2. Weighted Loss Function

October 17, 2024

1 Counting Labels & Weighted Loss Function

As you saw in the lecture videos, one way to avoid having class imbalance impact the loss function is to weight the losses differently. To choose the weights, you first need to calculate the class frequencies.

For this exercise, you'll just get the count of each label. Later on, you'll use the concepts practiced here to calculate frequencies in the assignment!

As before, calculate the two terms that make up the loss function. Notice that you are working with more than one class (represented by columns). In this case, there are two classes.

Start by calculating the loss for class `0`.

$$loss^{(i)} = loss_{pos}^{(i)} + los_{neg}^{(i)}$$

$$loss_{pos}^{(i)} = -1 \times weight_{pos}^{(i)} \times y^{(i)} \times log(\hat{y}^{(i)})$$

$$loss_{neg}^{(i)} = -1 \times weight_{neg}^{(i)} \times (1 - y^{(i)}) \times log(1 - \hat{y}^{(i)})$$

2 Compute class weights with scikit learn

[1]:
```
import os
import pytz
import random
import sklearn
import datetime
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from pprint import pprint
import matplotlib.pyplot as plt
from tensorflow.keras import backend as K
```

[2]:
```
data_dir  = "../../data/nih/"

try:
  from google.colab import drive
  drive.mount('/content/drive')
  !cp -r "/content/drive/MyDrive/machine learning/Coursera Medical XRays" /
↪content/

  data_dir  = '/content/Coursera Medical XRays/data/nih/'
  os.chdir('/content/drive/MyDrive/machine learning/Coursera Medical XRays/AI␣
↪for Medical Diagnosis/Week 1')
except:
  pass
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[3]:
```
!pwd
```

```
/content/drive/MyDrive/machine learning/Coursera Medical XRays/AI for Medical
Diagnosis/Week 1
```

[4]:
```
image_dir = data_dir + "images-small/"

train_csv = data_dir + "train-small.csv"
valid_csv = data_dir + "valid-small.csv"
test_csv  = data_dir + "test.csv"

target_w, target_h = 320,320
batch_size         = 64
```

2.1 Create text labels

[5]:
```
train_df = pd.read_csv(train_csv)
valid_df = pd.read_csv(valid_csv)
test_df  = pd.read_csv(test_csv)

labels_text = list(train_df.drop(['Image','PatientId'], axis=1).columns)
labels_text
```

[5]:
```
['Atelectasis',
 'Cardiomegaly',
 'Consolidation',
 'Edema',
 'Effusion',
 'Emphysema',
 'Fibrosis',
 'Hernia',
```

```
 'Infiltration',
 'Mass',
 'Nodule',
 'Pleural_Thickening',
 'Pneumonia',
 'Pneumothorax']
```

2.2 Create datasets in Numpy

[6]:
```python
for i in range(1):
    image = tf.keras.preprocessing.image.load_img(os.path.join(image_dir,␣
 ↪train_df['Image'].iloc[i]))
    image = tf.keras.preprocessing.image.img_to_array(image)

image.shape
```

[6]:
```
(1024, 1024, 3)
```

[7]:
```python
X_train = []

for i in range(len(train_df['Image'])):
    image = tf.keras.preprocessing.image.load_img(os.path.join(image_dir,␣
 ↪train_df['Image'].iloc[i]), target_size=(target_w, target_h))
    image = tf.keras.preprocessing.image.img_to_array(image)
    X_train.append(image)

X_train = np.array(X_train).astype(np.float32)

y_train = train_df[labels_text].values.astype(np.int32)  # Get the multi-label␣
 ↪output

# ----------------

X_valid = []
for i in range(len(valid_df['Image'])):
    image = tf.keras.preprocessing.image.load_img(os.path.join(image_dir,␣
 ↪valid_df['Image'].iloc[i]), target_size=(target_w, target_h))
    image = tf.keras.preprocessing.image.img_to_array(image)
    X_valid.append(image)

X_valid = np.array(X_valid).astype(np.float32)

y_valid = valid_df[labels_text].values.astype(np.int32)  # Get the multi-label␣
 ↪output

# ----------------
```

```
X_test = []

for i in range(len(test_df['Image'])):
    image = tf.keras.preprocessing.image.load_img(os.path.join(image_dir,␣
 ↪train_df['Image'].iloc[i]), target_size=(target_w, target_h))
    image = tf.keras.preprocessing.image.img_to_array(image)
    X_test.append(image)

X_test = np.array(X_test).astype(np.float32)

y_test = test_df[labels_text].values.astype(np.int32)  # Get the multi-label␣
 ↪output

