Final Project Report for CS 184A/284A, Fall 2024

Project Title: Classifying Skin Disease Types

List of Team Members:

- Timothy Quang Nguyen, 11722767, timotqn2@uci.edu
- Grace Wang, 51868219, gwang7@uci.edu
- Rachel Balta, 20763496, rbalta@uci.edu

1. Introduction and Problem Statement

Our system will receive images as input and output the corresponding class of skin disease for each image inputted. For our problem, the classes of skin disease include Herpes, Melanoma, Monkeypox, Sarampion (Measles), and Varicela (Chickenpox). These diseases can have similar symptoms and may be difficult to identify without formal testing. In addition, Melanoma is a type of skin cancer that can spread more easily to other parts of the body than other skin cancers, and there is currently a Monkeypox outbreak as well as a surge in Measles cases worldwide. The problem of correctly diagnosing these diseases is therefore largely prevalent and significant in the present day. Our system can assist in diagnosing these diseases via image classification of symptoms.

During our previous endeavor, we wanted to achieve a 90% or higher accuracy score using our own custom convolutional neural network (CNN) model for image classification, but after hours of frustrations and testing, we decided to use other models to help us achieve our goal. We learned it is easier to use transfer learning with existing models to achieve high accuracy scores than building our own custom CNN model from scratch. The main result of our project is we were able to achieve above 90% accuracy in classifying between the five types of skin disease given our dataset with a Data-Efficient Image Transformer model, after attempting to build a custom CNN model and experimenting with other models, namely ResNet-50, AlexNet, and a Vision Transformer.

2. Related Work

Previous approaches to address the problem of dermatological diagnosis, including diagnosing skin disease, have been machine learning and deep learning algorithms that identify and summarize repetitive features of skin lesions. Common algorithm models involve Convolutional Neural Networks (CNNs) and Support Vector Machines (SVMs), with segmentation (feature extraction) and classification methods. FCDN (Fully Convolutional Deconvolution Network) is a deep learning segmentation algorithm that achieved 99.53% accuracy in 2017, and BPNN is a deep learning classification algorithm that achieved 99.7% accuracy in 2020 for classifying skin diseases as neoplastic, inflammatory, or pigmented. The BPNN (Back Propagation Neural Network) methodology aforementioned employed a combination of SVMs and BPNNs, with an adjustment of elements by applying regularized Random Forests followed by image enhancement techniques (Zhang).

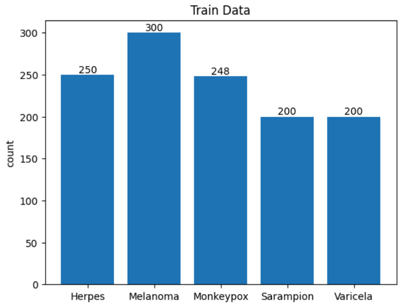
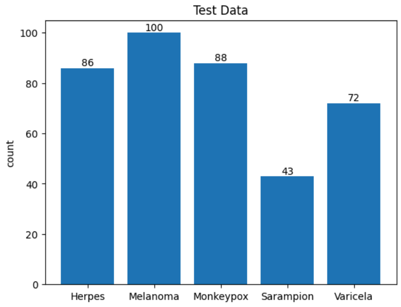
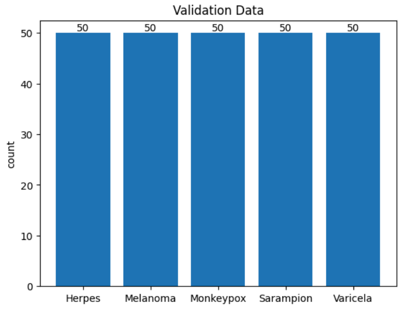
Our project fits in the context of earlier work in that the problem it addresses is classifying images as one of five skin diseases, which can be used to assist in identifying and diagnosing each of these diseases from other infections and conditions. We started with CNN models as well, that we customized or fine-tuned and trained, and then with recent advancements in transformer models, we also wanted to evaluate their performance on the problem of skin disease image classification, so we implemented some pre-trained versions of Vision and Data-Efficient Image Transformer models on the dataset. Because there already exist algorithms with over 99% accuracy from earlier work, rather than developing new algorithms, we wanted to practice and discover another model that can also accurately predict skin disease. Hence, we have our custom CNN model, as well as pre-built models like ResNet-50, AlexNet, and a few transformer models for image classification, for our project.

