Programming Assignment 5, DIT867, SP4 2024

Frida Jacobson
Gothenburg University
gusjacof@student.gu.se

Leo Holmgren
Gothenburg University
leoholmgren@me.com

Abstract

To create a system that can distinguish between an image depicting skin cancer *melanoma* (class MEL) and a benign, *nevus* skin lesion (class NV), binary classification of skin lesion images was performed. The system was trained on 6426 images of equal class sizes and validated on 1252 images with the same class distribution. A convolutional neural network (CNN) was set up with PyTorch, using two convolutional layers followed by one fully connected layer. After constructing a baseline model, several techniques were used to enhance performance of the model. Batch normalization increased the validation accuracy and led to faster convergence, dropout reduced overfitting tendencies and stabilized fluctuation in validation metrics, image augmentation significantly decreased the epoch to epoch metric fluctuations, and introducing residual connections improved training efficiency as well as training stability. In addition, the model using these improvements was pretrained on a resnet model using transfer learning. The best accuracy was achieved with the pre-trained ResNet model and the accuracy for an unseen test set (1366 images) was 84.99%.

1 Introduction

Skin cancer is a relatively common cancer variant and to make treatment efficient, it is important with an early diagnosis. In particular, it is important that the healthcare system can evaluate the skin lesions correctly, to avoid situations where a malign skin lesion is thought to be benign. For this task, machine learning can be of assistance. To explore the possibilities of AI-assisted classification of skin lesions, binary classification of labeled images depicting skin lesions was performed, in order to develop a system that can distinguish between malign and benign skin lesions. The classification task was performed with a convolutional neural network (CNN) and different methods to improve performance included feature normalization, dropout, data augmentation, and adding residual connections. Additionally, a pre-trained model was utilized to explore how transfer learning can be applied to the given data. These methods are further described below.

2 Methods

2.1 Data description

The data comprised three sets of labeled color images of size 128 x 128 pixels, all sets with an equal number of each class (MEL and NV). The images were labeled by skin cancer medical professionals. The training data (n = 6426) was used to train the models, and the hyperparameters were tuned by continuous validation of the generalization performance on the validation data (n = 1252). After all models had been trained, they were evaluated using the test data (n = 1366).

2.2 Model selection

Since the data was in the form of images, an excellent choice of method for classification was a convolutional neural network (CNN). CNNs can effectively capture spatial hierarchies of features through convolutional layers, and there are many options for improving performance. The images were loaded as tensors, in batches of size 256. The baseline model was kept simple and shallow

because of limitations in computational resources and because of the small size of the training data. The loss function BCEwithLogitsLoss was chosen because it combines the sigmoid activation function with the binary cross-entropy (BCE) loss, and it is often used in binary classification tasks. The chosen optimizer, stochastic gradient descent (SGD) was chosen because of it's simplicity and our previous knowledge. Learning rate was tuned to 0.001 and weight_decay (L2 regularization) was added with a value of 0.0001. The architecture and hyperparameters of the baseline model is described in Table 2 in Appendix I - Supplementary.

2.3 Feature normalization

Normalization is a crucial technique when training a CNN, since the scale of data features is adjusted, making them more uniform across different observations. By ensuring that no single feature dominates the training process, normalization helps to level the playing field for all features, in addition to improving the performance and stability of learning algorithms. This is why common normalization methods like batch normalization, layer normalization, and group normalization are widely used for neural networks. For the given image classification task, the performance of batch normalization and group normalization was compared. Layer normalization [1] is the least suitable method for imaging tasks and was therefore not tested.

In a model, normalization should be performed after the layer activation and before the next layer. If dropout is used it should come after normalization. Batch normalization improved both validation accuracy and loss compared to group normalization, as well as training efficiency. Since batch normalization proved better for the model and the given data, batch normalization was used for all layers in subsequent models.

2.4 Dropout

Regularization methods aim to prevent overfitting, and dropout is a regularization technique specifically designed for neural networks. Dropout randomly sets a fraction of the input units to 0 at each update during training, forcing the network to not rely too heavily on any individual neuron and helps the model generalize better to unseen data.

Dropout was added for all layers to reduce over-fitting tendencies and to improve training stability, especially since the dataset is small. Since the over-fitting tendencies and the training stability were improved, dropout was added to all layers in subsequent models. The model with dropout and batch normalization can be seen in Figure 2a.

2.5 Data augmentation

The size of the dataset is small, causing problems with overfitting, poor generalization and difficulties learning complex features. Data augmentation is a technique that is well suited to solve these issues with a small dataset. Different transformations are used on the images such as cropping, flipping, rotations, color changes, blur, and many more. Usually, multiple transformations are applied and they are specified to use random changes within a specified range of values. In this way, the dataset becomes more diverse, which is especially beneficial for a small dataset.

Data augmentation was used for the training set and training performed with the baseline model using batch normalisation and dropout. The transformations used were ColorJitter, Crop and Blur. The transformations were applied when loading the images with ImageLoader followed by ToTensor. ColorJitter and Blur were chosen since the images were taken in varying lighting conditions, thereby affecting the color and sharpness. Crop was used to improve the model's ability to find the important parts of the images.

2.6 Residual connections

Using residual connections is when skip connections are added to the model, enabling the neural network to learn to adjust the input features according to residuals, i.e. the difference between the input and the output layer in a residual block.

Residual connections are often used with deep neural networks to enhance performance and training efficiency, but adding residual connections to a shallow model can still be beneficial. While the improvements might not be as large as in very deep models, there is usually improvements in training stability and gradient flow during back propagation, potentially leading to better generalization. For a description of the model used with residual connections, see Appendix 1 - Supplementary.

2.7 Transfer learning

Transfer learning involves using a pre-trained model to provide fundamental knowledge for new classification tasks. We chose ResNet-18, a deep convolutional neural network, for our implementation. To address hardware capacity challenges, we are using external GPUs with CUDA cores for efficient training and optimization.[2]. A section detailing the implementation of this training can be found in Appendix 1 - Supplementary.

