CUSTOM EFFICIENT NEURAL NETWORK FOR HISTOPATHOLOGICAL LUNG CANCER CLASSIFICATION

Bonface Martin Oywa
UNICAF MSc. Computer Science, University of East London
martinoywa2@gmail.com

This paper explores the development and evaluation of a small custom image classifier for lung cancer histopathological images. Leveraging Deep Neural Networks, the model should aid enhance medical diagnosis efficiency. To optimize deployment, the architecture focuses on minimizing model size without resorting to post-development compression techniques. The algorithm demonstrates promising results, achieving a 91.89% reduction in parameter size compared to EfficientNet variant B0, with a deployable model parameter size of approximately 1.6 MegaBytes (MBs). Training results reveal superior performance, with a final accuracy of 99% and a loss of 0.018140. Despite longer training times compared to some variants, the model's effectiveness underscores the potential of tailored deep learning solutions in medical diagnostics, albeit with hardware-dependent considerations.

1. Introduction

1.1. Research Problem

Lung cancer classification with a small efficient neural network.

1.2. Objectives

1. Build a lung cancer image classifier.
2. Build a small and efficient model.
3. Evaluate the performance of the classifier on a test dataset.
4. Evaluate the performance of the classifier compared to models in the wild on similar data.
5. Deploy the classifier.

1.3. Background

Over the last few years, Deep Learning, a subset of Artificial Intelligence (AI) that utilizes the concept of Deep Neural Networks (DNNs), a form of Artificial Neural Networks designed to mimic the functioning of human brain neurons, has gained significant popularity. When provided with substantial amounts of data, these algorithms can achieve remarkable accuracy in tasks such as object identification. This capability has led to a wide range of applications implementations across various fields, including medicine, where deep learning is used for tasks such as consultancy via chatbots, drug discovery through protein folding analysis, and diagnostic procedures utilizing computer vision. In this paper, we will design, analyze, and evaluate a custom algorithm to

classify the presence or absence of cancer in lung cells.

Given that deployment is a crucial aspect of this project, we aim to ensure that the architecture design results in a small model size, thereby obviating the need for post-development compression techniques, which can be resource-intensive. Efficient deployment is vital because smaller models reduce computational resources and costs, improve latency, and enhance accessibility, making the technology more practical for real-world applications. Although not employed in this study, common model compression techniques include model pruning, which removes less significant weights; quantization, which reduces the precision of weights and activations; and knowledge distillation, where a smaller "student" model learns to replicate the behavior of a larger "teacher" model. These methods help maintain performance while significantly reducing model size, facilitating deployment on devices with limited computational power (Huyen, 2022, p. 206).

As the task is multiclass image classification, the algorithm will process batches of images, enabling efficient data handling and quicker processing times. The model will output a probability score for each class for every image, with the highest score indicating the predicted class.

2. Architecture

Since the task images classification as described in the introduction, the architecture of choice will be based on Convolutional Neural Networks (CNNs). This architecture will involve two blocks as described below;

1. **Feature Learning Block or the Backbone**. This normally contains the CNNs, Activation Functions and, Pooling Layers. CNNs are used for their ability to capture and extract spatial information from grid-like data structures, such as the pixel arrangements in images. These networks are particularly gread at identifying spatial hierarchies and patterns crucial for accurate image classification. To enhance their performance, they are integrated with Activation Functions, which introduce non-linearity, enabling the model to capture complex patterns and relationships within the data. Finally, the Pooling Layers are incorporated to perform downsampling on the extracted features, and reducing dimensionality by retaining essential information while discarding redundancies. These features are then used downstream for classification.
2. **Classification Block or the Head**. This typically includes a Flattening or Global Average Pooling (GAP) layer, Fully Connected or Dense layers, and an Output layer, which is also a Dense layer. Both Flattening and GAP serve to convert the output from the backbone of the network

into a vector embedding suitable for the Fully Connected layers, which require vectors as input. However, GAP will be preferred due to its efficiency, as it computes the average over the entire network feature map or output channels of the convolutional layers, resulting in a significantly shorter vector compared to Flattening. Moreover, unlike Flattening, GAP consistently produces a vector of fixed size regardless of the input image dimensions, thereby simplifying the architecture design by eliminating the need to calculate the input size for the subsequent layers. This vector is then fed into the Fully Connected layers, culminating in the Output layer, which generates the probability scores for the entire network.

The architecture will also contain the use of Batch Normalization to normalize feature maps between our convolutional layers to prevent them from varying wildly during training, because this would require large adjustments of the subsequent layers. It inherently normalizes the activations and keeps them much more stable during training, making the training more stable and the convergence faster (Ioffe and Szegedy, 2015).

To prevent overfitting, where the model becomes too tailored to the training data and struggles to generalize to unseen data, techniques such as Dropout and L2 regularization were employed. Dropout, as described by Hinton et al. (2012), involves randomly deactivating a portion of the network's neurons during training, which helps to prevent the model from becoming too reliant on any single neuron and encourages the learning of more robust features. L2 regularization, which will be discussed in detail in the optimizer section. These techniques were instrumental in stabilizing the training process, enabling the model to achieve excellent test scores by enhancing its ability to generalize well to new data.