print(f"{X_train.shape = } {X_valid.shape = } {X_test.shape = }")
print(f"{y_train.shape = }         {y_valid.shape = }          {y_test.shape =␣
 ↪}")
```

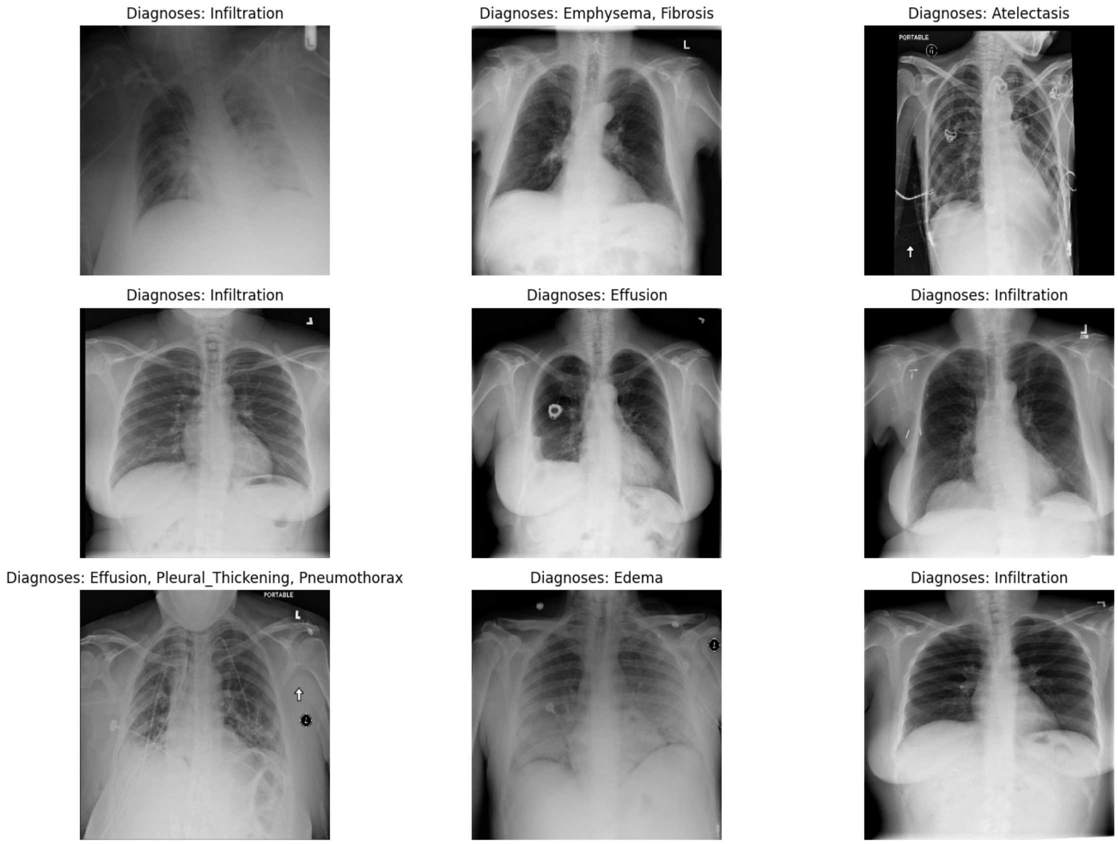
```
X_train.shape = (1000, 320, 320, 3) X_valid.shape = (109, 320, 320, 3)
X_test.shape = (420, 320, 320, 3)
y_train.shape = (1000, 14)          y_valid.shape = (109, 14)
y_test.shape = (420, 14)
```

3 Data Visualization

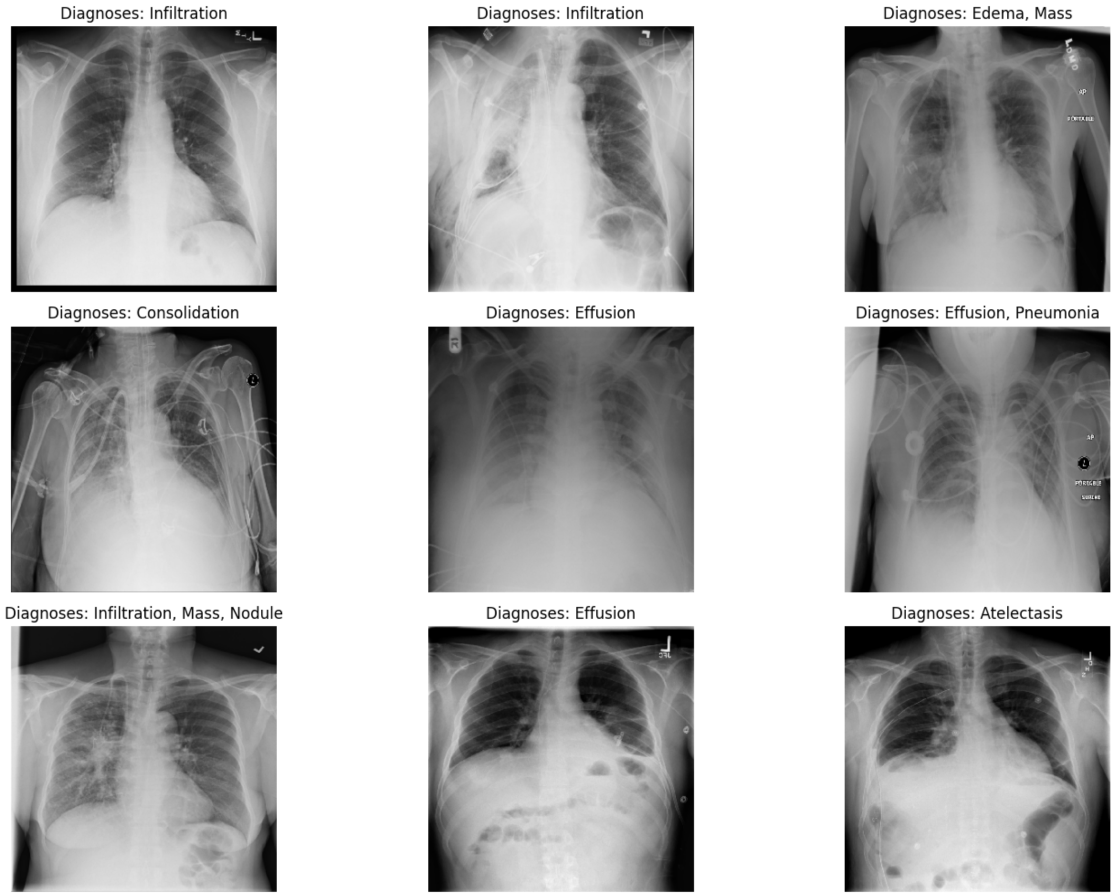
[8]:
```
def shuffled_list(n):
  '''Creates a shuffled list of integers from 0 to n-1'''
  lst = list(range(n))  # Create a list from 0 to n
  random.shuffle(lst)  # Shuffle the list in place
  return lst

