# PRACTICUM II
# Final Presentation

## Deep Learning for Image Detection of Breast Cancer

**Angela De La Mora**
**Regis University**
**August 2019**

# Project Overview

- Binary classification: cancer vs. no-cancer tissue in medical images
- Convolutional Neural Network architecture (aka: CovNet / CNN)
- Evolutionary algorithms to find optimal hyperparameters
- Total image count: 277,524

- Dataset based on seminal paper published in 2014
- Dataset downloaded from a republishing blog/author (Janowczyk, 2015)
- Comparison with original researchers Cruz-Roa et al. (2014):

| | Precision | Recall | F-Score | Specificity | Balanced Accuracy |
|---|---|---|---|---|---|
| **Orig. paper:** | 0.6540 | **0.7960** | 0.7180 | 0.8886 | 0.8423 |
| **Our project:** | **0.7617** | 0.7926 | **0.7768** | **0.9017** | **0.8471** |

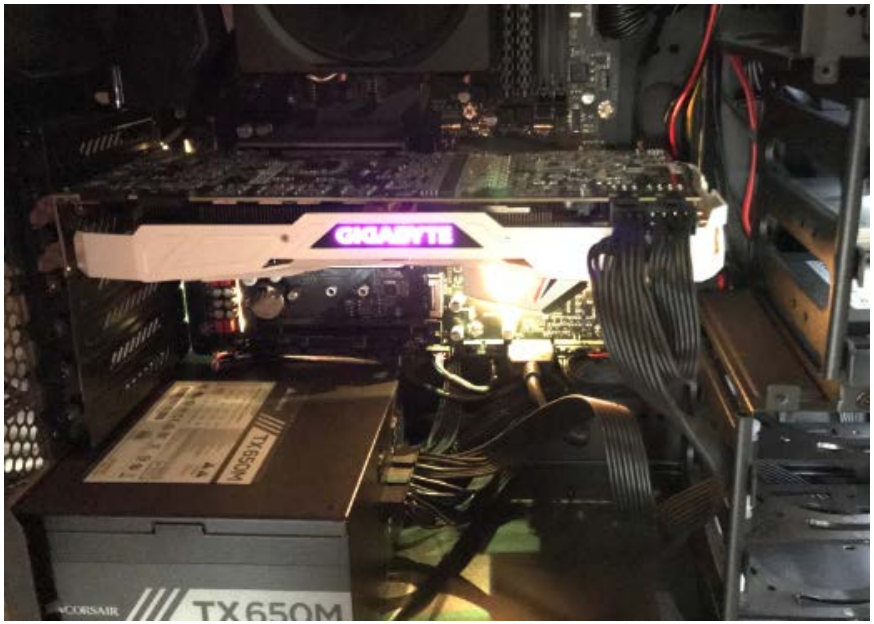
# Project Steps

- **W1 -** Project research, selection and writing of proposal.
- **W2 -** Prepare hardware/software on local PC.
- **W3 -** Data gathering, cleaning and splitting (training, validation and test sets).
- **W4 -** Exploratory architecture development; exploratory model training.
- **W5 -** Model training; further iterations in architecture and hyperparameters.
- **W6 -** Fine tune model training.
- **W7 -** Compare performance metrics and finalize model selection.
- **W8 -** Build presentation slides, upload to GitHub and present project.

# Hardware and software requirements

**Hardware:**

- GPU:  Nvidia 1080 Ti / 3584 cores / 11 GB RAM
- CPU:  AMD Ryzen 2700X / 3.7 GHz / 8 cores / 16 threads
- RAM:  16 GB

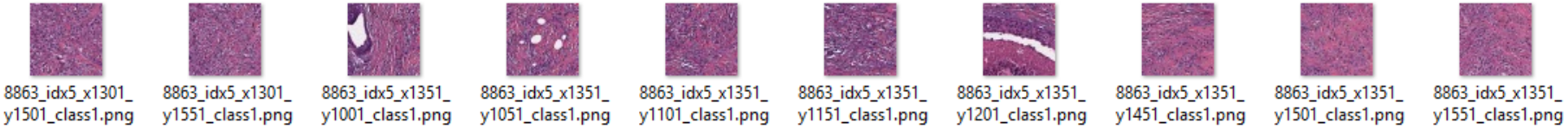



**Software:**

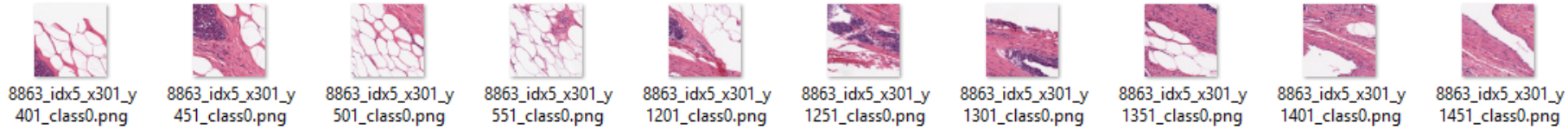
- Anaconda / Python 3.6
- Tensorflow 1.1 / Keras
- Nvidia Cuda 9.0 / cuDNN
- Jupyter Notebook

# Data collection and preparation

- Image counts:
  - o Negative class: 198,738
  - o Positive class: 78,786
  - o Total: 277,524
- Dataset size: 1.6 GB
- Image size: 50 x 50 pixels
- Mount slides of breast cancer specimens at 40x magnification
- Example positive class images (cancer):




- Example negative class images (no cancer):



