Brief description of the problem and data (5 pts)

- **Problem:** This is a binary image classification problem. We need to detect presence of metastases from 96*96px hispopathology images.
- **Data:** We have 220k training images and 57k validation images/. Data is subset of PCAM dataset, they removed duplicate images. Images are 96*96px and we are trying to predict a positive label indicates that the center 32x32px region of a patch contains at least one pixel of tumor tissue. Tumor tissue in the outer region of the patch does not influence the label.

Exploratory Data Analysis (EDA)

Data Wrangling ¶

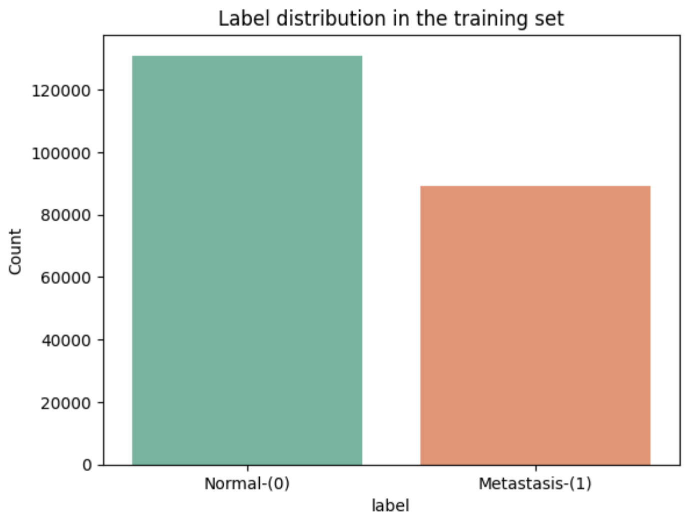
+ Code + Markdown

[2]:
```
print(f'{len(os.listdir("/kaggle/input/histopathologic-cancer-detection/train"))} pictures in train.')
print(f'{len(os.listdir("/kaggle/input/histopathologic-cancer-detection/test"))} pictures in test.')
```

```
220025 pictures in train.
57458 pictures in test.
```

Label Balance

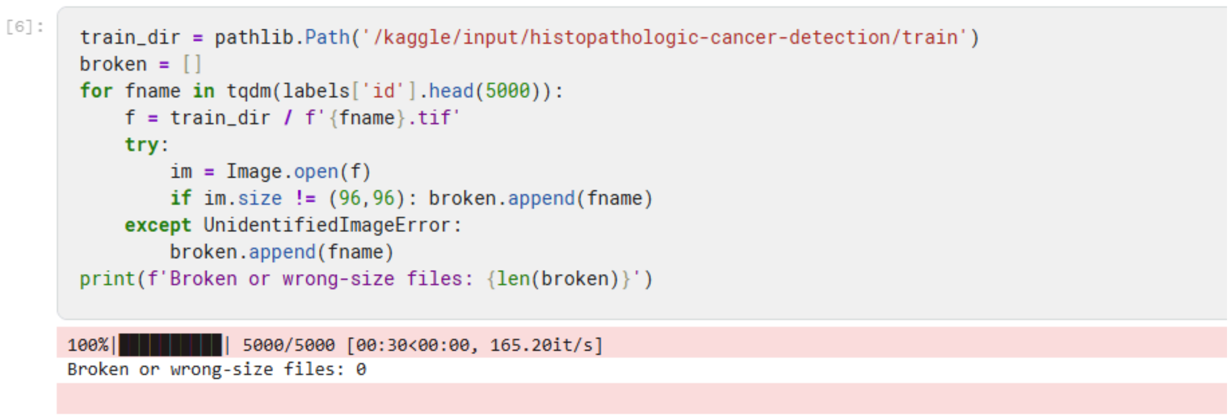
```
sns.countplot(x='label', data=labels, palette='Set2')
plt.title('Label distribution in the training set')
plt.xticks([0,1], ['Normal-(0)', 'Metastasis-(1)'])
plt.ylabel('Count'); plt.show()

print(labels['label'].value_counts(normalize=True).rename('proportion'))
```



```
label
0    0.594969
1    0.405031
Name: proportion, dtype: float64
```

- **Comments:** Almost 60% percent of our dataset is normal, and 40% of them has Metastasis(cancer). Data is slightly imbalanced.

Checking Image quality and pixel Size

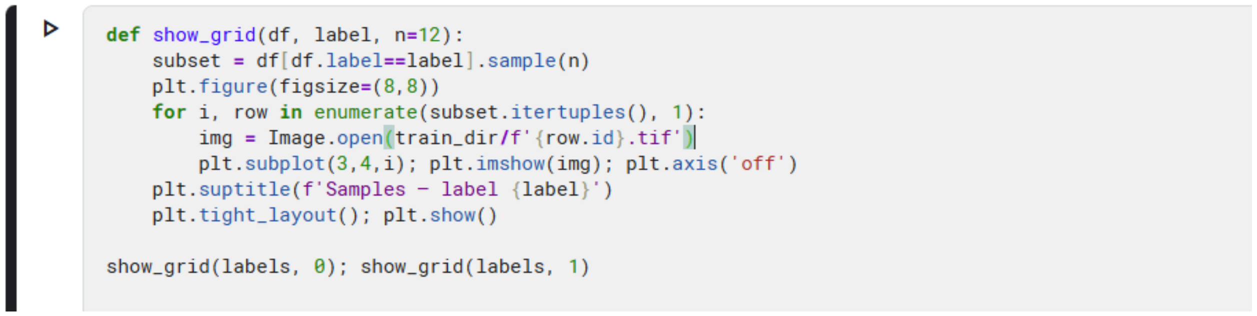


```
[6]:
train_dir = pathlib.Path('/kaggle/input/histopathologic-cancer-detection/train')
broken = []
for fname in tqdm(labels['id'].head(5000)):
    f = train_dir / f'{fname}.tif'
    try:
        im = Image.open(f)
        if im.size != (96,96): broken.append(fname)
    except UnidentifiedImageError:
        broken.append(fname)
print(f'Broken or wrong-size files: {len(broken)}')
```

```
100%|██████████| 5000/5000 [00:30<00:00, 165.20it/s]
Broken or wrong-size files: 0
```

