Anthony Banaag (an285473@ucf.edu) · Jaden Barnwell (ja753523@ucf.edu)
Rylee Albrecht (ry474630@ucf.edu) · Evan Chang (ev184047@ucf.edu)

Introduction

Classifying cell types usually requires significant time and expertise, but machine learning models can streamline this process. Our models can classify images into one of 19 possible cell types (e.g., breast, lung, cervix) and determine whether they are benign or exhibit characteristics of specific cancers (e.g., metaplastic, parabasal, melanoma). This technology is particularly valuable in the medical field, especially oncology. We will compare four different models that were trained to correctly classify images of cells. Based on our evaluation metrics, we will decide which mode is most efficient and accurate when it comes to identifying cancer patterns, thus being able to help patients faster and more effectively.

Our dataset is "CellNet: First Official Beta Test Version of the CellNet Medical Image Database". The images are 128x128x3 which gives us 49,152 attributes. We have a 0.75, 0.1, 0.15 training, validation, test split with 91,927 images in the training set, 12,254 in the validation set, and 18,394 images in the test set.

Evaluation Metrics

We evaluate our models based on a few metrics, computation time, test accuracy, weighted F1 score, and their confusion matrix. We used Python's time library to get the training start time and end time, then used the equation below to get the computation time.

$$Computation\,Time = startTime - endTime$$

The test accuracy was calculated by using our trained models to classify images in the test set. We took the number of true positives + true negatives and divided that by the true positives + true negatives + false positives + false negatives as shown in the equation below, which is equivalent to the correct classifications divided by the total number of predictions made.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The weighted F1 score uses several equations. First the F1 score is calculated for each class using precision and recall below. Then the weighted F1 score is calculated by using different weights for each class, which makes each class's contribution to the overall F1 score based on its proportion in the dataset, shown in the last equation below.

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \qquad F_1^{(weighted)} = \frac{\sum_{i=1}^{n} w_i \cdot F_{1i}}{\sum_{i=1}^{n} w_i}$$

We also use the confusion matrix to visualize all true positives, true negatives, false positives, and false negatives for each class.

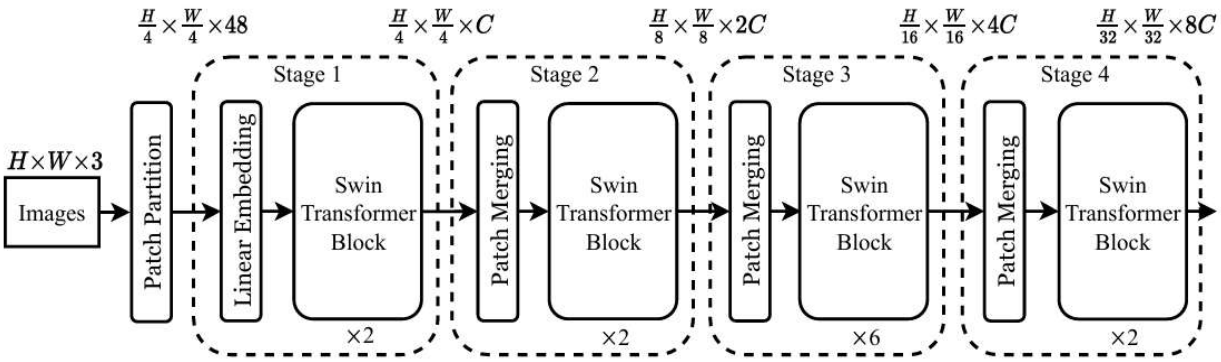
Methods

We implemented four different models in order to compare different methods and be able to recommend a high performing model. We will describe each neural network and report each of our evaluation metrics to help make an informed decision of the best model for classifying cell types.

SWIN Transformer

The SWIN Transformer, or shifted window transformer, was created to serve as a general-purpose backbone for computer vision. It is well-suited for large datasets due to its linear computational complexity due to a window based self-attention mechanism, scalability, and hierarchical design. The shifted window approach increases efficiency by confining self-attention computations to non-overlapping local windows while enabling connections across windows.

