Detecting Breast Cancer

Invasive Ductal Carcinoma (IDC) is the most common subtype of breast cancer. In order to predict patient outcome, histopathologists determine IDC's agressiveness grade by analyzing whole slide histopathology images. The purpose of the histopathologist is to identify regions indicating the presence of cancerous tissue. This task is time consumming and prone to human error, then it would be helpful to develop an automized classifier capable of discriminating cancerous tissue from non-cancerous tissue in an image.

This kind of model would be helpfull for hospitals and medical laboratories to increase sample speed analysis and help doctors take decisions faster. Also this kind of automized classifier will free some of the histopathologist time so he can focus on more important task or more complex samples.

The curated data is obtained directly from Kaggle.com (https://www.kaggle.com/paultimothymooney/breast-histopathology-images/data). It contains only whole slides color images divided by patient. Each whole slide image is segmented in patches of 50x50 pixels. Each of the patches corresponds either to an IDC region or a cancer free zone. Each image patch class is contained in its file name. The presence of IDC is indicated by a 1 while 0 corresponds to an IDC free area.

In [1]:

```
from keras.utils import np_utils
import numpy as np
from glob import glob
import cv2
import matplotlib.pyplot as plt
import os
%matplotlib inline
```

Using TensorFlow backend.

Checking GPU support:

In [2]:

```
import tensorflow as tf
# Checking GPU
if tf.test.gpu_device_name():
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
else:
    print("Please install GPU version of TF")
```

Default GPU Device: /device:GPU:0

The data is divided in folders were each folder corresponds to a patient.

In [3]:

```
# Path to data
path = "./Data/IDC_regular_ps50_idx5"
# Identify data folders
folders = [folder for folder in os.listdir(path)]

print("Number of patients: %d" %len(folders))
```

Number of patients: 279

Healthy tissue vs IDC tissue

As mentioned before, the class type for each 50x50 pixel image is indicated in its file name. The file name consists of three parts: a four digit number identifying the patient, a set of (x, y) coordinates corresponding to the region position in the whole slide, and finally the category class. The class can be either 0 or 1. 0 represents healthy tissue or absence of cancer while 1 corresponds to regions were IDC was found.

In [5]:

```python
# Healthy image
img_0 = cv2.imread(path + '/' + '8863' + '/' + '0' + '/' + '8863_idx5_x351_y751_class0.png')
# IDC image
img_1 = cv2.imread(path + '/' + '8863' + '/' + '1' + '/' + '8863_idx5_x1051_y1601_class1.png')
# Convert to RGB
img_0 = cv2.cvtColor(img_0, cv2.COLOR_BGR2RGB)
img_1 = cv2.cvtColor(img_1, cv2.COLOR_BGR2RGB)
# Plot images
fig = plt.figure(figsize = (16,8))
# Healthy image
ax1 = fig.add_subplot(121)
ax1.imshow(img_0)
_ = plt.xticks([])
_ = plt.yticks([])
_ = plt.title("Healthy Tissue", size = 20)
# IDC Image
ax2 = fig.add_subplot(122)
ax2.imshow(img_1)
_ = plt.xticks([])
_ = plt.yticks([])
_ = plt.title("IDC Region", size = 20)
# Images shape
print("Healthy image shape: {}".format(img_0.shape))
print("IDC image shape: {}".format(img_1.shape))
```
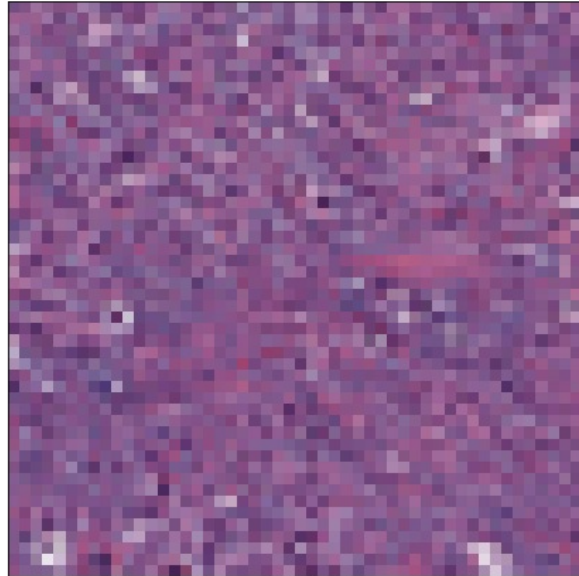
Healthy image shape: (50, 50, 3)
IDC image shape: (50, 50, 3)
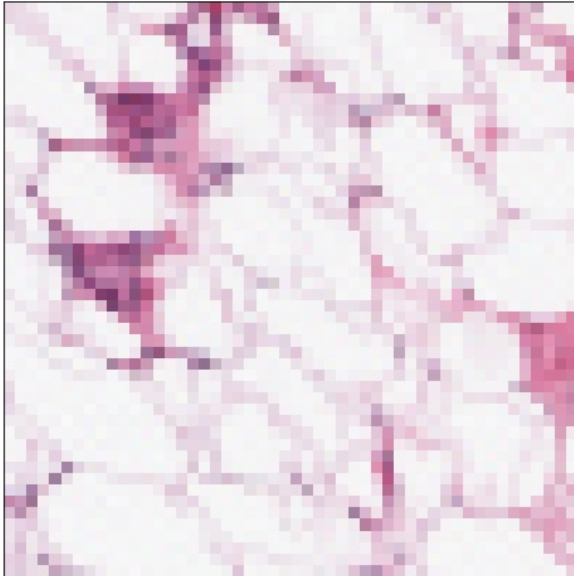
IDC regions appear to be denser and have darkers tones of violet compared to healthy tissue. This information can be used to define features in order to train a machine learning model. However a literature review (http://spie.org/Publications/Proceedings/Paper/10.1117/12.2043872) on this data set tells us that this not the best approach. A more reliable model uses CNN.

Loading images

Images are 50x50x3 where the third dimension corresponds to RGB color coding. This information is important to take it into account when loading the images and defining the CNN architecture.

In [6]:

```python
def path_to_tensor(img_path):
    # load images
    img = cv2.imread(img_path)
    # Convert to RGB
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    if img.shape == (50, 50, 3):
        # convert 3D tensor to 4D tensor with shape (1, 50, 50, 3) and return 4D tensor
        #return np.expand_dims(img, axis=0)
        return img.astype(np.float32) / 255
        #return img.reshape([50,50,3])
```

Each patient's folder contains two folders: a folder with non cancer images and a folder with cancerous tissue images.

In [7]:

```python
# Healthy tissue is in folder 0 and cancerous tissue in folder 1
folder_label = ['0', '1']
images = [] # stores images
labels = [] # stores image leabels

for folder in folders:
    # Read from folder 0 and create corresponding label array
    path_0 = path + '/' + folder + '/' + folder_label[0]
    images_0 = [path_to_tensor(path_0 + '/' + img_path) for img_path in os.listdir(path_0) \
            if path_to_tensor(path_0 + '/' + img_path) is not None]
    labels_0 = np.zeros(len(images_0), dtype=np.int8)
    images.extend(images_0)
    labels.extend(labels_0)
    # Read from folder 1 and create corresponding label array
    path_1 = path + '/' + folder + '/' + folder_label[1]
    images_1 = [path_to_tensor(path_1 + '/' + img_path) for img_path in os.listdir(path_1) \
            if path_to_tensor(path_1 + '/' + img_path) is not None]
    labels_1 = np.ones(len(images_1), dtype=np.int8)
    images.extend(images_1)
    labels.extend(labels_1)
```

In [8]:

```python
images = np.array(images)
labels = np.array(labels)

print("Number of images: {}".format(len(images)))
print("Number of labels: {}".format(len(labels)))

fig = plt.figure(figsize = (5,5))
_ = plt.imshow(images[0])
_ = plt.xticks([])
_ = plt.yticks([])
_ = plt.title("Class {} tissue".format(labels[0]), size = 20)
```
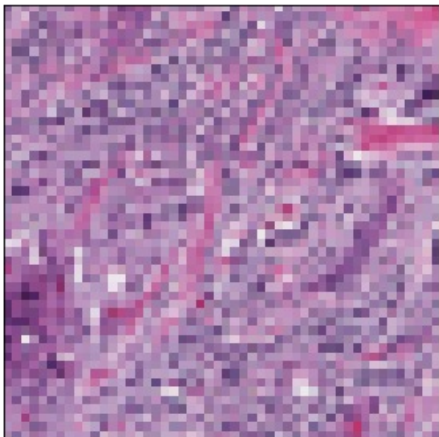
```
Number of images: 275222
Number of labels: 275222
```


Class 0 tissue

```
print("Number of 1 labels: {}".format(np.sum(labels)))
print("Number of 0 labels: {}".format(len(labels) - np.sum(labels)))
```

```
Number of 1 labels: 78768
Number of 0 labels: 196454
```

In this data set the classes are unbalanced with a ration of 2.5 to 1 in favor of label 0. So, in addition to the accuracy metric to assess the final model, it will be also important to check the F_1 score.

Metrics definition

To assess the fitness of the developed model, it is important to know how many labels in the validation set are predicted correctly. So, the metric of choice to evaluate the model is the accuracy. The accuracy calculates the number of correct predicted labels to the total number of labels. To explain this with an equation let's define first the following terms:

TP: true positives, it represents the images correctly identified as 1, meaning an IDC region.

FP: false postives, it denotes the images of healthy tissue incorrectly indentified as 1.

TN: true negatives, it refers to healthy tissue images correctly identified as 0.

FN: false negatives, it defines IDC region images identified as 0 or being healthy.

With theses definitions at hand, the accuracy is defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

In addition to the accuracy, given that the classes are unbalanced, a metric of interest is the F_1 score:

$$F_1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} = 2\frac{Precision.Recall}{Precision + Recall}$$

Where *Precision = TP/(TP+FP)* and *Recall=TP/(TP+FN)*.

Having determined the evaluation metrics, now it is important to define a benchmark model to assess the developed models. Given that the classes are unbalanced and that the negative classes dominates the positive classes, a model predicting that all the images correspond to a label of 0 is a good starting model.

Bechmark model

Images of category 0 are more common than the images with class 1, the ratio being approximately 2.5 to 1. Given that the classes are unbalanced at the advantage of label 0, then we can define a naive bencmark model were all the images are labeled as 0. Doing so results in the following values.

