Final Project Written Report:

Histopathologic Cancer Detection through Convolutional Neural Network and Resnet-18 & Medical Chatbot application through Graphical User Interface (GUI)

Group 1

Mingyuan Li (Kevin), Silan Deng (Cyrus), Jaechul Roh (Harry), Hao Dong (Hao Dong)

1. Introduction

The purpose of this written report is to demonstrate a user-friendly application that aids to classify whether an image of a pathology scan contains a metastatic cancer, along with a chatbot feature that helps the users to consult information about this specific cancer and for possible treatment. The data is from "Histopathologic Cancer Detection" Kaggle competition (Kaggle, 2022), which is a modified version of the PatchCamelyon benchmark. The main difference between the two datasets is that Kaggle helped us to filter out duplicate images. Our group further compared two types of Convolutional Neural Network (CNN) models, *Custom CNN* and *Resnet-18*, which we will further elaborate on the details later in this report. The code has been written in the kaggle notebook domain, since it provides us 30 hours of free GPU usage for a week, and the models were trained in the PyTorch Framework.

2. Terminology Definition

Metastatic cancer refers to the type of cancer that spreads from one part of the body to other parts. The original starting position is called the "Primary Cancer" (cancer.org, 2020).

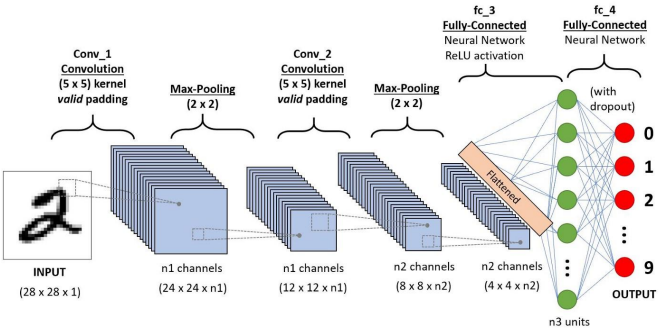


*Figure 1. Visual Illustration of Convolutional Neural Network. (*Saha, Sumit, 2021)

Convolutional Neural Network, also known as CNN or ConvNet, is one of the most common neural networks applied in the field of computer vision or image processing since it executes effective analysis in pixel data (Contributor). The neural network itself contains two main parts of layers: Convolutional (Conv.) Layer and Fully Connected (FC) Layer. After an image is processed by pixels within the convolutional layer, it processes a classification task within the fully connected layer (fc_3 and fc_4 from Figure 1), telling us which class the input image belongs to.

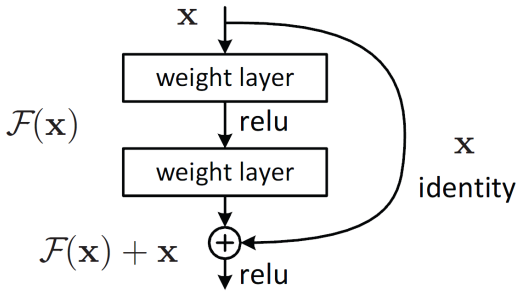


Figure 2. ResBlock Diagram (source: "Deep Residual Learning for Image Recognition" paper)

Residual Networks, most commonly known as Resnet, is a type of convolutional neural network first introduced by He Kaiming in 2015 in his paper "Deep Residual Learning for Image Recognition (Shakhadri, 2021). The most important feature of the Resnet is the ResBlock, which demonstrates the skip connection, where the input skips some of the layers and directly adds up to the output of the layers. The Resnet itself is built in many different forms: Resnet-18, Resnet-34, Resnet-50, Resnet-101, and Resnet-152. The numbers attached to "Resnet" indicate the number of layers used to build the whole neural network. The below image, *Figure 3,* illustrates the architectural structure of the Resnet-18. It expects a 32 by 32 pixel image as an input, extracting the result of the class of highest probability, out of 10 classes (The number of classes is set to 10 in this case since the CIFAR-10 dataset has 10 different classes) (Torchvision, 2022). We may clearly notice the skip connection at a certain point within the entire structure and such skip connection is processed depending on the condition set within the PyTorch class definition of the Resnet-18.

