```
#Load the modules
from glob import glob
import numpy as np
import pandas as pd
import keras,cv2,os

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, BatchNormalization, Activation
from keras.layers import Conv2D, MaxPool2D

from tqdm import tqdm_notebook,trange
import matplotlib.pyplot as plt

import gc #garbage collection, we need to save all the RAM we can
```

```
import pandas as pd
from glob import glob
import os

# Define data directories
root_dir = "../data/"  # Modify this path for your local setup
test_dir = root_dir + 'test/'
train_dir = root_dir + 'train/'

# Create DataFrame with image file paths
image_df = pd.DataFrame({'image_path': glob(os.path.join(train_dir, '*.tif'))})  # Gather all TIFF files

# Extract unique identifiers from file paths
image_df['unique_id'] = image_df.image_path.map(lambda x: x.split('/')[3].split(".")[0])  # Parse file names for IDs

# Import label information
label_info = pd.read_csv(root_dir + "train_labels.csv")  # Load label data from CSV

# Combine image paths with corresponding labels
combined_df = image_df.merge(label_info, on="unique_id")  # Join datasets based on unique ID

# Display initial rows of the resulting DataFrame
print(combined_df.head(3))  # Show first three entries for verification
```

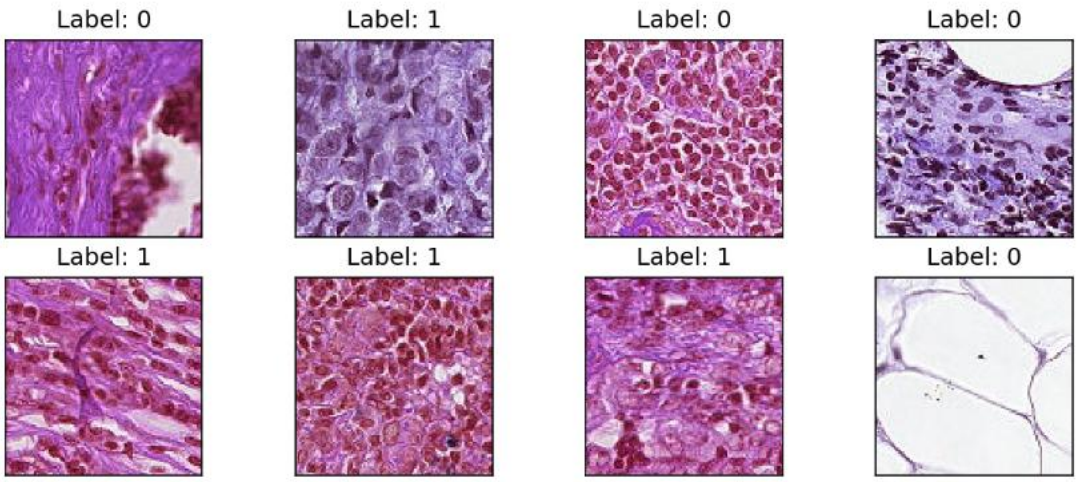
0	../input/train/9dd758d956606454ae92ff95a538c1d...	9dd758d956606454ae92ff95a538c1de8be6bc3c	0
1	../input/train/4a7af151b3b65f8cb24351d1df6fc51...	4a7af151b3b65f8cb24351d1df6fc5130ecbc934	0
2	../input/train/c5e2d6be658f80396e54c502b68342c...	c5e2d6be658f80396e54c502b68342c66f9d4386	0

```
import numpy as np
import cv2
from tqdm.notebook import tqdm_notebook

def fetch_image_data(sample_size, dataframe):
    """
    Retrieves a specified number of images and their corresponding labels from the given dataframe.