In [52]:

```
TP = 0 # true positive
FP = 0 # false postive
TN = len(labels) - np.sum(labels) # true negative
FN = np.sum(labels) # false negative

print("True postives: {}".format(TP))
print("False postives: {}".format(FP))
print("True negatives: {}".format(TN))
print("False negatives: {}\n".format(FN))

accuracy = (TP + TN) / (TP + TN + FP + FN)
print("Accuracy: {:.2f}%\n".format(accuracy*100))
```

```
True postives: 0
False postives: 0
True negatives: 196454
False negatives: 78768

Accuracy: 71.38%
```

In [73]:

```python
precision_neg = TN / (TN + FN)
print("Scores for healthy tiddue images (negatives labels):")
print("\t\t Precision: {:.2f}%".format(precision*100))
recall_neg = TN / (TN + FP)
print("\t\t Recall: {:.2f}%".format(recall*100))
f1 = 2 / (1 / precision_neg + 1 / recall_neg)
print("\t\t F1 score: {:.2f}%".format(f1*100))
```

```
Scores for healthy tiddue images (negatives labels):
                Precision: 71.38%
                Recall: 100.00%
                F1 score: 83.30%
```

The benchmark model has then an accuracy of 71.38% and and F_1 score of 83.30% for the negaitve labels. An F_1 score for the positve labels is undefined given that no positve images are detected under this model.

Having defined a benchmark model now we can proceed to develop a more reliable model.

CNN Architectures

Defining train and test sets

In order to train the models and determine how they behave, the images need to be divided in two: a training set and a validation set. In this case 90% of the images are held for training while 10% of them are assigned for testing.

Given that the classes are unbalanced, the 'stratify' parameter of the train_test_split (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) method from scikit-learn is used.

In [10]:

```python
from sklearn.model_selection import train_test_split

# Set seed for random calculations
seed = 42
np.random.seed(seed)
# SPlit between train and test set
idx_train, idx_test = train_test_split(np.arange(labels.shape[0]), test_size=0.1, \
                                        random_state=seed, shuffle=True, stratify=labels)
```

Using an existing architecture: transfer learning

The first model and more direct approach to try consists in applying transfer learning. Transfer learning uses a pre-trained model to be applied in a new data set by introducing minor modifications. This method allows saving training time in otherwise highly time consuming models.

The CNN architecture VGG19 (https://arxiv.org/abs/1409.1556) is chosen given that it accepts images of 50x50 pixels and it has been shown to give good results on the ImageNet (http://image-net.org/) data set.

Our data set doesn't corresponds to images from ImageNet, however the pre-trained weights already now to recognize complex patterns in images, so its use may be useful to train our model.

ImageNet is a data set with 1000 categories, so in order to be able to use the pre-trained network, the last full connected layers are ignored and only the convolutional layers are imported.

In [17]:

```python
from keras.applications.vgg19 import VGG19
```

In [18]:

```
model = VGG19(include_top=False, input_shape=(50,50,3))
model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 50, 50, 3)	0
block1_conv1 (Conv2D)	(None, 50, 50, 64)	1792
block1_conv2 (Conv2D)	(None, 50, 50, 64)	36928
block1_pool (MaxPooling2D)	(None, 25, 25, 64)	0
block2_conv1 (Conv2D)	(None, 25, 25, 128)	73856
block2_conv2 (Conv2D)	(None, 25, 25, 128)	147584
block2_pool (MaxPooling2D)	(None, 12, 12, 128)	0
block3_conv1 (Conv2D)	(None, 12, 12, 256)	295168
block3_conv2 (Conv2D)	(None, 12, 12, 256)	590080
block3_conv3 (Conv2D)	(None, 12, 12, 256)	590080
block3_conv4 (Conv2D)	(None, 12, 12, 256)	590080
block3_pool (MaxPooling2D)	(None, 6, 6, 256)	0
block4_conv1 (Conv2D)	(None, 6, 6, 512)	1180160
block4_conv2 (Conv2D)	(None, 6, 6, 512)	2359808
block4_conv3 (Conv2D)	(None, 6, 6, 512)	2359808
block4_conv4 (Conv2D)	(None, 6, 6, 512)	2359808
block4_pool (MaxPooling2D)	(None, 3, 3, 512)	0
block5_conv1 (Conv2D)	(None, 3, 3, 512)	2359808
block5_conv2 (Conv2D)	(None, 3, 3, 512)	2359808
block5_conv3 (Conv2D)	(None, 3, 3, 512)	2359808
block5_conv4 (Conv2D)	(None, 3, 3, 512)	2359808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0

```
Total params: 20,024,384
Trainable params: 20,024,384
Non-trainable params: 0
```

Bottlenck Features

Now that the weights corresponding to VGG19 convolutional layers have been imported, the next step is to calculate the bottleneck features. The term bottleneck features refers to the output of the last convolutional layer when the train and test images are fed to the CNN network.

In [19]:

```
# Bottleneck features for training set
bottleneck_features_train = model.predict(images[idx_train])
np.save('./bottleneck_features/bottleneck_features_train_VGG19.npy', \
        bottleneck_features_train)
# Bottleneck features for test set
bottleneck_features_test = model.predict(images[idx_test])
np.save('./bottleneck_features/bottleneck_features_test_VGG19.npy', \
        bottleneck_features_test)
```

In [20]:

```
# Check shapes
bottleneck_features_train = np.load('./bottleneck_features/bottleneck_features_train_VGG19.npy')
print("Bottleneck train features shape: {}".format(bottleneck_features_train.shape))

bottleneck_features_test = np.load('./bottleneck_features/bottleneck_features_test_VGG19.npy')
print("Bottleneck train features shape: {}".format(bottleneck_features_test.shape))
```

```
Bottleneck train features shape: (247699, 1, 1, 512)
Bottleneck train features shape: (27523, 1, 1, 512)
```

Now that the bottleneck features have been calculated, it is possible to define a model architecture corresponding to the full connected layers that were dropped from VGG19.

The final output consists of one category which is the probability for an image to belong to class 1. The activation function for the last layer is the 'sigmoid' because its range is between 0 and 1.

In [143]:

```
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()
model.add(Flatten(input_shape= (1,1,512)))
model.add(Dropout(0.2))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
# Ouput layer
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

model.summary()
```

Layer (type)	Output Shape	Param #
flatten_9 (Flatten)	(None, 512)	0
dropout_6 (Dropout)	(None, 512)	0
dense_25 (Dense)	(None, 256)	131328
dropout_7 (Dropout)	(None, 256)	0
dense_26 (Dense)	(None, 128)	32896
dense_27 (Dense)	(None, 1)	129

```
Total params: 164,353
Trainable params: 164,353
Non-trainable params: 0
```

To train the model, advantage is taken from Keras callback methods. ModelCheckpoint (https://keras.io/callbacks/#modelcheckpoint) allows to save the weights corresponding to the best epoch. EarlyStopping (https://keras.io/callbacks/#earlystopping) stops the training if the model loss doesn't improve after a determined number of epochs. These methods avoid training the model more than necessary and save the weights from the best loss found.

Given that the classes are not balanced, the parameter 'class_weight' from de fit (https://keras.io/models/model/#fit) method is set to $0:1, 1:2.5$ in order to account for the difference ratio between 0 and 1 labels.

In [144]:

```python
from keras.callbacks import ModelCheckpoint, EarlyStopping

epochs = 100
batch_size = 256

stopping_condition = EarlyStopping(patience=6)
checkpointer = ModelCheckpoint(filepath='./saved_models/weights.best.CNN_arch_VGG19.hdf5', \
                               verbose=1, save_best_only=True)

history = model.fit(bottleneck_features_train, labels[idx_train], \
        validation_split=0.2, \
        epochs=epochs, batch_size=batch_size, class_weight={0:1, 1:2.5},\
        callbacks=[checkpointer, stopping_condition], verbose=1)
```

```
Train on 198159 samples, validate on 49540 samples
Epoch 1/100
198159/198159 [==============================] - 2s 12us/step - loss: 0.7613 - acc: 0.7479 - va
l_loss: 0.6919 - val_acc: 0.7836

Epoch 00001: val_loss improved from inf to 0.69188, saving model to ./saved_models/weights.best
.CNN_arch_VGG19.hdf5
Epoch 2/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.7264 - acc: 0.7678 - va
l_loss: 0.6859 - val_acc: 0.7739

Epoch 00002: val_loss improved from 0.69188 to 0.68588, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 3/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.7161 - acc: 0.7712 - val
_loss: 0.6772 - val_acc: 0.7960

Epoch 00003: val_loss improved from 0.68588 to 0.67719, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 4/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.7085 - acc: 0.7746 - va
l_loss: 0.6689 - val_acc: 0.7895

Epoch 00004: val_loss improved from 0.67719 to 0.66887, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 5/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.7046 - acc: 0.7760 - va
l_loss: 0.6704 - val_acc: 0.8070

Epoch 00005: val_loss did not improve from 0.66887
Epoch 6/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6990 - acc: 0.7783 - val
_loss: 0.6676 - val_acc: 0.7953

Epoch 00006: val_loss improved from 0.66887 to 0.66765, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 7/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6971 - acc: 0.7792 - va
l_loss: 0.6650 - val_acc: 0.7965

Epoch 00007: val_loss improved from 0.66765 to 0.66500, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 8/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6939 - acc: 0.7790 - va
l_loss: 0.6615 - val_acc: 0.8111

Epoch 00008: val_loss improved from 0.66500 to 0.66153, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 9/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6907 - acc: 0.7812 - va
l_loss: 0.6632 - val_acc: 0.8056

Epoch 00009: val_loss did not improve from 0.66153
Epoch 10/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6915 - acc: 0.7811 - val
_loss: 0.6585 - val_acc: 0.7942

Epoch 00010: val_loss improved from 0.66153 to 0.65847, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 11/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6865 - acc: 0.7827 - va
l_loss: 0.6547 - val_acc: 0.7938

Epoch 00011: val_loss improved from 0.65847 to 0.65467, saving model to ./saved_models/weights.
```

```
best.CNN_arch_VGG19.hdf5
Epoch 12/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6864 - acc: 0.7821 - val
_loss: 0.6568 - val_acc: 0.8032

Epoch 00012: val_loss did not improve from 0.65467
Epoch 13/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6855 - acc: 0.7820 - va
l_loss: 0.6596 - val_acc: 0.8089

Epoch 00013: val_loss did not improve from 0.65467
Epoch 14/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6803 - acc: 0.7840 - va
l_loss: 0.6580 - val_acc: 0.7914

Epoch 00014: val_loss did not improve from 0.65467
Epoch 15/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6793 - acc: 0.7833 - val
_loss: 0.6497 - val_acc: 0.8014

Epoch 00015: val_loss improved from 0.65467 to 0.64970, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 16/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6781 - acc: 0.7838 - val
_loss: 0.6575 - val_acc: 0.7793

Epoch 00016: val_loss did not improve from 0.64970
Epoch 17/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6769 - acc: 0.7846 - va
l_loss: 0.6504 - val_acc: 0.8058