2.8 Model validation

The training was performed for a default of 50 epochs, and after each epoch the values for training loss, validation loss and validation accuracy were calculated and stored. These metrics were then plotted after each training session and the results served as feedback for adjustments made to the model. To be able to compare the additions of techniques to the baseline model, the architecture and the baseline hyperparameters were not altered.

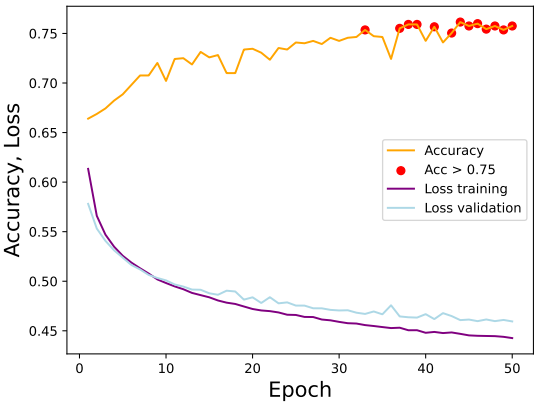
3 Results

3.1 Feature normalization

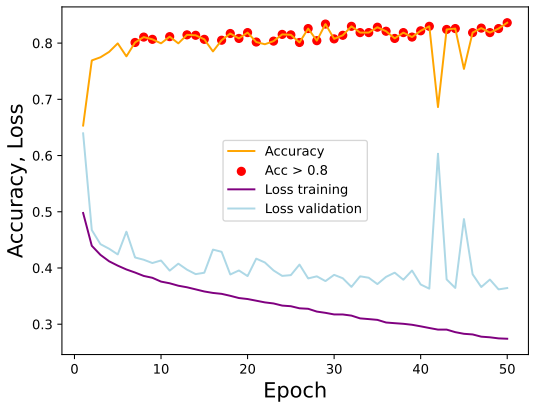
Batch normalization and group normalization were used with the baseline model to accelerate convergence and stabilize training. Comparison of the results revealed that both techniques improved the baseline model in a similar fashion. However, the batch normalization resulted in higher validation accuracy and lower training loss values, which prompted the decision to apply batch normalization to the model. A comparison between training of the baseline model with the two feature normalization techniques can be seen in Figure 1.

3.2 Data Augmentation

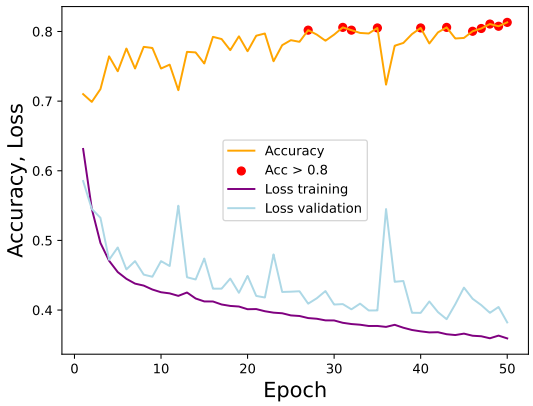
Data augmentation, using RandomResizedCrop, GaussianBlur and ColorJitter, affected validation accuracy not so much in peak values but by decreasing the fluctuations in accuracy seen in the previous models. Also, the loss values for both training and validation sets were lower compared to the same model without data augmentation (baseline model with batch normalization and dropout). It was therefore surprising that the test set accuracy for the model was lower than for the same model with non-augmented data. Training metrics and a comparison to models with residual



(a) Baseline model



(b) Addition of batch normalization to baseline



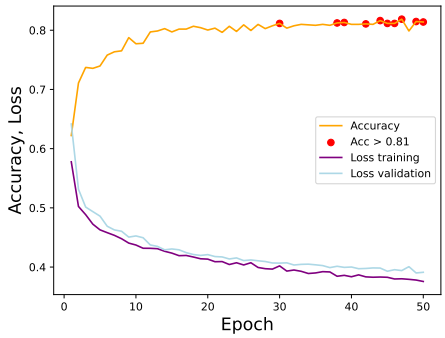
(c) Addition of group normalization to baseline

Figure 1: Comparison of training metrics for batch and group normalization against the baseline model.

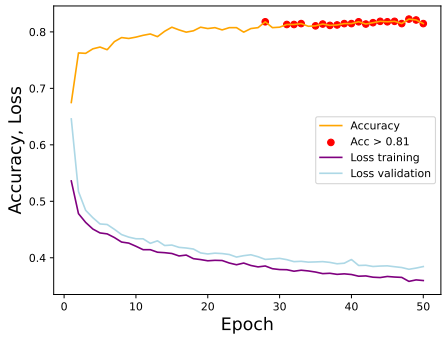
connections and a pre-trained resnet18 model can be seen in Figure 2.

3.3 Adding Residual Connections

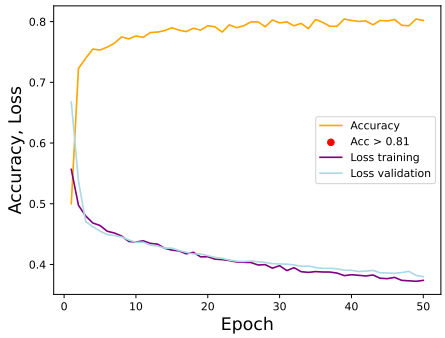
Adding residual connections in addition to batch normalization, dropout and data augmentation,



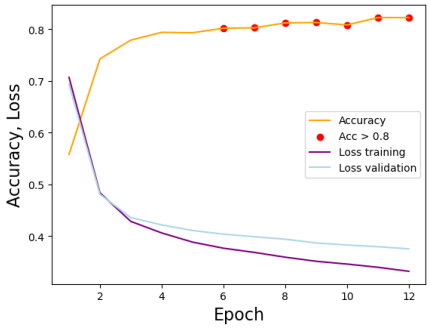
(a) Baseline model with batch normalization and dropout.