3. Data Sets

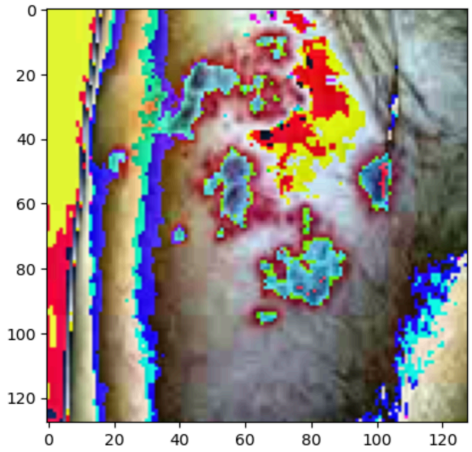
The data we will primarily be using in our research project is a dataset from Kaggle with images of several skin diseases, namely Herpes, Melanoma, Monkeypox, Measles, and Chickenpox. Each image in the dataset represents one of the five listed dermatological conditions. The majority of these images will be used to train our model, with a small portion used for validation, and the remaining for testing classifying different skin conditions. The images from the data were gathered from online image sources, the 2019 HAM10000 Challenge (skin disease dataset), Dermnet NZ (dermatological image sources), Hellenic Atlas, and Dermatological Atlas. The dataset is very lightweight, making it easy for us to conduct quick tests. However, having such a small dataset can lead to overfitting and lack of generalization, so we performed data augmentation when loading the dataset into our models.

URL to Dataset: https://www.kaggle.com/datasets/devdope/skin-disease-lightweight-dataset (Espinosa, Castilla, Lamont)

Our dataset already comes with a fixed train-validation-test partition. Therefore, when implementing our models, no preprocessing of data was done.

Training Data: 1198 images	Test Data: 389 images	Valid Data: 250
		

After transforming and loading the dataset, we checked a sample tensor image, which has size 3 x 128 x 128 (three channels for RGB). This is the sample image.



4. Description of Technical Approach

The methods and algorithms we used include initial data exploration by visualizing the distribution of image labels; utilizing training, validation, and testing splits of the data; shuffling the training dataset to minimize bias; cross-entropy loss and prediction accuracy for evaluation of the model; data augmentation to mitigate the issue of small dataset size; batch normalization to normalize the input for each layer; max pooling; ReLU activation; variations of the CNN machine learning model for image classification, including with residual blocks; variations of the transformer model; and different optimizers for feature weight learning, including Bayesian optimization, Adam, and stochastic gradient descent (SGD). We did not use any preprocessing software since our data was already processed for us.

In order to mitigate the small size of our dataset, we decided to implement data augmentation. This method increases the data variation by manipulating the given images by their position and pixel normalization. We resized, horizontally flipped, rotated, and normalized our data for each model we implemented.

To make our custom CNN model innovative and more capable, we decided to implement Bayes Optimization to automatically improve and train our hyper parameters. Similar to gradient descent, which adjusts the model weights during training, Bayesian Optimization finds the best hyperparameters. It uses the model's validation accuracy to create a probability model of its own function. This model then selects the next set of hyperparameters to evaluate, helping the search for optimal values and reducing loss. The method provided decent base parameters, but ultimately was not worth the time investment.

To achieve higher accuracies for our problem, we fine-tuned classical convolutional neural network models such as ResNet-50 and AlexNet, and a Vision Transformer (ViT) and Data-Efficient Image Transformer (DeiT). ResNet-50 uses a residual network architecture - it is a deep CNN model that uses residual learning to address the degradation of training accuracy with residual blocks, skip connections, and bottleneck layers. AlexNet is a CNN architecture consisting of eight layers - five convolutional and three fully-connected, interspersed with max pooling, dropout, normalization, and ReLU activation - that revolutionized GPU usage on training. Vision Transformers are transformers applied to sequences of image patches for computer vision applications. And Data-Efficient Image Transformers use a distillation token to produce comparable results as CNNs - they are vision transformers ideal for smaller datasets. We implemented all of these models from PyTorch's libraries using their pre-trained weights and versions.

Model Implemented	Validation Accuracy	Test Accuracy
Custom CNN	78.8%	73.3%
ResNet-50	82.0%	83.0%
AlexNet (Adam)	84.4%	83.5%
ViT	90.8%	87.7%
DeiT	91.6%	91.0%

For our ViT model, PyTorch has five different architectures for their vision transformer, and so we decided to use the base model with 12 layers and a hidden size of 768 because of our limited computing power using Google Colaboratory's free T4 GPU.