def display_images_with_diseases(X_train, y_train, labels_text,␣
 ↪normalized=False):
    num_imgs = 0
    plt.figure(figsize=(15, 10))  # 15 units wide, 10 units tall

    for i in shuffled_list(X_train.shape[0]):
        if num_imgs >= 9:
            break

        # Find images with at least one disease (at least one '1' in the label)
        if np.sum(y_train[i]) > 0:
            num_imgs += 1

            # Scale image back to [0, 255] if it's normalized
            img = X_train[i] * 255.0 if normalized else X_train[i]

            # Convert to integer for display if necessary
```

```
            img = img.astype(np.uint8)

            # Get the diagnosis for this image
            diagnoses = [labels_text[j] for j in range(len(y_train[i])) if␣
 ↪y_train[i][j] == 1]
            title_text = f"Diagnoses: {', '.join(diagnoses)}"

            plt.subplot(3, 3, num_imgs)  # Change to num_imgs to keep it within␣
 ↪1-9
            plt.imshow(img, cmap='gray')  # Assuming grayscale X-ray images
            plt.title(title_text)
            plt.axis('off')

    plt.tight_layout()  # Adjust layout to prevent overlap
    plt.show()

# Example usage:
# For normalized images in the range [0, 1]
print(f"\n{np.min((X_train/255.0)) = }\n{np.max((X_train/255.0)) = }")
print(f"{X_train.shape = }")
display_images_with_diseases(X_train / 255.0, y_train, labels_text,␣
 ↪normalized=True)

# For images in the range [0, 255]
print(f"\n{np.min(X_train) = }\n{np.max(X_train) = }")
print(f"{X_train.shape = }")
display_images_with_diseases(X_train, y_train, labels_text, normalized=False)
```

```
np.min((X_train/255.0)) = 0.0
np.max((X_train/255.0)) = 1.0
X_train.shape = (1000, 320, 320, 3)
```




```
np.min(X_train) = 0.0
np.max(X_train) = 255.0
X_train.shape = (1000, 320, 320, 3)
```



4 1. Custom Class Weights per Label (Per Disease)

For each of the 14 diseases, you can calculate class weights independently. This means you treat each column (disease) in your label array separately and compute weights for the presence (1) and absence (0) of each disease.

[9]:
```python
# Initialize a dictionary to store weights for each disease (label)
class_weights = {}

# Get the number of diseases (classes, labels)
num_classes = y_train.shape[1]

# Assuming y_train is a (num_samples, num_classes) array where each column corresponds to a disease
for i in range(num_classes):
    # Compute class weights for each disease separately
    class_weights[i] = sklearn.utils.class_weight.compute_class_weight(
        'balanced',
```

```
        classes=np.unique(y_train[:, i]),  # Get the unique values for the i-th␣
↪disease (0 and 1)
        y=y_train[:, i]                    # Pass the i-th disease's labels
    )

# Convert to a dictionary format
class_weights_dict = {i: dict(enumerate(class_weights[i])) for i in␣
↪range(num_classes)}

pprint(class_weights_dict)
```

```
{0: {0: 0.5592841163310962, 1: 4.716981132075472},
 1: {0: 0.5102040816326531, 1: 25.0},
 2: {0: 0.5170630816959669, 1: 15.151515151515152},
 3: {0: 0.508130081300813, 1: 31.25},
 4: {0: 0.573394495412844, 1: 3.90625},
 5: {0: 0.5065856129685917, 1: 38.46153846153846},
 6: {0: 0.5070993914807302, 1: 35.714285714285715},
 7: {0: 0.501002004008016, 1: 250.0},
 8: {0: 0.6060606060606061, 1: 2.857142857142857},
 9: {0: 0.5235602094240838, 1: 11.11111111111111},
 10: {0: 0.5285412262156448, 1: 9.25925925925926},
 11: {0: 0.5107252298263534, 1: 23.80952380952381},
 12: {0: 0.5050505050505051, 1: 50.0},
 13: {0: 0.5197505197505198, 1: 13.157894736842104}}
```

[10]:
```
sample_weights = sklearn.utils.class_weight.compute_sample_weight('balanced',␣
↪y_train)

print(f"       {y_train.shape = }\n{sample_weights.shape =␣
↪}\n\n{sample_weights[:10] = }")
```