(b) Addition of data augmentation.



(c) Addition of residual connection as well as data augmentation.



(d) Pre-trained with Resnet18, using all previous techniques.

Figure 2: Comparison of the baseline model with batch normalization and dropout, vs the same model with data augmentation, and residual connections model with data augmentation.

did not affect the performance much except stabilizing the training. The major contribution was the increased training speed. The result compared to other techniques can be seen in Figure 2.

3.4 Transfer Learning

We used transfer learning with ResNet18, a well-known model, to improve efficiency. Training on a GPU made a significant difference, providing better results in just 10-12 epochs compared to training our own models on a CPU. Both the training and validation loss was decreasing, suggesting that the model was converging. This is a positive indicator as it means the model is learning effectively.The validation loss decreases but at a slower pace compared to the training loss, this is typical as the validation loss often plateaus or decreases more slowly. However, there was litle evidence of overfitting since the validation loss did not increase.

Table 1: Performance metrics of models with the applied techniques. Each model includes all techniques from previous models.

Model Name	Test accuracy	Precision	Recall	F1 Score
Baseline model	0.7599	0.7116	0.8741	0.7845
+ Batch normalisation, dropout	0.8015	0.8025	0.8171	0.7975
+ Data augmentation	0.8228	0.8050	0.8521	0.8279
+ Residual connections	0.8272	0.8041	0.8653	0.8336
+ ResNet18	0.8499	0.8329	0.8755	0.8537

4 Conclusion

In this report, we build, train and evaluate a binary classifier for prediction of cancer in images of skin lesions. A CNN model was trained, and a number of performance-enhancing techniques were accumulatively applied, such as batch and group normalization, dropout, data augmentation, residual connections, and transfer learning. The baseline model was indeed improved and the baseline model was indeed improved and the best accuracy for the test set was 82.72% achieved with the baseline model and residual connections, not far from the pre-trained model that managed to achieve an accuracy of 84.99% on the test set. To apply this tool in a healthcare environment, the tool would need to have a dramatically higher accuracy. Our work can be improved by tuning of the applied techniques and/or longer model training. Other actions that may further enhance the performance is cross-validation and trying different optimizers (such as Adam).

5 Limitations

Developing a CNN for skin cancer detection presents significant challenges and limitations. A key obstacle is the scarcity of labeled data, stemming from privacy concerns, the need for expert labeling, and the rarity of certain skin cancer types. This shortage can lead to overfitting, where the model excels on training data but falters on unseen data. Data collected from one or multiple hospitals can significantly impact the model's effectiveness. Incorporating data from multiple hospitals allows for a wider representation of imaging equipment, lighting conditions, and patient demographics, which might enhance the model's generalizability. However, it is important to recognize that this approach introduces variability that the model must handle. Similarly, the time period from which the data is collected plays a crucial role. Data from different time periods might reflect changes in imaging technology and diagnostic standards. Moreover, the wide variation in skin color and texture, influenced by factors such as ethnicity, age, and skin conditions, poses a substantial challenge. A CNN trained on a dataset lacking diversity may struggle to generalize across different populations. Therefore, a diverse dataset is crucial to ensure accurate skin cancer detection across various skin tones and textures. Furthermore, limitations in computational power can constrain the model's depth and scope for experimentation. The development of deeper models and extensive hyperparameter tuning necessitate substantial computational resources, ultimately impacting the model's performance. Efficiently training complex CNN architectures is optimal on high-performance GPUs, but limiting access to that kind of resource is probably one of the most common limitations.

6 Ethical Considerations

When handling patient data, special consideration must be taken to anonymize data and handle identification with strict confidentiality. It is also important to ensure that consent has been given by the patients to participate in the study and that there is possibility to withdraw the consent. In such a case, the model would have to be retrained without the withdrawn image. Another important aspect regarding the data is about bias and fairness; if it represents all patient groups where the model will be used. Are the images mainly from patients with either light or dark skin complexion, or are the images an appropriate mix? The same can be said about age, since skin in young patients differs from that in older patients. If the training data does not accurately represent the patients who will benefit from the classification tool, it is discrimination. When a tool is to be used in situations where the outcome has a severe impact on a person's life, it is very important that the model is robust and rigorously validated and that the accuracy is high. Additionally, it is important that the tool will be used to complement a diagnostic decision made by healthcare professionals, and not replace such a decision. Clear guidelines should be made in advance about accountability when it comes to misdiagnoses. With that said, transparency about the model is of great importance for building trust towards the AI tool among healthcare professionals, something that is usually difficult because of skepticism. Another aspect to consider is how the tool will affect the working roles for the healthcare professionals, but as previously mentioned, as long as the use of the tool is to complement rather than replace human decisions, this matter would be under control. In summary, the data must be anonymous, unbiased and given with consent, and it is difficult but important to find a good middle ground between using AI tools in cases such as diagnostics, and keeping the human mind as the main decision-maker.

References

[1] www.pinecone.io. *Build Better Deep Learning Models with Batch and Layer Normalization*. Retrieved 2024-05-13. 2024. URL: Https://www.pinecone.io/learn/batch-layer-normalization/.

[2] Xu Han et al. *Pre-trained models: Past, present and future*. Retrieved May 23, 2024. 2021. URL: https://www.sciencedirect.com/science/article/pii/S2666651021000231.

7 Appendix I - Supplementary

The baseline model architecture is described in Table 2.

Table 2: Description of the baseline model architecture.

