	Method	Accuracy
Paul Mooney	CNN	0.838 (max)
nh4cl	CNN	0.8376
Kartik Sharma	CNN	0.85766
ilovescience	CNN	0.8709

Possible Improvements and Refinements.

- Try handmade segmentation with data augmentation for U-Net.
- Greater RAM: raw images and proper hyperparameterization (Talos).
- Keras has the ability to read images from directory: raw images.
- Add blank pixels to obtain square images.

Conclusion

- Masking and U-Net did not provide any additional accuracy, which was disappointing.
- Global and channel standardizations are useful to push accuracies higher. Standardization is necessary to converge reliably.
- Overall, it seems that FCN's can push accuracy higher but suffer from stability issues (in terms of convergence) when compared to similarly structures CNN's.
- It is possible to automate the detection and classification of blood cell subtypes by using ML algorithms.

The End.

References

Blood cell dataset: https://www.kaggle.com/paultimothymooney/blood-cells

Scaling/standardization: https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning/

Otsu thresholding: https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html

U-Net Implementation - https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47

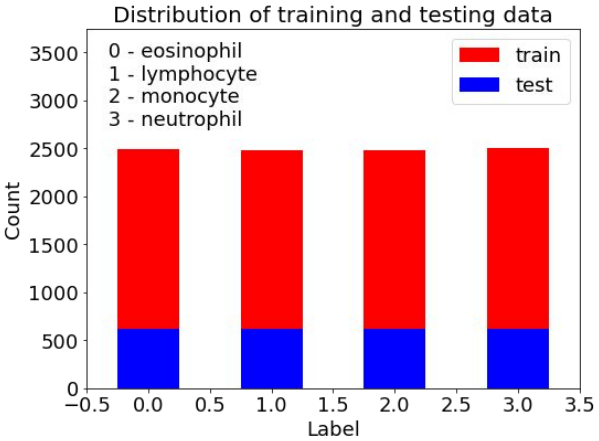
Image U-Net - Ronneberger, O., Fischer, P., and Brox, T. U-Net: Convolutional Networks for Biomedical Image Segmentation. [arXiv:1505.04597v1]

CNN - Lecture notes and sample class notebook.

FCN - https://towardsdatascience.com/implementing-a-fully-convolutional-network-fcn-in-tensorflow-2-3c46fb61de3b

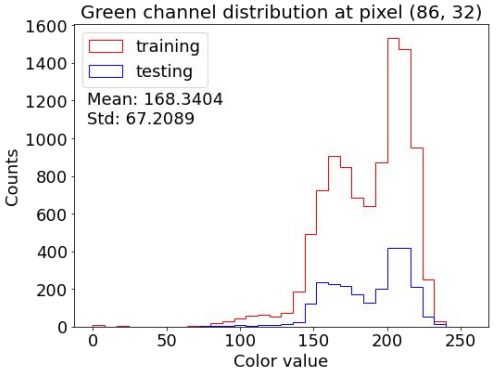
Appendix - Data Preprocessing

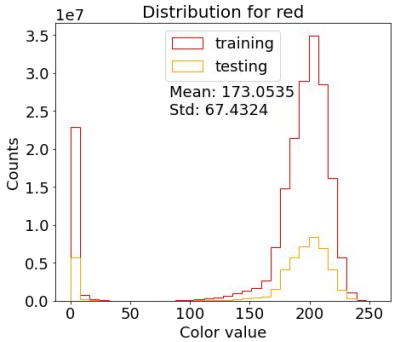
1) Check data is balanced for the train and test data. Data had already been augmented in the set to provide roughly equal distributions.
2) Image compression/resizing: 320x240 → 160x120 pixels (done using Pillow). Tried full resolution, but cumbersome and unreliable (RAM crashes, can only train in samples of 1000 which don't converge). With lower resolutions and 'squaring', we lose more information.
3) Standardizing the data: (best case) the training data was standardize so the mean of each rgb channel for all images was 0 and the variance equal to 1. This values were safe and then applied to standardize the test data.

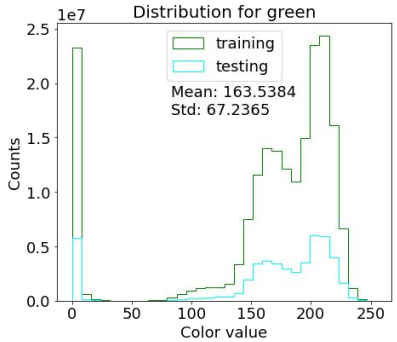


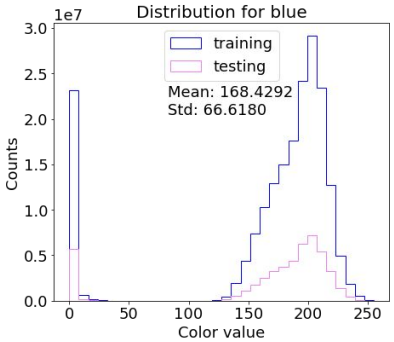
Appendix: Scaling

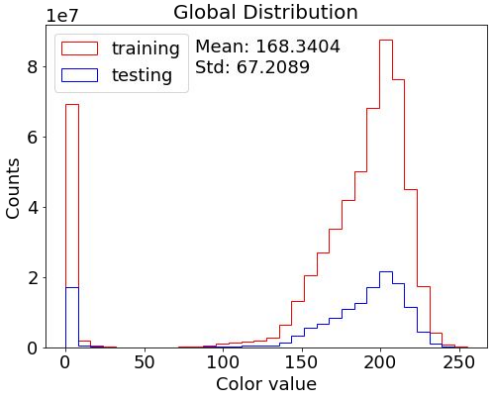
- Four different ways of scaling the train data:
  1) Pixel/channel scaling- Standardize each pixel/channel individually across all images
  2) Global scaling- Standardize all pixels and channels using one distribution.
  3) Channel scaling- Standardize each channel.
  4) Normalization: divide everything by 256 to obtain values between 0 and 1.