# Data collection and preparation

**Categories:**

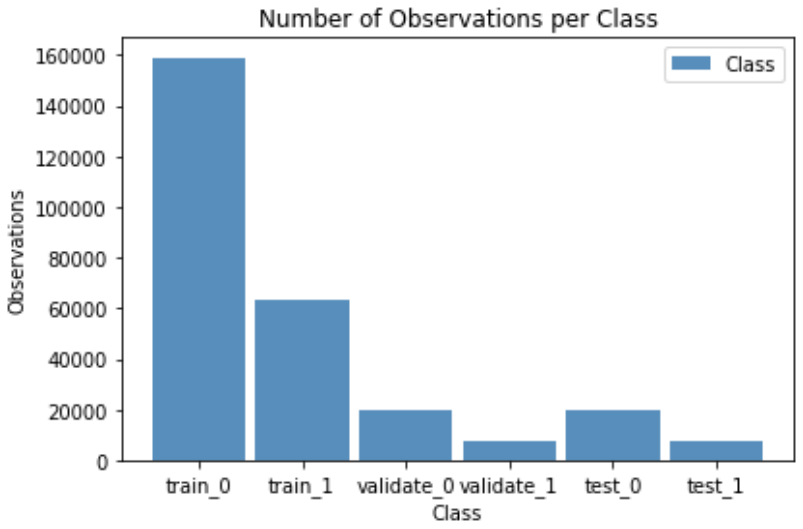
- Negative class: “0”
- Positive class: “1”

**Data splitting by set, class and number of images:**

- **Training set** – Class 0: 158,991 / Class 1: 63,029
- **Validation set** – Class 0: 19,874 / Class 1: 7,878
- **Test set** – Class 0: 19,873 / Class 1: 7,879

**Highlights:**

- Images selected at random
- Class imbalance present



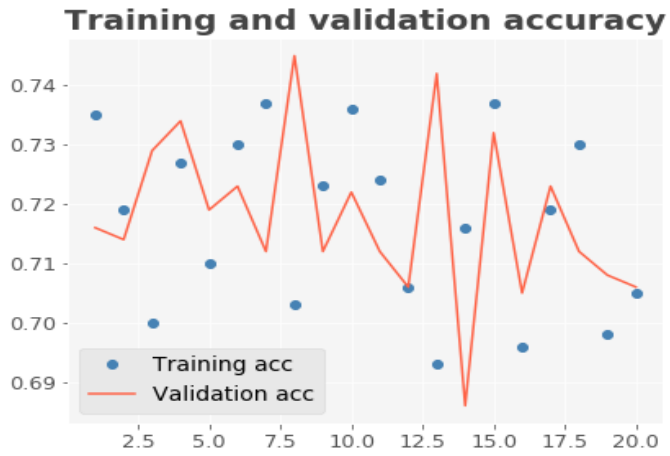
# Exploratory model architectures

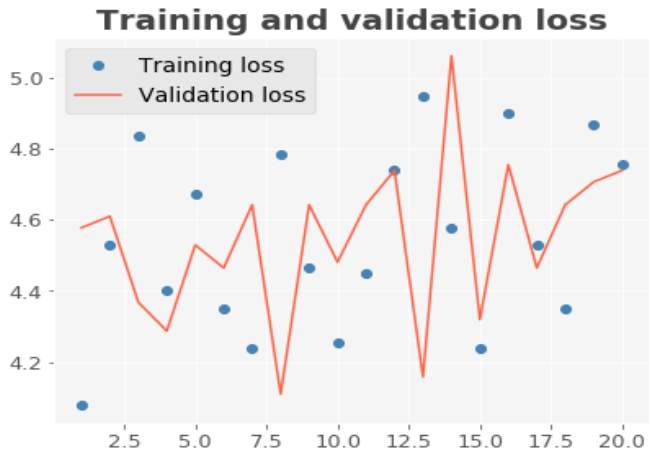
- Attempted a few simple model architectures to gauge performance
- Seven models compared
- **First model** not able to learn:

```
train_model(model_1, lr=0.01, epochs=20, epoch_steps=20, valid_steps=20, batch_size=50, image_size=(50, 50))
```

```
def model_1(lr, image_size):
    model = Sequential()

    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(image_size[0], image_size[1], 3)))

    model.add(Conv2D(64, (3, 3), activation='relu'))

    model.add(Flatten())

    model.add(Dense(128, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=lr), metrics=['acc'])
    model.summary()
    return model
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 48, 48, 32)        896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 46, 46, 64)        18496
_________________________________________________________________
flatten_1 (Flatten)          (None, 135424)            0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               17334400
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 17,353,921
Trainable params: 17,353,921
Non-trainable params: 0
```