```
       y_train.shape = (1000, 14)
sample_weights.shape = (1000,)

sample_weights[:10] = array([1.24397710e-04, 3.36950770e-02, 5.79930276e-03,
1.24397710e-04,
       5.86446346e-04, 1.65105877e+00, 1.24397710e-04, 1.24397710e-04,
       1.24397710e-04, 1.24397710e-04])
```

5 2. Using a Custom Loss Function

Keras doesn't natively support class weights in multi-label classification problems (where each sample can have multiple diseases), but you can manually adjust the loss function to include class weights. Here's an approach where you can adjust the binary cross-entropy loss for each label:

[11]:
```
def weighted_binary_crossentropy(class_weights):
    # Custom binary cross-entropy loss with class weights
    def loss(y_true, y_pred):
        # For each label (disease), apply class weights
        epsilon = 1e-7
        loss    = 0
        for i in range(len(class_weights)):
            weight_for_0 = class_weights[i][0]
            weight_for_1 = class_weights[i][1]
            loss += -tf.reduce_mean(weight_for_0 * (1 - y_true[:, i]) * tf.math.
↪log(1 - y_pred[:, i] + epsilon) +
                                    weight_for_1 *      y_true[:, i]  * tf.math.
↪log(    y_pred[:, i] + epsilon))
        return loss / len(class_weights)  # Average the loss over all labels
    return loss
```

6 Building the Model with Functional API

[12]:
```
# Input layer for RGB images
inputs = tf.keras.layers.Input(shape=(None, None, 3))  # Input shape for␣
↪arbitrary image size

# Data augmentation block (without flipping)
x = tf.keras.layers.RandomRotation(0.01)(inputs)
x = tf.keras.layers.RandomZoom(height_factor=(-0.09, 0.09))(x)
x = tf.keras.layers.RandomTranslation(height_factor=0.09, width_factor=0.09)(x)
x = tf.keras.layers.RandomContrast(factor=0.09)(x)
x = tf.keras.layers.RandomBrightness(factor=0.09)(x)

# Resize the image and normalize pixel values
x = tf.keras.layers.Resizing(target_h, target_w)(x)

# # When using one of the DenseNet variations for transfer learning
# # Instead of:
x = tf.keras.layers.Rescaling(1.0/255.0)(x)  # Normalize pixel values to [0, 1]
# # Use:
# x = tf.keras.layers.Lambda(tf.keras.applications.densenet.preprocess_input)(x)

# First Convolutional Block
x = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', kernel_regularizer=tf.
↪keras.regularizers.l2(0.001))(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)

# Second Convolutional Block
x = tf.keras.layers.Conv2D(64, (3, 3), activation='relu')(x)
```

```
x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)

# Third Convolutional Block
x = tf.keras.layers.Conv2D(128, (3, 3), activation='relu')(x)
x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)

# Fourth Convolutional Block
x = tf.keras.layers.Conv2D(256, (3, 3), activation='relu')(x)
x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)

# Fifth Convolutional Block
x = tf.keras.layers.Conv2D(512, (3, 3), activation='relu')(x)
x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)

# Flatten/GlobalAveragePooling2D
# x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.GlobalAveragePooling2D()(x)

# Fully connected layers
x = tf.keras.layers.Dense(256, activation='relu')(x)
x = tf.keras.layers.Dropout(0.5)(x)
x = tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=tf.keras.
↪regularizers.l2(0.01))(x)
x = tf.keras.layers.Dropout(0.5)(x)
x = tf.keras.layers.Dense(64, activation='relu')(x)
x = tf.keras.layers.Dropout(0.5)(x)
x = tf.keras.layers.Dense(32, activation='relu')(x)
x = tf.keras.layers.Dropout(0.5)(x)

# Output layer: 14 units (one for each label), sigmoid activation for␣
↪multi-label classification
outputs = tf.keras.layers.Dense(14, activation='sigmoid')(x)

# Build the model using the Functional API
model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
```

- Precision: Important when you want to minimize false positives (e.g., you don't want to mis
- Recall: Critical when false negatives are unacceptable (e.g., you don't want to miss a dis
- F1 Score: Balanced metric when both precision and recall are important.
- AUC: Useful for evaluating the model's ability to distinguish between classes across diffe

[13]:
```
def f1_score(y_true, y_pred):
    # Reset states before calculating precision and recall
    precision_metric.reset_state()
    recall_metric.reset_state()

    # Update state based on true and predicted values
```

```
    precision_metric.update_state(y_true, y_pred)
    recall_metric.update_state(y_true, y_pred)

    precision = precision_metric.result()
    recall    = recall_metric.result()

    # Calculate F1 score
    return 2 * (precision * recall) / (precision + recall + K.epsilon())

# model.compile(
#     optimizer='adam',
#     loss=weighted_binary_crossentropy(class_weights_dict),
#     metrics=['AUC', 'Precision', 'Recall', f1_score]
# )

lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-5,
    decay_steps=10000,
    decay_rate=0.96,
    staircase=True)
optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule)