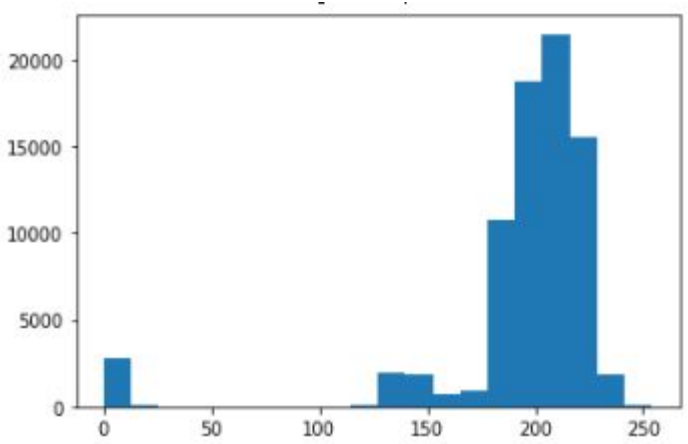


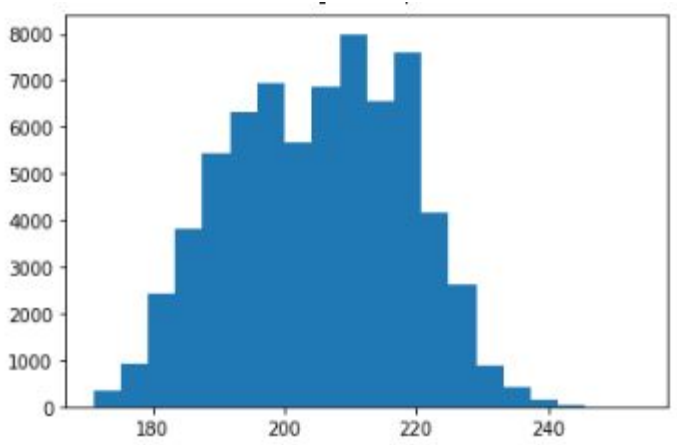
Appendix: Scaling

```python
from sklearn.preprocessing import StandardScaler

# The first and original scaler
# Finds a distribution for each channel in each pixel across all the images.
scaler_original = StandardScaler()
scaler_original.fit(np.asarray([image.flatten() for image in train_images]))

# Scales across all channels, all pixels, and all images
scaler_global = StandardScaler()
scaler_global.fit(np.asarray([image.flatten() for image in train_images]).flatten().reshape(-1, 1))

# Scales across each channel for all pixels and all images
# The greens of all the images are scaled the same, the blues of all images are scaled the same, ...
scaler_channels = [StandardScaler(), StandardScaler(), StandardScaler()]
scaler_channels[0].fit(np.asarray([image[..., 0].flatten() for image in train_images]).flatten().reshape(-1, 1))
scaler_channels[1].fit(np.asarray([image[..., 1].flatten() for image in train_images]).flatten().reshape(-1, 1))
scaler_channels[2].fit(np.asarray([image[..., 2].flatten() for image in train_images]).flatten().reshape(-1, 1))
```

Appendix: Segmentation

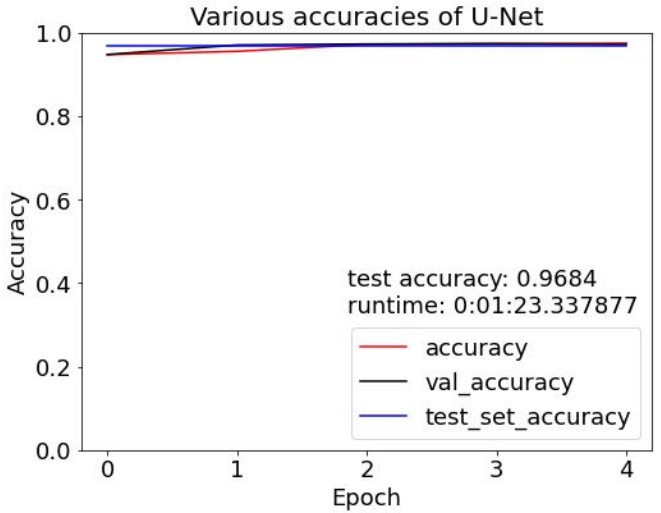
Based on Otsu's threshold. The threshold value is found between the two distributions for each channel. The graph on the left shows the full pixel distribution for one channel. The graph on the right shows the result after applying Otsu's threshold.





Appendix - U-Net Structure

U-Net accuracy/convergence result and structure:




```python
def output_UNET(input_layer, start_neurons): # 120, 160
    conv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(input_layer) # 120, 160
    conv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(conv1) # 120, 160
    pool1 = MaxPooling2D((2, 2))(conv1) # 60, 80
    pool1 = Dropout(0.25)(pool1) # 60, 80
    conv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(pool1)
    conv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(conv2)
    pool2 = MaxPooling2D((2, 2))(conv2) # 30, 40
    pool2 = Dropout(0.5)(pool2)
    conv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(pool2)
    conv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(conv3)
    pool3 = MaxPooling2D((2, 2))(conv3) # 15, 20
    pool3 = Dropout(0.5)(pool3)
    convm = Conv2D(start_neurons * 8, (3, 3), activation="relu", padding="same")(pool3)
    convm = Conv2D(start_neurons * 8, (3, 3), activation="relu", padding="same")(convm)
    deconv3 = Conv2DTranspose(start_neurons * 4, (3, 3), strides=(2, 2), padding="same")(convm)
    uconv3 = concatenate([deconv3, conv3])
    uconv3 = Dropout(0.5)(uconv3)
    uconv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(uconv3)
    uconv3 = Conv2D(start_neurons * 4, (3, 3), activation="relu", padding="same")(uconv3)
    deconv2 = Conv2DTranspose(start_neurons * 2, (3, 3), strides=(2, 2), padding="same")(uconv3)
    uconv2 = concatenate([deconv2, conv2])
    uconv2 = Dropout(0.5)(uconv2)
    uconv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(uconv2)
    uconv2 = Conv2D(start_neurons * 2, (3, 3), activation="relu", padding="same")(uconv2)
    deconv1 = Conv2DTranspose(start_neurons * 1, (3, 3), strides=(2, 2), padding="same")(uconv2)
    uconv1 = concatenate([deconv1, conv1])
    uconv1 = Dropout(0.5)(uconv1)
    uconv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(uconv1)
    uconv1 = Conv2D(start_neurons * 1, (3, 3), activation="relu", padding="same")(uconv1)
    output_layer = Conv2D(1, (1,1), padding="same", activation='sigmoid')(uconv1)
    return output_layer

img_size_target_x, img_size_target_y = train_images[0].shape[:2]

input_layer = Input((img_size_target_x, img_size_target_y, 1))
output_layer = output_UNET(input_layer, 16)

def build_model_UNET():
    return keras.models.Model(input_layer, output_layer)
```