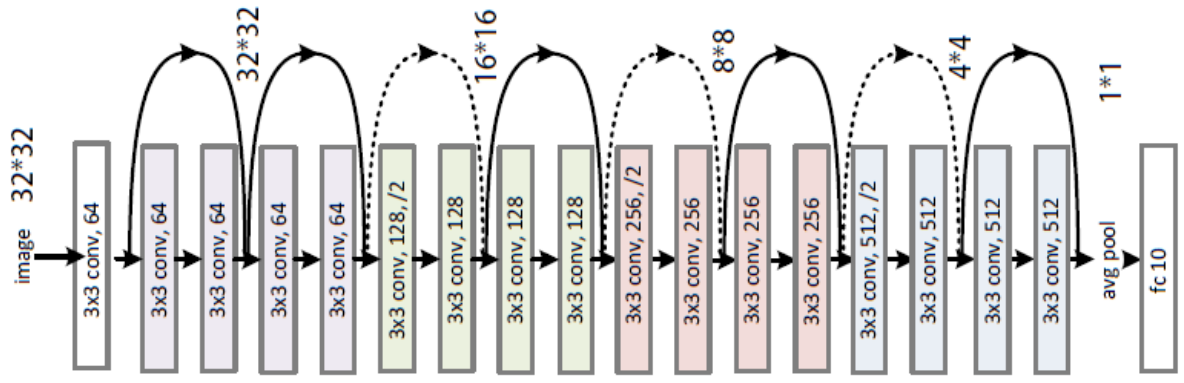


Figure 3. Resnet-18 Architectural Diagram for CIFAR-10 data (source: "Deep Residual Learning for Image Recognition" paper)

3. Body paragraph

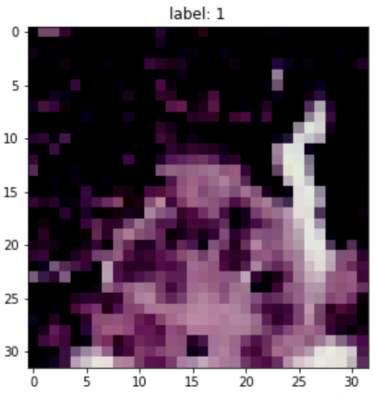
The project is structured in two major parts: image classification task using one of the convolutional neural networks and the graphical user interface along with the chatbot system. The details of the project structure will be elaborated in this following section.

3.1 Image Dataset

To begin with, the pathology scan collected from Kaggle had exactly 220,025 images (6.3GB), which we were able to check from the length of the "train_labels.csv" file. The table consists of "id" and "label" columns, where "id" column contains the image file name, and "label" column contains either the number 1s or 0s (1 representing that cancer exists within the scan). Then, we have separated the data with an 8 to 2 ratio of training and validation data. The specific number of images can be found from the numbers specified **on the right.**

```
train size:  176020
test size:  44005
```

In order to properly train the images in the PyTorch framework, we first have to create a class that can be used to call the images by passing the "Dataset" parameter (Pytorch, 2022). We called it the "ImageDataset" class.The class contains the "initializer:", "getitem" and the "length" function. This is the feature that makes PyTorch an object-oriented programming language. The initializer function allows us to self define the variables within the class, where we had dataframe values, the image directory, and the torch transform. We first pass all the parameters to this class and the "getitem" class will be able to return the image in torch tensor format with a specific transform applied image and the label value. The torch transform function is applied in order to make all the images in the same format. In our case, we resized all the images to a 32 by 32 image just in case there is an image with any other size, applied random horizontal flip with a probability of 50%, changed the pixel to tensor form, and normalized a tensor image [0.5, 0.5, 0.5] and standard deviation [0.5, 0.5, 0.5] (PyTorch, 2022).



We also created a function called "imshow" in order to display the image along with the correct label as shown **above**. We first create an object using the ImageDataset class. Unpacking the object will give us the tensor image and the label, which we defined in the ImageDataset class. The DataLoader function will use this Dataset object and separate it into a number of batches (PyTorch, 2022), which we defined as 128 for the training dataset and 64 for the validation dataset. We also applied shuffle as true for the training DataLoader, to make a stronger neural network, but false for the validation DataLoader to get rid of the confusion factor.

3.2 The Neural Networks

We have compared two different types of neural networks to choose which one gave us a better output: Custom CNN that we defined by ourselves and modified Resnet-18.