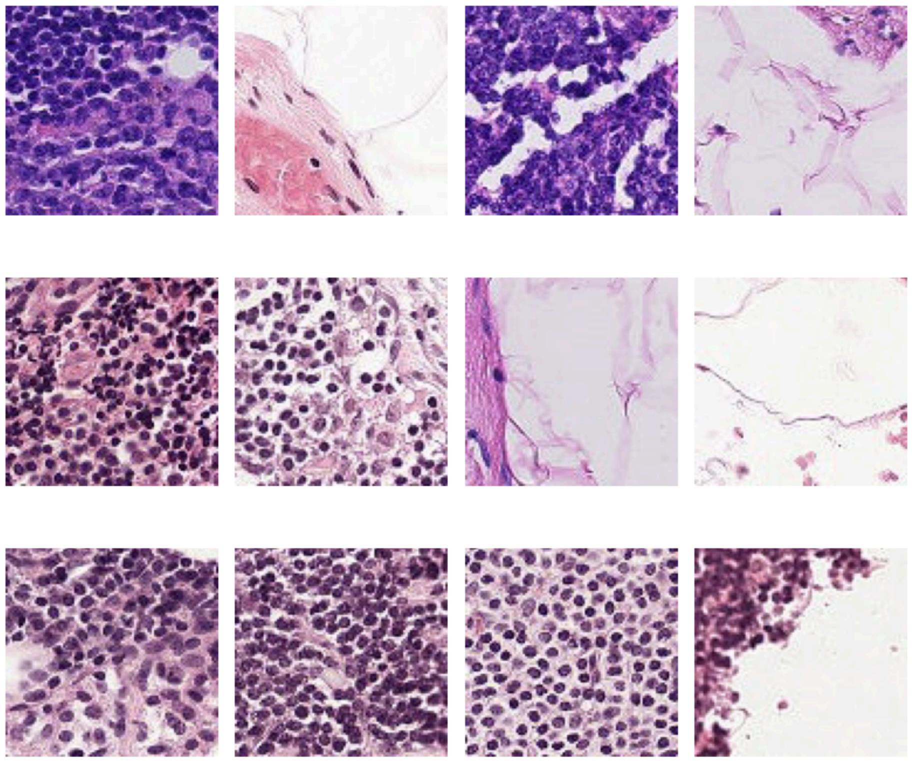
- **Comments:** Here I checked pixel size of the portion of the train dataset, and it passed the test, data is clean sizes are correct.

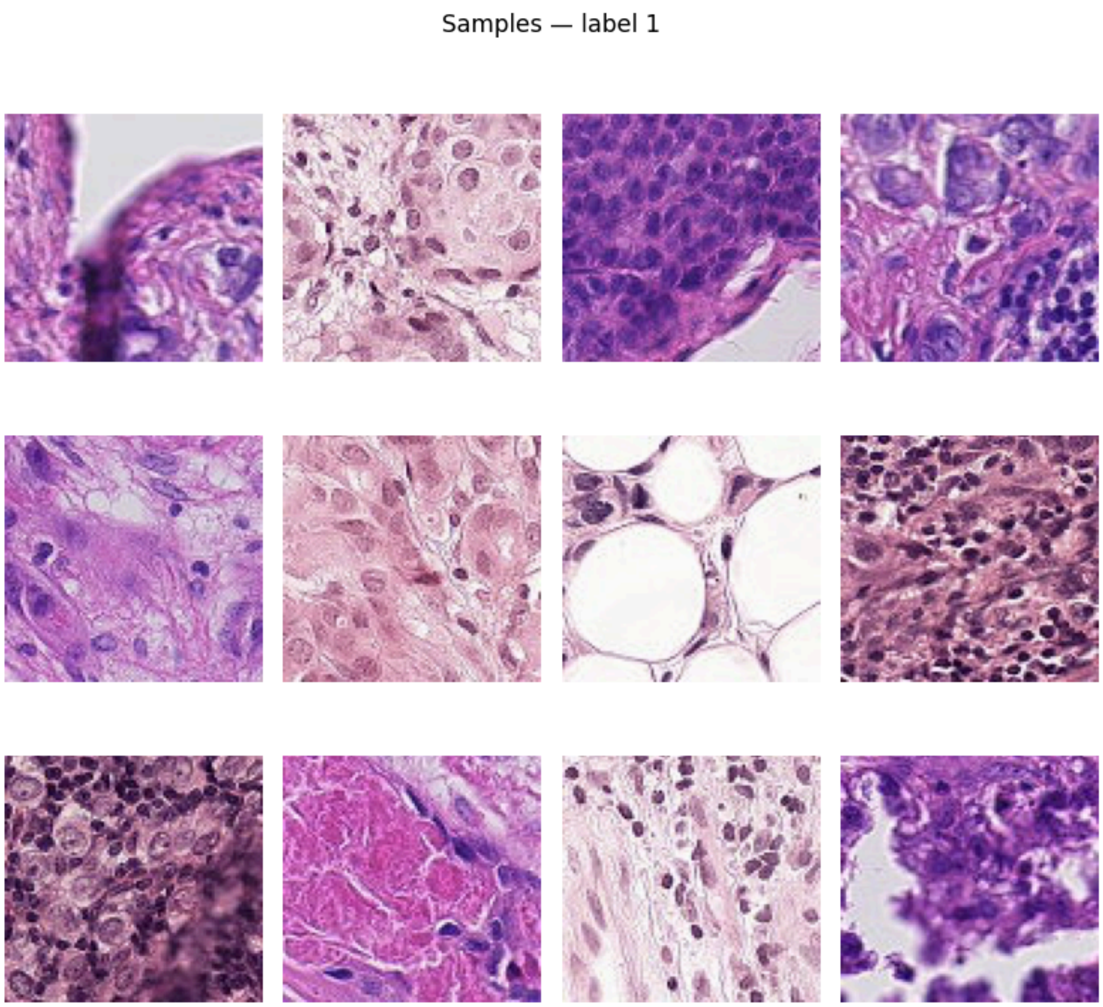
Creating Visual and Observing Image Samples



```
def show_grid(df, label, n=12):
    subset = df[df.label==label].sample(n)
    plt.figure(figsize=(8,8))
    for i, row in enumerate(subset.itertuples(), 1):
        img = Image.open(train_dir/f'{row.id}.tif')
        plt.subplot(3,4,i); plt.imshow(img); plt.axis('off')
    plt.suptitle(f'Samples - label {label}')
    plt.tight_layout(); plt.show()

show_grid(labels, 0); show_grid(labels, 1)
```