Epoch 00017: val_loss did not improve from 0.64970
Epoch 18/100
198159/198159 [==============================] - 2s 11us/step - loss: 0.6754 - acc: 0.7846 - va
l_loss: 0.6514 - val_acc: 0.7908

Epoch 00018: val_loss did not improve from 0.64970
Epoch 19/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6750 - acc: 0.7861 - va
l_loss: 0.6515 - val_acc: 0.8079

Epoch 00019: val_loss did not improve from 0.64970
Epoch 20/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6719 - acc: 0.7869 - va
l_loss: 0.6540 - val_acc: 0.8131

Epoch 00020: val_loss did not improve from 0.64970
Epoch 21/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6727 - acc: 0.7849 - val
_loss: 0.6485 - val_acc: 0.8109

Epoch 00021: val_loss improved from 0.64970 to 0.64846, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 22/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6689 - acc: 0.7884 - val
_loss: 0.6514 - val_acc: 0.7979

Epoch 00022: val_loss did not improve from 0.64846
Epoch 23/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6691 - acc: 0.7863 - va
l_loss: 0.6486 - val_acc: 0.7980

Epoch 00023: val_loss did not improve from 0.64846
Epoch 24/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6673 - acc: 0.7878 - va
l_loss: 0.6507 - val_acc: 0.8031

Epoch 00024: val_loss did not improve from 0.64846
Epoch 25/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6661 - acc: 0.7883 - val
_loss: 0.6510 - val_acc: 0.8033

Epoch 00025: val_loss did not improve from 0.64846
Epoch 26/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6655 - acc: 0.7879 - va
l_loss: 0.6462 - val_acc: 0.7997

Epoch 00026: val_loss improved from 0.64846 to 0.64622, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 27/100
```

```
198159/198159 [==============================] - 2s 10us/step - loss: 0.6649 - acc: 0.7875 - va
l_loss: 0.6604 - val_acc: 0.8167

Epoch 00027: val_loss did not improve from 0.64622
Epoch 28/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6646 - acc: 0.7892 - val
_loss: 0.6438 - val_acc: 0.8125

Epoch 00028: val_loss improved from 0.64622 to 0.64375, saving model to ./saved_models/weights.
best.CNN_arch_VGG19.hdf5
Epoch 29/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6616 - acc: 0.7882 - val
_loss: 0.6554 - val_acc: 0.7972

Epoch 00029: val_loss did not improve from 0.64375
Epoch 30/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6626 - acc: 0.7868 - va
l_loss: 0.6482 - val_acc: 0.8037

Epoch 00030: val_loss did not improve from 0.64375
Epoch 31/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6614 - acc: 0.7874 - va
l_loss: 0.6474 - val_acc: 0.7991

Epoch 00031: val_loss did not improve from 0.64375
Epoch 32/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6596 - acc: 0.7893 - va
l_loss: 0.6480 - val_acc: 0.8043

Epoch 00032: val_loss did not improve from 0.64375
Epoch 33/100
198159/198159 [==============================] - 2s 10us/step - loss: 0.6597 - acc: 0.7891 - va
l_loss: 0.6531 - val_acc: 0.7985

Epoch 00033: val_loss did not improve from 0.64375
Epoch 34/100
198159/198159 [==============================] - 2s 9us/step - loss: 0.6581 - acc: 0.7884 - val
_loss: 0.6495 - val_acc: 0.8086

Epoch 00034: val_loss did not improve from 0.64375
```

In [139]:

```python
# Visualize the training and validation loss
plt.figure(figsize=(20, 10))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.yticks(size=15)
plt.ylabel('Loss', size=20)
plt.xticks(size=15)
plt.xlabel('Epoch', size=20)
plt.legend(['train', 'validation'], loc='upper right', fontsize=20)
plt.title('Model Loss, no regularization', size=25)
plt.show()
# test accuracy 80.4164%
```
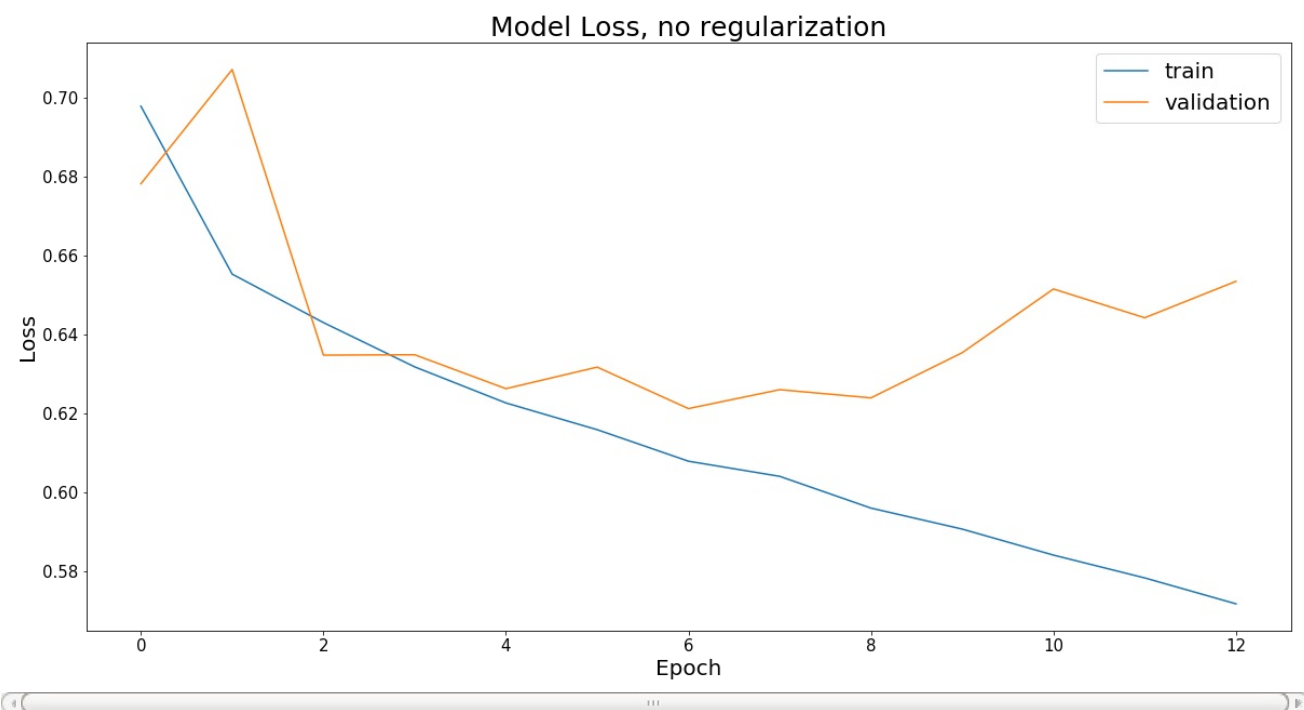

The plot above shows the train and validation losses behavior as the model trains. In this particular case regularization was not applied. In other words, Dropout (https://keras.io/layers/core/#dropout) layers were not applied. With no regularization, the curve shows overfitting: the loss improves on the training set but diverges on the validation set as the number of epochs increases. The accuracy on the test set is about 80.4164%.

The overfitting indication tells us that we need to apply regularization to the model. Below the model training beahvior is shown).

In [145]:

```python
# Visualize the training and validation loss
plt.figure(figsize=(20, 10))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.yticks(size=15)
plt.ylabel('Loss', size=20)
plt.xticks(size=15)
plt.xlabel('Epoch', size=20)
plt.legend(['train', 'validation'], loc='upper right', fontsize=20)
plt.title('Model Loss, with regularization', size=25)
plt.show()
```
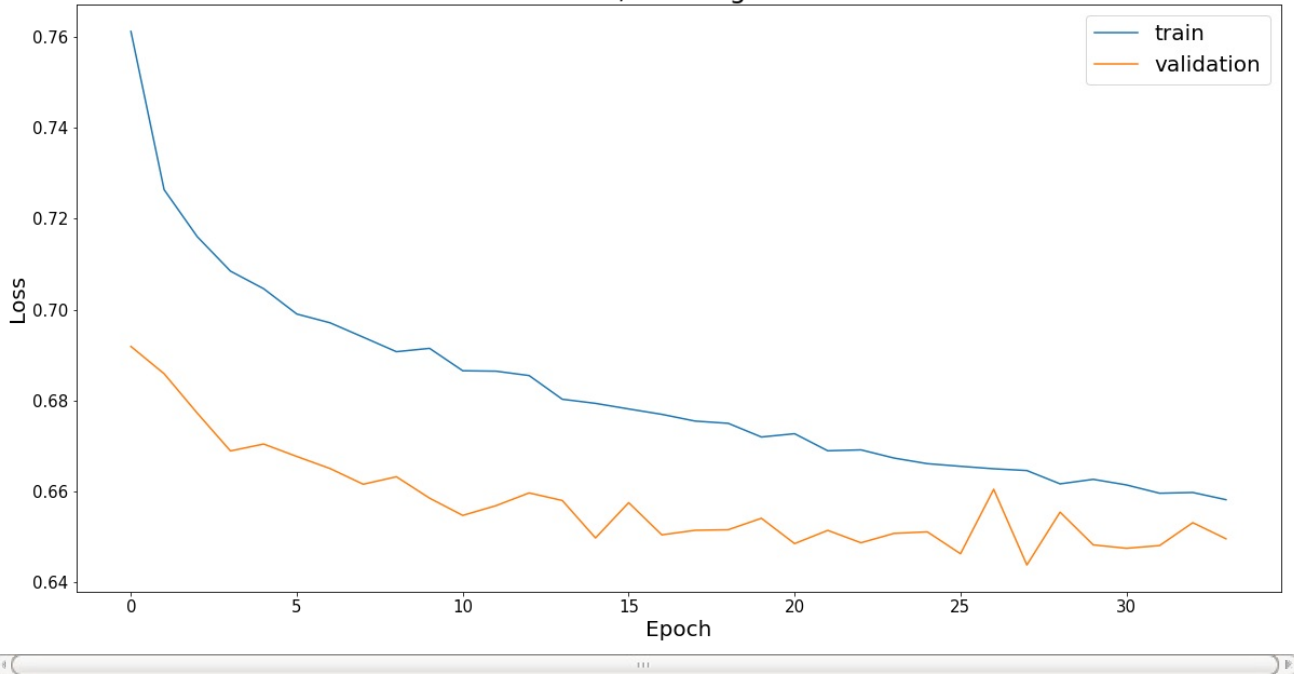

With regularization there no overfitting. However, the train loss decreases constantly while the validation loss doesn't show improvement after epoch 15.

Transfer learning evaluation metrics values

In [21]:

```python
from keras.models import load_model

model = load_model('./saved_models/weights.best.CNN_arch_VGG19.hdf5')
```

In [22]:

```python
# get predictions for each image in test set
predictions = [np.int(np.round(model.predict(np.expand_dims(feature, axis=0)))) \
               for feature in bottleneck_features_test]
```

In [23]:

```python
# report test accuracy
test_accuracy = np.sum(np.array(predictions)==labels[idx_test])/len(predictions)
print('Test accuracy: %.4f%%' %(100*test_accuracy))
```

Test accuracy: 80.6271%

```
In [25]:
```

```python
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import confusion_matrix, roc_curve, auc

y_test = labels[idx_test]
y_pred = np.array(predictions)

confusion = confusion_matrix(y_test, y_pred)

precision, recall, f1_score, support = precision_recall_fscore_support(y_test, y_pred, beta=1)

y_pred_prob = [np.float32(model.predict(np.expand_dims(feature, axis=0))) \
                 for feature in bottleneck_features_test]
y_pred_prob = np.array(y_pred_prob).reshape(-1)

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

auc_roc = auc(fpr, tpr)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr, label='CNN')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('CNN with Image Augmentation ROC Curve')
plt.show()

print("Area under ROC: {:.4f}".format(auc_roc))
print("Precision for 0 and 1: {:.2f}% and {:.2f}%".format(precision[0]*100, precision[1]*100))
print("Recall for 0 and 1: {:.2f}% and {:.2f}%".format(recall[0]*100, recall[1]*100))
print("F1 score for 0 and 1: {:.2f}% and {:.2f}%".format(f1_score[0]*100, f1_score[1]*100))
print("Confusion matrix:\n {}".format(confusion))
```
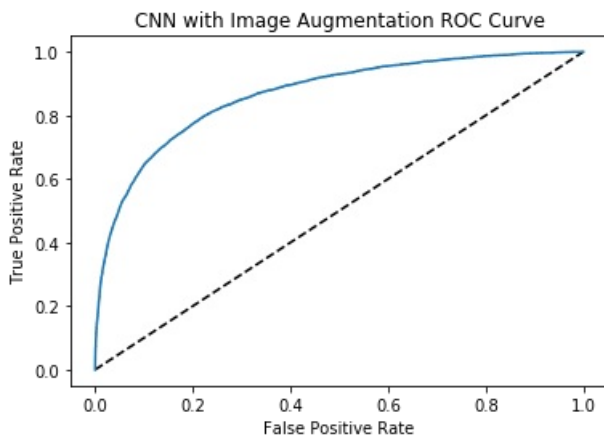


```
Area under ROC: 0.8664
Precision for 0 and 1: 88.69% and 64.10%
Recall for 0 and 1: 83.51% and 73.43%
F1 score for 0 and 1: 86.02% and 68.45%
Confusion matrix:
 [[16407  3239]
 [ 2093  5784]]
```

Using transfer learning, the best obtained accuracy is 80.6271% and the best F_1 on the 0 labels is 86.02%. For the the positive labels F_1 is 68.45%.

This is better than the benchmark model. If all the weights from the VGG19 are trained from scratch, it is expected to get better results. But that task is too heavy for a home laptop. So, instead it can be worth to try to do better by defining our own CNN architecture.

Custom made CNN architecture

The architecture is composed of 3 convolutional layers, a flattened layer, two dense layers and the output layer. The first 4 layers have first a convolutional layer with a kernel of 3x3 pixels. 3x3 was chosen given that this was used in the VGG architecture with good results. The 3x3 pooling window at each of the first 3 layers is also inspired from VGG. VGG uses 2x2 but 3x3 runs faster, so it slightly increases the training speed. Laos I consider that it should be a good practice to have a pooling window of the same size as the convolutional window. The activation function chosen for the convolutional layers is 'relu' and the dropout sub-layer for each convolutional layer is inspired from AlexNet (http://vision.stanford.edu/teaching/cs231b_spring1415/slides/alexnet_tugce_kyunghee.pdf). The purpose of the dropout is to avoid overfitting and to introduce regularization.

The input layer shape is based on the input images sizes (50x50x3). From layers 1 to 3, the number of filters progresses as follows: 32, 64, 128. When dealing with algorithms I feel comfortable moving by factors of 2. For all the 3 CNN layers the padding was set as 'same' to keep the filter sizes as the size of the inputs to each layer.

For the third layer, its output is flatten before going to the denses layer.s So, a flattening layer was used for this purpose.

The output layer, which is a dense layer, has a size of 1 corresponding to the probability of getting a label of 1. The the 'sigmoid' activation function was chosen because it is output range is $[0, 1]$.

In [153]:

```python
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### Architecture
# Layer 1 : input layer
model.add(Conv2D(filters=32, kernel_size=3, padding='same', \
                activation='relu', input_shape=(50,50,3)))
model.add(MaxPooling2D(pool_size=3))
model.add(Dropout(0.2))
# Layer 2
model.add(Conv2D(filters=64, kernel_size=3, padding='same', \
                activation='relu'))
model.add(MaxPooling2D(pool_size=3))
model.add(Dropout(0.2))
# Layer 3
model.add(Conv2D(filters=128, kernel_size=3, padding='same', \
                activation='relu'))
model.add(MaxPooling2D(pool_size=3))
model.add(Dropout(0.2))

#model.add(GlobalAveragePooling2D('channels_last'))

#Flat layers
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
# Output layer
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 50, 50, 32)        896
_____
max_pooling2d_5 (MaxPooling2 (None, 16, 16, 32)        0
_____
dropout_11 (Dropout)         (None, 16, 16, 32)        0
_____
conv2d_6 (Conv2D)            (None, 16, 16, 64)        18496
_____
max_pooling2d_6 (MaxPooling2 (None, 5, 5, 64)          0
_____
dropout_12 (Dropout)         (None, 5, 5, 64)          0
_____
conv2d_7 (Conv2D)            (None, 5, 5, 128)         73856
_____
max_pooling2d_7 (MaxPooling2 (None, 1, 1, 128)         0
_____
dropout_13 (Dropout)         (None, 1, 1, 128)         0
_____
flatten_10 (Flatten)         (None, 128)               0
_____
dense_28 (Dense)             (None, 64)                8256
_____
dense_29 (Dense)             (None, 32)                2080
_____
dense_30 (Dense)             (None, 1)                 33
=================================================================
Total params: 103,617
Trainable params: 103,617
Non-trainable params: 0
_____
```

In [154]:

```python
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

In [ ]:

Train the model:

In [155]:

```python
from keras.callbacks import ModelCheckpoint, EarlyStopping

epochs = 100
batch_size = 512

stopping_condition = EarlyStopping(patience=6)
checkpointer = ModelCheckpoint(filepath='./saved_models/weights.best.CNN_arch.hdf5',
                               verbose=1, save_best_only=True)

history = model.fit(images[idx_train], labels[idx_train], \
        validation_split=0.1, \
        epochs=epochs, batch_size=batch_size, class_weight={0:1, 1:2.5},\
        callbacks=[checkpointer, stopping_condition], verbose=1)
```

```
Train on 222929 samples, validate on 24770 samples
Epoch 1/100
222929/222929 [==============================] - 15s 65us/step - loss: 0.7051 - acc: 0.7785 - v
al_loss: 0.7714 - val_acc: 0.6437

Epoch 00001: val_loss improved from inf to 0.77135, saving model to ./saved_models/weights.best
.CNN_arch.hdf5
Epoch 2/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.6330 - acc: 0.8072 - v
al_loss: 0.6956 - val_acc: 0.7062

Epoch 00002: val_loss improved from 0.77135 to 0.69556, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 3/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.6082 - acc: 0.8152 - v
al_loss: 0.6900 - val_acc: 0.7011

Epoch 00003: val_loss improved from 0.69556 to 0.68996, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 4/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5835 - acc: 0.8234 - v
```

```
al_loss: 0.7422 - val_acc: 0.6383

Epoch 00004: val_loss did not improve from 0.68996
Epoch 5/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5718 - acc: 0.8273 - v
al_loss: 0.6882 - val_acc: 0.6991

Epoch 00005: val_loss improved from 0.68996 to 0.68816, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 6/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5618 - acc: 0.8306 - v
al_loss: 0.6680 - val_acc: 0.7010

Epoch 00006: val_loss improved from 0.68816 to 0.66799, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 7/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5545 - acc: 0.8332 - v
al_loss: 0.6594 - val_acc: 0.7156

Epoch 00007: val_loss improved from 0.66799 to 0.65944, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 8/100
222929/222929 [==============================] - 14s 63us/step - loss: 0.5470 - acc: 0.8350 - v
al_loss: 0.5672 - val_acc: 0.7997

Epoch 00008: val_loss improved from 0.65944 to 0.56717, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 9/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5433 - acc: 0.8367 - v
al_loss: 0.5896 - val_acc: 0.7675

Epoch 00009: val_loss did not improve from 0.56717
Epoch 10/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5370 - acc: 0.8391 - v
al_loss: 0.5709 - val_acc: 0.7911

Epoch 00010: val_loss did not improve from 0.56717
Epoch 11/100
222929/222929 [==============================] - 14s 61us/step - loss: 0.5364 - acc: 0.8388 - v
al_loss: 0.5958 - val_acc: 0.7665

Epoch 00011: val_loss did not improve from 0.56717
Epoch 12/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5315 - acc: 0.8397 - v
al_loss: 0.5425 - val_acc: 0.8076

Epoch 00012: val_loss improved from 0.56717 to 0.54255, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 13/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5287 - acc: 0.8399 - v
al_loss: 0.5870 - val_acc: 0.7700

Epoch 00013: val_loss did not improve from 0.54255
Epoch 14/100
222929/222929 [==============================] - 14s 61us/step - loss: 0.5271 - acc: 0.8411 - v
al_loss: 0.6056 - val_acc: 0.7599

Epoch 00014: val_loss did not improve from 0.54255
Epoch 15/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5191 - acc: 0.8434 - v
al_loss: 0.5850 - val_acc: 0.7828

Epoch 00015: val_loss did not improve from 0.54255
Epoch 16/100
222929/222929 [==============================] - 14s 61us/step - loss: 0.5182 - acc: 0.8430 - v
al_loss: 0.6512 - val_acc: 0.7191

Epoch 00016: val_loss did not improve from 0.54255
Epoch 17/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5158 - acc: 0.8445 - v
al_loss: 0.5740 - val_acc: 0.7776

Epoch 00017: val_loss did not improve from 0.54255
Epoch 18/100
222929/222929 [==============================] - 14s 63us/step - loss: 0.5136 - acc: 0.8450 - v
al_loss: 0.5192 - val_acc: 0.8322

Epoch 00018: val_loss improved from 0.54255 to 0.51917, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 19/100
```

```
222929/222929 [==============================] - 14s 63us/step - loss: 0.5081 - acc: 0.8468 - v
al_loss: 0.5291 - val_acc: 0.8186

Epoch 00019: val_loss did not improve from 0.51917
Epoch 20/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5105 - acc: 0.8455 - v
al_loss: 0.5958 - val_acc: 0.7588

Epoch 00020: val_loss did not improve from 0.51917
Epoch 21/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5086 - acc: 0.8467 - v
al_loss: 0.5131 - val_acc: 0.8379

Epoch 00021: val_loss improved from 0.51917 to 0.51311, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 22/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5041 - acc: 0.8479 - v
al_loss: 0.5225 - val_acc: 0.8279

Epoch 00022: val_loss did not improve from 0.51311
Epoch 23/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5048 - acc: 0.8480 - v
al_loss: 0.5078 - val_acc: 0.8260

Epoch 00023: val_loss improved from 0.51311 to 0.50776, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 24/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.5016 - acc: 0.8481 - v
al_loss: 0.5130 - val_acc: 0.8315

Epoch 00024: val_loss did not improve from 0.50776
Epoch 25/100
222929/222929 [==============================] - 14s 63us/step - loss: 0.5002 - acc: 0.8491 - v
al_loss: 0.5168 - val_acc: 0.8264

Epoch 00025: val_loss did not improve from 0.50776
Epoch 26/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.4962 - acc: 0.8507 - v
al_loss: 0.5109 - val_acc: 0.8297

Epoch 00026: val_loss did not improve from 0.50776
Epoch 27/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.4998 - acc: 0.8491 - v
al_loss: 0.5078 - val_acc: 0.8349

Epoch 00027: val_loss did not improve from 0.50776
Epoch 28/100
222929/222929 [==============================] - 14s 63us/step - loss: 0.4964 - acc: 0.8504 - v
al_loss: 0.5348 - val_acc: 0.8087

Epoch 00028: val_loss did not improve from 0.50776
Epoch 29/100
222929/222929 [==============================] - 14s 61us/step - loss: 0.4915 - acc: 0.8511 - v
al_loss: 0.5052 - val_acc: 0.8418

Epoch 00029: val_loss improved from 0.50776 to 0.50516, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 30/100
222929/222929 [==============================] - 14s 63us/step - loss: 0.4902 - acc: 0.8522 - v
al_loss: 0.4933 - val_acc: 0.8519

Epoch 00030: val_loss improved from 0.50516 to 0.49331, saving model to ./saved_models/weights.
best.CNN_arch.hdf5
Epoch 31/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.4900 - acc: 0.8516 - v
al_loss: 0.5153 - val_acc: 0.8225

Epoch 00031: val_loss did not improve from 0.49331
Epoch 32/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.4864 - acc: 0.8525 - v
al_loss: 0.5243 - val_acc: 0.8172

Epoch 00032: val_loss did not improve from 0.49331
Epoch 33/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.4891 - acc: 0.8518 - v
al_loss: 0.4998 - val_acc: 0.8314

Epoch 00033: val_loss did not improve from 0.49331
Epoch 34/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.4869 - acc: 0.8530 - v
```

```
al_loss: 0.4981 - val_acc: 0.8364

Epoch 00034: val_loss did not improve from 0.49331
Epoch 35/100
222929/222929 [==============================] - 14s 62us/step - loss: 0.4829 - acc: 0.8540 - v
al_loss: 0.5103 - val_acc: 0.8329

Epoch 00035: val_loss did not improve from 0.49331
Epoch 36/100
222929/222929 [==============================] - 15s 68us/step - loss: 0.4812 - acc: 0.8539 - v
al_loss: 0.5253 - val_acc: 0.8240

Epoch 00036: val_loss did not improve from 0.49331
```
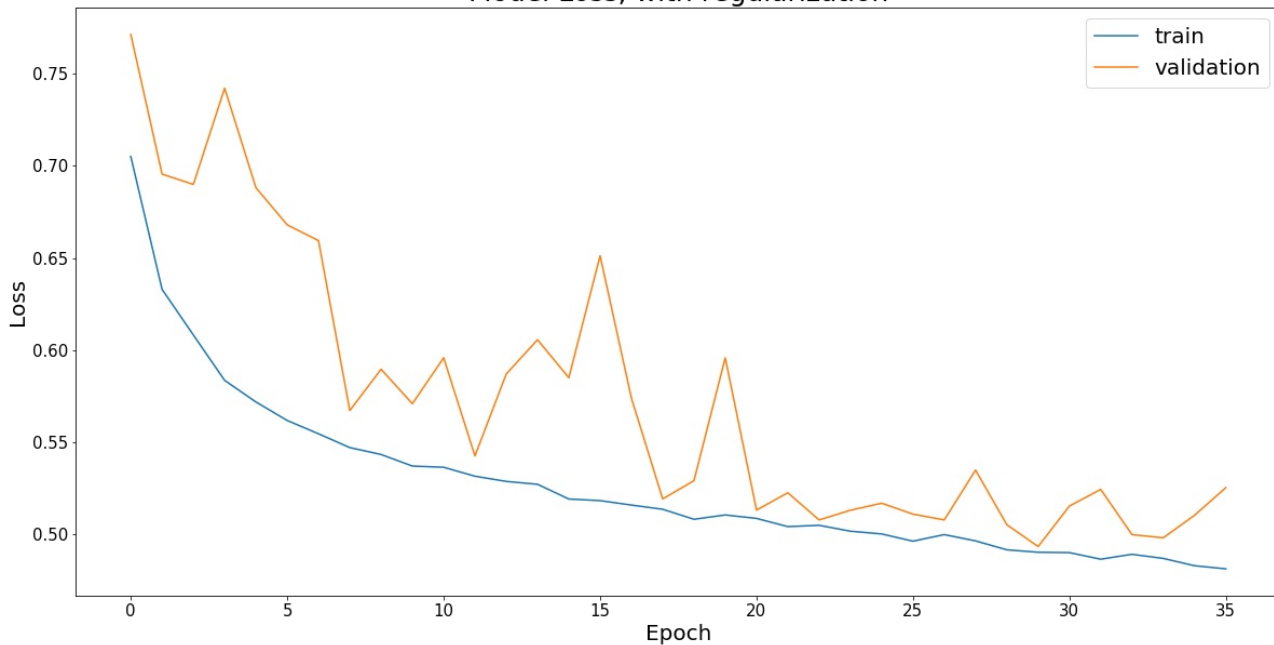
In [157]:

```python
# Visualize the training and validation loss
plt.figure(figsize=(20, 10))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.yticks(size=15)
plt.ylabel('Loss', size=20)
plt.xticks(size=15)
plt.xlabel('Epoch', size=20)
plt.legend(['train', 'validation'], loc='upper right', fontsize=20)
plt.title('Model Loss, with regularization', size=25)
plt.show()
```


The custom architecture obtains better losses than the transfer learning model. The validation loss tends to present high jumps during training but approaches the training loss as the number of epochs increases.

Custom CNN evaluation metrics values

Load model with best validation loss:

In [26]:

```python
model = load_model('./saved_models/weights.best.CNN_arch.hdf5')
```

Test model in validation set:

In [27]:

```python
# get predictions for each image in test set
predictions = [np.int(np.round(model.predict(np.expand_dims(image, axis=0)))) \
               for image in images[idx_test]]
```

In [28]:

```
# report test accuracy
test_accuracy = np.sum(np.array(predictions)==labels[idx_test])/len(predictions)
print('Test accuracy: %.4f%%' %(100*test_accuracy))
```

Test accuracy: 84.9726%

In [29]:

```
y_test = labels[idx_test]
y_pred = np.array(predictions)

confusion = confusion_matrix(y_test, y_pred)

precision, recall, f1_score, support = precision_recall_fscore_support(y_test, y_pred, beta=1)

y_pred_prob = [np.float32(model.predict(np.expand_dims(image, axis=0))) \
               for image in images[idx_test]]
y_pred_prob = np.array(y_pred_prob).reshape(-1)

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

auc_roc = auc(fpr, tpr)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr, label='CNN')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('CNN ROC Curve')
plt.show()

print("Area under ROC: {:.4f}".format(auc_roc))
print("Precision for 0 and 1: {:.2f}% and {:.2f}%".format(precision[0]*100, precision[1]*100))
print("Recall for 0 and 1: {:.2f}% and {:.2f}%".format(recall[0]*100, recall[1]*100))
print("F1 score for 0 and 1: {:.2f}% and {:.2f}%".format(f1_score[0]*100, f1_score[1]*100))
print("Confusion matrix:\n {}".format(confusion))
```
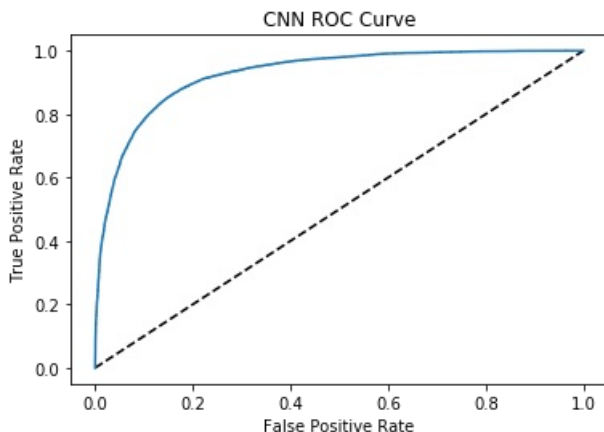


```
Area under ROC: 0.9263
Precision for 0 and 1: 93.63% and 69.19%
Recall for 0 and 1: 84.71% and 85.63%
F1 score for 0 and 1: 88.95% and 76.53%
Confusion matrix:
 [[16642  3004]
 [ 1132  6745]]
```

The custom CNN network resuls in an accuracy of 84.9726% and an F_1 score in healthy images (class 0) of 88.95%. For the IDC images (label 1) the obtained F_1 is 76.53%.

This is better than the transfer learning model using VGG19. Now given that we know that the classes are unbalanced, let's train a similar architecture by performing image augmention.

Custom architecture with image augmentation

Image augmentation increases the number of avalaible images by applying random rotations, shifts and flips to the existing images to generate new ones.

In [162]:

```python
from keras.preprocessing.image import ImageDataGenerator

# Create and configure image generator
data_gen_train = ImageDataGenerator(rotation_range=90.0, \
                                    width_shift_range=0.2, \
                                    height_shift_range=0.2, \
                                    horizontal_flip=True)

# Fit image generator to data
data_gen_train.fit(images[idx_train])
```

Given that augmented data is used, this time the method fit_generator (https://keras.io/models/model/#fit_generator) is used. For fit_generator the validation data needs to be specified directly. It cannot represent a percentage of the training input. Then we divide the training data into a train and validation sets.