5. Software

We mainly used Python and its available machine learning libraries for our project code, such as Scikit-Learn and PyTorch, as well as numpy, matplotlib, and seaborn for data exploration. We used GitHub for version control to collaborate on Jupyter Notebook, like on the homeworks. For better computational power, we used Google Colaboratory to train and run pre-trained models. We referenced PyTorch documentation; articles explaining training and evaluation of machine learning models (CNN and pre-trained PyTorch) on Medium, DigitalOcean, and other online sources; YouTube tutorials on these models; and additional online search query results.

a) **Code/Scripts Our Team Wrote** (referencing code examples in articles/videos)

i) Data exploration: visualization and sizes/shapes
ii) Data augmentation
iii) Training process function definition
iv) Evaluation: confusion matrix, accuracy calculation (to be used with validation and testing processes) function definitions
v) Custom CNN model definition
vi) Bayesian Optimization parameters definitions
vii) Data setup (organization) for models

b) **Code/Scripts Written By Others** (from the PyTorch hub and other imported Python libraries/packages)

i) Library/package functions called in the code our team wrote, e.g., DataLoader, precision_score
ii) Bayesian Optimization
iii) Model definitions, including loss function (criterion: cross-entropy) and optimizers (Adam, SGD)
iv) ResNet-50
v) AlexNet
vi) ViT
vii) DeiT

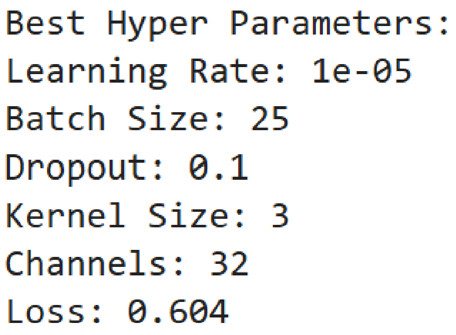
6. Experiments and Evaluation

To use our data for the first time, Timothy manually copied files from the Kaggle dataset into a directory in the same folder as our Jupyter notebook on GitHub, so that later imports of the data could just extract images from that directory.

Then prior to model experimentation, we first performed data augmentation via transforms.Compose() from PyTorch's torchvision library and converted and transformed images by resizing and randomly flipping and rotating them.

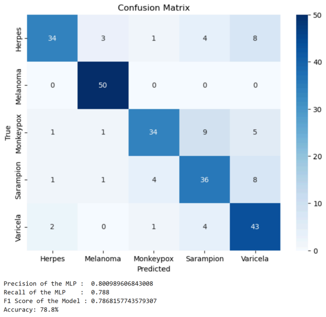
Our experiment setup for each model we implemented follows a similar pattern: (1) organize the data for the model via transforms.Compose() from PyTorch's torchvision library, i.e., resize, convert to tensor, and normalize pixel values like the pretrained model, with any additional changes as needed; (2) load the training, validation, and test datasets in parallel with three subprocesses, shuffling the training data and taking batch sizes of 25; (3) declare the model, modifying any layers to adapt to our problem with five classes; (4) set the number of epochs, learning rate, criterion, and optimizer for training; (5) train the model; (6) output the confusion matrix, precision, recall, F1 score, and accuracy of the model on the validation dataset; (7) output the confusion matrix, precision, recall, F1 score, and accuracy of the model on the test dataset. For most of our models, we trained them on 50 epochs with 0.0001 learning rate, CrossEntropyLoss criterion, and an Adam optimizer.

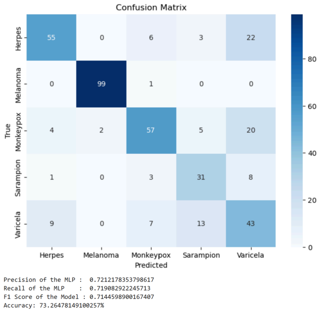
For our custom CNN model, we defined five convolutional layers and three fully connected layers, with a 0.1 dropout probability, three kernels, and 32 channels. To implement Bayesian Optimization, we defined how we wanted it to change our model, and tested it with values for the each of the following model hyperparameters: learning rate (0.01-0.00001), batch size (25, 50, 100, or 200), dropout rate (0.1-0.7), number of kernels (3, 5, or 7), and number of channels (8, 16, 32, 64, or 128), over 10 iterations, each with 20 epochs. The results of Bayesian Optimization were the following (on the right), which improved our custom CNN model accuracy by about 5%. The model complexity of five convolutional layers and three fully connected layers was chosen due to the low accuracy despite varying hyperparameters which implied a need for more model complexity. Dropout probability, kernel size, and amount of channels were decided based on the results of Bayesian Optimization after the model was deemed complex enough in retrospect to lack of hardware processing power.



```
Best Hyper Parameters:
Learning Rate: 1e-05
Batch Size: 25
Dropout: 0.1
Kernel Size: 3
Channels: 32
Loss: 0.604
```