Appendix - CNN Structure and Hyperparameters

The structure of the CNN was the following:

```
def create_model_half_image(kernel, dropout):
  model = Sequential()
  model.add(Conv2D(filters=64, strides=1, padding='same', activation='relu', input_shape=(rows, colm, dim)))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=(3,4), strides=None))
  model.add(Conv2D(filters=128, kernel_size=kernel, strides=1, padding='same', activation='relu' ))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=2, strides=None))
  model.add(Conv2D(filters=256, kernel_size=kernel, strides=1, padding='same', activation='relu' ))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=2, strides=None))
  model.add(Conv2D(filters=512, kernel_size=kernel, strides=1, padding='same', activation='relu'))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=2, strides=None))
  model.add(Conv2D(filters=1024, kernel_size=kernel-1, strides=1, padding='same', activation='relu'))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=2, strides=None))
  model.add(Dropout(rate=dropout)) #0.4
  model.add(Flatten())
  model.add(Dense(units=128, activation='relu'))
  model.add(Dense(units=4, activation='softmax'))
  model.summary()
  return model
```

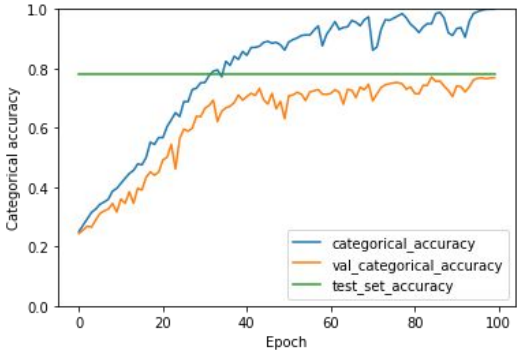
Appendix - FCN Structure and Hyperparameters

The structure of the FCN was the following:

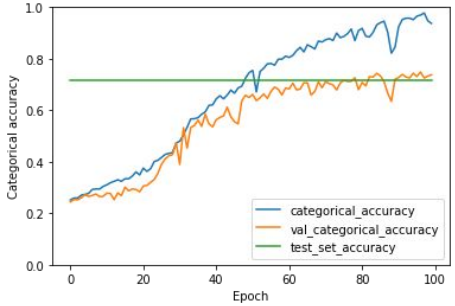
```
def create_model_half_image_fcn(kernel, dropout):
  model = Sequential()
  model.add(Conv2D(filters=64, kernel_size=kernel, strides=1, padding='same', activation='relu', input_shape=(rows, colm, dim)))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=(3,4), strides=None))
  model.add(Conv2D(filters=128, kernel_size=kernel, strides=1, padding='same', activation='relu'))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=2, strides=None))
  model.add(Conv2D(filters=256, kernel_size=kernel, strides=1,padding='same', activation='relu'))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=2, strides=None)
  model.add(Conv2D(filters=512, kernel_size=kernel, strides=1, padding='same', activation='relu' ))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=2, strides=None)
  model.add(Conv2D(filters=1024, kernel_size=kernel, strides=1, padding='same', activation='relu'))
  model.add(Dropout(rate=dropout))
  model.add(MaxPooling2D(pool_size=5, strides=None))
  model.add(Dropout(rate=dropout))
  model.add(Conv2D(filters=64, kernel_size=1, strides=1, padding='same', activation='relu'))
  model.add(Dropout(rate=dropout))
  model.add(Conv2D(filters=4, kernel_size=1, strides=1, padding='same', activation='softmax',))
  model.add(GlobalMaxPooling2D())
```