    """
    # Initialize array for storing image data (sample_size, 96x96 pixels, RGB channels)
    image_array = np.zeros([sample_size, 96, 96, 3], dtype=np.uint8)

    # Extract labels and convert to numpy array
    label_array = np.squeeze(dataframe[['label']].values)[:sample_size]

    # Iterate through dataframe rows and load images
    for index, entry in tqdm_notebook(dataframe.iterrows(), total=sample_size):
        if index == sample_size:
            break
        image_array[index] = cv2.imread(entry['image_path'])

    return image_array, label_array
```

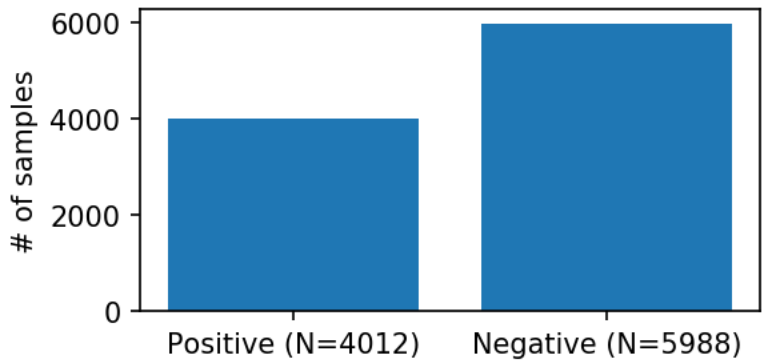
```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:7: FutureWarning: Method .as_matr
ix will be removed in a future version. Use .values instead.
  import sys
```

```
# Import a subset of 10,000 image samples
SAMPLE_SIZE = 10000
image_data, labels = fetch_image_data(sample_size=SAMPLE_SIZE, dataframe=combined_df)
```

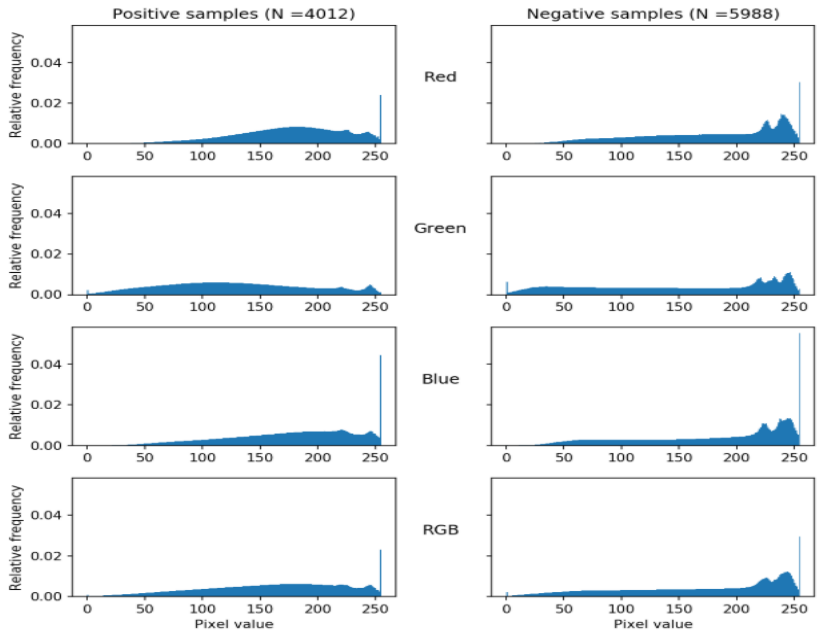
```
import matplotlib.pyplot as plt
import numpy as np

# Set up the plot layout
display_fig = plt.figure(figsize=(12, 5), dpi=120)

# Set random seed for reproducibility
np.random.seed(42)

# Select and display random images
for plot_index, random_idx in enumerate(np.random.randint(0, SAMPLE_SIZE, 8)):
    # Create subplot for each image
    subplot = display_fig.add_subplot(2, 4, plot_index + 1, xticks=[], yticks=[])

    # Show image and its corresponding label
    plt.imshow(image_data[random_idx])
    subplot.set_title(f'Label: {labels[random_idx]}')

plt.tight_layout()
plt.show()
```



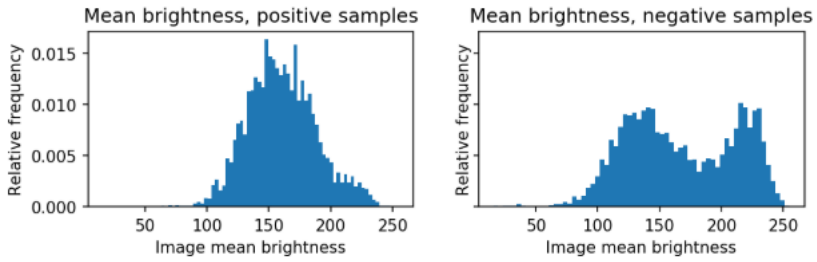

```
import matplotlib.pyplot as plt

# Create figure for the bar chart
chart_fig = plt.figure(figsize=(5, 3), dpi=120)

# Calculate class distribution
negative_count = (labels == 0).sum()
positive_count = (labels == 1).sum()

# Plot bar chart of class distribution
plt.bar([0, 1], [positive_count, negative_count])

# Customize x-axis labels
plt.xticks([0, 1], [f"Positive (N={positive_count})", f"Negative (N={negative_count})"])

# Add y-axis label
plt.ylabel("# of samples")

# Display the plot
plt.tight_layout()
plt.show()
```



```python
import matplotlib.pyplot as plt

# Define number of histogram bins
bin_count = 255

# Create figure and subplots
fig, axes = plt.subplots(4, 2, sharey=True, figsize=(10, 10), dpi=120)

# Plot histograms for each color channel and combined RGB
channels = ['Red', 'Green', 'Blue', 'RGB']
for i, channel in enumerate(channels):
    if channel == 'RGB':
        axes[i, 0].hist(class_1_images.flatten(), bins=bin_count, density=True)
        axes[i, 1].hist(class_0_images.flatten(), bins=bin_count, density=True)
    else:
        axes[i, 0].hist(class_1_images[:,:,:,i].flatten(), bins=bin_count, density=True)
        axes[i, 1].hist(class_0_images[:,:,:,i].flatten(), bins=bin_count, density=True)

    axes[i, 1].set_ylabel(channel, rotation='horizontal', labelpad=35, fontsize=12)