The metrics we evaluated for each model were precision, recall, F1 score, and accuracy. Precision is the proportion of all positive classifications made that are actually positive, meaning the proportion of all images classified as one skin disease type that are actually that skin disease type. Recall is the proportion of all actual positives that were correctly classified as positives, or the true positive rate, meaning the proportion of all images of each skin disease type that were correctly classified as that skin disease type. F1 score measures the harmonic mean of precision and recall, balancing their importance, and is more indicative than accuracy in a class-imbalanced dataset. Accuracy is the proportion of all classifications that were correct and was the main evaluation criteria for our models. Our primary goal for this project was to implement a model that would achieve an accuracy of 90% or higher. Our custom CNN model performed the poorest, but we were able to practice building it and experiment with Bayesian optimization as a tool for model development. Comparing the results of all the pretrained models we implemented, AlexNet performed slightly better than ResNet-50, though ResNet-50 had a higher recall score, and DeiT performed the best out of all four. With the AlexNet model, we experimented with the SGD and Adam optimizers when training as well, and overall, the Adam optimizer produced better results than SGD (SGD with 0.005 weight decay and 0.9 momentum).

<table>
<tr><td colspan="2">Custom CNN</td></tr>
<tr><td>Validation</td><td>Test</td></tr>
<tr><td>Accuracy: 78.8%


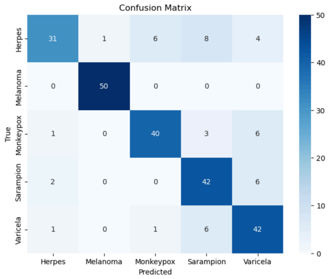
Precision of the MLP : 0.800989606843008
Recall of the MLP   : 0.788
F1 Score of the Model : 0.7868157743579307
Accuracy: 78.8%</td><td>Accuracy: 73.2%


Precision of the MLP : 0.7212178353798617
Recall of the MLP   : 0.719082922245713
F1 Score of the Model : 0.7144598900167407
Accuracy: 73.26478149100257%</td></tr>
</table>

ResNet-50

Validation

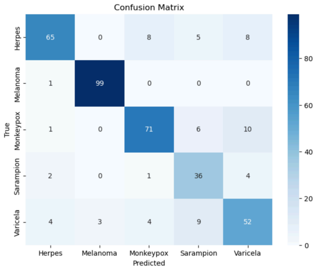
Accuracy: 82.0%



```
Precision of the MLP :  0.8306345220356818
Recall of the MLP    :  0.82
F1 Score of the Model : 0.8185346044521499
Accuracy: 82.0%
```

Test

Accuracy: 83.03%

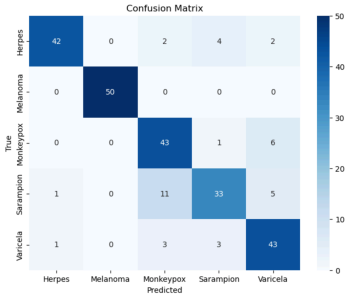


```
Precision of the MLP :  0.8103594269992337
Recall of the MLP    :  0.8224127319708716
F1 Score of the Model : 0.8125981944879829
Accuracy: 83.03341902313625%
```

AlexNet (Adam) (20 epochs, 0.00001 learning rate)

Validation

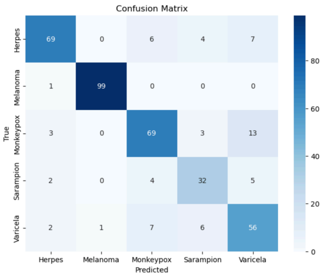
Accuracy: 84.4%



```
Precision of the MLP :  0.851218841101024
Recall of the MLP    :  0.844
F1 Score of the Model : 0.843840665391275
Accuracy: 84.4%
```

Test

Accuracy: 83.5%

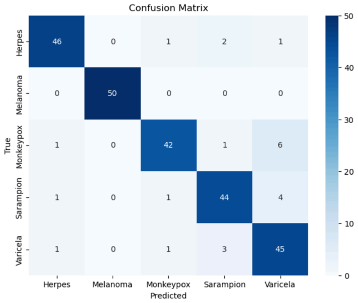


```
Precision of the MLP :  0.8181797226603429
Recall of the MLP    :  0.8196760629551327
F1 Score of the Model : 0.817805617242121
Accuracy: 83.54755784061696%
```