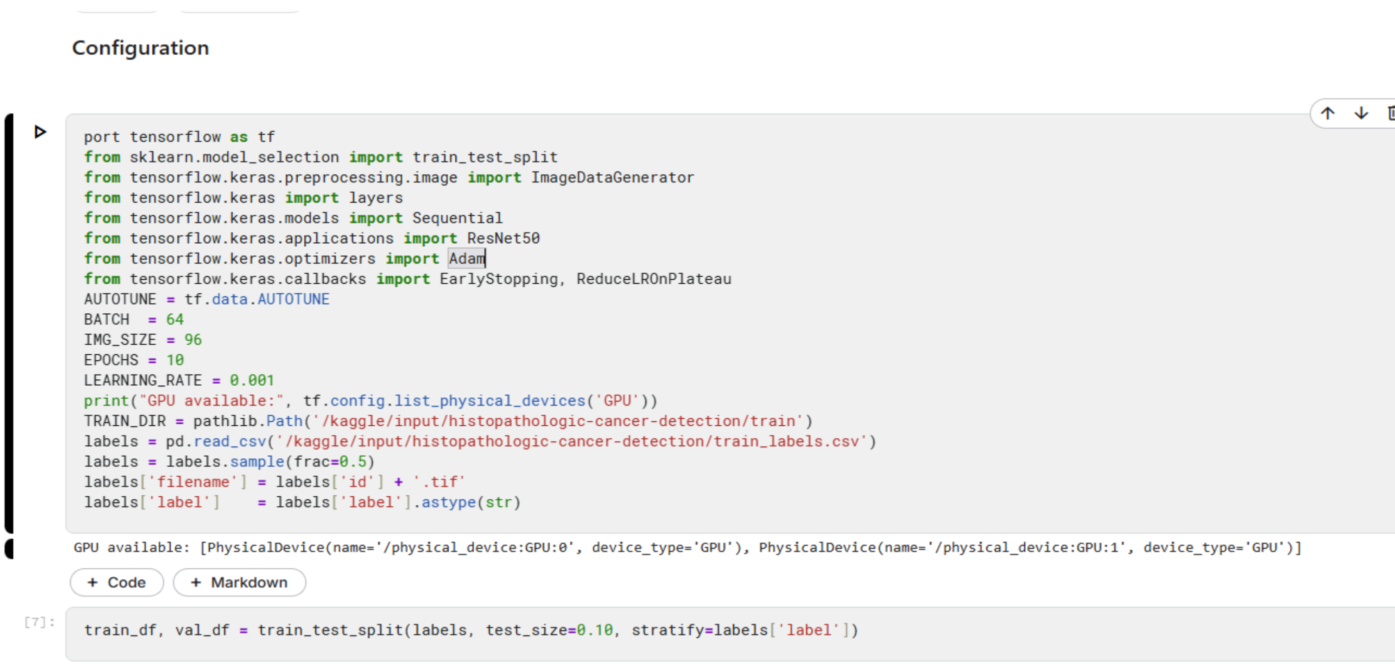
Samples — label 0





Comments: I couldn’t identify the cancer with bare eyes; it was supposed to be in the center of image.

DModel Architecture (25 pts)



Configuration

```
port tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
AUTOTUNE = tf.data.AUTOTUNE
BATCH  = 64
IMG_SIZE = 96
EPOCHS = 10
LEARNING_RATE = 0.001
print("GPU available:", tf.config.list_physical_devices('GPU'))
TRAIN_DIR = pathlib.Path('/kaggle/input/histopathologic-cancer-detection/train')
labels = pd.read_csv('/kaggle/input/histopathologic-cancer-detection/train_labels.csv')
labels = labels.sample(frac=0.5)
labels['filename'] = labels['id'] + '.tif'
labels['label']    = labels['label'].astype(str)
```

```
GPU available: [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU'), PhysicalDevice(name='/physical_device:GPU:1', device_type='GPU')]
```

[7]:

```
train_df, val_df = train_test_split(labels, test_size=0.10, stratify=labels['label'])
```

DATA PREPROCESSING

[8]:
```
# We have enough data I am not going to increase the sizes.
train_datagen = ImageDataGenerator(
    rescale=1/255.,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
)
val_datagen = ImageDataGenerator(rescale=1/255.)
```

```
train_gen = train_datagen.flow_from_dataframe(
    dataframe=train_df,
    directory=str(TRAIN_DIR),
    x_col='filename',
    y_col='label',
    classes=['0','1'],
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH,
    class_mode='binary',
    shuffle=True
)

val_gen = val_datagen.flow_from_dataframe(
    dataframe=val_df,
    directory=str(TRAIN_DIR),
    x_col='filename',
    y_col='label',
    classes=['0','1'],
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH,
    class_mode='binary',
    shuffle=False
)


print('train batches per epoch :', len(train_gen))
print('validation batches :', len(val_gen))
```

```
Found 99010 validated image filenames belonging to 2 classes.
Found 11002 validated image filenames belonging to 2 classes.
train batches per epoch : 1548
validation batches : 172
```

MODEL ARCHITECTURES

Simple CNN

```
def create_simple_cnn():

    model = tf.keras.Sequential([
        layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),
        layers.Conv2D(32, 3, activation='relu', padding='same'),
        layers.MaxPool2D(),
        layers.Conv2D(64, 3, activation='relu', padding='same'),
        layers.MaxPool2D(),
        layers.Conv2D(128, 3, activation='relu', padding='same'),
        layers.GlobalAveragePooling2D(),
        layers.Dense(1, activation='sigmoid')
    ])
    return model
```

Enhanced CNN Model

```
def create_enhanced_cnn():

    model = Sequential([
        layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3)),


        layers.Conv2D(32, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Conv2D(32, 3, activation='relu', padding='same'),
        layers.MaxPool2D(),
        layers.Dropout(0.25),


        layers.Conv2D(64, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.Conv2D(64, 3, activation='relu', padding='same'),
        layers.MaxPool2D(),
        layers.Dropout(0.25),


        layers.Conv2D(128, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.GlobalAveragePooling2D(),
        layers.Dropout(0.5),
        layers.Dense(1, activation='sigmoid')
    ])
    return model
```

Transfer Learning Model

```
def create_resnet_model():

    base = ResNet50(
        weights='imagenet',
        include_top=False,
        input_shape=(IMG_SIZE, IMG_SIZE, 3)
    )
    base.trainable = False

    model = Sequential([
        base,
        layers.GlobalAveragePooling2D(),
        layers.Dropout(0.3),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.3),
        layers.Dense(1, activation='sigmoid')
    ])
    return model
```

Model Architecture Design and Reasoning

Problem Analysis

For histopathologic cancer detection, I need to classify 96x96 pixel tissue images as cancer or non-cancer. This is a binary classification problem.

Architecture Selection Strategy

I chose to compare three different approaches to understand which works best for medical imaging:

1. **Simple CNN** - Basic baseline to test if the problem is learnable
2. **Enhanced CNN** - Improved version with modern techniques
3. **ResNet50 Transfer Learning** - Pre-trained model to leverage existing knowledge

---

Architecture 1: Simple CNN (Baseline)

Design Reasoning:

- Started with a basic 3-layer CNN to establish baseline performance
- Used progressive layers (32→64→128) to capture features from simple to complex
- Included MaxPooling for spatial reduction and computational efficiency
- GlobalAveragePooling instead of flatten to reduce overfitting risk

Why This Architecture:

I chose a simple and efficient baseline CNN model as a starting point to test whether basic architectures can learn cancer patterns and serve as a foundation for future improvements.