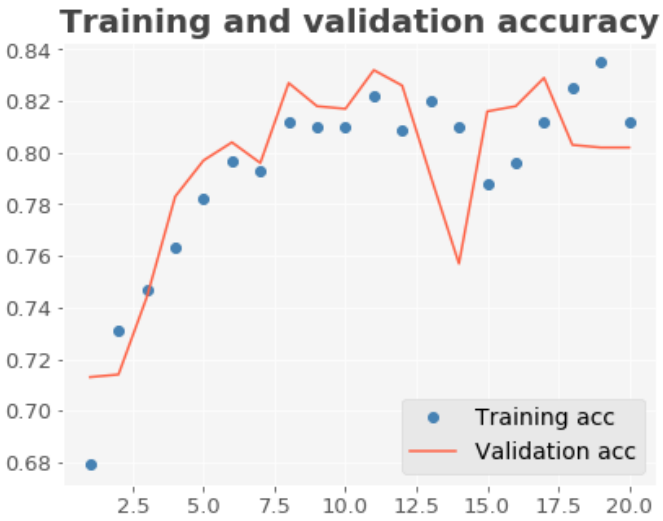
# Exploratory model architectures

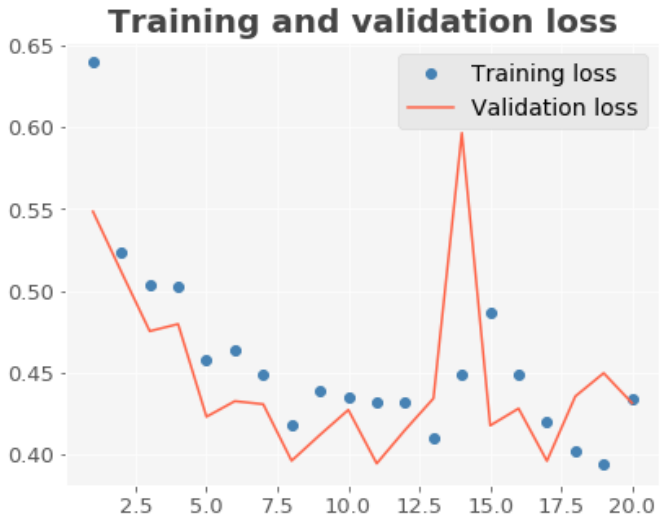
- Hyperparameter combinations were tweaked
- **The 3rd model** shows some improvement:

```
train_model(model_3, lr=0.0001, epochs=20, epoch_steps=20, valid_steps=20, batch_size=50, image_size=(50, 50))
```

```
def model_3(lr, image_size):
    model = Sequential()

    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(image_size[0], image_size[1], 3)))

    model.add(Conv2D(64, (3, 3), activation='relu'))

    model.add(Flatten())

    model.add(Dense(128, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=lr), metrics=['acc'])
    model.summary()
    return model
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 48, 48, 32)        896
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 46, 46, 64)        18496
_________________________________________________________________
flatten_3 (Flatten)          (None, 135424)            0
_________________________________________________________________
dense_5 (Dense)              (None, 128)               17334400
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 129
=================================================================
Total params: 17,353,921
Trainable params: 17,353,921
Non-trainable params: 0
```

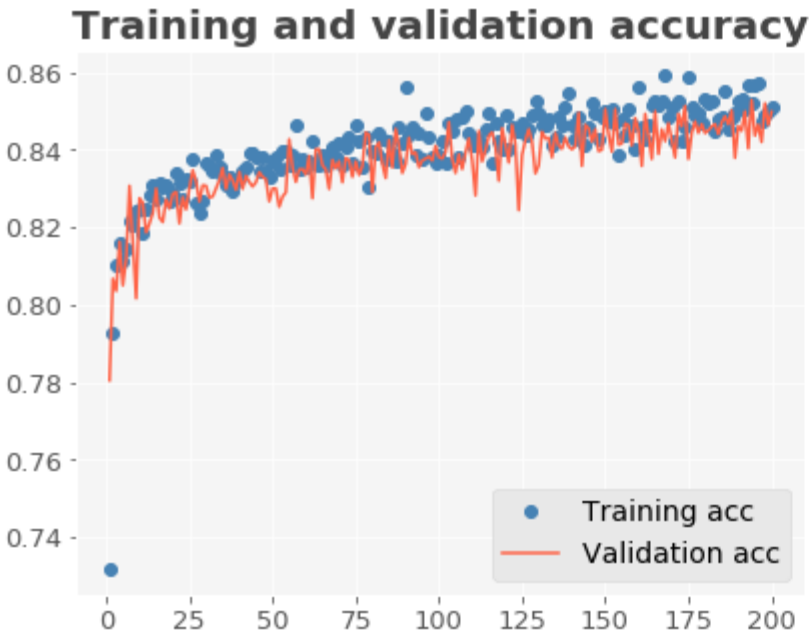


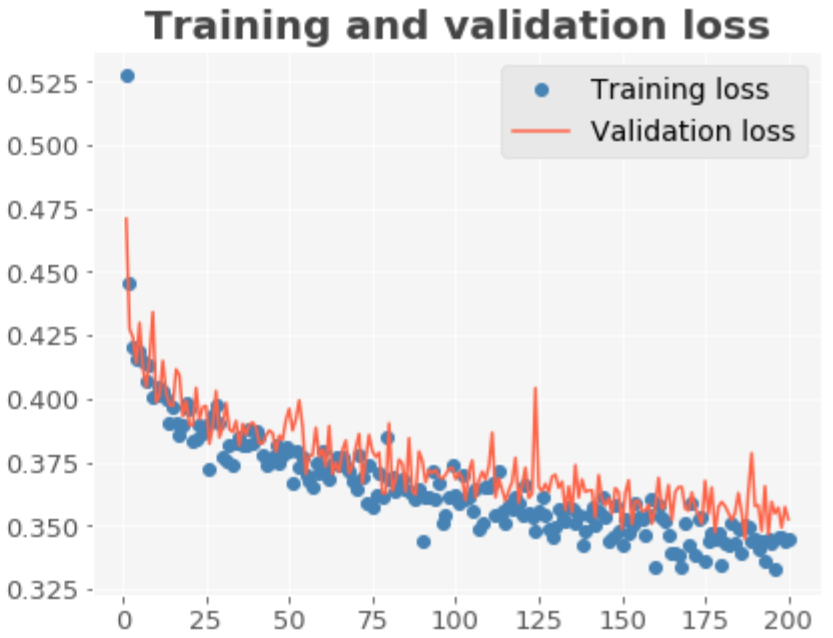



# Exploratory model architectures

**The 6th model** starts showing signs of overfitting, where the training accuracy is consistently higher than the validation accuracy. The same applies to the training loss, which is consistently lower than validation loss:

```
train_model(model_6, lr=0.00001, epochs=200, epoch_steps=200, valid_steps=200, batch_size=50, image_size=(50, 50))
```



# Exploratory model architectures

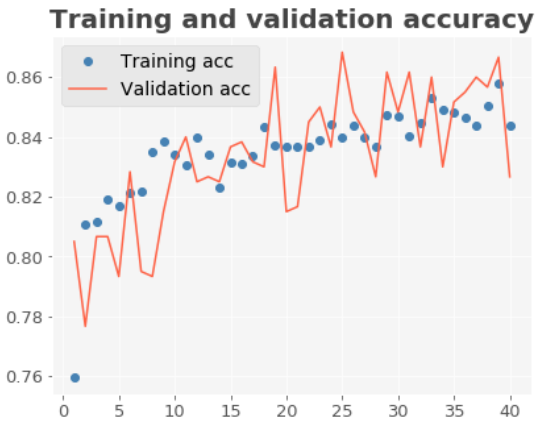
**The 7th model**, incorporates more layers and regularization (dropout) to combat overfitting by reducing the number of parameters from 17 million to about 300K.