ViT (SGD with 0.005 weight decay and 0.9 momentum)

Validation

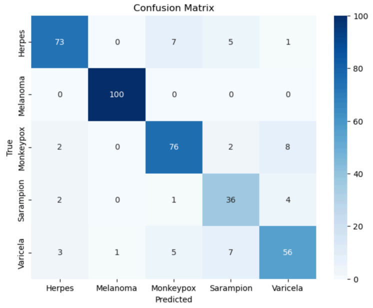
Accuracy: 90.8%



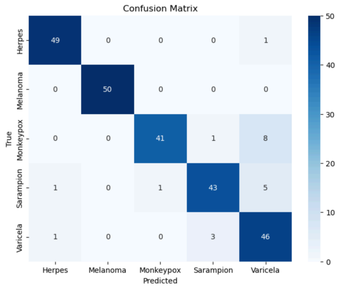
```
Precision of the MLP :  0.9111360544217687
Recall of the MLP    :  0.908
F1 Score of the Model : 0.9085120118764607
Accuracy: 90.8%
```

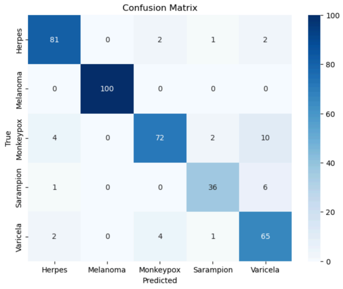
Test

Accuracy: 87.6%



```
Precision of the MLP :  0.8576251594138407
Recall of the MLP    :  0.8654921306084097
F1 Score of the Model : 0.8603639599159576
Accuracy: 87.66066838046272%
```

<table>
<tr><td colspan="2">DeiT</td></tr>
<tr><td>Validation</td><td>Test</td></tr>
<tr><td>Accuracy: 91.6%


Precision of the MLP : 0.923707014720782
Recall of the MLP   : 0.916
F1 Score of the Model : 0.9169125904074047
Accuracy: 91.6%</td><td>Accuracy: 91.0%


Precision of the MLP : 0.9053327997303899
Recall of the MLP   : 0.9000058726802914
F1 Score of the Model : 0.9009367838428239
Accuracy: 91.00257069408741%</td></tr>
</table>

7. Discussion and Conclusion

Through this project, we practiced applying CNN to a real-world application and learned about more models, different techniques for optimization, and the importance of collaboration in the machine learning field.

Our inspiration for implementing a Vision Transformer and Data-Efficient Image Transformer for our problem actually arose from hearing other groups' approaches in their projects, and while researching these models, the expression "transfer learning" appeared, all of which emphasized the significance of an open, collaborative ML/DL and AI community, and building on previous research and well-established models.

A surprising result was the magnitude of the difference in accuracy between our custom CNN model and well-researched models. We expected it would be difficult to build our own custom CNN model, but the amount of time it took to train and learn our model, compared to its results, was more expensive than originally anticipated. This lent a greater appreciation for the pre-trained models provided by PyTorch's libraries, and for the research and development process in creating those models. We also noticed that some models weren't able to surpass a certain accuracy for our problem, so it was interesting to note that some models might have limits on how much they can be improved for a given application. We also learned that more complex models generally perform better for our problem (when comparing the sizes of .pth files containing our trained models' parameters), probably because our problem is complex too.

To make more progress on this problem, we may try implementing other image classification models like UNet; further fine-tuning the parameters of the models we already have implemented, particularly for the DeiT, to try and get higher accuracy; adding a bottleneck layer to our custom CNN model to try and improve it more; and defining a more rigorous approach of evaluating models, such as keeping how the training dataset is shuffled consistent across different models, evaluating how that affects the performance and comparison of models, and looking at other evaluation metrics. It may also be helpful to try and recreate the models that achieved over 99% accuracy in earlier relevant work for our problem, with segmentation, support vector machines, and random forests.

8. Individual Contributions

Timothy created the data loaders to process and change the dataset preparing it to be used. Timothy created the functions to train models, test model accuracies, and implemented Bayesian Optimization. Timothy also created the code to produce Confusion Matrices and implemented Bayesian Optimization. Timothy also created and fine-tuned the Custom CNN model, the DeiT model, and ResNet-50 model. Timothy assisted with fine-tuning the AlexNet Model and ViT Model. Timothy assisted with detail for the project proposal.