Early Attempts with Different Compressions and Unscaled Data

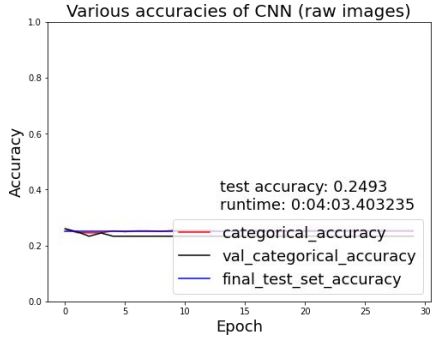
30 x 30 image



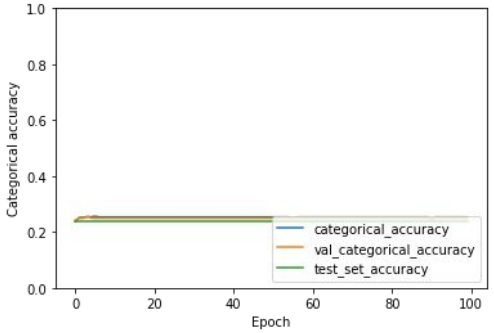
30 x 40



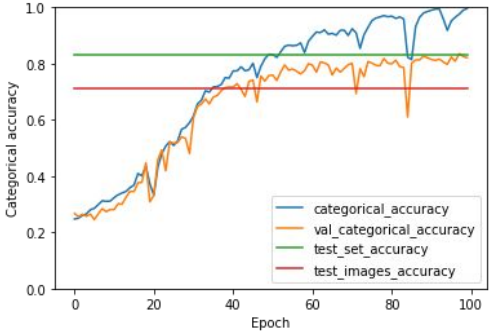
120 x 160

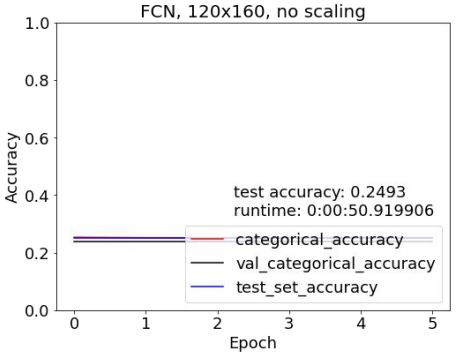


60 x 60 image

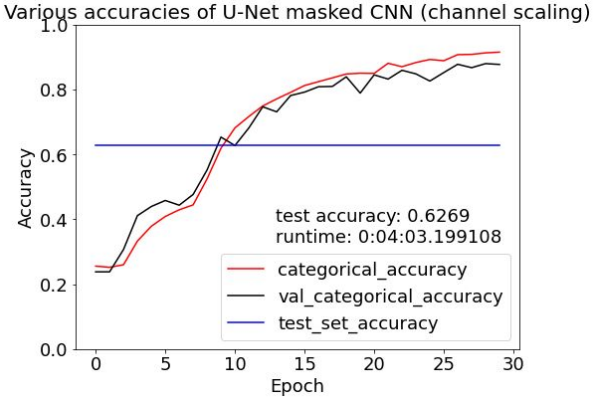


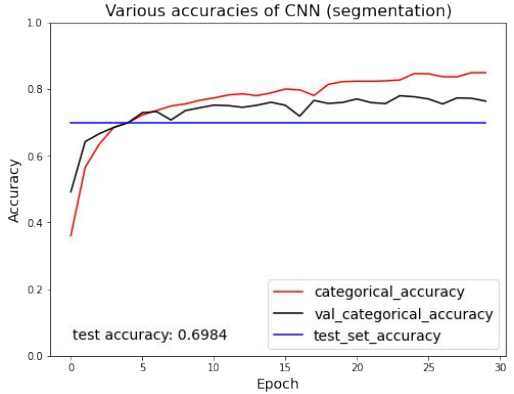
60 x 80

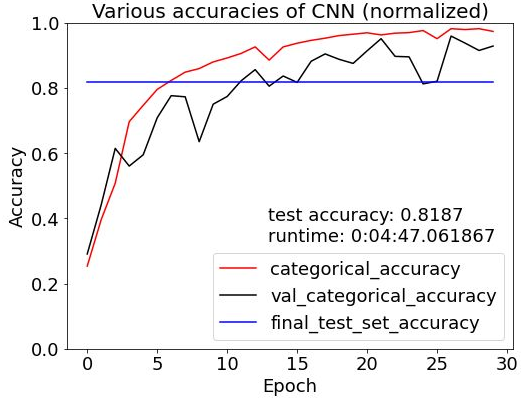


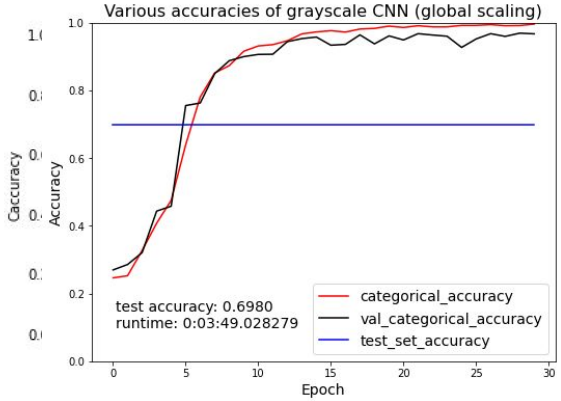


Appendix: Plots of various results CNN (1 of 2)

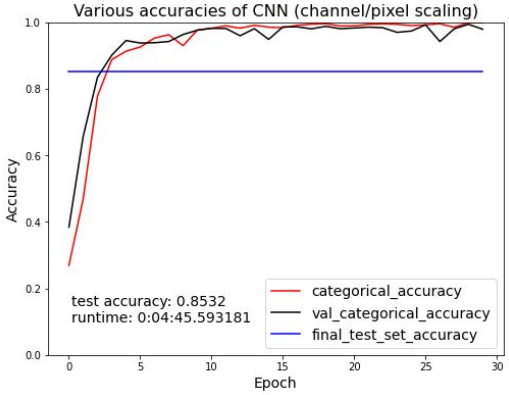


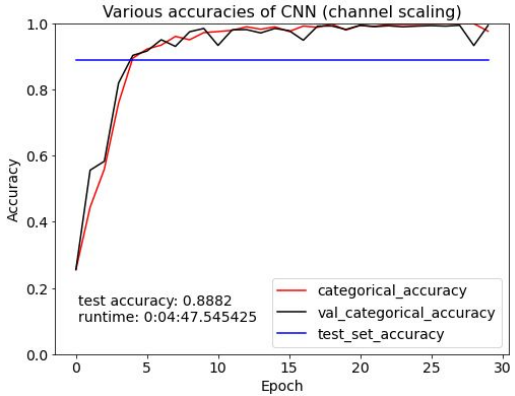


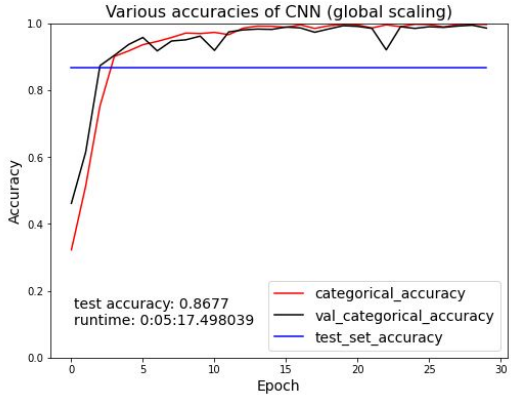




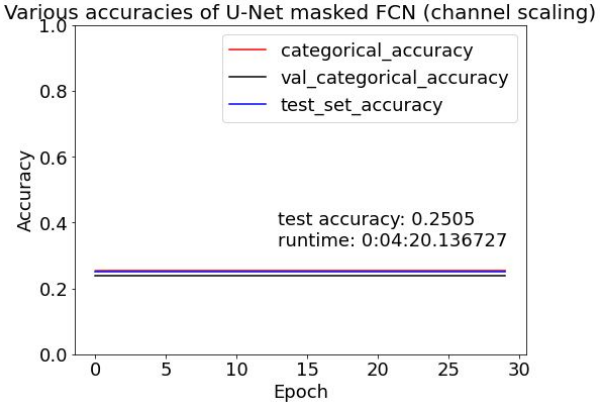
Appendix: Plots of various results CNN (2 of 2)