3.2.1 Custom-CNN

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

Figure 3. CNN output image calculation formula (Stack Overflow, 2019)

The whole structure of the Custom-CNN is the following:

Convolutional Layer

Conv1(in_channel = 3, out_channel=6, kernen_size=5, stride=1)

Max Pool (2, 2)

BatchNorm(6)

Conv2 (in_channel = 6, out_channel=16, kernen_size=5)

Max Pool (2, 2)

BatchNorm(16)

Fully Connected Layer

image_flatten (-1, 16 * 5 * 5)

FC1 (16 * 5 * 5, 120)

FC2 (120, 84)

FC3 (84, 2)

The first convolutional layer receives an image size of (32 x 32 x 3) with an input channel 3, which represents RGB. The first convolutional layer has a kernel size of 3 and default stride of 1, with an output channel of 6. Using the formula we can calculate the output image of this layer will be (28 x 28 x 6). Then it goes through max_pool of (2 x 2), which outputs (14 x 14 x 6), half the pixel size of the image before. Going through batch normalization will not affect the image itself, because the purpose of it is to speed up the training process or to reduce the probability of getting into a local optimum

problem. Batch normalization only normalizes the activation value of the activation function, or normalizing the output value, which naturally does not affect the output image size. The calculation continues in the same way as described above. The main part of the FC layer is where we flatten the image using the "view" function. This is to make the tensor size in one dimensional so that it can be processed in the fully connected layer. The final FC layer will output 2 probabilities for two classes of the dataset. Then in the training phase, we will choose the class that had a higher probability using the torch max function.

3.2.2 Resnet-18

```
class modified_resnet18(nn.Module):
    def __init__(self):
        super(modified_resnet18, self).__init__()
        self.resnet18 = torchvision.models.resnet18(pretrained=True)

        for param in self.resnet18.parameters():
            param.requres_grad = False

        modified_fc = nn.Linear(in_features=512, out_features=2)
        self.resnet18.fc = modified_fc

    def forward(self, x):
        return self.resnet18(x)
```

We could have created the Resnet-18 model from scratch, but in order to save time and to use the model that can output a higher accuracy measure, we have downloaded the resnet18 model from torchvision package, setting true for pretrained. The original Resnet-18 model has the final output feature as 1000, which we just have to change to "2" since we have 2 classes. As shown in the Python code, we first download the resnet18 model and modify the output feature. However, we first have to fix the feature extraction layer (Conv layer, Max Pool, Batch Normalization, etc.), so that it does not calculate the weights and just modify the output feature (Pytorch, 2022). Our group has chosen the resnet-18 mode because other deeper neural networks could have led to a longer period of training process, or even computing memory. Also, deeper layers may lead to an overfitting issue, which will be further discussed in the evaluation section of this paper.

3.3. Training and Validation phase

```
for epoch in range(num_epochs):  # loop over the dataset multiple times
    ############ Train phase ############
    net.train()
    running_loss = 0.0 # record the loss
    running_corrects = 0    # record correct prediction for classification
    for inputs, labels in tqdm(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs) ## step 1.

        loss = criterion(outputs, labels) ## step 2.

        optimizer.zero_grad() ## clear the previous gradients
        loss.backward() ## step 3. backpropagation - compute gradient
        optimizer.step() ## step 4. w = w - eta*w.grad -> update the model
        running_loss += loss.item()

        _, preds = torch.max(outputs.data, 1)
        running_corrects += torch.sum(preds == labels.data)  ## step 2 - measure accuracy
    ## train epoch loss and accuracy
    train_loss = running_loss / len(train_loader)
    train_acc = running_corrects / float(len(train_loader.dataset))
```

Figure 4.1. Training phase within the "train_model" function (PyTorch, 2022)

For both the training and the validation phase, I have created a function called “train_model”. In the training epoch, the model passed through the parameter is set as “net.train()” and records every loss and correct regardless of the batch. Within each epoch, we go through a loop of train_loader, which has a batch size of 128. We used the cross entropy loss, which is one of the most common loss functions for the classification model, to calculate the difference between the real label and the predicted output. “loss.item()” and “running accuracy” are both variables for the loss and accuracy measures each iteration. We have used them in both the training and the validation phase.

```
############ Validation phase ############
with torch.no_grad():
    net.eval()
    running_loss = 0.0  # loss for validation dataset
    running_corrects = 0    # correct prediction for validation dataset
    for inputs, labels in tqdm(valid_loader):
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = net(inputs) ## step 1
        loss = criterion(outputs, labels) ## step 2 - loss
        running_loss += loss.item()

        _, preds = torch.max(outputs.data, 1)
        running_corrects += torch.sum(preds == labels.data) ## step 2 - measure accuracy
    ## validation epoch loss and accuracy
    valid_loss = running_loss / len(valid_loader)
    valid_acc = running_corrects / float(len(valid_loader.dataset))
```