In [163]:

```python
idx_train_2, idx_val = train_test_split(np.arange(idx_train.shape[0]), test_size=0.1, \
                                        random_state=seed, shuffle=True, stratify=labels[idx_train])
```

Same architecture as before:

In [164]:

```python
model = Sequential()

### Architecture
# Layer 1 : input layer
model.add(Conv2D(filters=32, kernel_size=3, padding='same', \
            activation='relu', input_shape=(50,50,3)))
model.add(MaxPooling2D(pool_size=3))
model.add(Dropout(0.2))
# Layer 2
model.add(Conv2D(filters=64, kernel_size=3, padding='same', \
            activation='relu'))
model.add(MaxPooling2D(pool_size=3))
model.add(Dropout(0.2))
# Layer 3
model.add(Conv2D(filters=128, kernel_size=3, padding='same', \
            activation='relu'))
model.add(MaxPooling2D(pool_size=3))
model.add(Dropout(0.2))

#model.add(GlobalAveragePooling2D('channels_last'))

#Flat layers
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
# Output layer
model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 50, 50, 32)        896
_____
max_pooling2d_8 (MaxPooling2 (None, 16, 16, 32)        0
_____
dropout_14 (Dropout)         (None, 16, 16, 32)        0
_____
conv2d_9 (Conv2D)            (None, 16, 16, 64)        18496
_____
max_pooling2d_9 (MaxPooling2 (None, 5, 5, 64)          0
_____
dropout_15 (Dropout)         (None, 5, 5, 64)          0
_____
conv2d_10 (Conv2D)           (None, 5, 5, 128)         73856
_____
max_pooling2d_10 (MaxPooling (None, 1, 1, 128)         0
_____
dropout_16 (Dropout)         (None, 1, 1, 128)         0
_____
flatten_11 (Flatten)         (None, 128)               0
_____
dense_31 (Dense)             (None, 64)                8256
_____
dense_32 (Dense)             (None, 32)                2080
_____
dense_33 (Dense)             (None, 1)                 33
=================================================================
Total params: 103,617
Trainable params: 103,617
Non-trainable params: 0
_____
```

In [165]:

```
epochs = 100
batch_size = 256

stopping_condition = EarlyStopping(patience=6)
checkpointer = ModelCheckpoint(filepath='./saved_models/weights.best.CNN_arch_augmented.hdf5',
                               verbose=1, save_best_only=True)

history = model.fit_generator(data_gen_train.flow(images[idx_train_2], labels[idx_train_2], \
                                batch_size=batch_size), \
                    validation_data=(images[idx_val], labels[idx_val]), \
                    steps_per_epoch=len(idx_train_2)//batch_size, \
                    epochs=epochs, class_weight={0:1, 1:2.5},\
                    callbacks=[checkpointer, stopping_condition], verbose=1)
```

```
Epoch 1/100
870/870 [==============================] - 146s 168ms/step - loss: 0.7010 - acc: 0.7799 - val_l
oss: 0.4798 - val_acc: 0.7914

Epoch 00001: val_loss improved from inf to 0.47978, saving model to ./saved_models/weights.best
.CNN_arch_augmented.hdf5
Epoch 2/100
870/870 [==============================] - 143s 164ms/step - loss: 0.6222 - acc: 0.8095 - val_l
oss: 0.5142 - val_acc: 0.7533

Epoch 00002: val_loss did not improve from 0.47978
Epoch 3/100
870/870 [==============================] - 145s 167ms/step - loss: 0.5984 - acc: 0.8188 - val_l
oss: 0.5968 - val_acc: 0.7031

Epoch 00003: val_loss did not improve from 0.47978
Epoch 4/100
870/870 [==============================] - 144s 165ms/step - loss: 0.5821 - acc: 0.8245 - val_l
oss: 0.4705 - val_acc: 0.7918

Epoch 00004: val_loss improved from 0.47978 to 0.47052, saving model to ./saved_models/weights.
best.CNN_arch_augmented.hdf5
Epoch 5/100
870/870 [==============================] - 139s 160ms/step - loss: 0.5728 - acc: 0.8276 - val_l
oss: 0.4635 - val_acc: 0.7925

Epoch 00005: val_loss improved from 0.47052 to 0.46350, saving model to ./saved_models/weights.
best.CNN_arch_augmented.hdf5
Epoch 6/100
870/870 [==============================] - 138s 159ms/step - loss: 0.5688 - acc: 0.8290 - val_l
```

oss: 0.3789 - val_acc: 0.8425

Epoch 00006: val_loss improved from 0.46350 to 0.37890, saving model to ./saved_models/weights.
best.CNN_arch_augmented.hdf5
Epoch 7/100
870/870 [==============================] - 127s 146ms/step - loss: 0.5593 - acc: 0.8320 - val_l
oss: 0.4567 - val_acc: 0.7980

Epoch 00007: val_loss did not improve from 0.37890
Epoch 8/100
870/870 [==============================] - 107s 123ms/step - loss: 0.5601 - acc: 0.8322 - val_l
oss: 0.4378 - val_acc: 0.8090

Epoch 00008: val_loss did not improve from 0.37890
Epoch 9/100
870/870 [==============================] - 105s 121ms/step - loss: 0.5553 - acc: 0.8334 - val_l
oss: 0.4564 - val_acc: 0.8030

Epoch 00009: val_loss did not improve from 0.37890
Epoch 10/100
870/870 [==============================] - 99s 114ms/step - loss: 0.5518 - acc: 0.8340 - val_lo
ss: 0.5760 - val_acc: 0.7260

Epoch 00010: val_loss did not improve from 0.37890
Epoch 11/100
870/870 [==============================] - 96s 110ms/step - loss: 0.5482 - acc: 0.8347 - val_lo
ss: 0.4835 - val_acc: 0.7961

Epoch 00011: val_loss did not improve from 0.37890
Epoch 12/100
870/870 [==============================] - 92s 106ms/step - loss: 0.5464 - acc: 0.8359 - val_lo
ss: 0.3691 - val_acc: 0.8450

Epoch 00012: val_loss improved from 0.37890 to 0.36908, saving model to ./saved_models/weights.
best.CNN_arch_augmented.hdf5
Epoch 13/100
870/870 [==============================] - 92s 106ms/step - loss: 0.5450 - acc: 0.8372 - val_lo
ss: 0.3925 - val_acc: 0.8377

Epoch 00013: val_loss did not improve from 0.36908
Epoch 14/100
870/870 [==============================] - 92s 106ms/step - loss: 0.5390 - acc: 0.8386 - val_lo
ss: 0.3956 - val_acc: 0.8392

Epoch 00014: val_loss did not improve from 0.36908
Epoch 15/100
870/870 [==============================] - 96s 110ms/step - loss: 0.5402 - acc: 0.8377 - val_lo
ss: 0.4089 - val_acc: 0.8248

Epoch 00015: val_loss did not improve from 0.36908
Epoch 16/100
870/870 [==============================] - 95s 109ms/step - loss: 0.5372 - acc: 0.8374 - val_lo
ss: 0.4103 - val_acc: 0.8281

Epoch 00016: val_loss did not improve from 0.36908
Epoch 17/100
870/870 [==============================] - 96s 110ms/step - loss: 0.5358 - acc: 0.8379 - val_lo
ss: 0.3927 - val_acc: 0.8438

Epoch 00017: val_loss did not improve from 0.36908
Epoch 18/100
870/870 [==============================] - 102s 118ms/step - loss: 0.5322 - acc: 0.8404 - val_l
oss: 0.4238 - val_acc: 0.8250

Epoch 00018: val_loss did not improve from 0.36908

In [166]:

```
# Visualize the training and validation loss
plt.figure(figsize=(20, 10))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.yticks(size=15)
plt.ylabel('Loss', size=20)
plt.xticks(size=15)
plt.xlabel('Epoch', size=20)
plt.legend(['train', 'validation'], loc='upper right', fontsize=20)
plt.title('Model Loss, Augmented Images with regularization', size=25)
plt.show()
```
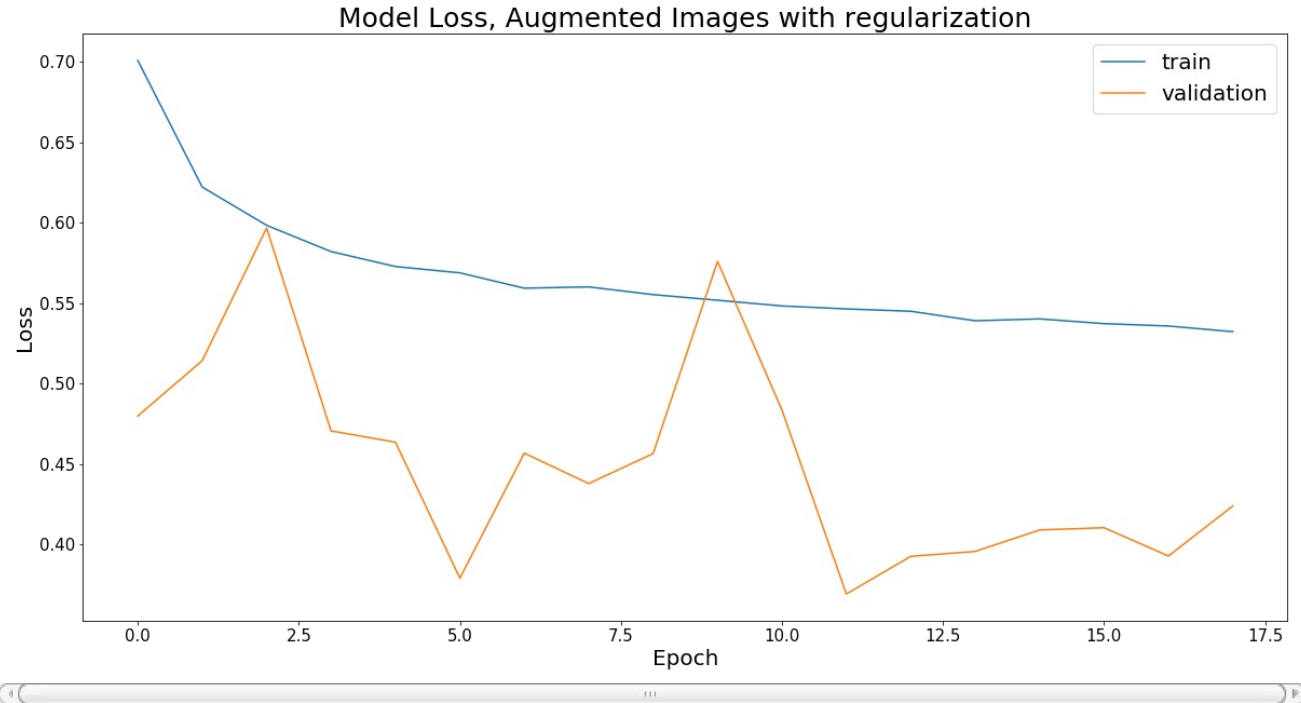

The plot belows shows that introducing image augmentation for this data set doesn't give any advantage. The training is slower and the train loss is no better than without augmentation and the validation loss doesn't seem to stabilize.