Optimized baseline model	Details
Convolutional layer 3 → 16 *channels*	kernel size 3x3, stride 1
Activation function	ReLU
Max-pooling	2x2, stride 2
Convolutional layer 16 → 32 *channels*	kernel size 3x3, stride 1
Activation function	ReLU
Max-pooling	2x2, stride 2
Fully connected layer 32x32x32 → 100 *channels*	
Activation function	ReLU
Output layer 100 → 1	
(Sigmoid activation only when computing probas)	

Dropout was added to all layers after trying out options for feature normalization. Since fully connected layers usually have more parameters and are more prone to overfitting than convolutional layers, a higher dropout rate of 0.5 was set to prevent overfitting. Convolutional layers usually have fewer parameters and capture spatial features, so a lower dropout rate of 0.2 was used for these layers to not lose information during training.

Residual connections requires the code to be implemented as a class for the residual block and model. Using nn.Sequential with residual connections is difficult because of the skip connections required for residual blocks. We realize of course that writing all models as parts of the same class would have been more structured. The architecture for residual block can be seen in Table 3 below.

Table 3: Description of the residual block architecture.

Layer	**Details**
Residual Block 1	
Convolutional layer	$16 \rightarrow 32$ channels, kernel size 3×3, stride 2, padding 1
Batch Normalization	Applied after conv layer
Activation function	ReLU
Convolutional layer	$32 \rightarrow 32$ channels, kernel size 3×3, stride 1, padding 1
Batch Normalization	Applied after conv layer
Residual Connection	
Downsample	1×1 conv layer if stride or channels change, followed by BatchNorm
Activation function	ReLU

For transfer learning, the pre-trained model was trained using an Nvidia GeForce RTX 3070 Ti with 6144 CUDA cores. CUDA (Compute Unified Device Architecture) cores are parallel processors within NVIDIA GPUs designed to handle complex computations required for graphics rendering, scientific simulations, and machine learning tasks. In deep learning, CUDA cores are utilized to speed up the training of neural networks by parallellizing the computation of matrix operations and other tasks. The other models, not used for transfer learning, were also trained on the GPU in question if training was performed longer than 50 epochs.

8 Appendix II - Code

Python was used for the assignment, with the modules Pytorch, Torchvision, Pandas, Numpy, Matplotlib, and Scikit-learn.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import torch
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
from torch import nn, optim, argmax
from sklearn.metrics import accuracy_score, precision_score, recall_score,
↪  f1_score, confusion_matrix

#check for gpu
if torch.backends.mps.is_available():
   mps_device = torch.device("mps")
   x = torch.ones(1, device=mps_device)
   print (x)
else:
   print ("MPS device not found.")

# check gpu run time

import time

# GPU
start_time = time.time()

# syncrocnize time with cpu, otherwise only time for oflaoding data to gpu
↪  would be measured
torch.mps.synchronize()

a = torch.ones(4000,4000, device="mps")
for _ in range(200):
   a +=a

elapsed_time = time.time() - start_time
print( "GPU Time: ", elapsed_time)

from google.colab import drive
drive.mount('/content/drive')

# Load data

transform = transforms.Compose([
    transforms.ToTensor()
])
```

```
data_train = datasets.ImageFolder('a5_data_new/train', transform=transform)
trainDataLoader = DataLoader(data_train, batch_size=256, shuffle=True)

data_val = datasets.ImageFolder('a5_data_new/val', transform=transform)
valDataLoader = DataLoader(data_val, batch_size=256, shuffle=False)

data_test = datasets.ImageFolder('a5_data_test/test', transform=transform)
testDataLoader = DataLoader(data_test, batch_size=256, shuffle=False)

# Check data

print(data_train, data_val, data_test)

all_labels = [label.item() for _, labels in trainDataLoader for label in
↪  labels]
print(len([label for label in all_labels if label == 0]))

img, label = data_val[1044]
img = img.numpy().transpose(1, 2, 0) # transpose needed to set the color
↪  dimension 3 last instead of first
print(img.shape) # 128 x 128 x 3
print(label, img) # values are tensor values between 0 and 1
plt.imshow(img)

# Functions

def cnn_train(model, loss_fn, optimizer, epochs=50,
↪  trainDataLoader=trainDataLoader, valDataLoader=valDataLoader):

    acc_list = []
    loss_list = []
    loss_val_list = []

    for epoch in range(epochs):

        print('Epoch ', epoch +1)
        model.train()
        total = 0
        running_loss = 0

        for images, labels in trainDataLoader:

            optimizer.zero_grad() # Set initial gradient to 0

            output = model(images) # Use model on images
            output = output.squeeze(dim=1)
            labels = labels.float()

            loss = loss_fn(output, labels) # Calculate loss comparing
            ↪  predicted and true labels from train set
            loss.backward() # Calculate new weights depending on loss
```

```
            optimizer.step() # Update parameters

            total += labels.size(0)
            running_loss += loss.item() * labels.size(0)

        loss_train = running_loss / total
        loss_list.append(loss_train)

        model.eval()
        correct = 0
        val_total = 0
        val_running_loss = 0

        with torch.no_grad():
            for images, labels in valDataLoader: # Iterate through test
             ↪  dataset

                output = model(images) # Use model on images
                output = output.squeeze(dim=1)
                labels = labels.float()

                val_loss = loss_fn(output, labels) # Calculate loss comparing
                 ↪  predicted and true labels from test set

                probs = torch.sigmoid(output)
                y_pred = (probs > 0.5).float()
                correct += (y_pred == labels).sum().item() # Number of correct
                 ↪  pred, add to total
                val_total += labels.size(0) # Add batch size to total

                val_running_loss += val_loss.item() * labels.size(0)

            accuracy = correct / val_total # divide total correct by total
             ↪  samples
            acc_list.append(accuracy)
            loss_val_average = val_running_loss / val_total
            loss_val_list.append(loss_val_average)

        print(f'Loss: {loss_train}, Loss validation set: {loss_val_average},
         ↪  Accuracy: {accuracy}')

    print('Done!')
    return acc_list, loss_list, loss_val_list

from torch import nn, optim
import torch
import torch.nn.init as init
import torch.nn.utils as utils
from torchvision.models import resnet18

def cnn_gpu_train(resnet_model, num_epochs, loss_fn, optimizer):
```

```python
# Clear cache
torch.cuda.empty_cache()