# # For multi-label classification with class imbalance, focal loss might␣
↪perform better than binary cross-entropy.
# # Focal loss reduces the relative loss for well-classified examples, focusing␣
↪more on hard-to-classify examples.
# def weighted_focal_loss(class_weights, gamma=2., alpha=0.25):
#     def focal_loss_fixed(y_true, y_pred):
#         epsilon = tf.keras.backend.epsilon()
#         y_pred = tf.clip_by_value(y_pred, epsilon, 1. - epsilon)

#         # Calculate focal loss for each label with individual class weights
#         loss = 0
#         for i in range(len(class_weights)):
#             weight_for_0 = class_weights[i][0]  # Class weight for 0 (no␣
↪disease)
#             weight_for_1 = class_weights[i][1]  # Class weight for 1 (disease␣
↪present)

#             # Compute cross entropy loss for both classes
#             cross_entropy_0 = weight_for_0 * (1 - y_true[:, i]) * tf.math.
↪log(1 - y_pred[:, i] + epsilon)
#             cross_entropy_1 = weight_for_1 * y_true[:, i] * tf.math.
↪log(y_pred[:, i] + epsilon)

#             # Combine the losses using focal loss formula
```

```
#             loss += - (alpha * tf.pow(1 - y_pred[:, i], gamma) *␣
↪cross_entropy_1 +
#                         (1 - alpha) * tf.pow(y_pred[:, i], gamma) *␣
↪cross_entropy_0)

#         return tf.reduce_mean(loss, axis=-1)
#     return focal_loss_fixed

# model.compile(optimizer=optimizer,
#                loss=weighted_focal_loss(class_weights_dict),   # Pass class␣
↪weights to focal loss
#                metrics=['AUC', 'Precision', 'Recall', f1_score])

# model.compile(optimizer=optimizer, loss='binary_crossentropy',␣
↪metrics=['AUC', 'Precision', 'Recall', f1_score])

# def combined_bce_dice_loss(y_true, y_pred):
#     bce = tf.keras.losses.BinaryCrossentropy()(y_true, y_pred)
#     dice = 1 - (2 * tf.reduce_sum(y_true * y_pred) + 1) / (tf.
↪reduce_sum(y_true + y_pred) + 1)
#     return bce + dice
# model.compile(optimizer=optimizer, loss=combined_bce_dice_loss,␣
↪metrics=['accuracy', 'precision', 'recall', 'AUC'])

def focal_loss(gamma=1.5, alpha=0.25):
    def focal_loss_fixed(y_true, y_pred):
        epsilon       = tf.keras.backend.epsilon()
        y_pred        = tf.clip_by_value(y_pred, epsilon, 1. - epsilon)
        cross_entropy = -y_true * tf.math.log(y_pred)
        loss          = alpha * tf.pow(1 - y_pred, gamma) * cross_entropy
        return tf.reduce_mean(loss, axis=-1)
    return focal_loss_fixed

# Define precision and recall metrics outside the function
precision_metric = tf.keras.metrics.Precision()
recall_metric    = tf.keras.metrics.Recall()

# Use focal loss to address class imbalance
model.compile(optimizer=optimizer,
              loss=focal_loss(gamma=2., alpha=0.25),
              metrics=['AUC', 'Precision', 'Recall', f1_score])  # Custom F1␣
↪function


# Print model summary
model.summary()
```

```
Model: "functional"

 Layer (type)                    Output Shape                            ␣
↪Param #

 input_layer (InputLayer)        (None, None, None, 3)                     ␣
↪   0

 random_rotation (RandomRotation)     (None, None, None, 3)                     ␣
↪   0

 random_zoom (RandomZoom)        (None, None, None, 3)                     ␣
↪   0

 random_translation              (None, None, None, 3)                     ␣
↪   0
 (RandomTranslation)                                                       ␣
↪

 random_contrast (RandomContrast)     (None, None, None, 3)                     ␣
↪   0

 random_brightness (RandomBrightness)  (None, None, None, 3)                     ␣
↪   0

 resizing (Resizing)             (None, 320, 320, 3)                       ␣
↪   0

 rescaling (Rescaling)           (None, 320, 320, 3)                       ␣
↪   0

 conv2d (Conv2D)                 (None, 318, 318, 32)                      ␣
↪896

 batch_normalization             (None, 318, 318, 32)                      ␣
↪128
 (BatchNormalization)                                                      ␣
↪

 max_pooling2d (MaxPooling2D)    (None, 159, 159, 32)                      ␣
↪   0

 conv2d_1 (Conv2D)               (None, 157, 157, 64)                    ␣
↪18,496
```