The entire architecture consists of five Convolution Blocks in the backbone, each comprising a Convolution layer, a Batch Normalization layer, a ReLU activation function, and a Max Pooling layer. In the head of the network, a single Fully Connected Block includes a GAP layer, two hidden layers with Dropout for regularization, and the Output layer, as illustrated in Figure 2.0. The following code output provides a summary of the entire network, detailing the number of trainable parameters and the estimated total size of the model parameters.

```
----------------------------------------------------------------
      Layer (type)               Output Shape         Param #
================================================================
          Conv2d-1        [30, 16, 384, 384]             448
     BatchNorm2d-2        [30, 16, 384, 384]              32
```

```
      MaxPool2d-3        [30, 16, 192, 192]               0
          Conv2d-4        [30, 32, 192, 192]           4,640
     BatchNorm2d-5        [30, 32, 192, 192]              64
      MaxPool2d-6          [30, 32, 96, 96]               0
          Conv2d-7          [30, 64, 96, 96]          18,496
     BatchNorm2d-8          [30, 64, 96, 96]             128
      MaxPool2d-9          [30, 64, 48, 48]               0
         Conv2d-10         [30, 128, 48, 48]          73,856
    BatchNorm2d-11         [30, 128, 48, 48]             256
     MaxPool2d-12         [30, 128, 24, 24]               0
         Conv2d-13         [30, 256, 24, 24]         295,168
    BatchNorm2d-14         [30, 256, 24, 24]             512
     MaxPool2d-15         [30, 256, 12, 12]               0
AdaptiveAvgPool2d-16           [30, 256, 1, 1]               0
        Linear-17                 [30, 128]          32,896
       Dropout-18                 [30, 128]               0
        Linear-19                  [30, 64]           8,256
       Dropout-20                  [30, 64]               0
        Linear-21                   [30, 3]             195
================================================================
Total params: 434,947
Trainable params: 434,947
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 50.62
Forward/backward pass size (MB): 2354.21
Params size (MB): 1.66
Estimated Total Size (MB): 2406.49
----------------------------------------------------------------
```

Code Output 2.0 Model Architecture Summary via Torch Summary Tool (https://pypi.org/project/torch-summary)

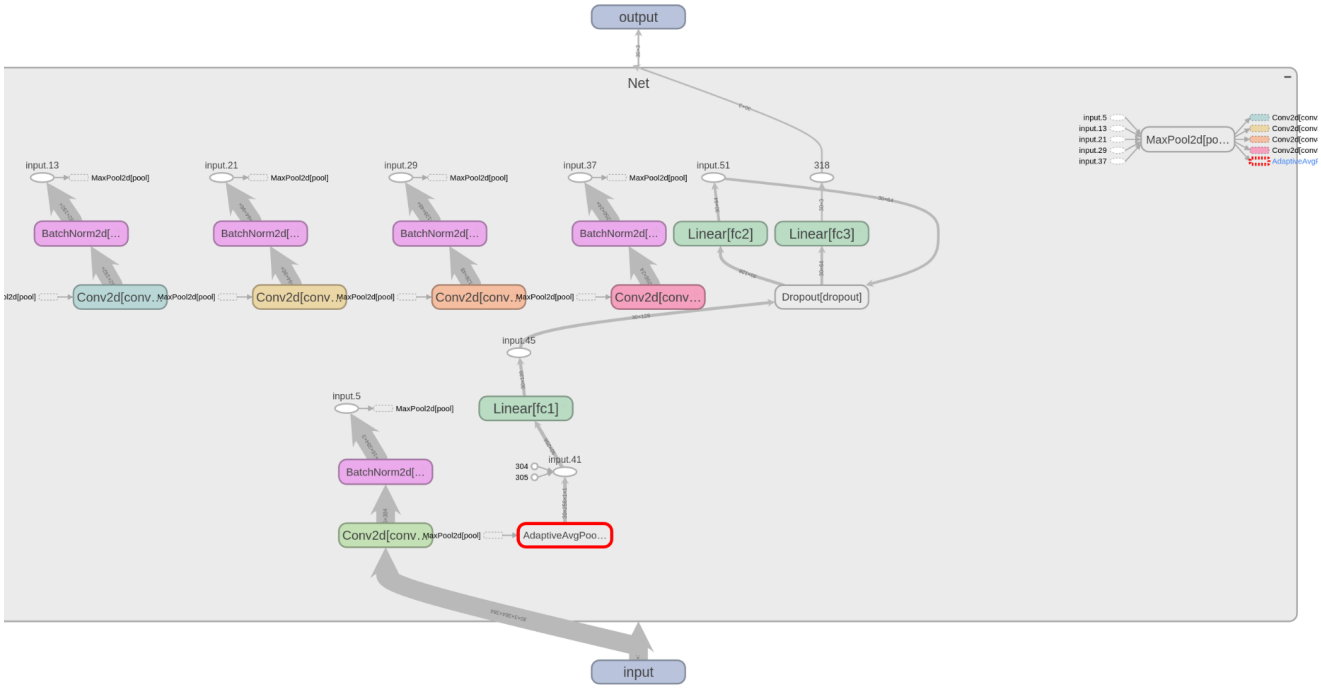

Figure 2.0 Architecture Visualization via Tensor Board Tool (https://www.tensorflow.org/tensorboard)

3. Dataset

The dataset selected for this task is the Lung and Colon Cancer Histopathological Image Dataset (LC25000), which comprises 25,000 color images with dimensions of 768 by 768 pixels, categorized into five classes: colon adenocarcinoma, benign colonic tissue, lung adenocarcinoma, lung squamous cell carcinoma, and benign lung tissue (Borkowski et al., 2019). In this dataset, adenocarcinoma and carcinoma represent cancerous cells, whereas benign tissues are non-cancerous. For the classifier evaluated in this paper, only the lung cancer histopathological data were utilized, focusing on distinguishing between lung adenocarcinoma, lung squamous cell carcinoma, and benign lung tissue as the three categories in the multiclass classification task.