Grace researched previous research done on the skin classification problem, wrote much of the project proposal, assisted with finalizing the presentation, implemented AlexNet with SGD, and assisted with the final project report, which was a joint effort. To implement AlexNet, Grace searched how to use alexnet in PyTorch and followed sample code for the model pre-trained on ImageNet. Because AlexNet was implemented after the custom CNN and ResNet-50 models, Grace also reused much of the code already in the project that Timothy wrote, such as for training and evaluation.

Rachel assisted with the dataset search, assisted with the project proposal, created the presentation, implemented the Visual Transformer Model in the algorithm, and assisted with the final project report which was a group effort. Because ResNet-50 and AlexNet were both working before the ViT, Rachel based off her implementation of what was already written as well as google how to call for ViT from Pytorch. Much of the code was reused from other previous model implementations.

References:

Ammar, Muhammad. “Transfer Learning with PyTorch - Muhammad Ammar - Medium.” Medium, 7 Sept. 2023, medium.com/@muhammad2000ammar/mastering-transfer-learning-with-pytorch-d1521f3a6a6e. Accessed 11 Dec. 2024.

“Data-Efficient Architectures and Training for Image Classification.” GitHub, 11 Apr. 2023, github.com/facebookresearch/deit/blob/main/README_deit.md.

Dosovitskiy, Alexey, et al. “An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale.” ArXiv.org, 3 June 2021, https://doi.org/10.48550/arXiv.2010.11929.

Espinosa, E.G., Castilla, J.S.R., Lamont, F.G. (2025). Skin Disease Pre-diagnosis with Novel Visual Transformers. In: Figueroa-García, J.C., Hernández, G., Suero Pérez, D.F., Gaona García, E.E. (eds) Applied Computer Sciences in Engineering. WEA 2024. Communications in Computer and Information Science, vol 2222. Springer, Cham. https://doi.org/10.1007/978-3-031-74595-9_10.

Hargurjeet. (2022, January 6). 7 Best Techniques to Improve the accuracy of CNN W/O Overfitting. Medium. https://gurjeet333.medium.com/7-best-techniques-to-improve-the-accuracy-of-cnn-w-o-overfitting-6db06467182f.

He, Kaiming, et al. Deep Residual Learning for Image Recognition. Cornell University, Dec. 2015, https://doi.org/10.48550/arxiv.1512.03385.

Inkawhich, Matthew. “Saving and Loading Models.” PyTorch, pytorch.org/tutorials/beginner/saving_loading_models.html. Accessed 12 Dec. 2024.

“Optimizing Vision Transformer Model for Deployment — PyTorch Tutorials 2.0.1+Cu117 Documentation.” Pytorch.org, pytorch.org/tutorials/beginner/vt_tutorial.html.

Ronneberger, Olaf, et al. “U-Net: Convolutional Networks for Biomedical Image Segmentation.” ArXiv (Cornell University), Cornell University, May 2015, https://doi.org/10.48550/arxiv.1505.04597.

Touvron, Hugo, et al. “Training Data-Efficient Image Transformers & Distillation through Attention.” ArXiv.org, 15 Jan. 2021, https://doi.org/10.48550/arXiv.2012.12877.

Zhang, Junpeng et al. “Recent Advancements and Perspectives in the Diagnosis of Skin Diseases Using Machine Learning and Deep Learning: A Review.” *Diagnostics (Basel, Switzerland)* vol. 13,23 3506. 22 Nov. 2023, doi:10.3390/diagnostics13233506.

Appendix

Output of training our custom CNN model:

```
Epoch: 1, 1.5360856453577678
Epoch: 2, 1.3992215022444725
Epoch: 3, 1.2788121476769447
Epoch: 4, 1.185199573636055
Epoch: 5, 1.092029083520174
Epoch: 6, 1.0280181504786015
Epoch: 7, 0.9583469368517399
Epoch: 8, 0.9080223763982455
Epoch: 9, 0.8606226295232773
Epoch: 10, 0.8040196436146895
Epoch: 11, 0.7709810584783554
Epoch: 12, 0.7356949485838413
Epoch: 13, 0.710061514750123
Epoch: 14, 0.6783936135470867
Epoch: 15, 0.6592731593797604
Epoch: 16, 0.62844351430734
Epoch: 17, 0.6143576676646868
Epoch: 18, 0.5622767545282841
Epoch: 19, 0.5451500757286946
Epoch: 20, 0.5482096932828426
Epoch: 21, 0.5110504515469074
Epoch: 22, 0.4930246341973543
Epoch: 23, 0.4722013510763645
```

```
Epoch: 24, 0.4563008003557722

Epoch: 25, 0.4397513335570693

Epoch: 26, 0.4408169922729333

Epoch: 27, 0.4008515203992526

Epoch: 28, 0.41515213375290233

Epoch: 29, 0.3942177888626854

Epoch: 30, 0.37453668005764484

Epoch: 31, 0.34631295191744965

Epoch: 32, 0.33865587692707777

Epoch: 33, 0.3077650365109245

Epoch: 34, 0.33104387018829584

Epoch: 35, 0.3098770085101326

Epoch: 36, 0.292105533182621

Epoch: 37, 0.28628918590644997

Epoch: 38, 0.29463189948971075

Epoch: 39, 0.2826036160501341

Epoch: 40, 0.2561951889656484

Epoch: 41, 0.25532912424144644

Epoch: 42, 0.2762870873945455

Epoch: 43, 0.2608495481933157

Epoch: 44, 0.23987820651382208

Epoch: 45, 0.23619396262802184

Epoch: 46, 0.2253456449446579

Epoch: 47, 0.22222986041257778

Epoch: 48, 0.20669180372109017
```

```
Epoch: 49, 0.2040824469489356
Epoch: 50, 0.18294540350325406
```