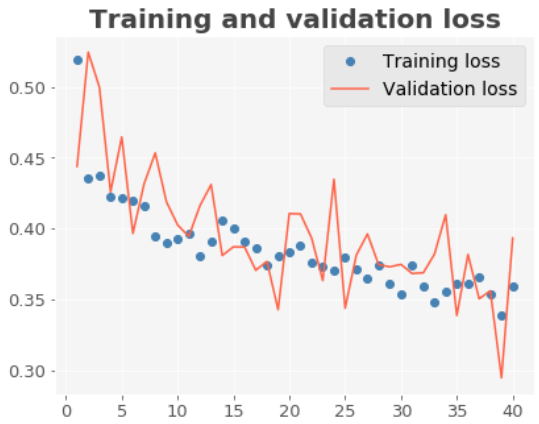
```
train_model(model_7, lr=0.001, epochs=40, epoch_steps=140, valid_steps=10, batch_size=60, image_size=(50, 50))
```

```
def model_7(lr, image_size):
    model = Sequential()

    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(image_size[0], image_size[1], 3)))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))

    model.add(Flatten())
    model.add(Dropout(0.5))

    model.add(Dense(512, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=lr), metrics=['acc'])
    model.summary()
    return model
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_13 (Conv2D) | (None, 48, 48, 32) | 896 |
| max_pooling2d_1 (MaxPooling2 | (None, 24, 24, 32) | 0 |
| conv2d_14 (Conv2D) | (None, 22, 22, 64) | 18496 |
| max_pooling2d_2 (MaxPooling2 | (None, 11, 11, 64) | 0 |
| conv2d_15 (Conv2D) | (None, 9, 9, 128) | 73856 |
| max_pooling2d_3 (MaxPooling2 | (None, 4, 4, 128) | 0 |
| conv2d_16 (Conv2D) | (None, 2, 2, 128) | 147584 |
| max_pooling2d_4 (MaxPooling2 | (None, 1, 1, 128) | 0 |
| flatten_7 (Flatten) | (None, 128) | 0 |
| dropout_1 (Dropout) | (None, 128) | 0 |
| dense_13 (Dense) | (None, 512) | 66048 |
| dense_14 (Dense) | (None, 1) | 513 |

Total params: 307,393
Trainable params: 307,393
Non-trainable params: 0



# Evolutionary algorithms

• A systematic approach to find an optimal neural network model requires exploring a large combination of architectures and hyperparameters, which is very difficult to do manually (Miikkulainen et al., 2019).

• Exploring 5 hyperparameter combinations (learning rate, number of epochs, steps per epoch, validation steps, and batch size) within known reasonable value bounds yields 100K possible models.

• Adding model architecture exploration (number of layers, nodes per layer, regularization, pooling, activation functions, etc.) yields a combinatorial explosion of over 10 billion models.

• Using 1,000 GPU’s and a conservative average time of 5 minutes to train each model would require almost a million hours, or about 100 years to explore the whole solution space using brute force with current technology!

• Evolutionary algorithms use concepts from natural selection processes, especially neuroevolution, and applies them to the process of breeding the most fit models over the course of multiple generations until the best model is found for the task at hand (Stanley et al., 2019).

# Evolutionary hyperparameter selection

- Use evolutionary algorithms instead of brute force or manual exploration

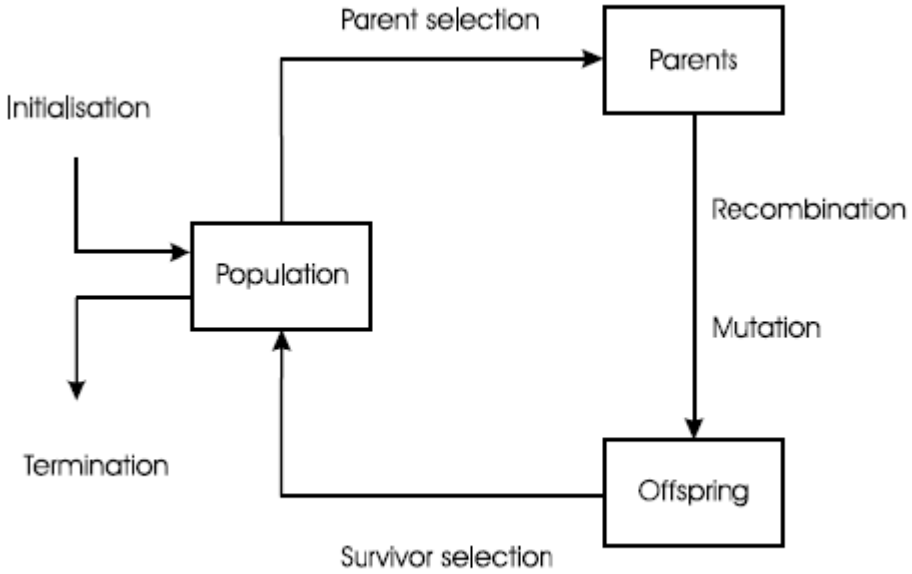



**Fig. 1.** Flowchart of evolutionary algorithm development (Eiben & Smith, 2015)

- Due to time constraints, only explore hyperparameters
- Use best model architecture found during model exploration (model 7)
- Hyperparameter values explored:

| | |
|---|---|
| Learning rate: | 5 values (0.1, 0.01, 0.001, 0.0001, 0.00001) |
| Batch size: | 10 values (20, 40, 60, 80, 100, 120, 140, 160, 180, 200) |
| Epochs: | 10 values (10, 20, 30, …, 80, 90, 100) |
| Steps per epoch: | 20 values (10, 20, 30, …, 180, 190, 200) |
| Validation steps: | 10 values (10, 20, 30, …, 80, 90, 100) |

# Evolutionary hyperparameter selection

A **first generation** of 1,000 algorithms was initialized with values selected at random