Lung adenocarcinoma and lung squamous cell carcinoma are two common types of lung cancer with distinct histological and molecular characteristics. Adenocarcinoma typically originates in the outer regions of the lungs and is associated with glandular patterns, while squamous cell carcinoma arises in the bronchial epithelium and is characterized by keratinization. Both are malignant and can metastasize, often presenting with symptoms such as coughing, chest pain, and shortness of breath. In contrast, benign lung tissue refers to non-cancerous cells and structures within the lungs, typically exhibiting normal cellular morphology and function. While benign tissue may occasionally present as nodules or masses on imaging studies, it lacks the invasive properties and potential

for metastasis seen in malignant tumors (Wang, Liu and Li, 2022).

To address authenticity and privacy concerns, the authors of the dataset have ensured that the images were captured from pathology glass slides and have been de-identified, thereby complying with the Health Insurance Portability and Accountability Act (HIPAA) and validation standards. De-identification involves removing any Personally Identifiable Information (PII) that could potentially reveal the identity of the individuals whose cells are represented in the dataset. HIPAA compliance guarantees the protection of this information (Edemekong, Haydel, and Annamaraju, 2022), ensuring that the dataset can be used in research without compromising patient privacy.

According to Borkowski et al. (2019), the original images in this dataset initially measured 1024 by 768 pixels, comprising 750 total lung tissues and 500 total colon tissues, with 250 images for each class. To expand the dataset to 25,000 images, the authors utilized the Augmentor Python software package (https://augmentor.readthedocs.io/en/stable/). Augmentation techniques applied included left and right rotations of up to 25 degrees with a probability of 1.0 and horizontal and vertical flips with a probability of 0.5. These augmentations effectively increased the dataset's diversity and variability, enhancing the trained models' robustness and generalization capability. Examples of these are shown below;