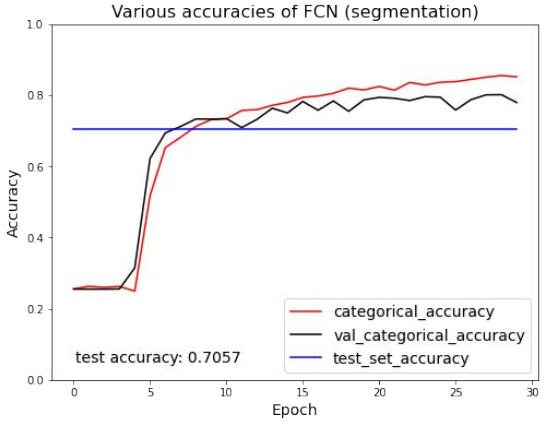


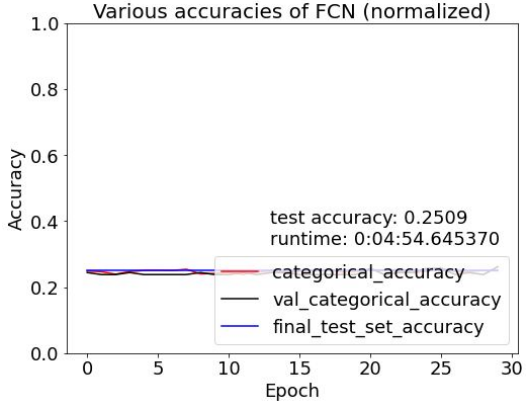


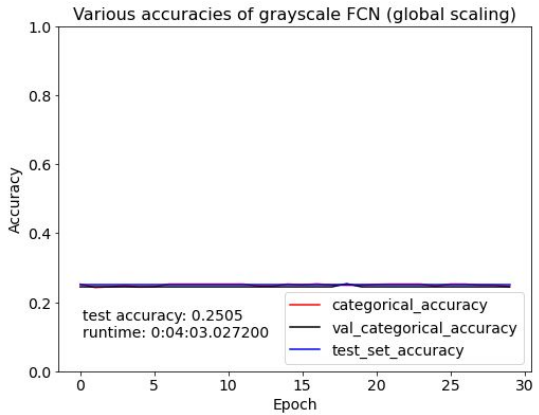


Appendix: Plots of various accuracies FCN (1 of 2)

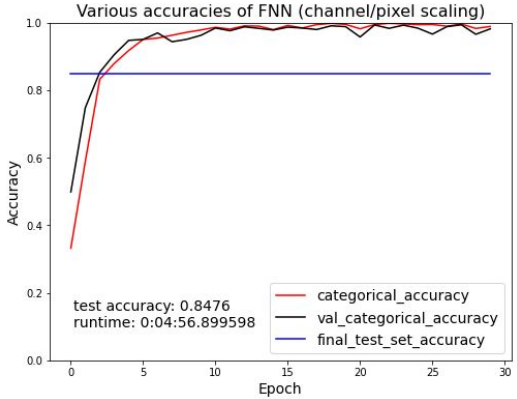


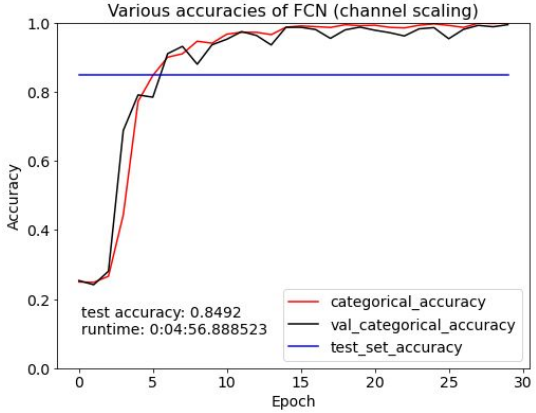


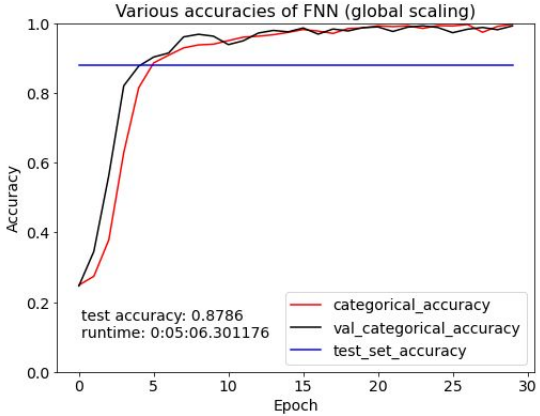




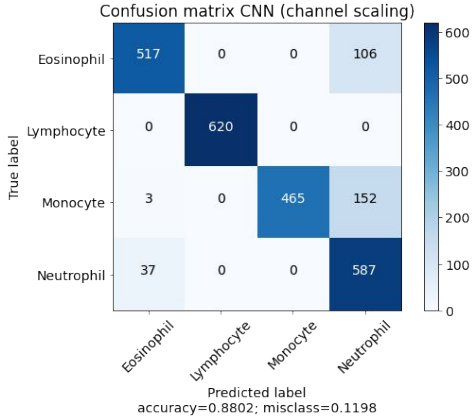
Appendix: Plots of various accuracies FCN (2 of 2)