Output of training ResNet-50 model:

```
Epoch: 1, 1.4759794895847638
Epoch: 2, 1.2561026414235432
Epoch: 3, 1.109113768984874
Epoch: 4, 0.9999419463177522
Epoch: 5, 0.9176950653394064
Epoch: 6, 0.864570926874876
Epoch: 7, 0.8062255171438059
Epoch: 8, 0.7793282369772593
Epoch: 9, 0.7351978508134683
Epoch: 10, 0.7108428633461396
Epoch: 11, 0.6810735054314137
Epoch: 12, 0.6624477822333574
Epoch: 13, 0.6502353108177582
Epoch: 14, 0.6241559243450562
Epoch: 15, 0.6124755752583345
Epoch: 16, 0.6052146373937527
Epoch: 17, 0.5837534684687853
Epoch: 18, 0.5639183595776558
Epoch: 19, 0.568071011453867
Epoch: 20, 0.5555130715171496
Epoch: 21, 0.5391060852756103
Epoch: 22, 0.5195994631697735
Epoch: 23, 0.5261760937670866
```

```
Epoch: 24, 0.5077656358480453

Epoch: 25, 0.5090879881754518

Epoch: 26, 0.5006280777355036

Epoch: 27, 0.49869956634938717

Epoch: 28, 0.4872417102257411

Epoch: 29, 0.4758760780096054

Epoch: 30, 0.4643389377743006

Epoch: 31, 0.47344232195367414

Epoch: 32, 0.46209118360032636

Epoch: 33, 0.45669274652997655

Epoch: 34, 0.452030910179019

Epoch: 35, 0.4420078129818042

Epoch: 36, 0.45002375294764835

Epoch: 37, 0.43966136282930773

Epoch: 38, 0.4355060380573074

Epoch: 39, 0.43547170733412105

Epoch: 40, 0.43412759838004905

Epoch: 41, 0.43178171416123706

Epoch: 42, 0.4239898690332969

Epoch: 43, 0.4142509534334143

Epoch: 44, 0.4243873627856374

Epoch: 45, 0.40554628645380336

Epoch: 46, 0.4110144355023901

Epoch: 47, 0.41993974801152945

Epoch: 48, 0.3886151167874535
```

Epoch: 49, 0.40336034664263326

Epoch: 50, 0.398219616462787

Performance of AlexNet with SGD optimizer (10 epochs, 0.001 learning rate, 0.005 weight decay, 0.9 momentum):