```
# Create first generation
first_generation(1000).tail()
```

| | batch_size | steps_per_epoch | validation_steps | epochs | learning_rate | train_loss | train_acc | val_loss | val_acc | time | gen | alive |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **995** | 200 | 170 | 90 | 50 | 0.001 | | | | | | 1 | True |
| **996** | 160 | 140 | 10 | 80 | 0.001 | | | | | | 1 | True |
| **997** | 160 | 170 | 100 | 80 | 0.100 | | | | | | 1 | True |
| **998** | 180 | 190 | 100 | 70 | 0.001 | | | | | | 1 | True |
| **999** | 140 | 40 | 10 | 50 | 0.001 | | | | | | 1 | True |

**Survival of the fittest:** After training, only the top 10% performing algorithms was selected

```
# Keep top performing algorithms
df_fittest = select_fittest(top_percent=10)
df_fittest.nlargest(10, 'val_acc')
```

| | batch_size | steps_per_epoch | validation_steps | epochs | learning_rate | train_loss | train_acc | val_loss | val_acc | time | gen | alive |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **653** | 100 | 190 | 10 | 90 | 0.0001 | 0.317655 | 0.865789 | 0.292571 | 0.894000 | 0:11:41 | 1 | True |
| **994** | 140 | 180 | 50 | 50 | 0.0010 | 0.320372 | 0.864048 | 0.315853 | 0.874857 | 0:10:14 | 1 | True |
| **693** | 140 | 160 | 10 | 80 | 0.0010 | 0.315224 | 0.867411 | 0.319296 | 0.874613 | 0:13:23 | 1 | True |
| **605** | 60 | 200 | 30 | 80 | 0.0010 | 0.326009 | 0.861417 | 0.316879 | 0.873889 | 0:19:31 | 1 | True |
| **760** | 200 | 190 | 40 | 100 | 0.0001 | 0.298031 | 0.874026 | 0.304660 | 0.873125 | 0:39:18 | 1 | True |
| **963** | 140 | 200 | 90 | 90 | 0.0001 | 0.312412 | 0.867250 | 0.306135 | 0.872778 | 0:34:51 | 1 | True |
| **548** | 80 | 200 | 30 | 100 | 0.0010 | 0.313608 | 0.866063 | 0.324402 | 0.872083 | 0:21:46 | 1 | True |
| **996** | 160 | 140 | 10 | 80 | 0.0010 | 0.309834 | 0.872634 | 0.339324 | 0.871875 | 0:12:31 | 1 | True |
| **276** | 100 | 170 | 50 | 80 | 0.0001 | 0.335177 | 0.857294 | 0.311656 | 0.871600 | 0:24:30 | 1 | True |
| **215** | 180 | 130 | 90 | 100 | 0.0010 | 0.304454 | 0.868462 | 0.312375 | 0.871169 | 0:37:29 | 1 | True |

# Evolutionary hyperparameter selection

• For the **second generation,** surviving algorithms were matched at random and their **chromosomes** (hyperparameters) passed to the next generation (**genetic inheritance**) with a 50% chance of inheriting a chromosome from either parent.

• Each set of parents produce 5 **offspring** at most, excluding offspring with chromosomal combinations previously observed in the population.

• A random chromosome **mutation** was assigned to one child per parent pair, promoting variety in the population and exploration of new values vs. exploitation of known best values.

```
# example of parents
df_parents
```

| | batch_size | steps_per_epoch | validation_steps | epochs | learning_rate | train_loss | train_acc | val_loss | val_acc | time | gen | alive |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **963** | 140 | 200 | 90 | 90 | 0.0001 | 0.312412 | 0.867250 | 0.306135 | 0.872778 | 0:34:51 | 1 | True |
| **923** | 80 | 110 | 50 | 90 | 0.0001 | 0.334619 | 0.858523 | 0.333084 | 0.854750 | 0:07:29 | 1 | True |