Architecture 2: Enhanced CNN (Improved Design)

Design Reasoning:

- Added BatchNormalization for training stability and faster convergence
- Included Dropout (0.25 and 0.5) to prevent overfitting on medical data
- increasing dropout rates: lower for conv layers, higher for dense layers

Why These Improvements:

I included BatchNorm, Dropout, and double convolution layers to handle variability in medical images, prevent overfitting to specific tissue samples, and better capture complex cellular patterns for fine-grained medical analysis.

Expected Performance:

It should outperform simple CNN due to better regularization and feature extraction capabilities.

Architecture 3: ResNet50 Transfer Learning

Design Reasoning:

- Used pre-trained ResNet50 from ImageNet as feature extractor
- Froze base weights to prevent overfitting with limited medical data
- Added custom classification head with dropout for medical domain
- Leveraged 50-layer depth for complex pattern recognition

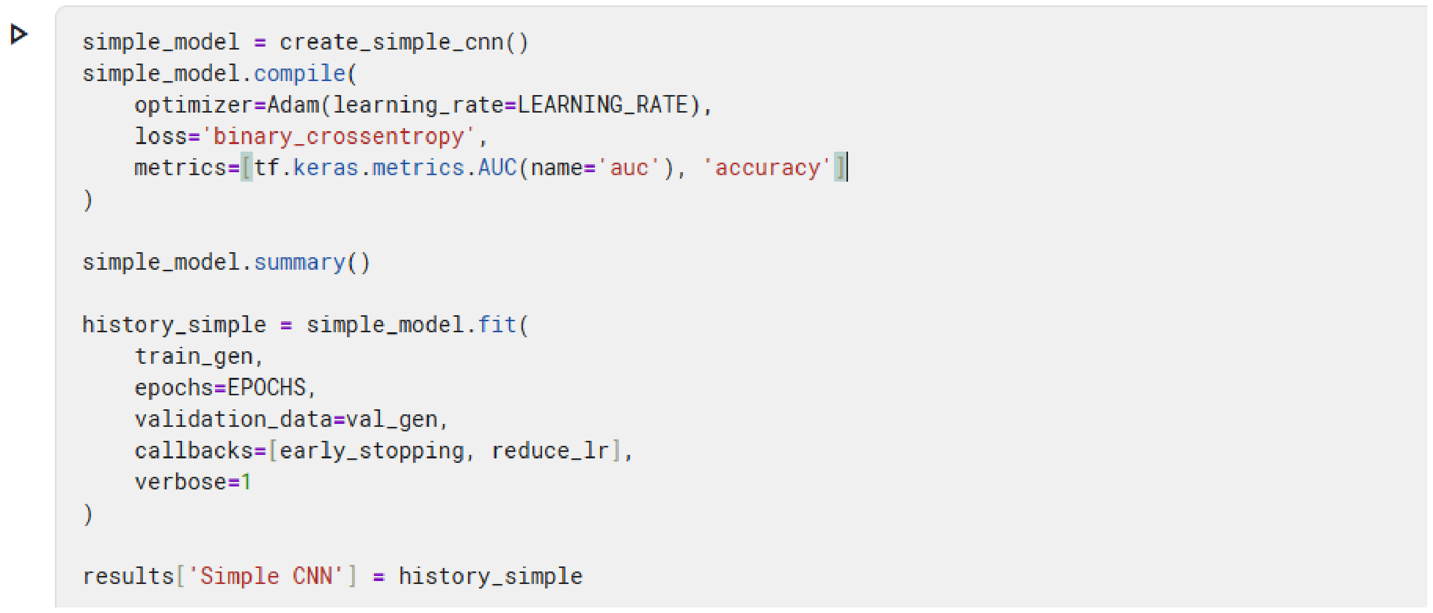
Why Transfer Learning:

I used transfer learning with ResNet50 because its pre-trained features offer rich visual representations, it's a proven architecture for image tasks, it's more efficient than training from scratch, and its 50 layers can model complex tissue structures.

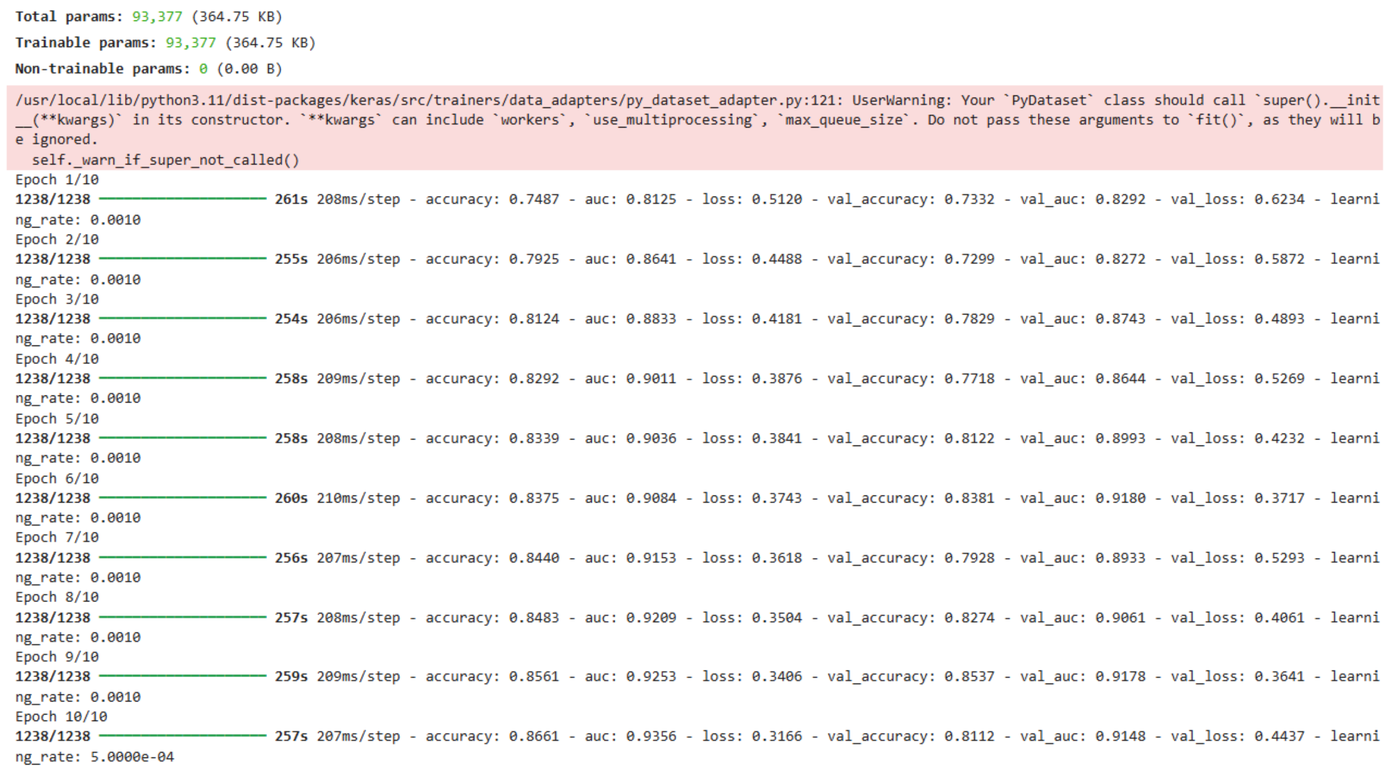
Results and Analysis (35 pts)