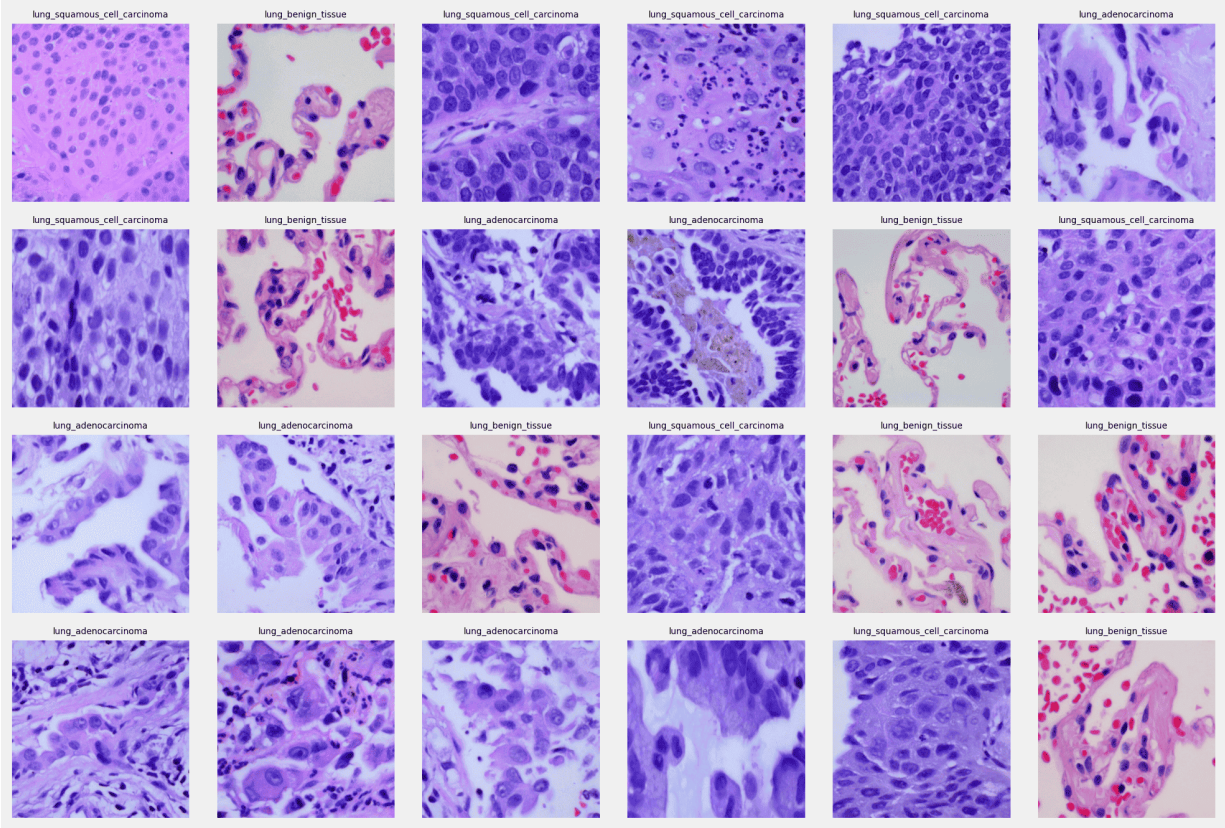

Figure 3.0 Original Dataset Sample Images

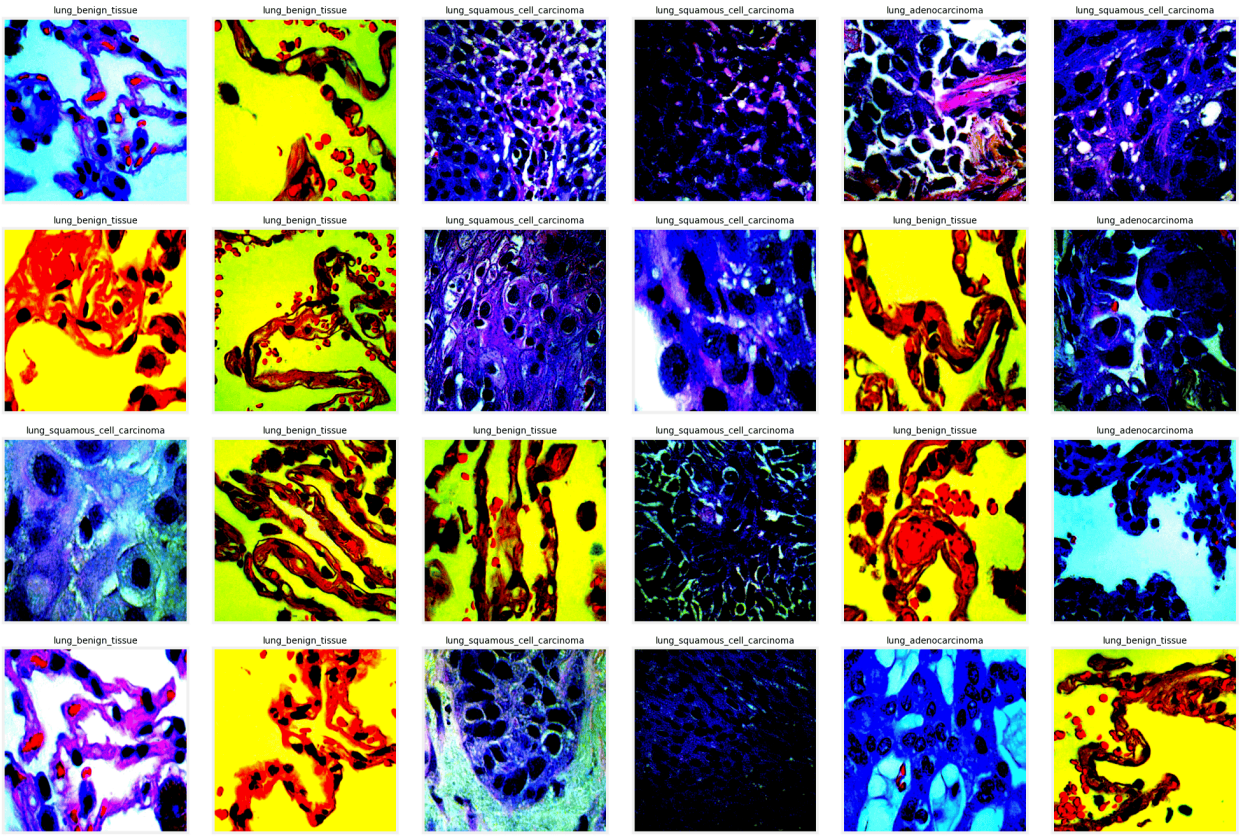

Fig 3.1 Normalized Dataset Sample Images

Since PyTorch was the chosen framework for modeling the classifier, additional augmentations were performed in the code as follows;

1. The image sizes were halved, from 768 to 384 pixels, to accommodate the GPU storage limitation of 12GB during training.
2. All images were converted to Tensors, which are multidimensional matrices, serving as the primary data structure in most deep learning frameworks for modeling various data forms.
3. To facilitate model training within a smaller memory space, pixel values in the images were normalized to fall between 0 and 1 using the average mean and standard deviation calculated from the entire dataset. This normalization process aids in stabilizing and accelerating the training process.

A visualization of sample data from the training set is depicted in Figure 3.1 (above). This is a representation of the preprocessed data fed into the model

From the visualization provided, some notable observations can be made. Benign tissue images appear lighter and have a bright yellow color, often featuring red patches, which makes them relatively easy to identify visually. On the other hand, cancerous adenocarcinoma and squamous cell carcinoma tissues exhibit a deeper blue-purple color, which can make them challenging to distinguish without labels due to their similarity in appearance.

The dataset was divided into training, validation, and test splits, comprising 10,500, 2,250, and 2,250 images, respectively, following a split format of 70%-15%-15%. PyTorch's random split functionality was utilized for this purpose, ensuring the generation of non-overlapping splits based on the specified split sizes (pytorch.org, n.d. a). This partitioning strategy enables the model to be trained, validated, and tested on distinct subsets of the data, facilitating comprehensive performance assessment across different stages of model development.

4. Algorithm

4.1. Introduction

To ensure reproducibility across different data splits and training iterations, a consistent random seed of 0 was applied throughout the entire notebook. The technical stack utilized includes PyTorch, Python, CUDA for GPU acceleration, and Streamlit for model deployment.

This section comprises two distinct algorithms: one for model development, encompassing data preprocessing to model evaluation, and another for model deployment, which involves uploading an image and obtaining results from the model. The architectural diagrams for these algorithms are illustrated in Figures 4.1.0 and 4.1.1, respectively. These diagrams should provide a visual representation of the workflow and components involved in each algorithm.