# Custom weight initialization function
def initialize_weights(model):
    for m in model.modules():
        if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
            init.kaiming_normal_(m.weight)
            if m.bias is not None:
                init.constant_(m.bias, 0)
        elif isinstance(m, nn.BatchNorm2d):
            init.constant_(m.weight, 1)
            init.constant_(m.bias, 0)


# Modify the final layer for binary classification
num_features = resnet_model.fc.in_features
resnet_model.fc = nn.Linear(num_features, 1)  # Output 1 value for binary
↪  classification

# Reinitialize weights
initialize_weights(resnet_model)

# Move model to the device
resnet_model = resnet_model.to(device)

# Initialize loss function and optimizer
# pre_loss_fn = nn.BCEWithLogitsLoss()
pre_loss_fn = loss_fn
# pre_optimizer = optim.SGD(resnet_model.parameters(), lr=0.001,
↪  momentum=0.9, weight_decay=1e-4)  # Lowered learning rate
pre_optimizer = optimizer # Lowered learning rate

early_stopping = EarlyStopping(patience=5, min_delta=0.01)

acc_list = []
loss_list = []
loss_val_list = []

print('Training Data Statistics:')
for images, labels in preTrainDataLoader:
    images, labels = images.to(device), labels.to(device).float()
    print(f'Images mean: {images.mean().item()}, std:
    ↪  {images.std().item()}')
    print(f'Labels mean: {labels.mean().item()}, std:
    ↪  {labels.std().item()}')
    break  # Just print stats for one batch to avoid excessive logging

print('Validation Data Statistics:')
for images, labels in preValDataLoader:
```

```
    images, labels = images.to(device), labels.to(device).float()
    print(f'Images mean: {images.mean().item()}, std:
    ↪ {images.std().item()}')
    print(f'Labels mean: {labels.mean().item()}, std:
    ↪ {labels.std().item()}')
    break  # Just print stats for one batch to avoid excessive logging

print('Initial model weights statistics:')
for name, param in resnet_model.named_parameters():
    if param.requires_grad:
        print(f'Layer: {name}, mean: {param.data.mean().item()}, std:
        ↪ {param.data.std().item()}')

for epoch in range(num_epochs):
    print('Epoch', epoch + 1)
    total = 0
    running_loss = 0

    resnet_model.train()  # Set model to training mode

    for images, labels in preTrainDataLoader:
        images, labels = images.to(device), labels.to(device).float()

        if torch.isnan(images).any() or torch.isinf(images).any() or
        ↪ torch.isnan(labels).any() or torch.isinf(labels).any():
            print("Data contains nan or inf values")
            print(f"Images: {images}")
            print(f"Labels: {labels}")
            continue

        pre_optimizer.zero_grad()  # Set initial gradient to 0

        output = resnet_model(images)  # Use resnet_model on images
        output = output.squeeze(dim=1)  # Ensure proper dimensions

        if torch.isnan(output).any() or torch.isinf(output).any():
            print("Model output contains nan or inf values")
            print(f"Output: {output}")
            continue

        loss = pre_loss_fn(output, labels)  # Calculate loss comparing
        ↪ predicted and true labels from train set

        if torch.isnan(loss) or torch.isinf(loss):
            print("Loss contains nan or inf values")
            print(f"Loss: {loss}")
            continue

        loss.backward()  # Calculate new weights depending on loss

        for name, param in resnet_model.named_parameters():
```

```
            if param.requires_grad:
                if torch.isnan(param.grad).any() or
                ↪  torch.isinf(param.grad).any():
                    print(f'Gradient contains nan or inf values in layer:
                    ↪  {name}')
                    print(f'Gradients: {param.grad}')
                    continue

        # Clip gradients to prevent exploding gradients
        utils.clip_grad_norm_(resnet_model.parameters(), max_norm=1.0)

        pre_optimizer.step()  # Update parameters

        total += labels.size(0)
        running_loss += loss.item() * labels.size(0)

    if total == 0:  # If all batches are skipped, avoid division by zero
        print("All batches were skipped, setting loss_train to NaN")
        loss_train = float('nan')
    else:
        loss_train = running_loss / total
    loss_list.append(loss_train)

    resnet_model.eval()  # Set model to evaluation mode
    correct = 0
    val_total = 0
    val_running_loss = 0

    with torch.no_grad():
        for images, labels in preValDataLoader:  # Iterate through
        ↪  validation dataset
            images, labels = images.to(device), labels.to(device).float()

            if torch.isnan(images).any() or torch.isinf(images).any() or
            ↪  torch.isnan(labels).any() or torch.isinf(labels).any():
                print("Validation data contains nan or inf values")
                print(f"Images: {images}")
                print(f"Labels: {labels}")
                continue

            output = resnet_model(images)  # Use model on images
            output = output.squeeze(dim=1)  # Ensure proper dimensions

            if torch.isnan(output).any() or torch.isinf(output).any():
                print("Validation model output contains nan or inf
                ↪  values")
                print(f"Output: {output}")
                continue

            val_loss = pre_loss_fn(output, labels)  # Calculate loss
            ↪  comparing predicted and true labels from test set
```