```
max_pooling2d_1 (MaxPooling2D)          (None, 78, 78, 64)
↪   0

conv2d_2 (Conv2D)                       (None, 76, 76, 128)
↪73,856

max_pooling2d_2 (MaxPooling2D)          (None, 38, 38, 128)
↪   0

conv2d_3 (Conv2D)                       (None, 36, 36, 256)
↪295,168

max_pooling2d_3 (MaxPooling2D)          (None, 18, 18, 256)
↪   0

conv2d_4 (Conv2D)                       (None, 16, 16, 512)
↪1,180,160

max_pooling2d_4 (MaxPooling2D)          (None, 8, 8, 512)
↪   0

global_average_pooling2d                (None, 512)
↪   0
(GlobalAveragePooling2D)
↪

dense (Dense)                           (None, 256)
↪131,328

dropout (Dropout)                       (None, 256)
↪   0

dense_1 (Dense)                         (None, 128)
↪32,896

dropout_1 (Dropout)                     (None, 128)
↪   0

dense_2 (Dense)                         (None, 64)
↪8,256

dropout_2 (Dropout)                     (None, 64)
↪   0

dense_3 (Dense)                         (None, 32)
↪2,080
```

```
 dropout_3 (Dropout)                  (None, 32)                                 ␣
↪  0

 dense_4 (Dense)                      (None, 14)                                 ␣
↪462
```

Total params: 1,743,726 (6.65 MB)

Trainable params: 1,743,662 (6.65 MB)

Non-trainable params: 64 (256.00 B)

7 Convert to tensors

[14]:
```
train_tf = tf.data.Dataset.from_tensor_slices((X_train, y_train))
valid_tf = tf.data.Dataset.from_tensor_slices((X_valid, y_valid))
test_tf  = tf.data.Dataset.from_tensor_slices((X_test, y_test))

# Cache the datasets (to avoid redundant operations in future epochs)
train_tf = train_tf.cache()
valid_tf = valid_tf.cache()
test_tf  = test_tf.cache()

# Shuffle the train dataset
train_tf = train_tf.shuffle(buffer_size=int(len(X_train)/3))  # Shuffle only␣
 ↪the training data

# from tensorflow_utils import combine_features
# # Map the combine_features function
# train_tf = train_tf.map(lambda x, y: (combine_features(x), y),␣
 ↪num_parallel_calls=tf.data.AUTOTUNE)
# valid_tf = valid_tf.map(lambda x, y: (combine_features(x), y),␣
 ↪num_parallel_calls=tf.data.AUTOTUNE)
# test_tf  = test_tf.map(lambda x, y:  (combine_features(x), y),␣
 ↪num_parallel_calls=tf.data.AUTOTUNE)

# Batch, prefetch for efficient processing
train_tf = train_tf.batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)
valid_tf = valid_tf.batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)
test_tf  = test_tf.batch(batch_size).prefetch(buffer_size=tf.data.AUTOTUNE)
```

[15]:
```
# Verify the shapes of the features and labels in the combined dataset
print("\nFor train_dataset")
for feature_batch, label_batch in train_tf.take(1):
    print(f"Combined Feature Batch Shape (batch size, width, height, layers):␣
 ↪{feature_batch.shape}")
    print(f"Label Batch Shape            (batch size, number of classes):    ␣
 ↪{label_batch.shape}")
```

```
For train_dataset
Combined Feature Batch Shape (batch size, width, height, layers): (64, 320, 320,
3)
Label Batch Shape            (batch size, number of classes):     (64, 14)
```

8 Fit the model

[16]:
```
%%time

from sklearn.utils.class_weight import compute_sample_weight

early_stopping = tf.keras.callbacks.EarlyStopping(
    verbose=1,
    monitor='val_loss',
    patience=15,
    mode='min',
    restore_best_weights=True)

from tensorflow_utils import PrintEveryNEpoch

# When fitting the model, use verbose=0 and include the custom callback
periodic_messages = PrintEveryNEpoch(n=150, timezone_str='America/New_York')

# ==========================
# Assuming train_tf is a tf.data.Dataset object and y_train is available
# Calculate sample weights for each sample based on class imbalance
sample_weights = compute_sample_weight('balanced', y_train)

# Convert sample_weights to a Tensor
sample_weights_tensor = tf.constant(sample_weights, dtype=tf.float32)

# Function to add sample weights to the dataset
def add_sample_weights(features, labels, index):
    weight = tf.gather(sample_weights_tensor, index)
    return features, labels, weight

# Apply the function to add sample weights to the dataset
```

```
index_ds              = tf.data.Dataset.range(len(y_train))  # Create an index␣
↪dataset
def map_function(data, idx):
    return add_sample_weights(data[0], data[1], idx)

train_tf_with_weights = tf.data.Dataset.zip((train_tf, index_ds)).
↪map(map_function)
# train_tf_with_weights = tf.data.Dataset.zip((train_tf, index_ds)).map(
#     lambda data, idx: add_sample_weights(data[0], data[1], idx)
# )
# ==========================

# Train the model
history = model.fit(train_tf_with_weights,
                    validation_data=valid_tf,
                    epochs=10000, batch_size=batch_size, verbose=0,
                    callbacks=[early_stopping, periodic_messages])
```

```
WARNING:tensorflow:AutoGraph could not transform <function map_function at
0x7fe1a2ebadd0> and will run it as-is.
Cause: Unable to locate the source code of <function map_function at
0x7fe1a2ebadd0>. Note that functions defined in certain environments, like the
interactive Python shell, do not expose their source code. If that is the case,
you should define them in a .py source file. If you are certain the code is
graph-compatible, wrap the call using @tf.autograph.experimental.do_not_convert.
Original error: could not get source code
To silence this warning, decorate the function with
@tf.autograph.experimental.do_not_convert
```