# Set titles and labels
axes[0, 0].set_title(f"Positive samples (N = {class_1_images.shape[0]})")
axes[0, 1].set_title(f"Negative samples (N = {class_0_images.shape[0]})")

for ax in axes[:, 0]:
    ax.set_ylabel("Relative Frequency")

axes[-1, 0].set_xlabel("Pixel Value")
axes[-1, 1].set_xlabel("Pixel Value")

# Adjust layout and display
plt.tight_layout()
plt.show()
```



```
# Configuration for histogram
bin_count = 32  # Reduced number of bins for smoother distribution

# Create subplots
fig, (ax_pos, ax_neg) = plt.subplots(1, 2, sharey=True, sharex=True, figsize=(10, 3), dpi=150)

# Calculate and plot histogram for positive samples
pos_brightness = np.mean(iceberg_samples, axis=(1,2,3))
ax_pos.hist(pos_brightness, bins=bin_count, density=True)
ax_pos.set_title("Mean brightness, positive samples")
ax_pos.set_xlabel("Image mean brightness")
ax_pos.set_ylabel("Relative frequenc")

# Calculate and plot histogram for negative samples
neg_brightness = np.mean(ship_samples, axis=(1,2,3))
ax_neg.hist(neg_brightness, bins=bin_count, density=True)
ax_neg.set_title("Mean brightness, positive samples")
ax_neg.set_xlabel("Image mean brightness")
ax_neg.set_ylabel("Relative frequenc")

# Adjust layout and display
plt.tight_layout()
plt.show()
```




```
import matplotlib.pyplot as plt
import numpy as np

# Set number of histogram bins for smoother visualization
bin_count = 50

# Create figure and subplots
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, sharex=True, figsize=(10, 3), dpi=120)

# Calculate and plot mean brightness histograms
ax1.hist(np.mean(class_1_images, axis=(1,2,3)), bins=bin_count, density=True)
ax2.hist(np.mean(class_0_images, axis=(1,2,3)), bins=bin_count, density=True)

# Set titles and labels
ax1.set_title("Average Intensity Distribution - Class 1")
ax2.set_title("Average Intensity Distribution - Class 0")

ax1.set_xlabel("Mean Image Intensity")
ax2.set_xlabel("Mean Image Intensity")

ax1.set_ylabel("Density")
ax2.set_ylabel("Density")

# Adjust layout and display
plt.tight_layout()
plt.show()
```

```
import gc

# Clear specific variables from memory
class_1_images = None
class_0_images = None

# Trigger garbage collection to free up memory
gc.collect()
```

```
import numpy as np

# Define the proportion of data to use for training
train_ratio = 0.75

# Set random seed for reproducibility
np.random.seed(123)

# Generate shuffled indices
shuffled_indices = np.random.permutation(labels.shape[0])

# Apply shuffling to both image data and labels
shuffled_images = image_data[shuffled_indices]
shuffled_labels = labels[shuffled_indices]

# Calculate the index to split the data
split_point = int(np.round(train_ratio * shuffled_labels.shape[0]))

# Split the data into training and validation sets
# (Note: we're not explicitly creating validation sets here, just preparing for the split)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:7: FutureWarning: Method .as_matr
ix will be removed in a future version. Use .values instead.