Custom CNN with imge augmentation evaluation metrics

In [167]:

```
model.load_weights('./saved_models/weights.best.CNN_arch_augmented.hdf5')
```

In [168]:

```
# get predictions for each image in test set
predictions = [np.int(np.round(model.predict(np.expand_dims(image, axis=0)))) \
               for image in images[idx_test]]
```

In [169]:

```
# report test accuracy
test_accuracy = np.sum(np.array(predictions)==labels[idx_test])/len(predictions)
print('Test accuracy: %.4f%%' %(100*test_accuracy))
```

Test accuracy: 84.5220%

In [171]:

```python
y_test = labels[idx_test]
y_pred = np.array(predictions)

confusion = confusion_matrix(y_test, y_pred)

precision, recall, f1_score, support = precision_recall_fscore_support(y_test, y_pred, beta=1)

y_pred_prob = [np.float32(model.predict(np.expand_dims(image, axis=0))) \
                for image in images[idx_test]]
y_pred_prob = np.array(y_pred_prob).reshape(-1)

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

auc_roc = auc(fpr, tpr)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr, label='CNN')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('CNN with Image Augmentation ROC Curve')
plt.show()

print("Area under ROC: {:.4f}".format(auc_roc))
print("Precision for 0 and 1: {:.2f}% and {:.2f}%".format(precision[0]*100, precision[1]*100))
print("Recall for 0 and 1: {:.2f}% and {:.2f}%".format(recall[0]*100, recall[1]*100))
print("F1 score for 0 and 1: {:.2f}% and {:.2f}%".format(f1_score[0]*100, f1_score[1]*100))
print("Confusion matrix:\n {}".format(confusion))
```
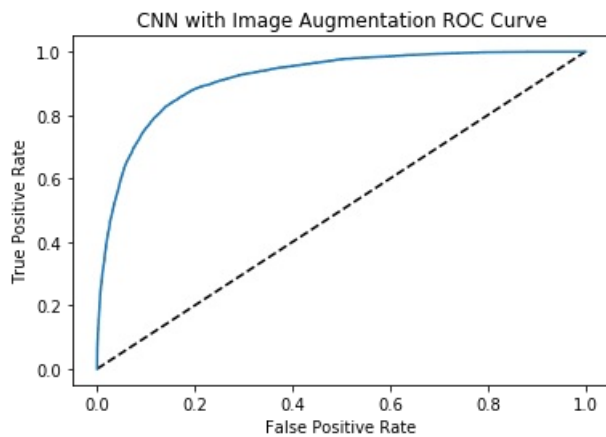


```
Area under ROC: 0.9166
Precision for 0 and 1: 92.88% and 68.88%
Recall for 0 and 1: 84.82% and 83.78%
F1 score for 0 and 1: 88.67% and 75.60%
Confusion matrix:
 [[16664  2982]
 [ 1278  6599]]
```

The custom CNN network using image augmentation resulted in an accuracy of 84.5220% and an F_1 score in the healthy images (class 0) of 88.67%. For the IDC images (label 1) the F_1 was 75.60%.

Results

The table below summarizes the results for each model considered in this project.

Model	Accuracy	F_1 score (labels 0)	F_1 score (labels 1)
Benchmark naive model	71.38%	83.30	-
Pre-trained VGG19	80.6271%	86.02%	68.45%
Custom CNN	84.9726%	88.95%	76.53%
Custom CNN with image augmentation	84.5220%	88.67%	75.60%

As the table shows, the custom CNN has the better metrics. The custom CNN with image augmentation behaves similarly however its training time is too slow. Interestingly, the pre-trained model didn't behave as well compared to the best models. This is certainly due to the fact that the histopathology images are tottaly different to those used in ImageNet.

Import custom CNN:

In [30]:

```
from keras.models import load_model

model = load_model('./saved_models/weights.best.CNN_arch.hdf5')
```

Let's plot again the ROC curve for the custom CNN:

In [68]:

```
# get predictions for each image in test set
predictions = [np.int(np.round(model.predict(np.expand_dims(image, axis=0)))) \
               for image in images[idx_test]]

y_test = labels[idx_test]
y_pred = np.array(predictions)

confusion = confusion_matrix(y_test, y_pred)

precision, recall, f1_score, support = precision_recall_fscore_support(y_test, y_pred, beta=1)

y_pred_prob = [np.float32(model.predict(np.expand_dims(image, axis=0))) \
               for image in images[idx_test]]
y_pred_prob = np.array(y_pred_prob).reshape(-1)

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

auc_roc = auc(fpr, tpr)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr, label='CNN')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Custom CNN ROC Curve')
plt.show()

print("Area under ROC: {:.4f}".format(auc_roc))
print("Precision for 0 and 1: {:.2f}% and {:.2f}%".format(precision[0]*100, precision[1]*100))
print("Recall for 0 and 1: {:.2f}% and {:.2f}%".format(recall[0]*100, recall[1]*100))
print("F1 score for 0 and 1: {:.2f}% and {:.2f}%".format(f1_score[0]*100, f1_score[1]*100))
print("Confusion matrix:\n {}".format(confusion))
```
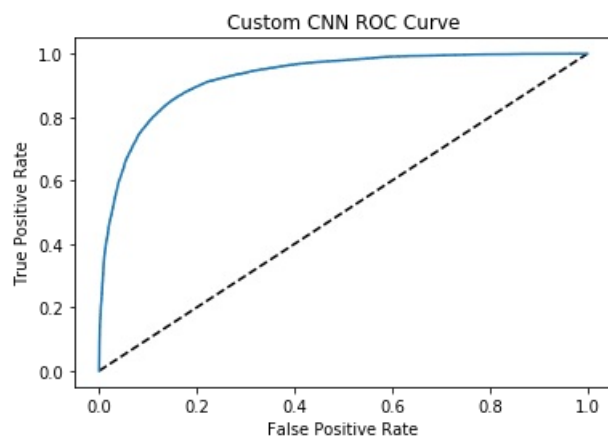


```
Area under ROC: 0.9263
Precision for 0 and 1: 93.63% and 69.19%
Recall for 0 and 1: 84.71% and 85.63%
F1 score for 0 and 1: 88.95% and 76.53%
Confusion matrix:
 [[16642  3004]
 [ 1132  6745]]
```

The false positive rate (FPR) can be improved by changing the threshold on the predictions. For example if the goal is to have a false positive rate smaller than 1%, then the threshold to classify an image as an IDC image must be:

In [87]:

```
# Find index for 1% threshold
thresh = 0.01
idx_1per = list(fpr).index(max([i for i in fpr if i < thresh]))
print("Threshold to ensure FPR is smaller than 1%%: %.6f" %fpr[idx_1per])
```

Threshold to ensure FPR is smaller than 1%: 0.009977

Predicitions and metrics using new threshold:

In [90]:

```
# Predictions with new threshold
predictions_1per = np.array(y_pred_prob >= thresholds[idx_1per])

# Metrics
confusion = confusion_matrix(y_test, predictions_1per)

precision, recall, f1_score, support = precision_recall_fscore_support(y_test, predictions_1per, beta=1)

accuracy = np.sum(np.array(predictions_1per)==labels[idx_test])/len(idx_test)

print("Accuracy: {:.2f}%%".format(accuracy*100))
print("Precision for 0 and 1: {:.2f}% and {:.2f}%".format(precision[0]*100, precision[1]*100))
print("Recall for 0 and 1: {:.2f}% and {:.2f}%".format(recall[0]*100, recall[1]*100))
print("F1 score for 0 and 1: {:.2f}% and {:.2f}%".format(f1_score[0]*100, f1_score[1]*100))
print("Confusion matrix:\n {}".format(confusion))
```

```
Accuracy: 80.69%%
Precision for 0 and 1: 79.16% and 93.36%
Recall for 0 and 1: 99.00% and 35.01%
F1 score for 0 and 1: 87.98% and 50.93%
Confusion matrix:
 [[19450   196]
 [ 5119  2758]]
```
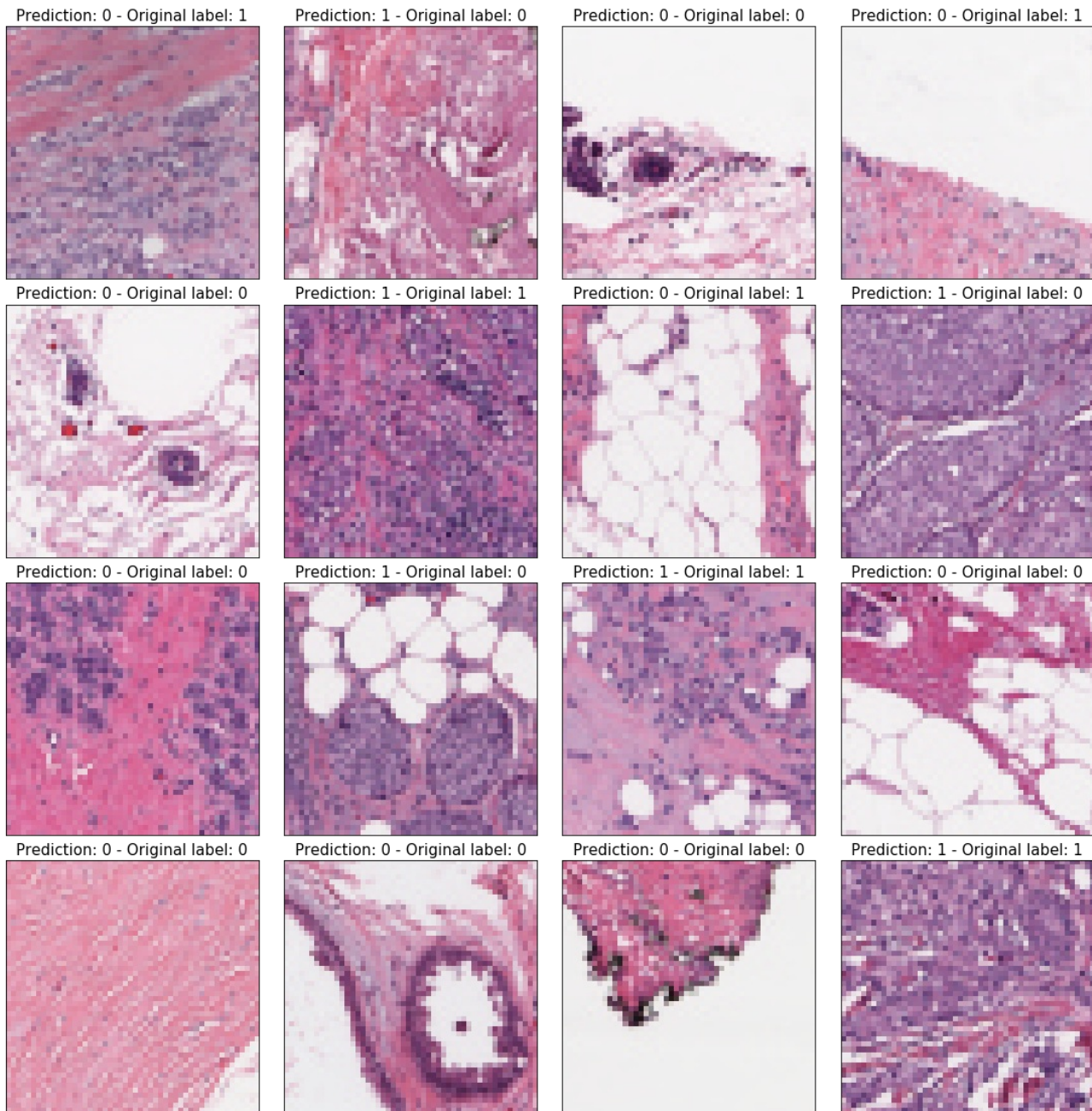
The new threshold increases the precision to 93.96% but decreases accuracy to 80.69%. Similarly it decreases the recall value for positive classes, indicating that adjusting the threshold for a higher precision also increases the false negative rate. This is confirmed by the confusion matrix.