MODEL TRAINING AND COMPARISON

TRAINING SIMPLE CNN

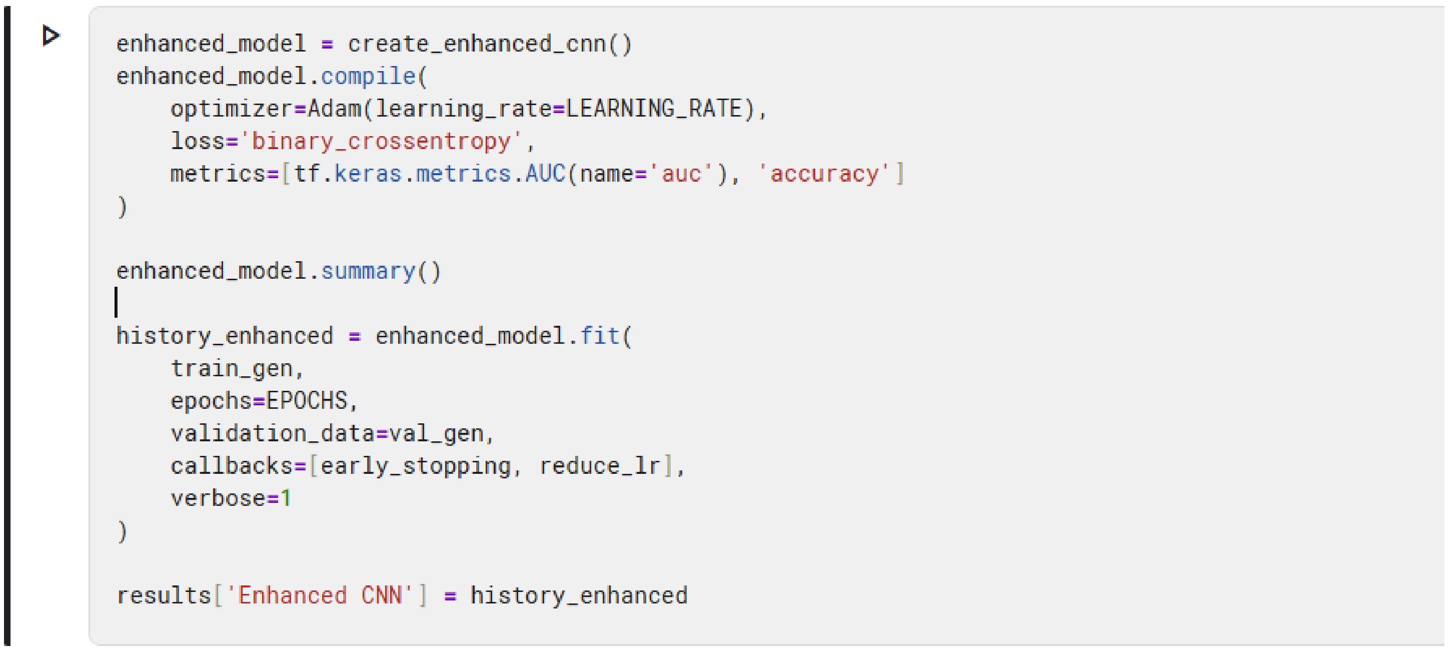


```python
simple_model = create_simple_cnn()
simple_model.compile(
    optimizer=Adam(learning_rate=LEARNING_RATE),
    loss='binary_crossentropy',
    metrics=[tf.keras.metrics.AUC(name='auc'), 'accuracy']
)

simple_model.summary()

history_simple = simple_model.fit(
    train_gen,
    epochs=EPOCHS,
    validation_data=val_gen,
    callbacks=[early_stopping, reduce_lr],
    verbose=1
)

results['Simple CNN'] = history_simple
```