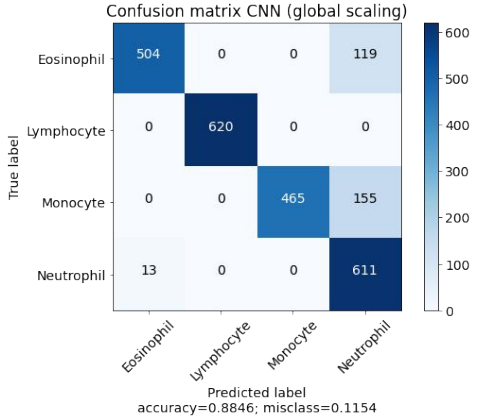


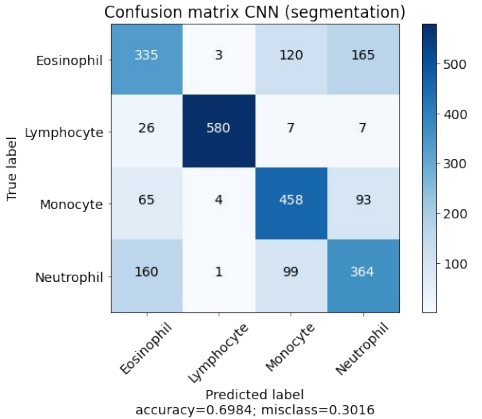


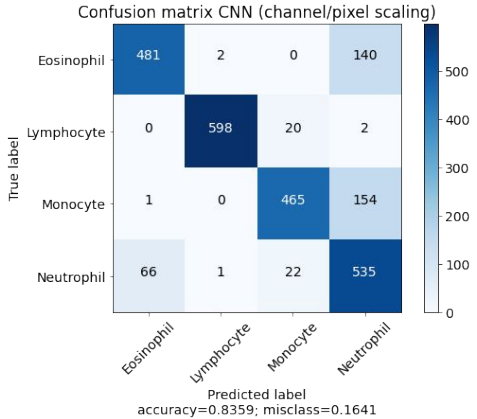


Appendix - Confusion matrices CNN

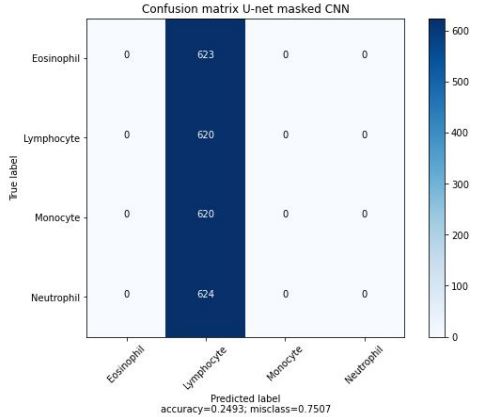




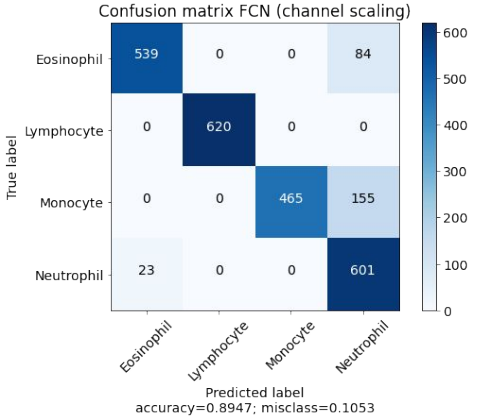


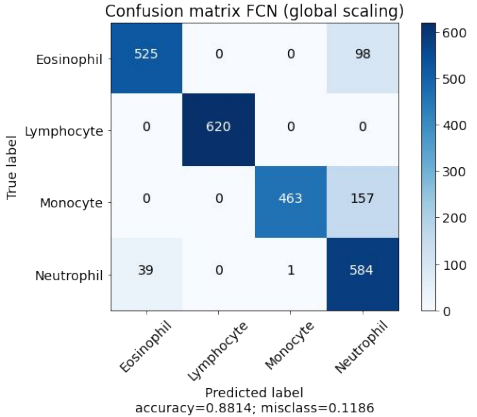


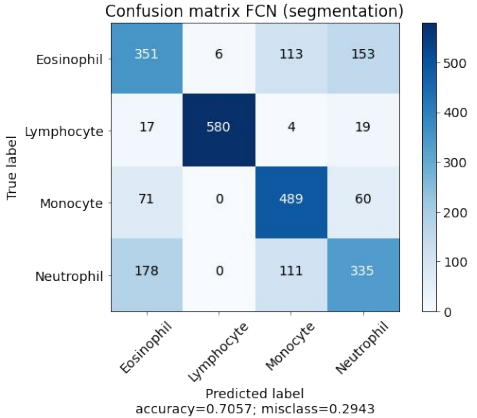
- The two most common errors are labeling monocyte and eosinophil cells as neutrophil cells.
- The labeling of lymphocyte cells is always the one with the maximum accuracy.

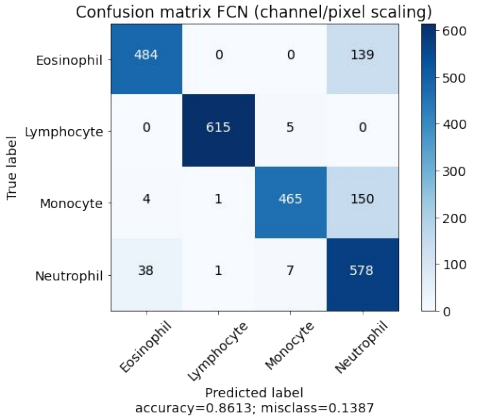


Appendix - Confusion matrices FCN

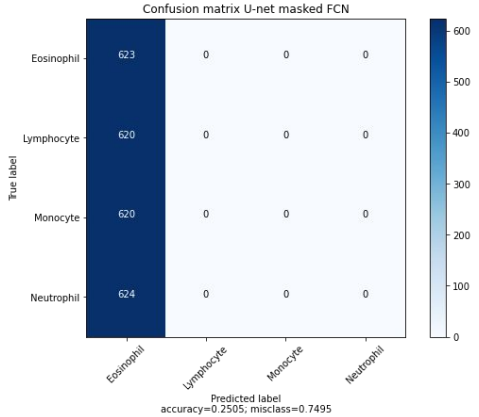








- The two most common errors are labeling monocyte and eosinophil cells as neutrophil cells.
- The labeling of lymphocyte cells is always the one with the maximum accuracy.



Optimization

- Optimizer: Adam (optimal), Adadelta (crashed), Adamax (worse)
- Adam learning rates: 0.1 (useless), 0.01(useless), 0.001, 0.0001(crashed)
- Dropout rate: 0.1, 0.2, 0.3, 0.4, 0,5 depends on kernel size and scaling maybe
- Kernel size: 2, 3, 2 seems to yield better results but is less stable, could be indicator for suboptimal structure
- Batch size: 32, 64, 128, 256, not sure, lower numbers look like they are worse.
- Also considered other compression rates and squaring the images, but since we got the 120 by 160 compression to work, we just moved on with this.
- Once we were set on a structure and saw that global and channel scaling got the best results, we started optimizing and logging results (see next three slides). Other results in the lead up to this are not included.