```
                if torch.isnan(val_loss) or torch.isinf(val_loss):
                    print("Validation loss contains nan or inf values")
                    print(f"Loss: {val_loss}")
                    continue

                probs = torch.sigmoid(output)
                y_pred = (probs > 0.5).float()
                correct += (y_pred == labels).sum().item()  # Number of
                ↪  correct predictions
                val_total += labels.size(0)  # Add batch size to total

                val_running_loss += val_loss.item() * labels.size(0)

            if val_total == 0:  # If all batches are skipped, avoid division
            ↪  by zero
                print("All validation batches were skipped, setting
                ↪  loss_val_average to NaN")
                loss_val_average = float('nan')
            else:
                loss_val_average = val_running_loss / val_total
            loss_val_list.append(loss_val_average)

        accuracy = correct / val_total if val_total != 0 else float('nan')
        acc_list.append(accuracy)

        print(f'Loss: {loss_train}, Loss validation set: {loss_val_average},
        ↪  Accuracy: {accuracy}')

         # Check early stopping condition
        early_stopping(val_loss)
        if early_stopping.early_stop:
            print(f'Early stopping at epoch {epoch+1}')
            break

    print('Done!')

    return(acc_list, loss_list, loss_val_list, resnet_model)

# function for plotting training and validation metrics
def plot_loss_acc(acc_list, loss_list, loss_val_list, threshold=1,
↪  fig_name='acc_loss.pdf'):
    acc_list = np.array(acc_list)
    acc_high_x = np.argwhere(acc_list >= threshold).flatten() + 1
    acc_high_y = [acc for acc in acc_list if acc >= threshold]
    ax_len = len(acc_list)
    plt.plot(np.arange(1, ax_len+1), acc_list, color='orange')
    plt.scatter(acc_high_x, acc_high_y, color='red')
    plt.plot(np.arange(1, ax_len+1), loss_list, color='purple')
    plt.plot(np.arange(1, ax_len+1), loss_val_list, color='lightblue')
    plt.xlabel('Epoch', size=16)
```

```
    plt.ylabel('Accuracy, Loss', size=16)
    plt.legend(['Accuracy', f'Acc > {threshold}', 'Loss training', 'Loss
    ↪  validation'])
    plt.savefig(fig_name)
    plt.show()

# function to evaluate the model
def predict_evaluate(model, dataloader):
    model.eval()
    predictions = []
    true_labels = []

    with torch.no_grad():
        for images, labels in dataloader:
            logits = model(images).squeeze(1)
            probs = torch.sigmoid(logits)
            preds = (probs > 0.5).float()
            predictions.extend(preds.cpu().numpy()) # Frida undrar, varför
            ↪  inte append? och vad är .cpu()
            true_labels.extend(labels.cpu().numpy())

    predictions = np.array(predictions)
    true_labels = np.array(true_labels)

    # Calculate metrics
    accuracy = accuracy_score(true_labels, predictions)
    precision = precision_score(true_labels, predictions)
    recall = recall_score(true_labels, predictions)
    f1 = f1_score(true_labels, predictions)
    conf_matrix = confusion_matrix(true_labels, predictions)

    # Print metrics
    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
    print('Confusion Matrix:')
    print(conf_matrix)

    return predictions, true_labels

# Using the function to get predictions and evaluate
# predictions, true_labels = predict_evaluate(model, testDataLoader)


# base model batch 256, ReLu for all

# define model
model_base = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=(3, 3), stride=1,
    ↪  padding=1), # Conv layer
```

```
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3, 3), stride=1,
    ↪  padding=1), # Conv layer
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(32768, 100),
    nn.ReLU(), # Activation function
    nn.Linear(100, 1)  # Single output neuron for binary classification
    # application of sigmoid in training loop before accuracy calculation only
)

# Loss function and optimizer method
loss_fn_base = nn.BCEWithLogitsLoss()
optimizer_base = optim.SGD(model_base.parameters(), lr=0.001,
↪  weight_decay=1e-4)

acc_list_base, loss_list_base, loss_val_list_base = cnn_train(model_base,
↪  loss_fn_base, optimizer_base)

# Plot results
plot_loss_acc(acc_list=acc_list_base, loss_list=loss_list_base,
↪  loss_val_list=loss_val_list_base ,threshold=0.70, fig_name='cnn_base.pdf')


# Adding batch normalization for all layers, same as base otherwise

# Define model
model_bn = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=(3, 3), stride=1,
    ↪  padding=1), # Conv layer
    nn.BatchNorm2d(16),
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3, 3), stride=1,
    ↪  padding=1), # Conv layer
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(32768, 100), # adjust dimenstion 7200
    nn.BatchNorm1d(100),
    nn.ReLU(), # Activation function
    nn.Linear(100, 1)  # Single output neuron for binary classification
)

# Loss function and optimizer method
loss_fn_bn = nn.BCEWithLogitsLoss()
optimizer_bn = optim.SGD(model_bn.parameters(), lr=0.001, weight_decay=1e-4)

# Train model
```

```
acc_list_bn, loss_list_bn, loss_val_list_bn = cnn_train(model_bn, loss_fn_bn,
 ↪  optimizer_bn)

# Plot results, threshold 0.80 for both bn and gn
plot_loss_acc(acc_list_bn, loss_list_bn, loss_val_list_bn, threshold=0.80,
 ↪  fig_name='cnn_bn.pdf')


# Trying group normalization to see if fluctuations can be decreased
# and to compare with batch norm, lr 0.001 and 50 epochs, loss_fn and
 ↪  optimizer as before

model_gn = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=(3, 3), stride=1,
     ↪  padding=1),
    nn.GroupNorm(num_groups=4, num_channels=16),  # Group Normalization with 4
     ↪  groups
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3, 3), stride=1,
     ↪  padding=1),
    nn.GroupNorm(num_groups=8, num_channels=32),  # Group Normalization with 8
     ↪  groups
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),

    nn.Flatten(),
    nn.Linear(32 * 32 * 32, 100),
    nn.ReLU(),
    nn.Linear(100, 1)
)