  import sys
```

```
# Network architecture parameters
conv_filter_size = (3,3)
max_pool_size = (2,2)
filters_layer1 = 128
filters_layer2 = 64
filters_layer3 = 32

# Regularization parameters for preventing overfitting
dropout_rate_conv = 0.3
dropout_rate_dense = 0.5

# Create the sequential model
cnn_model = Sequential()

# Convolutional block 3
cnn_model.add(Conv2D(filters_layer1, conv_filter_size, use_bias=False))
cnn_model.add(BatchNormalization())
cnn_model.add(Activation("relu"))
cnn_model.add(Conv2D(filters_layer1, conv_filter_size, use_bias=False))
cnn_model.add(BatchNormalization())
cnn_model.add(Activation("relu"))
cnn_model.add(MaxPool2D(pool_size = max_pool_size))
cnn_model.add(Dropout(dropout_rate_conv))

# Convolutional block 2
cnn_model.add(Conv2D(filters_layer2, conv_filter_size, use_bias=False))
cnn_model.add(BatchNormalization())
cnn_model.add(Activation("relu"))
cnn_model.add(Conv2D(filters_layer2, conv_filter_size, use_bias=False))
cnn_model.add(BatchNormalization())
cnn_model.add(Activation("relu"))
cnn_model.add(MaxPool2D(pool_size = max_pool_size))
cnn_model.add(Dropout(dropout_rate_conv))

# Convolutional block 1
cnn_model.add(Conv2D(filters_layer3, conv_filter_size, input_shape = (96, 96, 3)))
cnn_model.add(BatchNormalization())
cnn_model.add(Activation("relu"))
cnn_model.add(Conv2D(filters_layer3, conv_filter_size, use_bias=False))
cnn_model.add(BatchNormalization())
cnn_model.add(Activation("relu"))
cnn_model.add(MaxPool2D(pool_size = max_pool_size))
cnn_model.add(Dropout(dropout_rate_conv))

# Fully connected layer for feature extraction
cnn_model.add(Flatten())
cnn_model.add(Dense(256, use_bias=False))
cnn_model.add(BatchNormalization())
cnn_model.add(Activation("relu"))
cnn_model.add(Dropout(dropout_rate_dense))

# Output layer with sigmoid activation for binary classification
cnn_model.add(Dense(1, activation = "sigmoid"))
```

```
# Define the number of samples processed in each iteration
samples_per_batch = 50

# Configure the model for training
cnn_model.compile(
    # Use binary cross-entropy as the loss function for binary classification
    loss=keras.losses.binary_crossentropy,
    # Employ Adam optimizer with a specified learning rate
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    # Monitor accuracy during training
    metrics=['accuracy']
)
```

```
# Training configuration
num_epochs = 3
batch_size = 50  # Assuming this was defined earlier

# Iterate through epochs
for epoch_num in range(num_epochs):
    # Calculate number of batches per epoch
    batch_count = np.floor(train_data_end / batch_size).astype(int)

    # Initialize metrics for this epoch
    epoch_loss, epoch_accuracy = 0, 0

    # Use tqdm for progress tracking
    with trange(batch_count) as progress_bar:
        for batch_idx in progress_bar:
            # Determine batch boundaries
            batch_start = batch_idx * batch_size
            batch_end = batch_start + batch_size

            # Extract current batch
            features_batch = X[batch_start:batch_end]
            labels_batch = y[batch_start:batch_end]

            # Train model on current batch
            batch_metrics = cnn_model.train_on_batch(features_batch, labels_batch)

            # Update running metrics
            epoch_loss += batch_metrics[0]
            epoch_accuracy += batch_metrics[1]

            # Update progress bar
            progress_bar.set_description(f'Running training epoch {epoch_num + 1}/{num_epochs}')
            progress_bar.set_postfix(
                loss=f"{epoch_loss / (batch_idx + 1):.2f}",
                acc=f"{epoch_accuracy / (batch_idx + 1):.2f}"
            )

# Note: Data is not reshuffled between epochs due to in-place train/validation split
```

```
Running training epoch 0: 100%|██████████| 3520/3520 [06:28<00:00,  9.14it/s, acc=0.85, loss=0.