Appendix: Manual Hyperparametrization configurations (1 of 3)

Network	Dropout Rate	Kernel Sizes	Standardization	Batch Size	Accuracy (6 runs)
FCN	0.2	(2, 2, 2, 2, 2, 1, 1)	Global	32	0.874, std = 0.008
FCN	0.5	(2, 2, 2, 2, 2)	Channel	128	max .830, 4 non-conv
FCN	0.2	(2, 2, 2, 2, 2)	Channel	32	.869 - .898
CNN	0.2	(4, 3, 3, 4, 4)	Global	128	0.869, 1 non-conv
FCN	0.1	(2, 2, 2, 2, 2)	Channel	16	~0.88, 1 non-conv
FCN	0.2	(3, 3, 3, 3, 3, 1, 1)	Global	256	0.859, std = 0.029
CNN	0.1	(3, 3, 3, 3, 3)	Channel	32	Max .86, 3 non-conv
FCN	0.2	(4, 4, 3, 3, 2, 1, 1)	Global	128	0.874, std = 0.008
FCN	0.2	(3, 3, 2, 2, 1, 1, 1)	Global	128	0.879, std 0.005

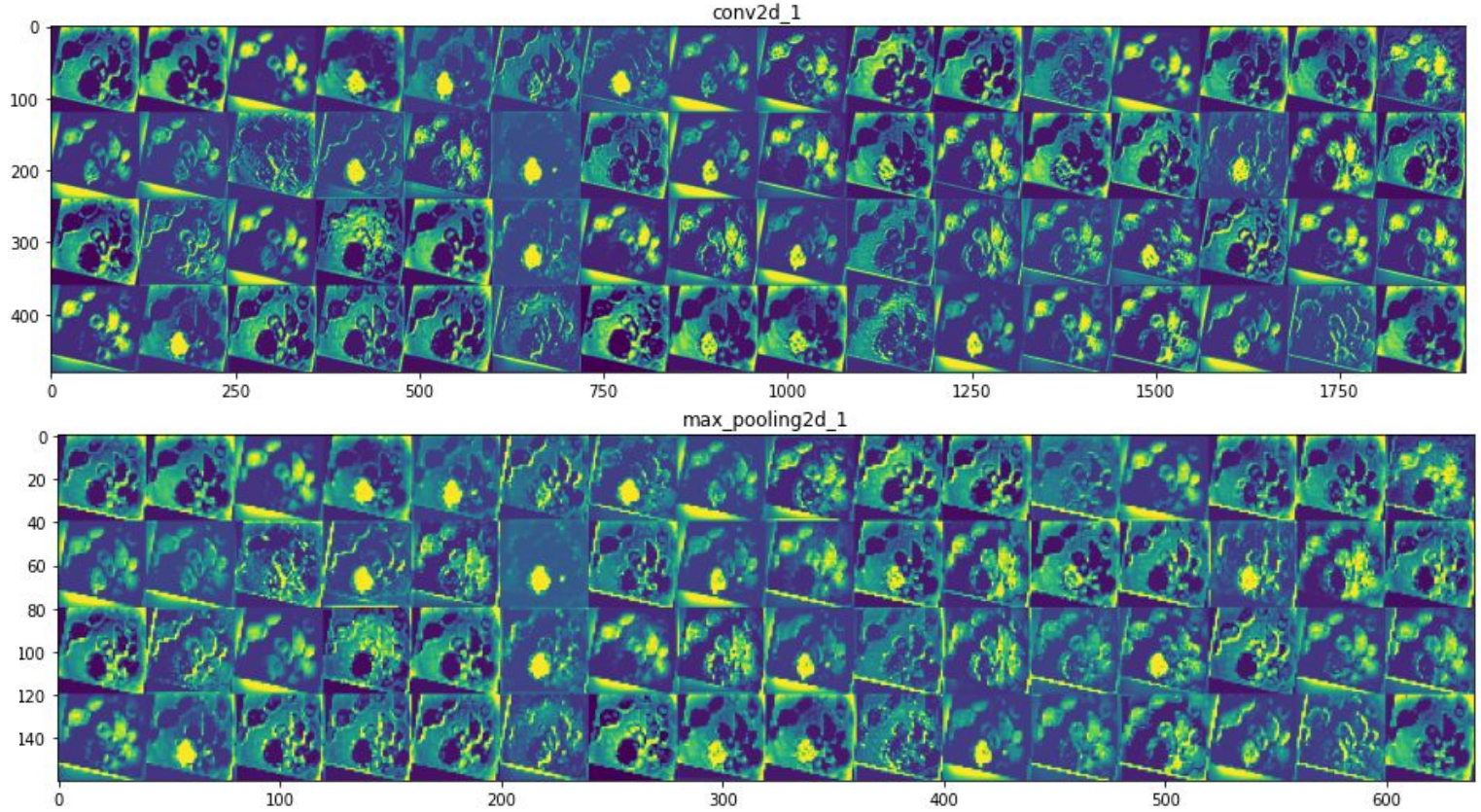
Appendix: Manual Hyperparametrization configurations (2 of 3)

Network	Dropout Rate	Kernel Sizes	Standardization	Batch Size	Accuracy (6 runs)
CNN	0.1	(3, 3, 3, 3, 3)	Channel	128	~0.86 (max 0.87)
FCN	0.3	(2, 2, 2, 2, 2, 1, 1)	Global	128	0.875, max 0.8914, 1 non conv
CNN	0.5	(3, 3, 3, 3, 3)	Channel	64	0.86 pm 0.02
FCN	0.3	(2, 2, 2, 2, 2, 1, 1)	Global	64	0.883, 1 non conv
CNN	0.3	(2, 2, 2, 2, 2)	Channel	128	0.879 pm 0.008, .89 max
FCN	0.4	(2, 2, 2, 2, 2, 1, 1)	Global	128	0.88, 4 non conv
FCN	0.2	(2, 2, 2, 2, 2, 1, 1)	Global	32	0.887, 0.006, 1 non
FCN	0.2	(4, 4, 2, 2, 2, 1, 1)	Global	32	0.865, 0.018, 1 non

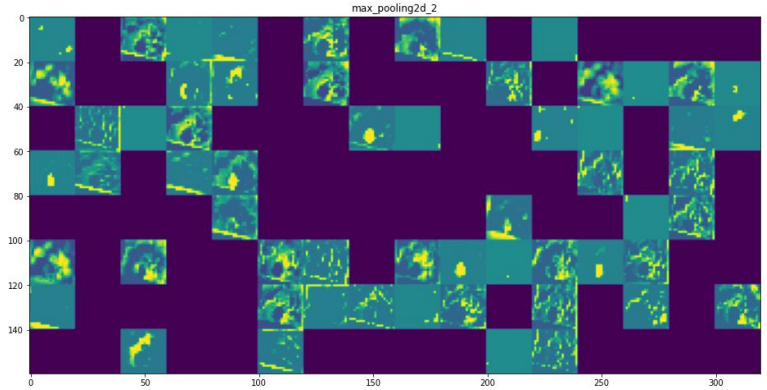
Appendix: Manual Hyperparametrization configurations (3 of 3)