```
# example of children
df_children
```

| | batch_size | steps_per_epoch | validation_steps | epochs | learning_rate | train_loss | train_acc | val_loss | val_acc | time | gen | alive |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 80 | 200 | 90 | 90 | 0.0001 | | | | | | 2 | True |
| **1** | 140 | 200 | 50 | 90 | 0.0001 | | | | | | 2 | True |
| **2** | 80 | 110 | 50 | 90 | 0.0001 | | | | | | 2 | True |
| **3** | 80 | 110 | 50 | 90 | 0.0001 | | | | | | 2 | True |
| **4** | 140 | 200 | 90 | 90 | 0.1000 | | | | | | 2 | True |

# Evolutionary hyperparameter selection

• Best-performing algorithms from previous generations were allowed to survive if their validation accuracy fell within the top 10% best-performing algorithms in the population.

• A total of 4 generations were evolved

• A total of 17 algorithms survived the **mass extinctions**

• Total training time for all generations (1326 algorithms): almost 10 days (237.5 hours)

• Best-performing algorithms, based on validation accuracy:

| | batch_size | steps_per_epoch | validation_steps | epochs | learning_rate | train_loss | train_acc | val_loss | val_acc | time | gen | alive |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **653** | 100 | 190 | 10 | 90 | 0.0001 | 0.317655 | 0.865789 | 0.292571 | 0.894000 | 0:11:41 | 1 | True |
| **1240** | 160 | 140 | 10 | 80 | 0.0010 | 0.309508 | 0.870714 | 0.298903 | 0.886250 | 0:12:35 | 3 | True |
| **1178** | 80 | 190 | 30 | 50 | 0.0010 | 0.333464 | 0.858750 | 0.292957 | 0.884583 | 0:05:13 | 2 | True |
| **1278** | 200 | 130 | 10 | 90 | 0.0010 | 0.302942 | 0.873192 | 0.296192 | 0.882000 | 0:17:44 | 3 | True |
| **1129** | 200 | 200 | 20 | 70 | 0.0001 | 0.311461 | 0.867375 | 0.295427 | 0.881832 | 0:29:50 | 2 | True |
| **1107** | 80 | 160 | 10 | 80 | 0.0010 | 0.321133 | 0.866250 | 0.336637 | 0.881250 | 0:10:42 | 2 | True |
| **1038** | 100 | 200 | 10 | 90 | 0.0010 | 0.306632 | 0.870800 | 0.317246 | 0.881000 | 0:14:31 | 2 | True |
| **1250** | 200 | 190 | 30 | 100 | 0.0010 | 0.284835 | 0.881026 | 0.298489 | 0.878500 | 0:28:42 | 3 | True |
| **1267** | 200 | 150 | 20 | 60 | 0.0010 | 0.310335 | 0.870633 | 0.298475 | 0.878250 | 0:13:40 | 3 | True |
| **1039** | 100 | 190 | 10 | 50 | 0.0001 | 0.344512 | 0.852526 | 0.317177 | 0.878000 | 0:07:40 | 2 | True |
| **1314** | 100 | 150 | 10 | 90 | 0.0010 | 0.311339 | 0.868067 | 0.298141 | 0.878000 | 0:12:12 | 4 | True |

• Some algorithms achieved comparable results in less time (i.e., #1178 vs. #1129)

# Performance metrics

- Predictions on the test set images were run using all of the 17 surviving algorithms
- New performance metrics introduced based on predictions:
  - **Accuracy**: How many predictions were correct
  - **Precision**: How many positive predictions were true positives
  - **Sensitivity/Recall**: How many positives were correctly identified (true positive rate)
  - **Specificity**: How many negatives were true negatives
  - **F1-Score**: Weighted average of sensitivity and precision
- Accuracy may not be the best performance metric, especially when class imbalance exists
- Top performing models, based on F1-Score:

| | model_name | cm | accuracy | precision | sensitivity | specificity | f1_score |
|---|---|---|---|---|---|---|---|
| 0 | 1250 | Predicted False True __all__\nActual ... | 0.876766 | 0.790413 | 0.770148 | 0.919036 | 0.780149 |
| 1 | 1312 | Predicted False True __all__\nActual ... | 0.870712 | 0.761678 | 0.792613 | 0.901676 | 0.776838 |
| 2 | 1278 | Predicted False True __all__\nActual ... | 0.874423 | 0.791302 | 0.757457 | 0.920797 | 0.774009 |

- Top performing models, based on Sensitivity/Recall:

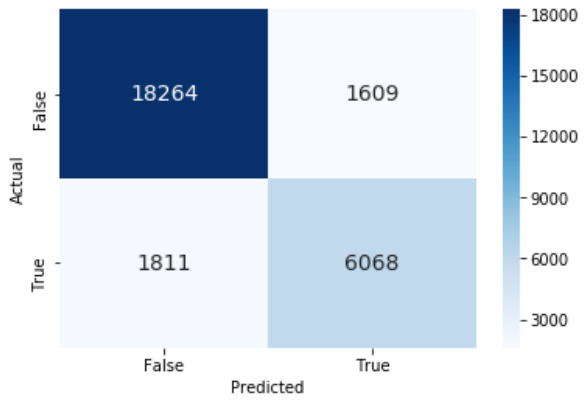
| | model_name | cm | accuracy | precision | sensitivity | specificity | f1_score |
|---|---|---|---|---|---|---|---|
| 0 | 1312 | Predicted False True __all__\nActual ... | 0.870712 | 0.761678 | 0.792613 | 0.901676 | 0.776838 |