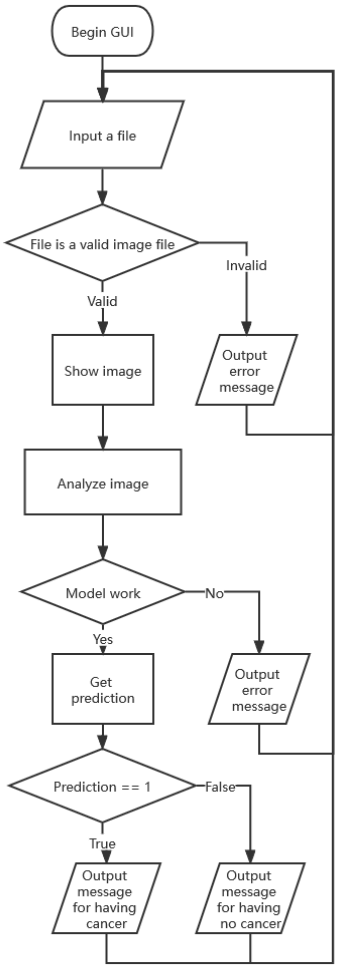
Figure 4.2. Validation phase within the train_model function

For the validation phase, we used “torch.no_grad()” and “net.eval()”, so that the model goes into the testing mode where it does not calculate new weight and the gradient (PyTorch, 2022), which means the model will not be trained. The process is almost the same as the training phase where the valid loader has a batch size of 32, since it has a smaller size of data. Another difference is that it does not go through the back propagation process, because we do not want to update the model as mentioned before.

3.4. The chatbot and GUI (Graphical User Interface)

We used a “tkinter” library in python to create a GUI, graphical user interface. After uploading a valid image on the GUI, users could get a predicted result and some advice from the GUI. The flow chart showed the main logic. A user would upload an image to the program. The program initially checked the image validation to make sure the trained model can process the file. If the image is valid, the program would show the image on the GUI and analyze the image by applying the model. Then, the program would get a prediction for the cancer and output corresponding messages on the GUI. Following this process, the program would return to the beginning and give an error message if the image is invalid or the model cannot process.

At the code level, we first created a GUI with frames, including an input frame, a canvas frame, an output frame and a medical information frame. For the input frame, it contained two parts, which were the “upload image” button and a small box. The “upload image” button would call a process function if users clicked the button. The process function consisted of generating a window for the user to upload a file, checking the file type, showing the image, analyzing the image, and outputting corresponding messages. The small box would show the file path and name of the uploaded file. The canvas frame consisted of an empty canvas which would occupy a place with suitable image size, and an image label which would show the uploaded image on the canvas. The output frame included an “about” button for the user to get information of this application, and a small box which would show a result generated from the model. The medical information frame has a label which simply shows a text of “medical Prediction”, and a box which could show a cancer message according to the generated result.



4. Results

We would like to demonstrate the result of this project in both the accuracy measure of the classification task and the implementation to the user interface.

```
Epoch [1/2], Train Loss: 0.4001, Train Acc: 0.8204, Valid Loss: 0.4058,  Valid Acc: 0.8205
Model saved to cifar_net.pt



Epoch [2/2], Train Loss: 0.3477, Train Acc: 0.8485, Valid Loss: 0.3769,  Valid Acc: 0.8352
Model saved to cifar_net.pt
Finished Training
```

Figure 5. Training Loss, Accuracy and Validation Loss, Accuracy for “Custom_CNN”

```
Epoch [1/2], Train Loss: 0.3246, Train Acc: 0.8617, Valid Loss: 0.3212,  Valid Acc: 0.8666



Epoch [2/2], Train Loss: 0.2638, Train Acc: 0.8915, Valid Loss: 0.2790,  Valid Acc: 0.8877
Finished Training
```

Figure 6. Training Loss, Accuracy and Validation Loss, Accuracy for “modified Resnet-18”

To begin with, we have used 2 epochs to train both CNN and modified Resnet-18. As shown in Figure 5, we can notice that training loss has decreased by 0.1 and the validation loss also decreased by 0.03. Both the training and the validation accuracy have increased by 0.02 and 0.01 respectively. Comparing the result of the two models, we have decided to use the modified Resnet-18 model instead of the Custom CNN we created. This is because although the rate of increase in accuracy and decrease in loss could seem similar, the more important factor that outweighs this specific explanation is the final result. Both the training validation accuracy reached almost 90% even with two epochs of training. Furthermore, the pretrained model will obviously be able to output a better classification result compared to the skeleton model built from scratch, since it has been trained with a large amount of data that normal computing power cannot process. The "pretrained" Resnet-18 model has been trained with 150GB size of ImageNet data (Torchvision, 2022), which will give an outstanding weight parameter, along with an incomparable classification result.