```
WARNING: AutoGraph could not transform <function map_function at 0x7fe1a2ebadd0>
and will run it as-is.
Cause: Unable to locate the source code of <function map_function at
0x7fe1a2ebadd0>. Note that functions defined in certain environments, like the
interactive Python shell, do not expose their source code. If that is the case,
you should define them in a .py source file. If you are certain the code is
graph-compatible, wrap the call using @tf.autograph.experimental.do_not_convert.
Original error: could not get source code
To silence this warning, decorate the function with
@tf.autograph.experimental.do_not_convert

Epoch 150 (2024-10-17 09:13:56):
AUC = 0.5613, Precision = 0.0564, Recall = 0.7259, f1_score = 0.1040, loss =
0.9496, val_AUC = 0.6761, val_Precision = 0.0634, val_Recall = 0.9383,
val_f1_score = 0.1196, val_loss = 0.9487

Epoch 300 (2024-10-17 09:18:16):
AUC = 0.5570, Precision = 0.0553, Recall = 0.7526, f1_score = 0.1021, loss =
0.4756, val_AUC = 0.6941, val_Precision = 0.0634, val_Recall = 0.9383,
```

```
val_f1_score = 0.1196, val_loss = 0.4755

Epoch 450 (2024-10-17 09:22:36):
AUC = 0.5721, Precision = 0.0568, Recall = 0.8104, f1_score = 0.1066, loss =
0.1999, val_AUC = 0.6897, val_Precision = 0.0634, val_Recall = 0.9383,
val_f1_score = 0.1196, val_loss = 0.2001

Epoch 600 (2024-10-17 09:26:56):
AUC = 0.5967, Precision = 0.0570, Recall = 0.8326, f1_score = 0.1067, loss =
0.0624, val_AUC = 0.7056, val_Precision = 0.0634, val_Recall = 0.9383,
val_f1_score = 0.1196, val_loss = 0.0629

Epoch 750 (2024-10-17 09:31:17):
AUC = 0.5947, Precision = 0.0562, Recall = 0.8474, f1_score = 0.1059, loss =
0.0118, val_AUC = 0.7134, val_Precision = 0.0634, val_Recall = 0.9383,
val_f1_score = 0.1196, val_loss = 0.0124

Epoch 900 (2024-10-17 09:35:36):
AUC = 0.6268, Precision = 0.0575, Recall = 0.8948, f1_score = 0.1084, loss =
0.0008, val_AUC = 0.7155, val_Precision = 0.0634, val_Recall = 0.9383,
val_f1_score = 0.1196, val_loss = 0.0013
Epoch 1000: early stopping
Restoring model weights from the end of the best epoch: 985.
CPU times: user 32min 31s, sys: 33.4 s, total: 33min 4s
Wall time: 29min 9s
```

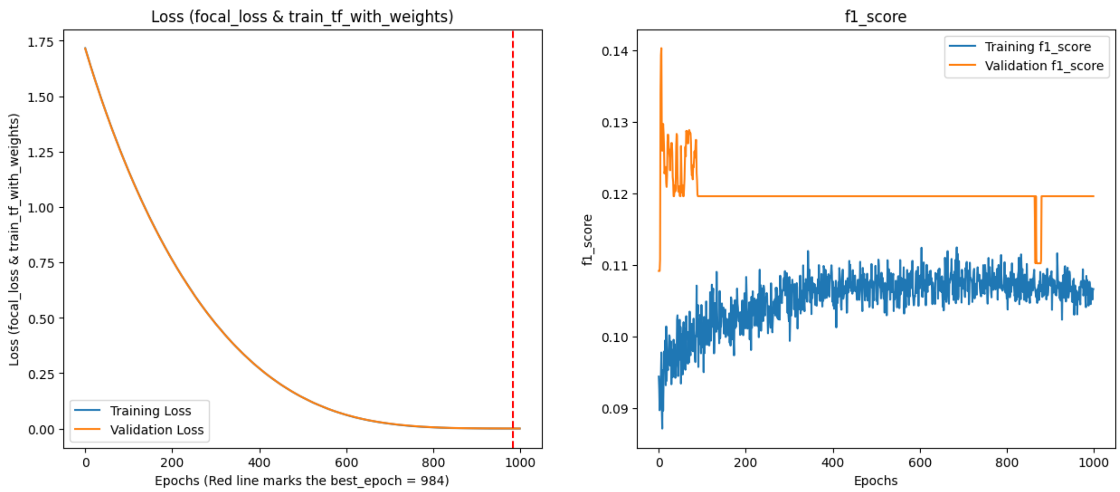
9 Print Loss Curves

[17]:
```python
history.history.keys()
```

[17]:
```
dict_keys(['AUC', 'Precision', 'Recall', 'f1_score', 'loss', 'val_AUC',
'val_Precision', 'val_Recall', 'val_f1_score', 'val_loss'])
```

[18]:
```python
losses_list=['loss', 'focal_loss & train_tf_with_weights', 'f1_score']
# losses_list=['loss', 'dice', 'accuracy']
# losses_list=['loss', 'binary_crossentropy', 'accuracy']
# losses_list=['loss', 'focal loss', 'accuracy']
# losses_list=['loss', 'weighted_binary_crossentropy', 'accuracy']
```

[19]:
```python
from tensorflow_utils import loss_curves

loss_curves(history, start_after=0, losses=losses_list,␣
↪best_epoch=early_stopping.best_epoch)
```



10 Test set evaluation

[20]:
```
model_eval = model.evaluate(test_tf)
```