| 1 | 1107 | Predicted False True __all__\nActual ... | 0.858605 | 0.736515 | 0.781571 | 0.889146 | 0.758374 |
| 2 | 653 | Predicted False True __all__\nActual ... | 0.864839 | 0.756908 | 0.771798 | 0.901726 | 0.764281 |

# Performance metrics

## Confusion Matrix Results

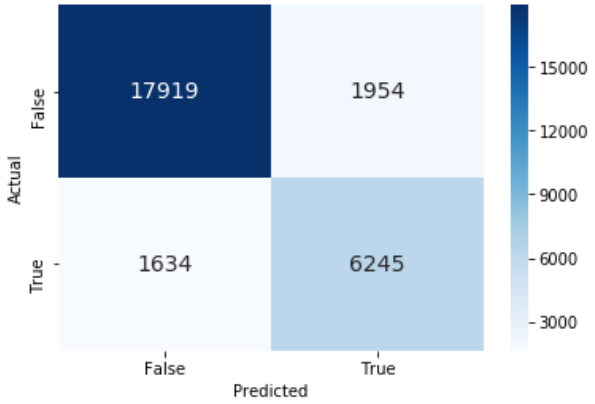
Best model based on F1-Score:

| Predicted | False | True | __all__ |
|---|---|---|---|
| Actual | | | |
| False | 18264 | 1609 | 19873 |
| True | 1811 | 6068 | 7879 |
| __all__ | 20075 | 7677 | 27752 |




Best model based on Sensitivity/Recall:

| Predicted | False | True | __all__ |
|---|---|---|---|
| Actual | | | |
| False | 17919 | 1954 | 19873 |
| True | 1634 | 6245 | 7879 |
| __all__ | 19553 | 8199 | 27752 |

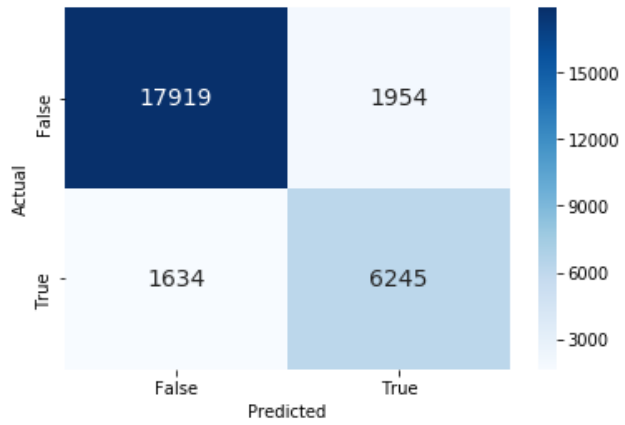


# Model selection

- Chosen model is the one based on highest sensitivity
- Maximizes true positives and minimizes false negatives
- Model based on F1-Score is a good balance, but our goal is to flag as many true positives as possible (here, false negatives are worse than false positives)

**Example of predicted images →**
(probabilities included for reference)

| | file | y_true | y_pred | y_pred_prob |
|---|---|---|---|---|
| 0 | 10253_idx5_x1001_y1501_class0.png | 0 | 0 | 0.024449 |
| 1 | 10253_idx5_x1051_y151_class0.png | 0 | 0 | 0.008627 |
| 2 | 10253_idx5_x1151_y551_class0.png | 0 | 0 | 0.316427 |
| 3 | 10253_idx5_x1151_y701_class0.png | 0 | 1 | 0.870326 |
| 4 | 10253_idx5_x1201_y1351_class0.png | 0 | 0 | 0.187531 |
| 5 | 10253_idx5_x1201_y401_class0.png | 0 | 0 | 0.041283 |
| 6 | 10253_idx5_x1201_y451_class0.png | 0 | 1 | 0.656394 |
| 7 | 10253_idx5_x1251_y1251_class0.png | 0 | 0 | 0.170695 |
| 8 | 10253_idx5_x1251_y901_class0.png | 0 | 1 | 0.623490 |
| 9 | 10253_idx5_x1301_y1101_class0.png | 0 | 1 | 0.760330 |




**← Confusion matrix of selected model**

# Prediction results

## True Negatives

17919

| | file | y_true | y_pred | y_pred_prob |
|---|---|---|---|---|
| 0 | 10253_idx5_x1001_y1501_class0.png | 0 | 0 | 0.024449 |
| 1 | 10253_idx5_x1051_y151_class0.png | 0 | 0 | 0.008627 |
| 2 | 10253_idx5_x1151_y551_class0.png | 0 | 0 | 0.316427 |
| 3 | 10253_idx5_x1201_y1351_class0.png | 0 | 0 | 0.187531 |
| 4 | 10253_idx5_x1201_y401_class0.png | 0 | 0 | 0.041283 |

## False Positives

1954

| | file | y_true | y_pred | y_pred_prob |
|---|---|---|---|---|
| 0 | 10253_idx5_x1151_y701_class0.png | 0 | 1 | 0.870326 |
| 1 | 10253_idx5_x1201_y451_class0.png | 0 | 1 | 0.656394 |
| 2 | 10253_idx5_x1251_y901_class0.png | 0 | 1 | 0.623490 |
| 3 | 10253_idx5_x1301_y1101_class0.png | 0 | 1 | 0.760330 |
| 4 | 10253_idx5_x1451_y1001_class0.png | 0 | 1 | 0.759450 |

## False Negatives

1634

| | file | y_true | y_pred | y_pred_prob |
|---|---|---|---|---|
| 0 | 10255_idx5_x151_y1051_class1.png | 1 | 0 | 0.234125 |
| 1 | 10256_idx5_x1601_y1101_class1.png | 1 | 0 | 0.217857 |
| 2 | 10256_idx5_x1851_y901_class1.png | 1 | 0 | 0.443880 |
| 3 | 10256_idx5_x1951_y1401_class1.png | 1 | 0 | 0.324580 |
| 4 | 10256_idx5_x2151_y951_class1.png | 1 | 0 | 0.479504 |

## True Positives

6245

| | file | y_true | y_pred | y_pred_prob |
|---|---|---|---|---|
| 0 | 10253_idx5_x601_y751_class1.png | 1 | 1 | 0.649673 |
| 1 | 10253_idx5_x751_y451_class1.png | 1 | 1 | 0.888613 |
| 2 | 10253_idx5_x751_y701_class1.png | 1 | 1 | 0.706819 |
| 3 | 10253_idx5_x751_y751_class1.png | 1 | 1 | 0.524811 |
| 4 | 10254_idx5_x1651_y1051_class1.png | 1 | 1 | 0.906722 |

# Prediction results

## True Negatives

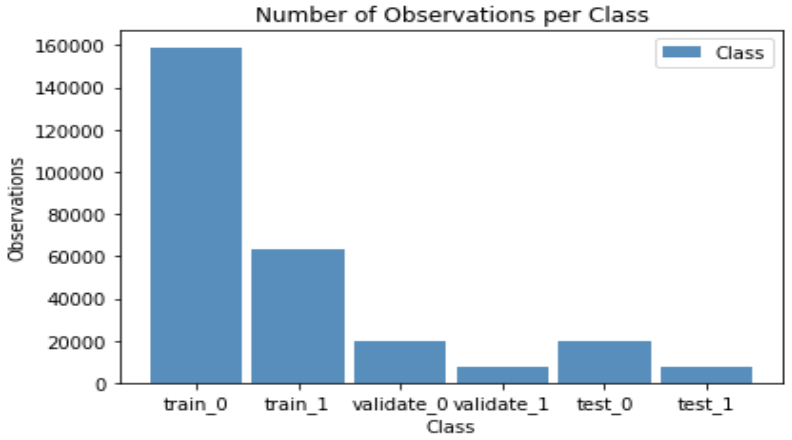
## False Negatives

## True Positives

## False Positives

# Class imbalance and data augmentation

Class imbalance observed during data analysis →




Weights can be added for class imbalance during training:

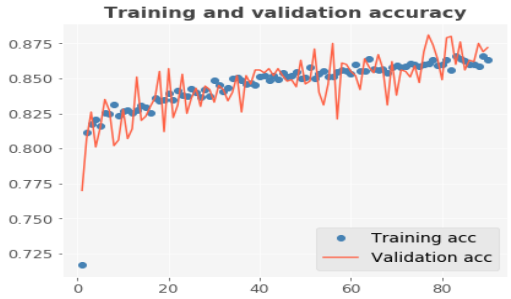
```
files_cnt = []
for category in categories:
    files = os.listdir(os.path.join(dir_train, category))
    files_cnt.append(int(len(files)))