Let's now check some of the image predictions by the custom CNN with the sklearn standard threshold.

In [295]:

```
fig = plt.figure(figsize=(15,15))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.1, wspace=0.1)
for i in range(4, 20):
    ax = fig.add_subplot(4, 4, i - 3, xticks=[], yticks=[])
    ax.imshow(images[idx_test[i]])
    _ = plt.title("Prediction: {} - Original label: {}".format(predictions[i], labels[idx_test[i]]), \
            size=15)
_ = plt.suptitle("Predictions and Original Labels", size = 20, y=1.05)
```

Predictions and Original Labels

Prediction: 0 - Original label: 1	Prediction: 1 - Original label: 0	Prediction: 0 - Original label: 0	Prediction: 0 - Original label: 1
Prediction: 0 - Original label: 0	Prediction: 1 - Original label: 1	Prediction: 0 - Original label: 1	Prediction: 1 - Original label: 0
Prediction: 0 - Original label: 0	Prediction: 1 - Original label: 0	Prediction: 1 - Original label: 1	Prediction: 0 - Original label: 0
Prediction: 0 - Original label: 0	Prediction: 0 - Original label: 0	Prediction: 0 - Original label: 0	Prediction: 1 - Original label: 1

Let's visualize separately the properly classified images and the misclassified images.

```python
# Display mislabeled images
fig = plt.figure(figsize=(15,6))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.1, wspace=0.1)
j = 1
for i in range(4, 41):
    if predictions[i] != labels[idx_test[i]]:
        ax = fig.add_subplot(2, 5, j, xticks=[], yticks=[])
        ax.imshow(images[idx_test[i]])
        _ = plt.title("Prediction: {} - Original label: {}".format(predictions[i], labels[idx_test[i]]), \
                size=12)
        j += 1
_ = plt.suptitle("Mislabeled Images", size = 20, y=1.1)
# Display correctly labeled images
fig = plt.figure(figsize=(15,6))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.1, wspace=0.1)
j = 1
for i in range(4, 20):
    if predictions[i] == labels[idx_test[i]]:
        ax = fig.add_subplot(2, 5, j, xticks=[], yticks=[])
        ax.imshow(images[idx_test[i]])
        _ = plt.title("Prediction: {} - Original label: {}".format(predictions[i], labels[idx_test[i]]), \
                size=12)
        j += 1
_ = plt.suptitle("Properly labeled Images", size = 20, y=1.1)
```

Mislabeled Images
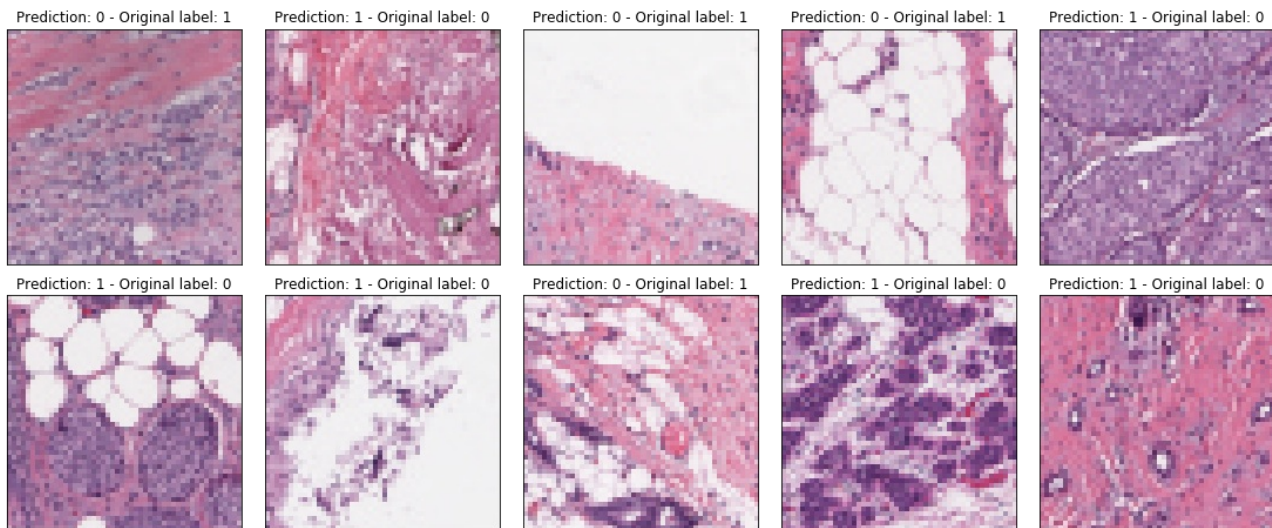
Properly labeled Images
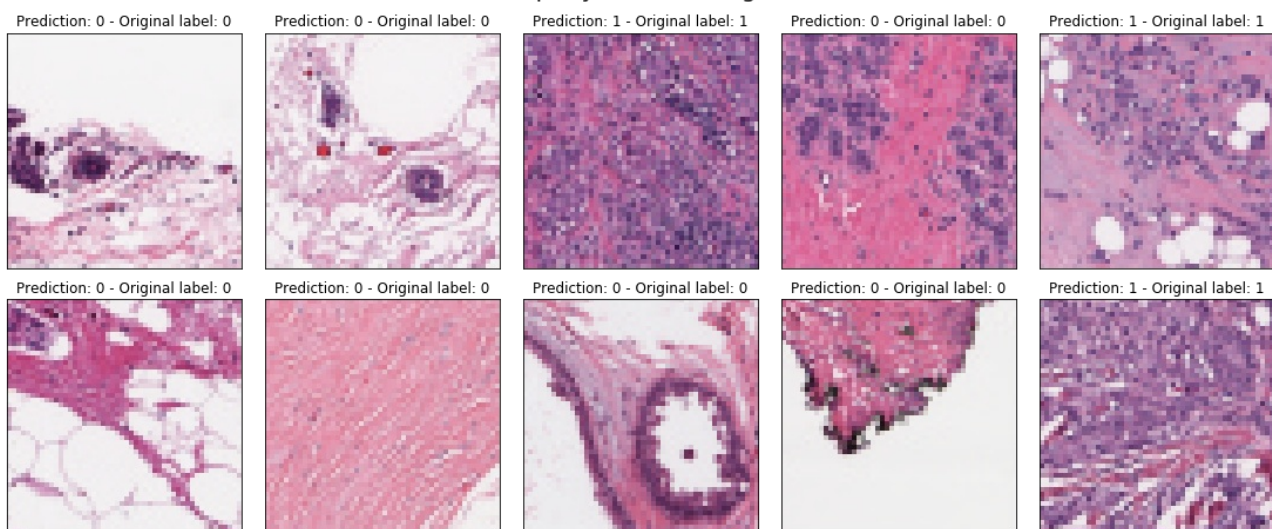
On the misclassified images it is difficult to visually determine why the algorithm failed. However, it seems that the model gives more weight to the violet tones than to the structure to labeled an image as being IDC. Similarly it seems that the algorithm considers that images with a lot of white areas and clear tones of violet to be an indication of a healthy tissue.

Conclusion

A model to classify images from and unbalanced data set was proposed. Given the imbalance, accuracy it is not the only metric of interest. The F_1 score also needs to be considered. For this particular problem, where it is important to accurately identify IDC regions (positive classes), precision is also a metric of choice.

If precision is very important for a user of this model, in order to improve this metric the threshold to classify an image as presenting IDC can be adjusted. Given that increasing the precision also increases the false negative rate (FNR), the user needs to consider the most appropriate balance between a high precision and the FNR. Also, given that the output of the model is the probability for an image of being from an IDC region, the histopathologist can define a range between 0 and 1 to choose which images need to be visually analyzed.

When developing the model it was found that transfer learning didn't perform as well as expected with the VGG19 architecture. This can be explained by the fact that the original weights from VGG19 are calculated using images from ImageNet (http://www.image-net.org/). ImageNet is an image data base consisting of 1000 categories of common objects and animals, and the histhopathology images used in this project are totally different. This explains the performance of the transfer learning model. Surely a full training of the VGG19 architecture using the histopathology images will provide a more reliable classifier.

The best results were obtained implementing a simple architecture using what is known to work properly in different famous CNN architectures. It is accepted that in most cases image aumentation tends to improve a model. However, for this particular data set it was not the case. In the present situation image augmentation just turned training into a very slow task with no gain. So, it seems that for this kind of images image augmentation should be avoided.

A quick analysis of the predictions in the validation data tends to hint that the model considers tones of violet and image structure to assign the probability of an image to be of cancerous tissue. Most likely a higher accuracy was not obtained given that, as noted in this paper (https://www.ncbi.nlm.nih.gov/pubmed/27563488), not all the IDC regions in the original image slides were fully indicated by the hysthopathologist. As noted at the beginning this is a very time consumming and arduous task, thus the interest on developing this kind of classifiers for the medical sciences.