# Loss function and optimizer method
loss_fn_gn = nn.BCEWithLogitsLoss()
optimizer_gn = optim.SGD(model_gn.parameters(), lr=0.001, weight_decay=1e-4)

acc_list_gn, loss_list_gn, loss_val_list_gn = cnn_train(model_gn, loss_fn_gn,
 ↪  optimizer_gn)

plot_loss_acc(acc_list_gn, loss_list_gn, loss_val_list_gn, threshold=0.80,
 ↪  fig_name='cnn_gn.pdf')


# Tried 5x5 kernel with decrease in performance, kept kernel 3x3

# model with dropout
model_bn_dropout2 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=(3, 3), stride=1,
     ↪  padding=1), # Conv layer
    nn.BatchNorm2d(16),
```

```
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Dropout(p=0.2),

    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3, 3), stride=1,
    ↪  padding=1), # Conv layer
    nn.BatchNorm2d(32),
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Dropout(p=0.2),

    nn.Flatten(),
    nn.Linear(32768, 100),
    nn.BatchNorm1d(100),
    nn.ReLU(), # Activation function
    nn.Dropout(p=0.5),
    nn.Linear(100, 1)  # Single output neuron for binary classification
)

# Loss function and optimizer method
loss_fn_bn_dropout2 = nn.BCEWithLogitsLoss()
optimizer_bn_dropout2 = optim.SGD(model_bn_dropout2.parameters(), lr=0.001,
↪  weight_decay=1e-4)

acc_list_bn_drop2, loss_list_bn_drop2, loss_val_list_bn_drop2 =
↪  cnn_train(model_bn_dropout2, loss_fn_bn_dropout2, optimizer_bn_dropout2)

plot_loss_acc(acc_list=acc_list_bn_drop2, loss_list=loss_list_bn_drop2,
↪  loss_val_list=loss_val_list_bn_drop2, threshold=0.81,
↪  fig_name='cnn_bn_drop.pdf')


# Image augmentation in addition to batch normalization and dropout

#transforms
transform = transforms.Compose([
    transforms.RandomResizedCrop(128, scale=(0.8, 1.0)), # remove random sized
    ↪  pieces of the second image dimension
    transforms.GaussianBlur(kernel_size=(5, 9), sigma=(0.1, 5)), # blur image
    ↪  randomly
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2,
    ↪  hue=0.2), # randomly change variables related to color
    transforms.ToTensor(),
])

data_train = datasets.ImageFolder('a5_data_new/train', transform=transform)
trainDataLoader = DataLoader(data_train, batch_size=256, shuffle=True)

data_val = datasets.ImageFolder('a5_data_new/val',
↪  transform=transforms.ToTensor())
valDataLoader = DataLoader(data_val, batch_size=256, shuffle=False)
```

```
# Define model (same as model_bn_dropout2, name change for clarity)
model_aug = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=16, kernel_size=(3, 3), stride=1,
    ↪  padding=1), # Conv layer
    nn.BatchNorm2d(16),
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Dropout(p=0.2),
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3, 3), stride=1,
    ↪  padding=1), # Conv layer
    nn.BatchNorm2d(32),
    nn.ReLU(), # Activation function
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Dropout(p=0.2),
    nn.Flatten(),
    nn.Linear(32768, 100),
    nn.BatchNorm1d(100),
    nn.ReLU(), # Activation function
    nn.Dropout(p=0.5),
    nn.Linear(100, 1)  # Single output neuron for binary classification
)

# Loss function and optimizer method
loss_fn_aug = nn.BCEWithLogitsLoss()
optimizer_aug = optim.SGD(model_aug.parameters(), lr=0.001, weight_decay=1e-4)

# Train model
acc_list_aug, loss_list_aug, loss_val_list_aug = cnn_train(model_aug,
↪  loss_fn_aug, optimizer_aug)

# plot training metrics for data augmentation
plot_loss_acc(acc_list=acc_list_aug, loss_list=loss_list_aug,
↪  loss_val_list=loss_val_list_aug, threshold=0.81, fig_name='cnn_aug.pdf')


# Adding residual connections
# The model architecture and hyperparamters are the same as before, with batch
↪  normalization, dropout and image augmentation

import torch.nn.functional as F

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
        ↪  stride=stride, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
        ↪  stride=1, padding=1)
```

```
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Ensure that the residual connection can be correctly added -
        ↪ dimensions
        self.downsample = nn.Sequential() # if dimensions in and out are the
        ↪ same, do nothing
        if stride != 1 or in_channels != out_channels: # otherwise apply conv
        ↪ layer without activation function
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
                ↪ stride=stride),
                nn.BatchNorm2d(out_channels)
            )

    # define method forward, the layers in the residual block (with batch
    ↪ norm)
    def forward(self, x):
        residual = self.downsample(x) # define residuals
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += residual # the input skips the two convolutions but is added to
        ↪ their output
        out = self.relu(out)
        return out

# Now that we have ResidualBlock, we can use it with the model (same model as
↪ model_aug)
class ResidualModel(nn.Module):
    def __init__(self):
        super(ResidualModel, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=(3, 3),
            ↪ stride=1, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(p=0.2)
        )

        # Put residual block between the conv layers, define dimensions
        ↪ according to that
        self.resblock1 = ResidualBlock(16, 32, stride=2)

        # Layer 2 activates residual block, and adds batch norm and dropout
        self.layer2 = nn.Sequential(
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.2)
```

```
        )

        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(32768, 100)  # Adjust the input size according to
        ↪  the final feature map size
        self.bn3 = nn.BatchNorm1d(100)
        self.fc2 = nn.Linear(100, 1)

    def forward(self, x):
        x = self.layer1(x) # conv2d -> batch norm -> relu -> maxpool ->
        ↪  dropout
        x = self.resblock1(x) # residual block conv2d -> ... conv2d -> batch
        ↪  norm -> add res -> relu
        x = self.layer2(x) # batch norm -> relu -> dropout
        x = self.flatten(x)
        x = self.fc1(x) # nn.linear
        x = self.bn3(x)
        x = F.relu(x)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.fc2(x) # nn.linear (output)
        return x

model_res = ResidualModel()