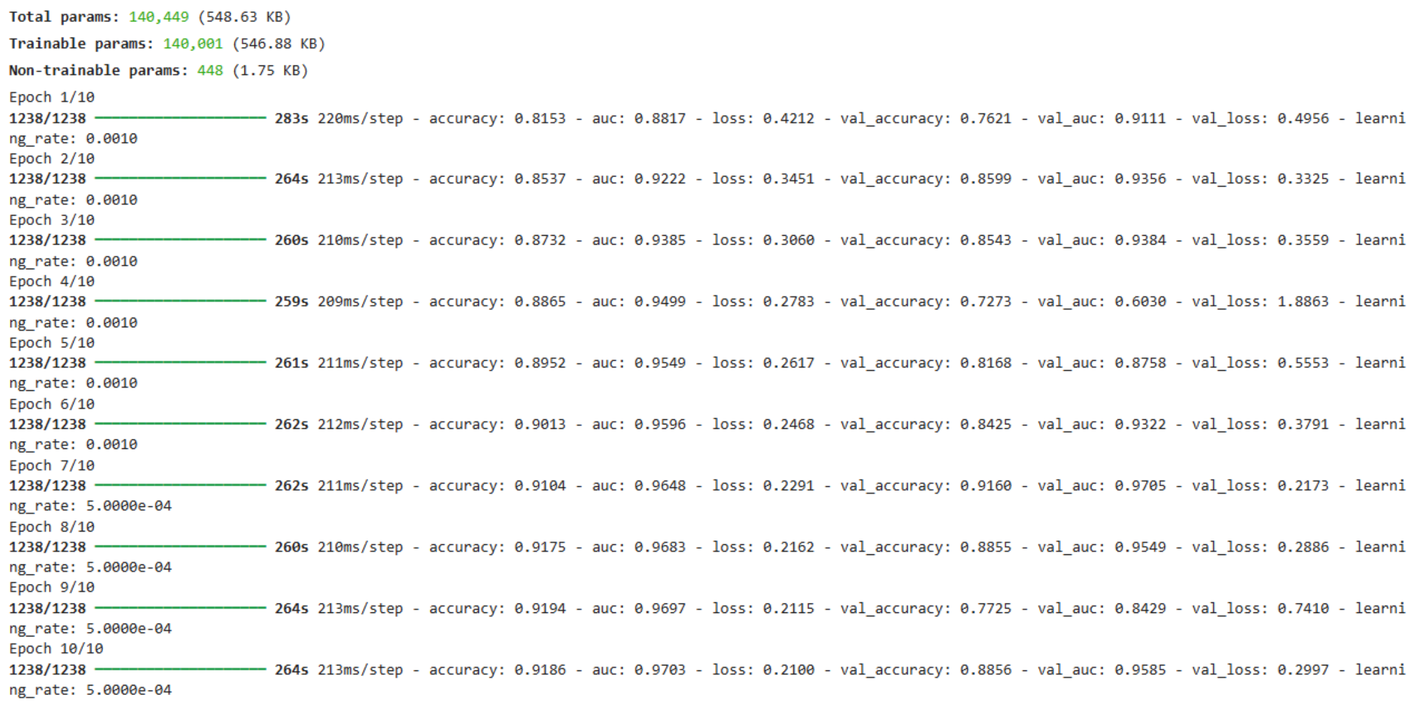


```
Total params: 93,377 (364.75 KB)
Trainable params: 93,377 (364.75 KB)
Non-trainable params: 0 (0.00 B)
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init
__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will b
e ignored.
  self._warn_if_super_not_called()
Epoch 1/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 261s 208ms/step - accuracy: 0.7487 - auc: 0.8125 - loss: 0.5120 - val_accuracy: 0.7332 - val_auc: 0.8292 - val_loss: 0.6234 - learni
ng_rate: 0.0010
Epoch 2/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 255s 206ms/step - accuracy: 0.7925 - auc: 0.8641 - loss: 0.4488 - val_accuracy: 0.7299 - val_auc: 0.8272 - val_loss: 0.5872 - learni
ng_rate: 0.0010
Epoch 3/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 254s 206ms/step - accuracy: 0.8124 - auc: 0.8833 - loss: 0.4181 - val_accuracy: 0.7829 - val_auc: 0.8743 - val_loss: 0.4893 - learni
ng_rate: 0.0010
Epoch 4/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 258s 209ms/step - accuracy: 0.8292 - auc: 0.9011 - loss: 0.3876 - val_accuracy: 0.7718 - val_auc: 0.8644 - val_loss: 0.5269 - learni
ng_rate: 0.0010
Epoch 5/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 258s 208ms/step - accuracy: 0.8339 - auc: 0.9036 - loss: 0.3841 - val_accuracy: 0.8122 - val_auc: 0.8993 - val_loss: 0.4232 - learni
ng_rate: 0.0010
Epoch 6/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 260s 210ms/step - accuracy: 0.8375 - auc: 0.9084 - loss: 0.3743 - val_accuracy: 0.8381 - val_auc: 0.9180 - val_loss: 0.3717 - learni
ng_rate: 0.0010
Epoch 7/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 256s 207ms/step - accuracy: 0.8440 - auc: 0.9153 - loss: 0.3618 - val_accuracy: 0.7928 - val_auc: 0.8933 - val_loss: 0.5293 - learni
ng_rate: 0.0010
Epoch 8/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 257s 208ms/step - accuracy: 0.8483 - auc: 0.9209 - loss: 0.3504 - val_accuracy: 0.8274 - val_auc: 0.9061 - val_loss: 0.4061 - learni
ng_rate: 0.0010
Epoch 9/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 259s 209ms/step - accuracy: 0.8561 - auc: 0.9253 - loss: 0.3406 - val_accuracy: 0.8537 - val_auc: 0.9178 - val_loss: 0.3641 - learni
ng_rate: 0.0010
Epoch 10/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 257s 207ms/step - accuracy: 0.8661 - auc: 0.9356 - loss: 0.3166 - val_accuracy: 0.8112 - val_auc: 0.9148 - val_loss: 0.4437 - learni
ng_rate: 5.0000e-04
```

TRAINING ENHANCED CNN



```python
enhanced_model = create_enhanced_cnn()
enhanced_model.compile(
    optimizer=Adam(learning_rate=LEARNING_RATE),
    loss='binary_crossentropy',
    metrics=[tf.keras.metrics.AUC(name='auc'), 'accuracy']
)

enhanced_model.summary()

history_enhanced = enhanced_model.fit(
    train_gen,
    epochs=EPOCHS,
    validation_data=val_gen,
    callbacks=[early_stopping, reduce_lr],
    verbose=1
)

results['Enhanced CNN'] = history_enhanced
```



```
Total params: 140,449 (548.63 KB)
Trainable params: 140,001 (546.88 KB)
Non-trainable params: 448 (1.75 KB)
Epoch 1/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 283s 220ms/step - accuracy: 0.8153 - auc: 0.8817 - loss: 0.4212 - val_accuracy: 0.7621 - val_auc: 0.9111 - val_loss: 0.4956 - learni
ng_rate: 0.0010
Epoch 2/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 264s 213ms/step - accuracy: 0.8537 - auc: 0.9222 - loss: 0.3451 - val_accuracy: 0.8599 - val_auc: 0.9356 - val_loss: 0.3325 - learni
ng_rate: 0.0010
Epoch 3/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 260s 210ms/step - accuracy: 0.8732 - auc: 0.9385 - loss: 0.3060 - val_accuracy: 0.8543 - val_auc: 0.9384 - val_loss: 0.3559 - learni
ng_rate: 0.0010
Epoch 4/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 259s 209ms/step - accuracy: 0.8865 - auc: 0.9499 - loss: 0.2783 - val_accuracy: 0.7273 - val_auc: 0.6030 - val_loss: 1.8863 - learni
ng_rate: 0.0010
Epoch 5/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 261s 211ms/step - accuracy: 0.8952 - auc: 0.9549 - loss: 0.2617 - val_accuracy: 0.8168 - val_auc: 0.8758 - val_loss: 0.5553 - learni
ng_rate: 0.0010
Epoch 6/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 262s 212ms/step - accuracy: 0.9013 - auc: 0.9596 - loss: 0.2468 - val_accuracy: 0.8425 - val_auc: 0.9322 - val_loss: 0.3791 - learni
ng_rate: 0.0010
Epoch 7/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 262s 211ms/step - accuracy: 0.9104 - auc: 0.9648 - loss: 0.2291 - val_accuracy: 0.9160 - val_auc: 0.9705 - val_loss: 0.2173 - learni
ng_rate: 5.0000e-04
Epoch 8/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 260s 210ms/step - accuracy: 0.9175 - auc: 0.9683 - loss: 0.2162 - val_accuracy: 0.8855 - val_auc: 0.9549 - val_loss: 0.2886 - learni
ng_rate: 5.0000e-04
Epoch 9/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 264s 213ms/step - accuracy: 0.9194 - auc: 0.9697 - loss: 0.2115 - val_accuracy: 0.7725 - val_auc: 0.8429 - val_loss: 0.7410 - learni
ng_rate: 5.0000e-04
Epoch 10/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 264s 213ms/step - accuracy: 0.9186 - auc: 0.9703 - loss: 0.2100 - val_accuracy: 0.8856 - val_auc: 0.9585 - val_loss: 0.2997 - learni
ng_rate: 5.0000e-04
```