35]
Running training epoch 1:  24%|██▍       | 846/3520 [01:31<04:49,  9.23it/s, acc=0.88, loss=0.
28]
```

```python
# Calculate the number of batches needed for processing
batch_count = np.floor((y.shape[0]-split_point) / samples_per_batch).astype(int) # Approximate calculation

# Initialize running metrics
running_loss, running_accuracy = 0, 0

# Use trange for a progress bar during iteration
with trange(batch_count) as progress_bar:
    for batch_num in progress_bar:
        # Determine the starting index for the current batch
        batch_start = batch_num * samples_per_batch

        # Extract the current batch of data and labels
        batch_data = X[batch_start:batch_start+samples_per_batch]
        batch_labels = y[batch_start:batch_start+samples_per_batch]

        # Evaluate the model on the current batch
        batch_metrics = cnn_model.test_on_batch(batch_data, batch_labels)

        # Update running metrics
        running_loss += batch_metrics[0]
        running_accuracy += batch_metrics[1]

        # Configure progress bar display
        progress_bar.set_description('Running training')
        progress_bar.set_postfix(
            loss="%.2f" % round(running_loss / (batch_num+1), 2),
            acc="%.2f" % round(running_accuracy / (batch_num+1), 2)
        )

# Print final validation results
print("Validation loss:", running_loss / batch_count)
print("Validation accuracy:", running_accuracy / batch_count)
```

```python
# Clear data from memory
feature_matrix = None
target_vector = None

# Invoke garbage collection to free up memory
import gc
gc.collect()
```

```
Running validation: 100%|██████████| 880/880 [00:33<00:00, 26.28it/s, acc=0.88, loss=0.34]

Validation loss: 0.3379617225251753
Validation accuracy: 0.8775227218866348
```

```python
# Set the directory for test images
test_image_directory = path + 'test/'

# Get all .tif files in the test directory
test_image_files = glob(os.path.join(test_image_directory, '*.tif'))

# Initialize DataFrame for predictions
prediction_results = pd.DataFrame()

# Set batch size for processing
images_per_batch = 5000

# Get total number of test images
total_images = len(test_image_files)

# Process test images in batches
for start_index in range(0, total_images, images_per_batch):
    end_index = start_index + images_per_batch
    print(f"Processing images: {start_index} - {end_index}")

    # Create DataFrame for current batch
    batch_df = pd.DataFrame({'image_path': test_image_files[start_index:end_index]})

    # Extract image IDs from filenames
    batch_df['image_id'] = batch_df.image_path.map(lambda x: x.split('/')[3].split(".")[0])

    # Load images for current batch
    batch_df['image_data'] = batch_df['image_path'].map(cv2.imread)

    # Convert image data to numpy array
    image_array = np.stack(batch_df["image_data"].values)

    # Generate predictions for current batch
    batch_predictions = cnn_model.predict(image_array, verbose=1)

    # Add predictions to DataFrame
    batch_df['predicted_label'] = batch_predictions

    # Append batch results to main results DataFrame
    prediction_results = pd.concat([prediction_results, batch_df[["image_id", "predicted_label"]]])

# Display first few rows of results
print(prediction_results.head())
```

```python
# Export prediction results to a CSV file
prediction_results.to_csv("model_predictions.csv", index=False, header=True)

# The CSV file will contain the image IDs and their corresponding predicted labels
```