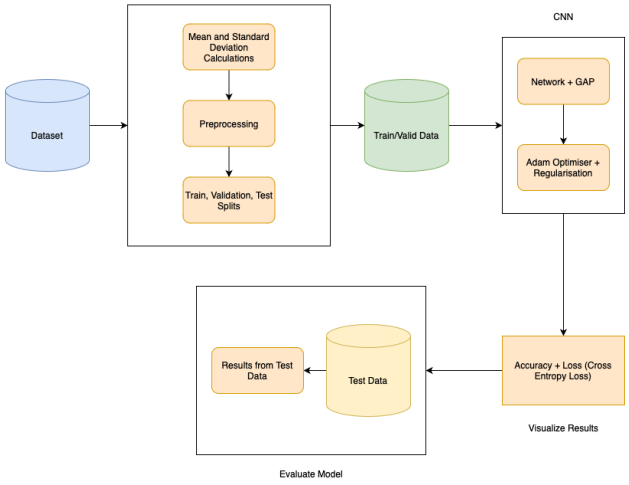

Figure 4.1.0 Model Development Algorithm Architecture

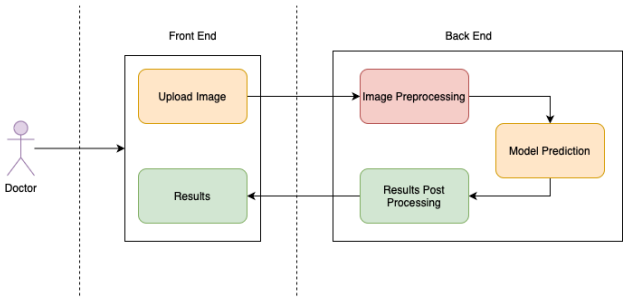

Figure 4.1.1 Model Deployment Algorithm Architecture

4.2. Model Development

The hyperparameters employed in this research included **epochs**, **batch size**, and **weight decay**. The epoch value was set to 30, allowing the model to undergo sufficient training without succumbing to overfitting. Regarding batch size, a value of 30 was chosen due to GPU memory constraints. Experiments with larger batch sizes resulted in GPU memory overflow during training, leading to incomplete iterations.

The weight decay hyperparameter, applied to the optimizer, introduced L2 regularization by adding a penalty term to the loss function proportional to the sum of the squared values of the model's weights. This regularization technique discourages large weights, promoting simpler models that generalize better to unseen data.

The optimizer of choice was Adam (Adaptive Moment Estimation) due to its capability to converge rapidly and handle noisy or sparse gradients effectively. Additionally, Adam eliminates the need for manual tuning of hyperparameters such as learning rate decay or momentum coefficient, simplifying its usage compared to other optimization algorithms (pytorch.org, n.d. b).

For the loss function, Cross Entropy Loss was selected, given the multiclass classification task. Validation loss values were utilized to determine the saving of trained model weights, ensuring that only models demonstrating convergence were preserved for deployment. The code output 4.0 provided below demonstrates this process.

Since the dataset was balanced by design, monitoring and evaluation of performance primarily focused on loss and accuracy metrics.

```
Epoch: 1    Training Loss: 0.365471    Training Accuracy: 0.868000    Validation Loss: 0.201693
Validation Accuracy: 0.915556
Validation loss decreased (inf --> 0.201693).  Saving model ...
Epoch: 2    Training Loss: 0.253896    Training Accuracy: 0.906571    Validation Loss: 0.134823
Validation Accuracy: 0.952889
Validation loss decreased (0.201693 --> 0.134823).  Saving model ...
Epoch: 3    Training Loss: 0.185743    Training Accuracy: 0.932381    Validation Loss: 0.125715
Validation Accuracy: 0.950667
```

Code Output 4.2.0 Model Saving on Validation Loss Decrease

4.3. Code Samples Model Development

Hosted code for model development can be found here https://github.com/martinoywa/turbo-guacamole.

```
class Net(nn.Module):
  def __init__(self):
      super(Net, self).__init__()
      # convolutional layers
      self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
      self.bn1 = nn.BatchNorm2d(16)
      self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
      self.bn2 = nn.BatchNorm2d(32)
      self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
      self.bn3 = nn.BatchNorm2d(64)
      self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
      self.bn4 = nn.BatchNorm2d(128)
      self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
      self.bn5 = nn.BatchNorm2d(256)
      # max pooling layer
      self.pool = nn.MaxPool2d(2, 2)
      # global average pooling
      self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
      # fully connected layer
      self.fc1 = nn.Linear(256, 128)
      self.fc2 = nn.Linear(128, 64)
      self.fc3 = nn.Linear(64, len(classes))
      # dropout
      self.dropout = nn.Dropout(p=0.5)

  def forward(self, x):
      x = self.pool(F.relu(self.bn1(self.conv1(x))))
      x = self.pool(F.relu(self.bn2(self.conv2(x))))
      x = self.pool(F.relu(self.bn3(self.conv3(x))))
      x = self.pool(F.relu(self.bn4(self.conv4(x))))
      x = self.pool(F.relu(self.bn5(self.conv5(x))))
      x = self.global_avg_pool(x)
      x = torch.flatten(x, 1)
      x = self.dropout(F.relu(self.fc1(x)))
      x = self.dropout(F.relu(self.fc2(x)))
      x = F.relu(self.fc3(x))
      return x
```

Code Sample 4.3.0 Custom Architecture PyTorch Implementation

```
model.train()
for data, target in trainloader:
  # move data to gpu
  if train_on_gpu:
      data, target = data.cuda(), target.cuda()
```

```
  # clear the gradients
  optimizer.zero_grad()
  # forward pass
  outputs = model(data)
  # calculate the loss
  loss = criterion(outputs, target)
  # accumulate gradients
  loss.backward()
  # update parameters
  optimizer.step()
  # store loss and accuracy for visualization
  train_loss += loss.item()*data.size(0)
  _, predicted = torch.max(outputs, 1)
  correct_train += (predicted == target).sum().item()
```