class_weights = {}
for label in range(len(files_cnt)):
    class_weights.update({label : round(files_cnt[label]/sum(files_cnt)*100)})


# start training
t0 = datetime.now()
with tf.device('/gpu:0'):
    history = model.fit_generator(gen_train, steps_per_epoch=epoch_steps, epochs=epochs, validation_data=gen_valid,
                                  validation_steps=valid_steps, class_weight=class_weights)
```

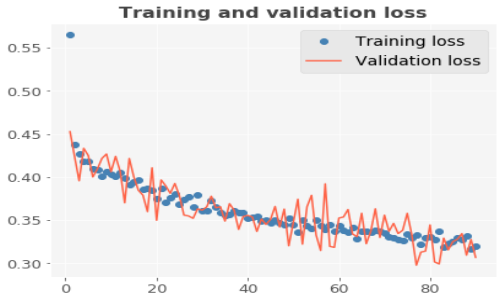
Data augmentation can be added during training on the fly →

```
datagen_train = ImageDataGenerator(rescale=1./255,
                                   horizontal_flip=True,
                                   vertical_flip=True,
                                   rotation_range=20,
                                   width_shift_range=0.02,
                                   height_shift_range=0.02,
                                   brightness_range=[0.2,1.0])
```

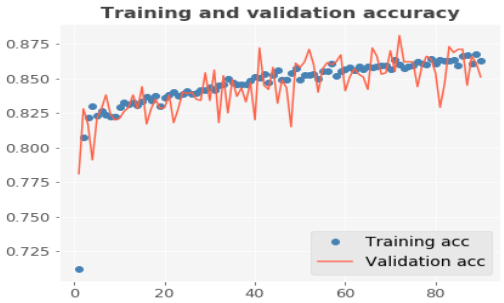
# Class imbalance and data augmentation

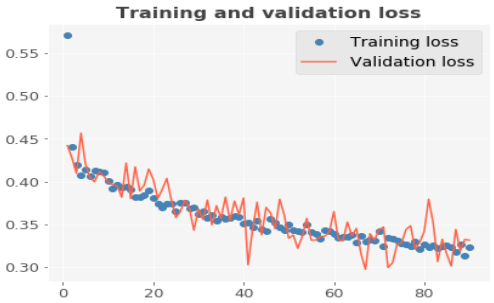
- Retrain best model with no imbalance correction or data augmentation (baseline model)



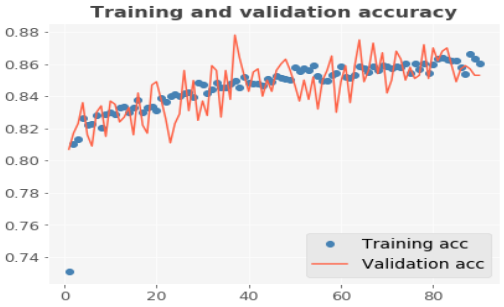



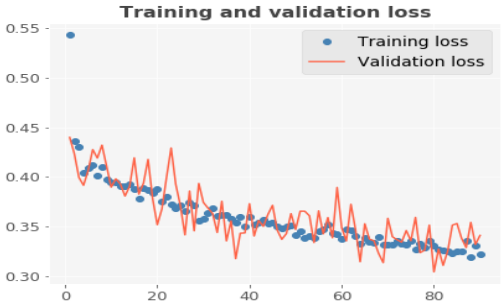

- Retrain best model with class imbalance correction only (no data augmentation)




- Retrain best model using class imbalance correction and data augmentation



# Class imbalance and data augmentation

- Correcting for class imbalance did not yield to an improved model
- Applying data augmentation also had not positive effect
- The model without correction or augmentation performed better

| | model_name | cm | accuracy | precision | sensitivity | specificity | f1_score |
|---|---|---|---|---|---|---|---|
| 0 | 2000 | Predicted False True __all__\nActual ... | 0.864586 | 0.771870 | 0.742480 | 0.912998 | 0.756890 |
| 1 | 2001 | Predicted False True __all__\nActual ... | 0.865884 | 0.779556 | 0.735626 | 0.917526 | 0.756954 |
| 2 | 2002 | Predicted False True __all__\nActual ... | 0.863649 | 0.792041 | 0.704785 | 0.926634 | 0.745870 |

# Conclusions

• Evolutionary algorithms are a quick and effective alternative to brute force and manual training for deep neural networks.

• Binary class imbalance at ~ 70% / 30% did not negatively impact model performance, likely because the small class contains 63K images.

• Misclassified images show very close resemblance to the predicted class and error could be due to deficiencies in the original data labeling (this point was also raised on paper by original researchers).

• Performance metrics indicate similar results in sensitivity/recall, specificity and balanced accuracy to the baseline model by original researchers; and an important improvement in precision and F-score.

# Future research

- Expand the use of evolutionary algorithms to include model architecture selection along with hyperparameters.

- To account for increased training time, extra GPU’s and RAM could be added, or model could be trained on scalable cloud virtual machines.

# References


Cruz-Roa, A., et al. (2014). Automatic detection of invasive ductal carciroma in whole slide images with Convolutional Neural Networks. Procedures of SPIE, 9041*(904103).* Retrieved from *https://www.researchgate.net/publication/263052166_Automatic_detection_of_invasive_ductal_carcinoma_in_whole_slide_images_with_Convolutional_Neural_Networks*

Eiben, A. E., & Smith, J. E. (2015). Introduction to Evolutionary Computing, 2nd Ed. Amsterdam, The Netherlands: Springer. Print.

Janowczyk, A. (2015). Invasive ductal carcinoma (IDC) segmentation. Retrieved from *http://www.andrewjanowczyk.com/use-case-6-invasive-ductal-carcinoma-idc-segmentation/*

# Resources

**Repository:**

https://github.com/gelidely/CNN_Breast_Cancer_Detection

**Software:**

https://www.tensorflow.org/

https://keras.io/

https://www.anaconda.com/

https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/

https://developer.nvidia.com/cuda-downloads

https://developer.nvidia.com/cudnn