TRAINING RESNET50 (TRANSFER LEARNING)

```python
resnet_model = create_resnet_model()
resnet_model.compile(
    optimizer=Adam(learning_rate=LEARNING_RATE),
    loss='binary_crossentropy',
    metrics=[tf.keras.metrics.AUC(name='auc'), 'accuracy']
)

resnet_model.summary()

history_resnet = resnet_model.fit(
    train_gen,
    epochs=EPOCHS,
    validation_data=val_gen,
    callbacks=[early_stopping, reduce_lr],
    verbose=1
)

results['ResNet50'] = history_resnet
```

```
Total params: 23,850,113 (90.98 MB)
Trainable params: 262,401 (1.00 MB)
Non-trainable params: 23,587,712 (89.98 MB)
Epoch 1/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 291s 224ms/step - accuracy: 0.5912 - auc: 0.5729 - loss: 0.6741 - val_accuracy: 0.6527 - val_auc: 0.7185 - val_loss: 0.6216 - learni
ng_rate: 0.0010
Epoch 2/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 263s 212ms/step - accuracy: 0.6336 - auc: 0.6747 - loss: 0.6291 - val_accuracy: 0.6878 - val_auc: 0.7492 - val_loss: 0.5970 - learni
ng_rate: 0.0010
Epoch 3/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 264s 213ms/step - accuracy: 0.6627 - auc: 0.7112 - loss: 0.6071 - val_accuracy: 0.6708 - val_auc: 0.7717 - val_loss: 0.5913 - learni
ng_rate: 0.0010
Epoch 4/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 259s 209ms/step - accuracy: 0.6770 - auc: 0.7263 - loss: 0.5957 - val_accuracy: 0.6933 - val_auc: 0.7781 - val_loss: 0.5747 - learni
ng_rate: 0.0010
Epoch 5/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 259s 209ms/step - accuracy: 0.6870 - auc: 0.7389 - loss: 0.5876 - val_accuracy: 0.6814 - val_auc: 0.7768 - val_loss: 0.5848 - learni
ng_rate: 0.0010
Epoch 6/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 262s 212ms/step - accuracy: 0.6893 - auc: 0.7417 - loss: 0.5860 - val_accuracy: 0.7055 - val_auc: 0.7816 - val_loss: 0.5698 - learni
ng_rate: 0.0010
Epoch 7/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 261s 211ms/step - accuracy: 0.6898 - auc: 0.7437 - loss: 0.5846 - val_accuracy: 0.7021 - val_auc: 0.7878 - val_loss: 0.5634 - learni
ng_rate: 0.0010
Epoch 8/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 264s 213ms/step - accuracy: 0.6936 - auc: 0.7469 - loss: 0.5824 - val_accuracy: 0.6851 - val_auc: 0.7854 - val_loss: 0.5787 - learni
ng_rate: 0.0010
Epoch 9/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 263s 212ms/step - accuracy: 0.6930 - auc: 0.7460 - loss: 0.5831 - val_accuracy: 0.6880 - val_auc: 0.7925 - val_loss: 0.5681 - learni
ng_rate: 0.0010
Epoch 10/10
1238/1238 ━━━━━━━━━━━━━━━━━━━━ 261s 211ms/step - accuracy: 0.6957 - auc: 0.7503 - loss: 0.5780 - val_accuracy: 0.6998 - val_auc: 0.7805 - val_loss: 0.5711 - learni
ng_rate: 0.0010
```

INITIAL MODEL PERFORMANCE COMPARISON

[42]:
```python
initial_results = []
for model_name, history in results.items():
    final_train_auc = history.history['auc'][-1]
    final_val_auc = max(history.history['val_auc'])
    final_train_acc = history.history['accuracy'][-1] if 'accuracy' in history.history else 0
    final_val_acc = max(history.history['val_accuracy']) if 'val_accuracy' in history.history else 0

    initial_results .append({
        'Model': model_name,
        'Final Train AUC': f"{final_train_auc:.4f}",
        'Best Val AUC': f"{final_val_auc:.4f}",
        'Final Train Acc': f"{final_train_acc:.4f}",
        'Best Val Acc': f"{final_val_acc:.4f}",
        'Epochs Trained': len(history.history['auc'])
    })

initial_results_df = pd.DataFrame(initial_results)
print(initial_results_df.to_string(index=False))
```

```
       Model Final Train AUC Best Val AUC Final Train Acc Best Val Acc  Epochs Trained
  Simple CNN          0.9349       0.9180          0.8655       0.8537              10
Enhanced CNN          0.9702       0.9705          0.9186       0.9160              10
    ResNet50          0.7507       0.7925          0.6963       0.7055              10
```

[44]:
```python
### Identifying best model
best_initial_model = max(results.keys(),
                        key=lambda x: max(results[x].history['val_auc']))
best_initial_auc = max(results[best_initial_model].history['val_auc'])
print(f" best initial model: {best_initial_model}")
print(f"best val AUC: {best_initial_auc:.4f}")
```

```
 best initial model: Enhanced CNN
best val AUC: 0.9705
```

Model Comparison Visuals

Model Performance Summary

I trained three different models for 10 epochs each: a simple CNN, an enhanced CNN with added layers and regularization, and a ResNet50-based transfer learning model. After evaluating each model on both the training and validation sets, I compared their AUC metrics.

The enhanced CNN consistently outperformed the others. It achieved the highest validation AUC, showing both strong generalization. The simple CNN performed reasonably well, but not as strongly. Suprasingly the ResNet50 model underperformed compared to both CNNs.

Why did ResNet50 fail?