Code Sample 4.3.1 Training Data Loop

```
model.eval()
for data, target in validloader:
  if train_on_gpu:
      data, target = data.cuda(), target.cuda()
  outputs = model(data)
  loss = criterion(outputs, target)
  valid_loss += loss.item()*data.size(0)
  _, predicted = torch.max(outputs, 1)
  correct_valid += (predicted == target).sum().item()
```

Code Sample 4.3.2 Validation Data Loop

```
# calculate average losses and accuracies
train_loss = train_loss/len(trainloader.dataset)
valid_loss = valid_loss/len(validloader.dataset)
train_accuracy = correct_train/len(trainloader.dataset)
valid_accuracy = correct_valid/len(validloader.dataset)
```

Code Sample 4.3.3 Accuracy and Loss Calculation (Train and Validation)

```
# save model if validation loss has decreased
if valid_loss <= valid_loss_min:
  print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
  valid_loss_min,
  valid_loss))
  torch.save(model.state_dict(), 'model.pt')
  valid_loss_min = valid_loss
```

Code Sample 4.3.4 Model Weights Saving Logic

```
# testing
test_loss = 0.0
class_correct = list(0. for i in range(3))
class_total = list(0. for i in range(3))

model.eval()
model.cpu()
for data, target in testloader:
    output = model(data)
    loss = criterion(output, target)
    test_loss += loss.item()*data.size(0)
    # get predictions
    _, pred = torch.max(output, 1)
    # compare predictions to ground truth
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.cpu().numpy())
    # calculate per class accuracies
    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# average test loss
test_loss = test_loss/len(testloader.dataset)
print(f'Test Loss: {test_loss:.6f}\n')

for i in range(3):
    if class_total[i] > 0:
        print(f'Test Accuracy of {classes[i]}:
{(class_correct[i]/class_total[i])*100}%
({np.sum(class_correct[i])}/{np.sum(class_total[i])})')
    else:
        print(f'Test Accuracy of {classes[i]}: N/A (no training examples)')

print(f'\nTest Accuracy (Overall):
{(np.sum(class_correct)/np.sum(class_total))*100}
({np.sum(class_correct)}/{np.sum(class_total)})')
```

Code Sample 4.3.5 Final Testing Data Model Evaluation

4.4. Model Deployment

A web application was selected as a great choice for deploying a model because of ease of development, accessibility and use across broad device types.

Streamlit (https://streamlit.io/) was selected because it offers a streamlined, efficient way to build and share data-driven web applications with minimal coding effort. It allows machine learning engineers to create interactive and visually appealing dashboards directly in Python, allowing the use of libraries like Pandas, NumPy, and Matplotlib without needing extensive web development plugins. It also provided free hosting capabilities via GitHub. The hosted web applications can be found here. https://lungcancerhistopathy.streamlit.app/

4.5. Code Samples Model Deployment

In the web application, the expected input for the model is images. Therefore, an upload option was implemented to allow users to upload their images for classification. Once an image is uploaded, the model performs inference to make predictions regarding the content of the image. The output of the model includes the predicted class or label for the image, as well as the inference time, providing users with insight into the efficiency of the classification process.

```
from pathlib import Path

import streamlit as st
import torch

from loader import model_loader
from preprocessor import preprocess

st.title("Lung Cancer Histopathological Images Classifier")
if st.checkbox("Data and Model Details"):
  st.subheader("Dataset Details")
  st.markdown("""
[https://arxiv.org/abs/1912.12142v1](https://arxiv.org/abs/1912.12142v1)""")

  st.markdown("---")

  st.subheader("Model Details")
  st.image("src/images/training_metrics_overtime.png", caption="Training Metrics Overtime")
  st.markdown("""
    ```
    Test Loss: 0.018140
    ```
    """)
  st.markdown("""
      ```
      Test Accuracy of Lung Adenocarcinoma: 99% (716/717)
      Test Accuracy of Lung Benign Tissue: 100% (763/763)
```
```

```
        Test Accuracy of Lung Squamous Cell Carcinoma: 97% (753/770)
        ```
        """)
    st.markdown("""
        ```
        Test Accuracy (Overall): 99% (2232/2250)
        ```
        """)
st.markdown("---")

# classes
classes = ["Lung Adenocarcinoma", "Lung Benign Tissue", "Lung Squamous Cell Carcinomaa"]

# checkpoint loader
model_path = Path("src/checkpoint/model_v4_0199.pt")
model = model_loader(model_path)

# file uploader
image_file = st.file_uploader("Upload Lung Image", type=["png", "jpg", "jpeg"])

# inference
if image_file is not None:
    # To read file as bytes:
    bytes_data = image_file.getvalue()

    # preprocessing and predictions
    image_tensor = preprocess(bytes_data)
    prediction = model.forward(image_tensor)

    # outputs
    _, pred = torch.max(prediction, 1)

    # file viewer
    st.image(bytes_data, caption="Uploaded image")
    st.markdown(f"""
    ```
    Predicted Class: {classes[pred.item()]}
    ```
    """)
```

Code Sample 4.5.0 Main Code

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import streamlit as st
```
```