Epoch: 1, 0.8531502485275269

Accuracy: 77.6%

Epoch: 2, 0.4337834392984708

Accuracy: 78.8%

Epoch: 3, 0.2868639199684064

Accuracy: 83.2%

Epoch: 4, 0.2173409047536552

Accuracy: 83.6%

Epoch: 5, 0.16300961728362987

Accuracy: 84.8%

Epoch: 6, 0.11849829888281722

Accuracy: 83.6%

Epoch: 7, 0.08039004887298991

Accuracy: 78.4%

Epoch: 8, 0.11270071496255696

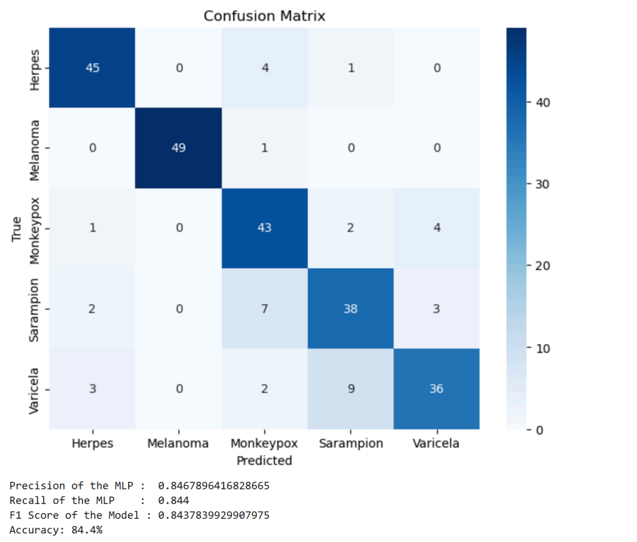
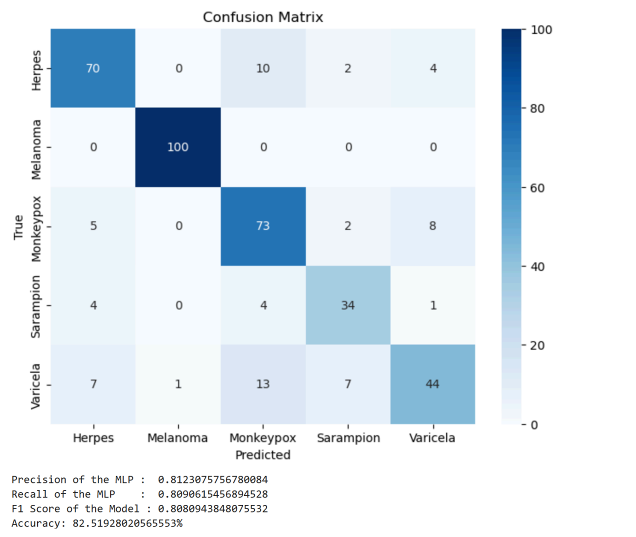
Accuracy: 83.6%

Epoch: 9, 0.04241911468367713

Accuracy: 82.0%

Epoch: 10, 0.0401855189508448

Accuracy: 84.4%

<table>
<tr><td colspan="2">AlexNet (SGD) (10 epochs, 0.001 learning rate, 0.005 weight decay, 0.9 momentum)</td></tr>
<tr><td>Validation</td><td>Test</td></tr>
<tr><td> Precision of the MLP : 0.8467896416828665
Recall of the MLP   : 0.844
F1 Score of the Model : 0.8437839929907975
Accuracy: 84.4%</td><td> Precision of the MLP : 0.8123075756780084
Recall of the MLP   : 0.8090615456894528
F1 Score of the Model : 0.8080943848075532
Accuracy: 82.51928020565553%</td></tr>
</table>

AlexNet model details:

AlexNet(

(features): Sequential(

(0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))

(1): ReLU(inplace=True)

(2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)

(3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))

(4): ReLU(inplace=True)

(5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)

(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(7): ReLU(inplace=True)

(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(9): ReLU(inplace=True)

(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

```
(11): ReLU(inplace=True)
(12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
(classifier): Sequential(
(0): Dropout(p=0.5, inplace=False)
(1): Linear(in_features=9216, out_features=4096, bias=True)
(2): ReLU(inplace=True)
(3): Dropout(p=0.5, inplace=False)
(4): Linear(in_features=4096, out_features=1024, bias=True)
(5): ReLU(inplace=True)
(6): Linear(in_features=1024, out_features=5, bias=True)
)
)
```

ViT model details:

```
VisionTransformer(
(conv_proj): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
(encoder): Encoder(
(dropout): Dropout(p=0.0, inplace=False)
(layers): Sequential(
(encoder_layer_0): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
```

```
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_1): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_2): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_3): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_4): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_5): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
```

```
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_6): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_7): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_8): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_9): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_10): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
```

```
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
(encoder_layer_11): EncoderBlock(
(ln_1): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(self_attention): MultiheadAttention(
(out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
)
(dropout): Dropout(p=0.0, inplace=False)
(ln_2): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
(mlp): MLPBlock(
(0): Linear(in_features=768, out_features=3072, bias=True)
(1): GELU(approximate='none')
(2): Dropout(p=0.0, inplace=False)
(3): Linear(in_features=3072, out_features=768, bias=True)
(4): Dropout(p=0.0, inplace=False)
)
)
)
```

```
(ln): LayerNorm((768,), eps=1e-06, elementwise_affine=True)
)
(heads): Sequential(
(head): Linear(in_features=768, out_features=5, bias=True)
)
)
```