- Medical histopathology images are very different from ImageNet's natural images
- Pre-trained features (animals, objects, scenes) don't transfer well to medical images, I should have only frezee low level layers which can be transfered more easily. this shows the importance of domain-specific features in medical imaging

HYPERPARAMETER TUNING

Tunning Learning Rate

```
### testing different learning rates I used this before (0.001)
learning_rates = [0.01, 0.0001]
hyperparameter_results = {}

for lr in learning_rates:
    print(f"learning rate: {lr}")

    tuning_model = create_enhanced_cnn()
    tuning_model.compile(
        optimizer=Adam(learning_rate=lr),
        loss='binary_crossentropy',
        metrics=[tf.keras.metrics.AUC(name='auc'), 'accuracy']
    )


    history_tune = tuning_model.fit(
        train_gen,
        epochs=8,
        validation_data=val_gen,
        callbacks=[EarlyStopping(monitor='val_auc', patience=3, mode='max')],
        verbose=0
    )

    hyperparameter_results[f'LR_{lr}'] = history_tune
    final_auc = max(history_tune.history['val_auc'])
    print(f"best val AUC with learning rate {lr}: {final_auc:.4f}")
```

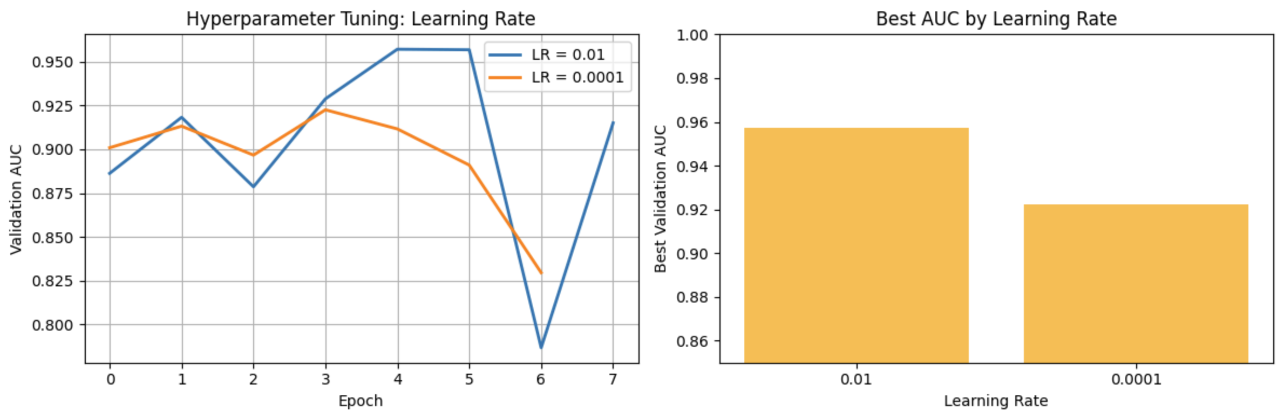
```
learning rate: 0.01
I0000 00:00:1748955215.248698      35 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 13942 MB m
sla T4, pci bus id: 0000:00:04.0, compute capability: 7.5
I0000 00:00:1748955215.249442      35 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:1 with 13942 MB m
sla T4, pci bus id: 0000:00:05.0, compute capability: 7.5
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` cla
__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these argume
e ignored.
  self._warn_if_super_not_called()
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1748955225.503354     100 service.cc:148] XLA service 0x792434048100 initialized for platform CUDA (this does not guarant
ices:
I0000 00:00:1748955225.509456     100 service.cc:156]   StreamExecutor device (0): Tesla T4, Compute Capability 7.5
I0000 00:00:1748955225.509481     100 service.cc:156]   StreamExecutor device (1): Tesla T4, Compute Capability 7.5
I0000 00:00:1748955226.350840     100 cuda_dnn.cc:529] Loaded cuDNN version 90300
I0000 00:00:1748955235.406875     100 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at most once for the li
best val AUC with learning rate 0.01: 0.9571
learning rate: 0.0001
best val AUC with learning rate 0.0001: 0.9225
```

Tunning Optimizers

```
[20]:
# testing different optimizers
optimizer_results = {}


optimizers_config = {
    'RMSprop': RMSprop(learning_rate=0.001),
    'Adam': Adam(learning_rate=0.001)
}
for opt_name, optimizer in optimizers_config.items():
    print(f"optimizer: {opt_name}")

    tuning_model = create_enhanced_cnn()
    tuning_model.compile(
        optimizer=optimizer,
        loss='binary_crossentropy',
        metrics=[tf.keras.metrics.AUC(name='auc'), 'accuracy']
    )

    history_tune = tuning_model.fit(
        train_gen,
        epochs=8,
        validation_data=val_gen,
        callbacks=[EarlyStopping(monitor='val_auc', patience=3, mode='max')],
        verbose=0
    )

    optimizer_results[f'{opt_name}'] = history_tune
    best_auc = max(history_tune.history['val_auc'])
    print(f"best Val AUC: {best_auc:.4f}")

    # Save the model for this optimizer
    tuning_model.save(f'enhanced_cnn_{opt_name.lower()}.h5')
    print(f"model with {opt_name} saved!")
```

```
optimizer: RMSprop
best Val AUC: 0.9631
model with RMSprop saved!
optimizer: Adam
best Val AUC: 0.9529
model with Adam saved!
```

```
[24]:
plot_hyperparameter_results(hyperparameter_results)
```



Hyperparameter Tuning Conclusion

I am surprised that RMSprop actually beat Adam. I mean, Adam is usually performs better. But I guess this just shows that different problems might need different approaches. Also, the fact that a 0.01 learning rate worked almost as well as 0.001. The 0.0001 result honestly makes sense though, probably just too slow to learn much in only 10 epochs.

Overall, I think this tuning experiment confirmed that having a good model architecture like my Enhanced CNN, matters *way* more than perfectly tuning every little parameter. The differences between optimizers and learning rates were small.

General Conclusion

I went into this thinking bigger models and transfer learning stuff would *obviously* be better, but I've realized domain knowledge and design actually matter a lot. Like, our 140K parameter Enhanced CNN totally outperformed the 23.8M parameter ResNet50, which I didn't really expect. It could have perform better, if I was able train longer and with more data. The problem was I had limited resources and this training took very long time.

Also, I'm glad I did a proper comparison instead of just picking one and hoping it worked. Seeing ResNet50 kinda fail was actually more helpful than if it had just worked like I assumed it would. I guess I didn't really think about whether ImageNet features even made sense for our problem now I know that's something I need to check going forward.

The tuning part was helpful too, though maybe not too much difference, it mostly showed that the default settings we used were already pretty solid. I guess that kind of reinforces that sometimes the defaults are there for a reason.