```
class Net(nn.Module):
  def __init__(self):
      super(Net, self).__init__()
      # convolutional layers
      self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
      self.bn1 = nn.BatchNorm2d(16)
      self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
      self.bn2 = nn.BatchNorm2d(32)
      self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
      self.bn3 = nn.BatchNorm2d(64)
      self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
      self.bn4 = nn.BatchNorm2d(128)
      self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
      self.bn5 = nn.BatchNorm2d(256)
      # max pooling layer
      self.pool = nn.MaxPool2d(2, 2)
      # global average pooling
      self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
      # fully connected layer
      self.fc1 = nn.Linear(256, 128)
      self.fc2 = nn.Linear(128, 64)
      self.fc3 = nn.Linear(64, 3)
      # dropout
      self.dropout = nn.Dropout(p=0.5)

  def forward(self, x):
      x = self.pool(F.relu(self.bn1(self.conv1(x))))
      x = self.pool(F.relu(self.bn2(self.conv2(x))))
      x = self.pool(F.relu(self.bn3(self.conv3(x))))
      x = self.pool(F.relu(self.bn4(self.conv4(x))))
      x = self.pool(F.relu(self.bn5(self.conv5(x))))
      x = self.global_avg_pool(x)
      x = torch.flatten(x, 1)
      x = self.dropout(F.relu(self.fc1(x)))
      x = self.dropout(F.relu(self.fc2(x)))
      x = F.relu(self.fc3(x))
      return x


@st.cache_data
def model_loader(path):
  model = Net()
  model.load_state_dict(torch.load(path, map_location='cpu'), strict=False)
  model.eval()
```