```
7/7              1s 170ms/step - AUC:
0.4455 - Precision: 0.1086 - Recall: 0.6498 - f1_score: 0.1870 - loss: 0.0037
```

[21]:
```
# y_pred = model.predict(test_tf)

# Adjust threshold post-training
y_pred = (model.predict(test_tf) > 0.3).astype(int)  # Set a higher threshold␣
↪than 0.5

y_true = []
for _, labels in test_tf:
    y_true.extend(labels.numpy())  # Append the binary labels to the y_true list

# Convert y_true to a numpy array to match the shape of y_pred
y_true = np.array(y_true)
```

```
7/7              0s 49ms/step
```

[22]:
```
y_pred_proba = model.predict(test_tf)
print(y_pred_proba[:5])  # Check the predicted probabilities
```

```
7/7              0s 28ms/step
[[0.87212694 0.6125994  0.66961634 0.70221823 0.79197    0.73665595
  0.3875863  0.47672176 0.8042248  0.8426775  0.7465116  0.671003
  0.41211006 0.7036664 ]
 [0.84072495 0.5973738  0.64871293 0.6777471  0.76108044 0.709282
  0.4040965  0.48063526 0.7730658  0.8107394  0.71933556 0.64988375
```

```
 0.42526615 0.6790275 ]
[0.82990646 0.5925506  0.64183486 0.67018515 0.7508594  0.7005422
 0.40940392 0.4819826  0.76289123 0.800019   0.71084696 0.64322853
 0.42938554 0.6715202 ]
[0.82006955 0.58853    0.63638085 0.66311646 0.74208105 0.6928324
 0.41364554 0.48295423 0.75365    0.7902024  0.7029314  0.6375453
 0.4328177  0.6643961 ]
[0.8369714  0.5956036  0.6460886  0.6751815  0.75747114 0.70626694
 0.40604448 0.4812615  0.76956797 0.8070673  0.7164531  0.6475235
 0.42678648 0.6764902 ]]
```

[23]:

```python
from sklearn.metrics import precision_score, recall_score, f1_score,␣
 ↪roc_auc_score, classification_report

# Compute precision, recall, and F1 for multi-label classification
precision = precision_score(y_true, y_pred, average='macro')  # or 'micro'
recall = recall_score(y_true, y_pred, average='macro')
f1 = f1_score(y_true, y_pred, average='macro')

# Compute AUC for multi-label classification
auc = roc_auc_score(y_true, y_pred, average='macro')

# Print results
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
print(f"AUC: {auc}")

# Generate detailed classification report
report = classification_report(y_true, y_pred)
print(report)
```

```
Precision: 0.13078231292517004
Recall: 1.0
F1 Score: 0.2311893277949307
AUC: 0.5
              precision    recall  f1-score   support

           0       0.14      1.00      0.25        60
           1       0.12      1.00      0.21        50
           2       0.13      1.00      0.22        53
           3       0.12      1.00      0.21        50
           4       0.13      1.00      0.22        53
           5       0.13      1.00      0.24        56
           6       0.15      1.00      0.25        61
           7       0.12      1.00      0.21        50
           8       0.14      1.00      0.25        59
           9       0.14      1.00      0.25        60
```

```
          10       0.13      1.00      0.23        54
          11       0.14      1.00      0.24        58
          12       0.12      1.00      0.21        50
          13       0.13      1.00      0.23        55

   micro avg       0.13      1.00      0.23       769
   macro avg       0.13      1.00      0.23       769
weighted avg       0.13      1.00      0.23       769
 samples avg       0.13      1.00      0.22       769
```

[24]:

```
from sklearn.metrics import multilabel_confusion_matrix

# Calculate confusion matrix for each label
conf_matrix = multilabel_confusion_matrix(y_true, y_pred)

# Print the confusion matrix for each label
print("Confusion Matrix for each label:")
for i, label in enumerate(labels_text):
    print(f"Confusion matrix for {label}:")
    print(conf_matrix[i])
    print("\n")
```

```
Confusion Matrix for each label:
Confusion matrix for Atelectasis:
[[  0 360]
 [  0  60]]



Confusion matrix for Cardiomegaly:
[[  0 370]
 [  0  50]]



Confusion matrix for Consolidation:
[[  0 367]
 [  0  53]]



Confusion matrix for Edema:
[[  0 370]
 [  0  50]]



Confusion matrix for Effusion:
[[  0 367]
 [  0  53]]
```

```
Confusion matrix for Emphysema:
[[  0 364]
 [  0  56]]


Confusion matrix for Fibrosis:
[[  0 359]
 [  0  61]]


Confusion matrix for Hernia:
[[  0 370]
 [  0  50]]


Confusion matrix for Infiltration:
[[  0 361]
 [  0  59]]


Confusion matrix for Mass:
[[  0 360]
 [  0  60]]


Confusion matrix for Nodule:
[[  0 366]
 [  0  54]]


Confusion matrix for Pleural_Thickening:
[[  0 362]
 [  0  58]]


Confusion matrix for Pneumonia:
[[  0 370]
 [  0  50]]


Confusion matrix for Pneumothorax:
[[  0 365]
 [  0  55]]
```

[24]:

```

```