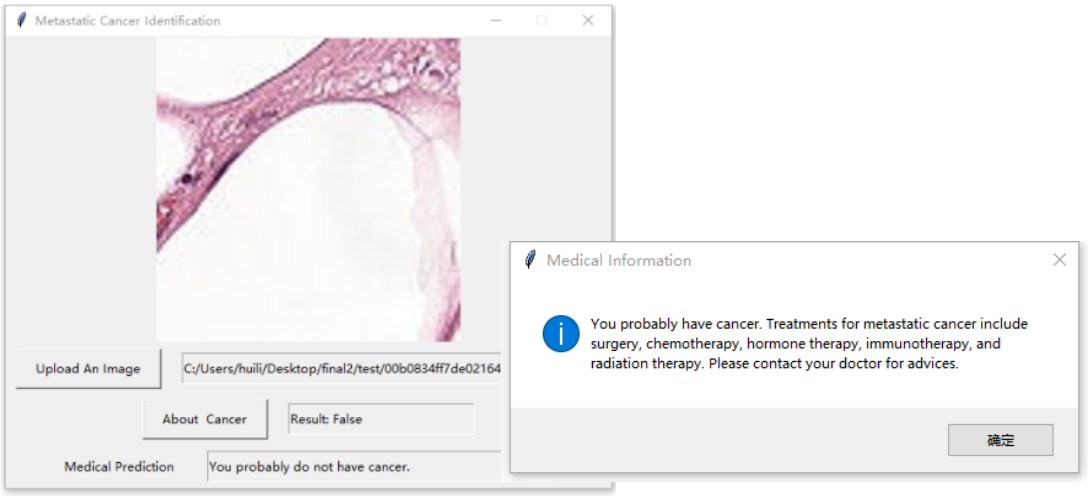


Figure 7. GUI output along with automatic medical prediction

Figure 7 demonstrates the GUI output of the medical prediction after when the user inputs their own pathology scan. As we can see, the result clearly shows the scan itself, the result of the image, and an automatically generated medical information. We further implemented a treatment service pop up that increases the interaction with the users.

5. Conclusion

In evaluation and conclusion, the result of both the neural network classification and the GUI has been successful. Our group is satisfied with the output in both tasks of the project. In order for further improvement in accuracy, we could have also trained the model with a higher number of epochs. However, this could have also led to an overfitting issue where the model only tends to predict correctly to the training data, while misclassifying the images that the model has not seen. This implies that if we tried to improve the performance within the training dataset, we could have faced a

weaker classification application in real life that could lead to a more severe situation for the actual patients. In addition, we could have improved slightly more for the chatbot system where it interacts with the users of the application by answering more complex questions, where the application itself acts as a real consultant or even a doctor.

As for an idea for further studies, our group believes that not all patients may possess a pathology scan of themselves. There might be symptoms or signs from their body that we can early detect histopathologic cancer without patients uploading the scan. If research in such areas is carried out successfully, patients will be able to diagnose a sign of a cancer, without even going to the hospital for the scan. Such symptoms may be used as a feature of the neural network. Furthermore, since we are using pathology scan, which is not an ordinary image we see, further studies could be done in structuring a specific neural network that is the best fit for this type of images, such as a combination of different neural networks, by using only using the strengths of each one of them.

6. Reference

- "Histopathologic Cancer Detection | Kaggle." *Kaggle*, 22 Jan. 2022, www.kaggle.com/c/histopathologic-cancer-detection.
- "Understanding Advanced and Metastatic Cancer." *Understanding Advanced and Metastatic Cancer*, www.cancer.org/treatment/understanding-your-diagnosis/advanced-cancer/what-is.html. Accessed 29 Jan. 2022.
- Contributor, TechTarget. "Convolutional Neural Network." *SearchEnterpriseAI*, 27 Apr. 2018, www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network.
- Shakhadri, Syed Abdul Gaffar. "What Is ResNet | Build ResNet from Scratch With Python." *Analytics Vidhya*, 8 June 2021, www.analyticsvidhya.com/blog/2021/06/build-resnet-from-scratch-with-python.
- "Torchvision.Models — Torchvision 0.11.0 Documentation." *Torchvision*, pytorch.org/vision/stable/models.html. Accessed 29 Jan. 2022.
- "Training a Classifier — PyTorch Tutorials 1.10.1+cu102 Documentation." *PyTorch*, pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html. Accessed 29 Jan. 2022.
- "How to Calculate Output Sizes after a Convolution Layer in a Configuration File?" *Stack Overflow*, 4 June 2019,

stackoverflow.com/questions/56450969/how-to-calculate-output-sizes-after-a-convolution-layer-in-a-configuration-file/56452756.

- Saha, Sumit. “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 Way.” *Medium*, 7 Dec. 2021, towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.