```
  return model
```

Code Sample 4.5.1 Model Loader

```
import io
from torchvision import transforms
from PIL import Image

# hyperparameters
n = 768 // 2
mean, std = [0.6703, 0.5346, 0.8518], [0.1278, 0.1731, 0.0729]


def preprocess(image_bytes):
  transform = transforms.Compose([
      transforms.Resize((n, n)),
      transforms.ToTensor(),
      transforms.Normalize(mean, std)
  ])
  # image_bytes are from the uploaded image
  image = Image.open(io.BytesIO(image_bytes)).convert('RGB')
  # sends a single image batch
  return transform(image).unsqueeze(0)
```

Code Sample 4.5.2 Image Preprocessor

4.6. Screenshots Model Deployment

Figure 4.6.0 Web Application Home Page

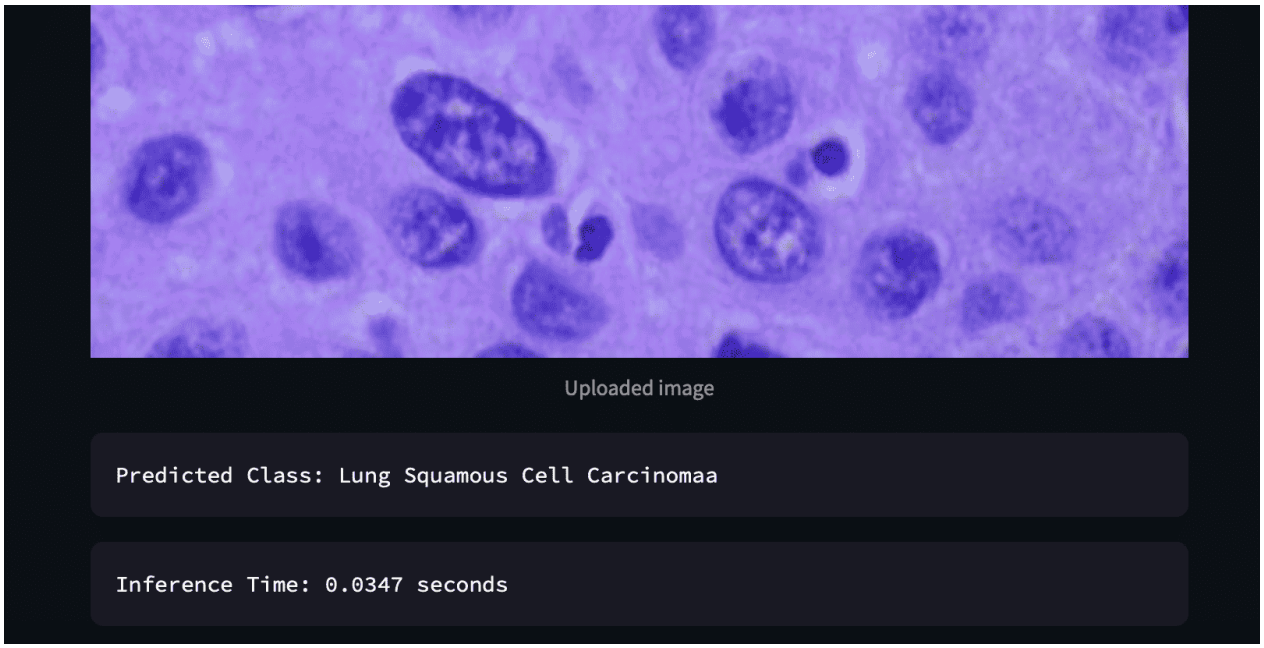

Figure 4.6.1 Web Application Results (Cropped Uploaded Image)

5. Results

Figure 5.0 The model's performance was evaluated over 30 epochs, with metrics such as accuracy and loss monitored to assess learning and generalization capabilities. At the conclusion of the training period, the model achieved a Training Loss of 0.052205 and a Training Accuracy of 98.12%, indicating a high degree of fit to the training data. Concurrently, the Validation Loss was 0.021515 and the Validation Accuracy was 99.2%, reflecting excellent generalization to unseen data. The close alignment between training and validation metrics implied the model effectively avoids overfitting, demonstrating robust predictive performance suitable for real-world applications.

The testing phase of the model involved evaluating overall test loss, accuracies per category, and overall accuracy to assess its real-world efficacy. Impressively, the model achieved an overall test accuracy of 99%, with a test loss of 0.018140. Notably, it demonstrated perfect accuracy in classifying benign tissues, achieving a perfect 100% (763 out of 763) accuracy rate and effectively eliminating false positives in this category. Regarding cancerous tissues, the model exhibited remarkable performance, achieving 99% accuracy in identifying lung adenocarcinoma samples (716 out of 717) and 97% accuracy in identifying lung squamous cell carcinoma samples (753 out of 770). These results highlight the model's exceptional precision and reliability across different tissue categories, culminating in an overall test accuracy of 99% (2232 out of 2250 samples). Thus, confirming its efficacy in accurately distinguishing between benign and various cancerous lung tissues, underscoring its potential utility in clinical settings.

6. Critical analysis

6.1. Introduction

To comprehensively evaluate this model, comparisons were drawn with other models utilized on the same dataset. The evaluation criteria included model sizes, performance during training, evaluation performance, and training times. By comparing these aspects, insights were gained into the relative strengths and weaknesses of different models. This comparative analysis enabled informed decisions regarding model selection and deployment.

6.2. Observations and Explanations

The total number of parameters achieved was approximately 430,000, compared to EfficientNet models described by (Anjum et al., 2023, Table 2), whose smallest model variant, B0, comprised 5,300,000 parameters, a reduction of 91.89%. This resulted in a deployable model parameter size of about 1.6 MegaBytes (MB), contrasting with a total parameter size of approximately 20.17 MB for variant B0. These calculations were conducted using the Torch Summary Tool, as detailed in the Architecture Section.

The custom model obtained a final Training Loss of 0.052205 and Training Accuracy of 98.12%, significantly outperforming all EfficientNet variants, with the best performance achieved by variant B2, which had a Training Loss of 0.07 and Training Accuracy of 97%. Similarly, the custom model achieved a final Validation Loss of 0.021515 and Validation Accuracy of 99.2%, surpassing the same variant B2 model (Anjum et al., 2023, Table 3). The custom model's overall test accuracy was 99%, with a test loss of 0.018140, demonstrating superior performance across the EfficientNet variants.

However, a drawback of this architecture was the training time, which amounted to approximately 45 minutes for 30 epochs—longer than only variants B3, B4, B6, and B7. In comparison, the best-performing variant B2 required only 9 minutes for 100 epochs. These discrepancies may be attributed to differences in image size (384 vs. 260 for variant B2) and hardware specifications, as we employed a locally running NVIDIA GeForce RTX 3060 with 3584 CUDA Cores and 12 GB GDDR6 Memory, intended for consumer graphics card use, while the Colab environment utilized an NVIDIA Tesla K80 with 4992 CUDA Cores and 24 GB GDDR5 Memory, designed for data centers, scientific computing, and machine learning applications (NVIDIA, n.d.).

7. Conclusion

In conclusion, this paper presents the design, analysis, and evaluation of a custom deep learning algorithm for the classification of lung cell cancer presence. By leveraging less CNNs blocks and GAP layers, the model demonstrates significant reductions in parameter size while maintaining high accuracy, showcasing its potential for efficient medical diagnosis. Despite longer training times compared to some variants, the algorithm's superior performance underscores the effectiveness.

For future work, several avenues of exploration emerge. Firstly, further optimization of the algorithm's training process could be pursued to mitigate training time. Additionally, the integration of advanced techniques such as transfer learning and ensemble methods may enhance model robustness and generalization to diverse datasets. Lastly, the application of the developed algorithm to real-world clinical settings warrants investigation, including rigorous validation against established diagnostic protocols.

8. References

Huyen, C. (2022). *Designing Machine Learning Systems*. 'O'Reilly Media, Inc.', p. 206.

Ioffe, S. and Szegedy, C. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. [online] arXiv.org. Available at: https://arxiv.org/abs/1502.03167.

Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R.R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580 [cs]*. [online] Available at: https://arxiv.org/abs/1207.0580.

Borkowski, A.A., Bui, M.M., Thomas, L.B., Wilson, C.P., DeLand, L.A. and Mastorides, S.M. (2019). Lung and Colon Cancer Histopathological Image Dataset (LC25000). *arXiv:1912.12142 [cs, eess, q-bio]*. [online] Available at: https://arxiv.org/abs/1912.12142v1.

Wang, W., Liu, H. and Li, G. (2022). What's the difference between lung adenocarcinoma and lung squamous cell carcinoma? Evidence from a retrospective analysis in a cohort of Chinese patients. *Frontiers in Endocrinology*, 13. doi:https://doi.org/10.3389/fendo.2022.947443.

Edemekong, P.F., Haydel, M.J. and Annamaraju, P. (2022). *Health insurance portability and accountability act (HIPAA)*. [online] Nih.gov. Available at: https://www.ncbi.nlm.nih.gov/books/NBK500019/.

pytorch.org. (n.d. a). *torch.utils.data — PyTorch 2.3 documentation*. [online] Available at: https://pytorch.org/docs/stable/data.html#torch.utils.data.random_split [Accessed 5 Jun. 2024].

pytorch.org. (n.d. b). *Adam — PyTorch 2.0 documentation*. [online] Available at:

https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam [Accessed 7 Jun. 2024].

Anjum, S., Ahmed, I., Asif, M., Hanan Aljuaid, Fahad Alturise, Yazeed Yasin Ghadi and Elhabob, R. (2023). Lung Cancer Classification in Histopathology Images Using Multiresolution Efficient Nets. *Computational Intelligence and Neuroscience*, 2023, pp.1–12. doi:https://doi.org/10.1155/2023/7282944.

NVIDIA. (n.d.). *Tesla K80*. [online] Available at: https://www.nvidia.com/en-gb/data-center/tesla-k80/. [Accessed 9 Jun. 2024].