# Print model summary
print(model_res)

# Training residual connections model
model_res = ResidualModel()

# Loss function and optimizer method
loss_fn_res = nn.BCEWithLogitsLoss()
optimizer_res = optim.SGD(model_res.parameters(), lr=0.001, weight_decay=1e-4)

# Train model
acc_list_res, loss_list_res, loss_val_list_res = cnn_train(model_res,
↪  loss_fn_res, optimizer_res)

# Plotting after adding residual connections
plot_loss_acc(acc_list_res, loss_list_res, loss_val_list_res, threshold=0.81,
↪  fig_name='cnn_res.pdf') # 0.81 to compare with previous results


# Implementing pre-trained models with PyTorch/Torchvision

# Function to clear memory on local gpu.
import gc
def clear_memory(device):

    def print_cuda_memory():
        allocated_memory = torch.cuda.memory_allocated(device)
```

```
        reserved_memory = torch.cuda.memory_reserved(device)
        print(f"Allocated memory: {allocated_memory / (1024 ** 3):.2f} GB")
        print(f"Reserved memory: {reserved_memory / (1024 ** 3):.2f} GB")

    # Before clearing cache
    print("Before clearing cache:")
    print_cuda_memory()

    # Clear cache
    torch.cuda.empty_cache()
    torch.cuda.reset_peak_memory_stats(device)
    torch.cuda.reset_accumulated_memory_stats(device)
    gc.collect()

    # After clearing cache
    print("After clearing cache:")
    print_cuda_memory()
    print(torch.cuda.memory_summary())

# Early stopping function to stop when improvment is to low.
class EarlyStopping:
    def __init__(self, patience=5, min_delta=0):
        """
        Args:
            patience (int): How many epochs to wait after last time validation
             ↪  loss improved. Default: 5
            min_delta (float): Minimum change in the monitored quantity to
             ↪  qualify as an improvement. Default: 0
        """
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True


# Set up for applying pre-trained model specifik weights on dataset.
from torch.utils.data import DataLoader
from torchvision import models
```

```
# Load pre-trained model weights
weights_id = models.ResNet18_Weights.IMAGENET1K_V1

# Define the transforms using pre-trained model's normalization parameters
# Check if weights_id.transforms() needs to be used directly
if hasattr(weights_id, 'transforms'):
    resnet_transforms = weights_id.transforms()
else:
    resnet_transforms = transforms.Compose([
        transforms.Resize((224, 224)),  # Resize to the size the model expects
        transforms.ToTensor()
        # transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
        ↪  0.225])
    ])

# Define a common transform function for both training and validation datasets
data_transforms = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to the size the model expects
    transforms.ToTensor()
    # transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
    ↪  0.225])
])

pre_data_train =
↪  datasets.ImageFolder('C:/Users/Freddie/Downloads/train/train',
↪  transform=data_transforms)
preTrainDataLoader = DataLoader(pre_data_train, batch_size=256, shuffle=True)
pre_data_val = datasets.ImageFolder('C:/Users/Freddie/Downloads/val/val',
↪  transform=data_transforms)
preValDataLoader = DataLoader(pre_data_val, batch_size=256, shuffle=False)
pre_data_test = datasets.ImageFolder('C:/Users/Freddie/Downloads/test/test',
↪  transform=data_transforms)
preTestDataLoader = DataLoader(pre_data_test, batch_size=256, shuffle=False)

# Set up for applying pre-trained model specifik weights on dataset.
from torch.utils.data import DataLoader
from torchvision import models

# Load pre-trained model weights
weights_id = models.ResNet18_Weights.IMAGENET1K_V1

# Define the transforms using pre-trained model's normalization parameters
# Check if weights_id.transforms() needs to be used directly
if hasattr(weights_id, 'transforms'):
    resnet_transforms = weights_id.transforms()
else:
    resnet_transforms = transforms.Compose([
        transforms.Resize((224, 224)),  # Resize to the size the model expects
        transforms.ToTensor()
        # transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
        ↪  0.225])
```

```
    ])

# Define a common transform function for both training and validation datasets
data_transforms = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to the size the model expects
    transforms.ToTensor()
    # transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
    ↪  0.225])
])

pre_data_train =
↪  datasets.ImageFolder('C:/Users/Freddie/Downloads/train/train',
↪  transform=data_transforms)
preTrainDataLoader = DataLoader(pre_data_train, batch_size=256, shuffle=True)
pre_data_val = datasets.ImageFolder('C:/Users/Freddie/Downloads/val/val',
↪  transform=data_transforms)
preValDataLoader = DataLoader(pre_data_val, batch_size=256, shuffle=False)
pre_data_test = datasets.ImageFolder('C:/Users/Freddie/Downloads/test/test',
↪  transform=data_transforms)
preTestDataLoader = DataLoader(pre_data_test, batch_size=256, shuffle=False)

# Load and tune the pretrained model

# Set Device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Load and modify the pretrained ResNet18 model
res_model = resnet18(pretrained=True)

# Define loss function and optimizer
loss_fn = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(res_model.parameters(), lr=0.001, weight_decay=1e-4)

num_epochs = 400
not_binary = True

# Run the training
acc, loss, val_loss, trained_model = cnn_gpu_train(not_binary, res_model,
↪  num_epochs, loss_fn, optimizer, preTrainDataLoader, preValDataLoader,
↪  device)


# Training model with image augmentation in addition to batch normalization
↪  and dropout on GPU

# Set Device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Loss function and optimizer method
loss_fn_bn = nn.BCEWithLogitsLoss()
optimizer_bn = optim.SGD(model_bn_dropout.parameters(), lr=0.001,
↪  weight_decay=1e-4)
```