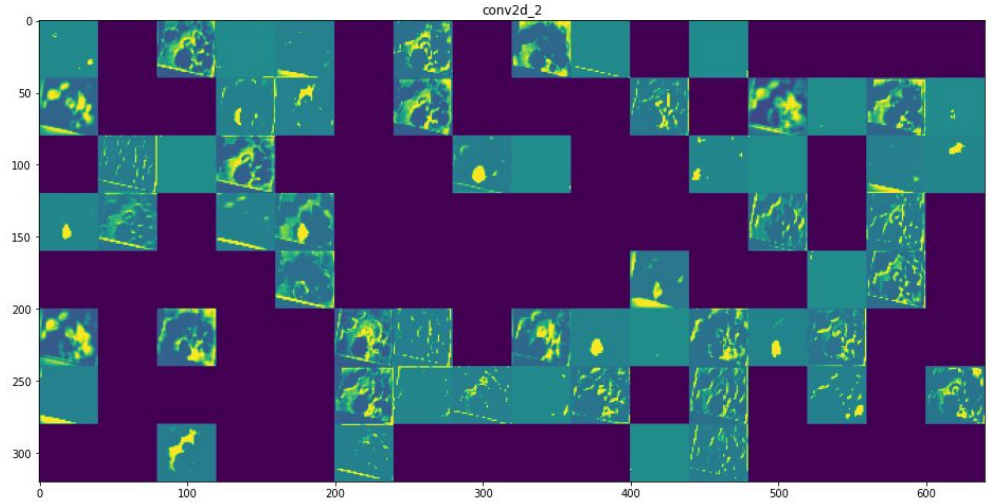
Network	Dropout Rate	Kernel Sizes	Standardization	Batch Size	Accuracy (6 runs)
CNN	0.3	(3, 3, 3, 3, 3)	Global	128	~0.88 (little variance)
FCN	0.2	(2, 2, 2, 2, 2, 1, 1)	Global	128	0.88, std = 0.01
CNN	0.2	(3, 3, 3, 2, 2)	Global	128	0.871
CNN	0.1	(2, 2, 2, 2, 2)	Channel	128	~0.87 (little var)
CNN	0.5	(2, 2, 2, 2, 2)	Channel	128	B/w 82% and 88.8%
FCN	0.2	(3, 3, 3, 2, 2, 1, 1)	Global	128	0.869, std = 0.02
FCN	0.1	(2, 2, 2, 2, 2)	Channel	128	.87 to 0.907
FCN	0.4	(3, 3, 3, 3, 3, 1, 1)	Global	128	~0.88, 1 non conv
FCN	0.2	(2, 2, 2, 2, 1, 1, 1)	Global	128	0.886, 2 non

Pictures from convolutional layers and max pooling layers (1of 4)

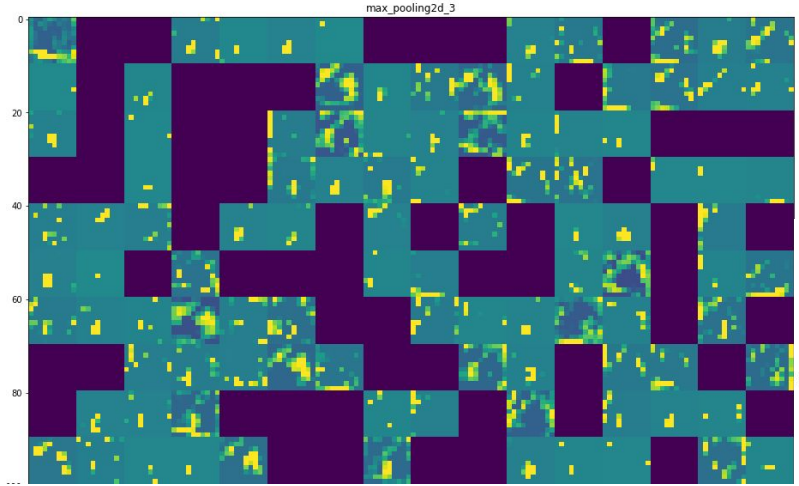


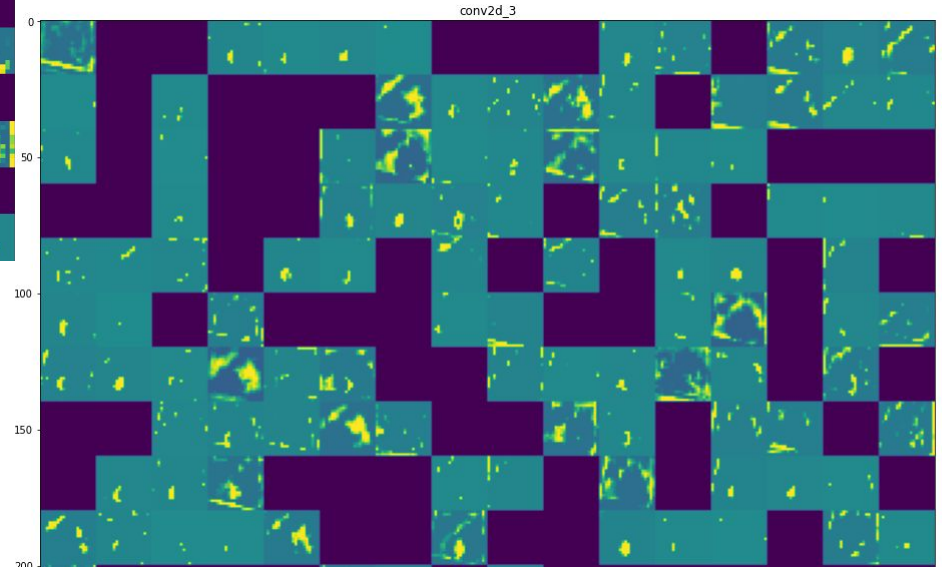
Pictures from convolutional layers and max pooling layers (2 of 4)





Pictures from convolutional layers and max pooling layers (3 of 4)





Pictures from convolutional layers and max pooling layers (4 of 4)