The figure below depicts the architecture of the model. It processes an image by splitting it into non-overlapping 4×4 patches, treating each as a token with an initial feature dimension of 48. A linear embedding layer projects these features to dimension C, forming Stage 1, where SWIN Transformer blocks operate at a resolution of (H/4)×(W/4). To create a hierarchical representation, patch merging layers progressively reduce the resolution and token count. Each merging layer combines 2×2 neighboring patches, downsamples the resolution by 2, and doubles the feature dimension (e.g., 2C2 in Stage 2, 4C4 in Stage 3, and so on). SWIN Transformer blocks refine the features at each stage, with final resolutions of (H/8)×(W/8), (H/16)×(W/16), and (H/32)×(W/32) for Stages 2, 3, and 4, respectively. This hierarchical structure mirrors typical convolutional networks like ResNet and VGG, enabling it to serve as a backbone for diverse vision tasks.

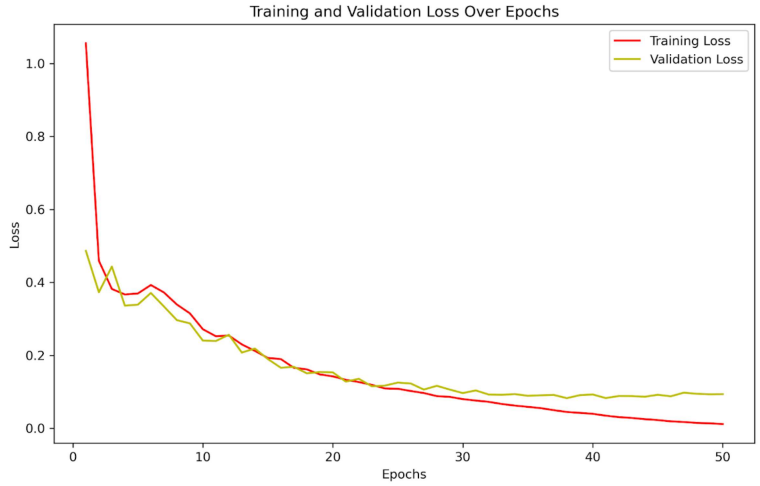


In order to run the model first you should clone both the CellNet_128 repository and the Cell_Classification repository. Detailed instructions about setting up the environment and packages can be found in the README.md in the Cell_Classification repository. Then navigate to the SWIN Transformer folder found at Cell_Classification/SWIN_transformer. From there you can run SWIN_transformer_GridSearch.ipynb and SWIN_transformer_train_eval.ipynb after changing a few root data folder variables described in the README.md.

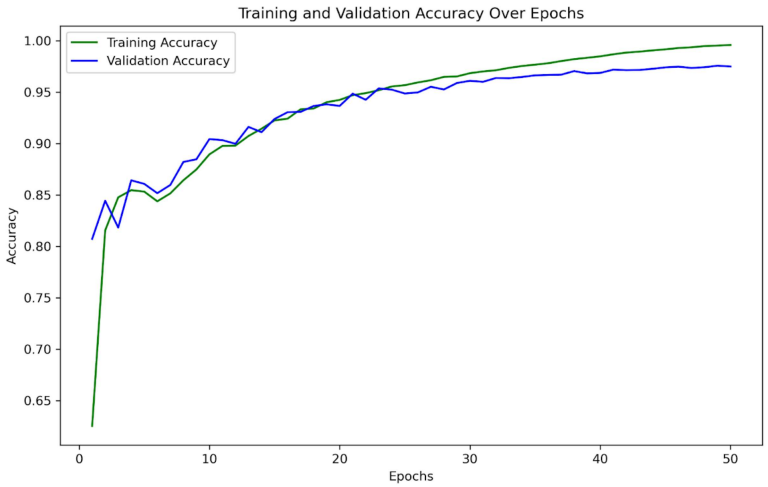
To tune the parameters, a subset of 5,000 of the images was used to allow for quicker training during optimization. First, the model was trained using all the default parameters: learning_rate=5e-5, batch_size=64, dropout=0, window_size=7, mlp_ratio=4, patch_size=4, embed_dim=96, num_heads=[3,6,12,24]. Then it was trained with a few variations, with different values for the embedding dimension, window size, patch size, and number of attention heads. This was to explore options of changing the architecture to decrease training time without sacrificing good accuracy. An embedding dimension of 48 and number of attention heads at [2, 4, 8, 16] did decrease the computation time quite a bit while maintaining similar accuracy. Changing the window size and patch size changed the accuracy more and did not help decrease training time much, so they were kept the same. Next, a grid search was performed to train the model using all combinations of three different learning rates, MLP ratios, and batch

sizes in order to find the combination that results in the best accuracy. The best accuracy came from the model with these parameters: learning_rate=1e-3, batch_size=64, dropout=0, window_size=7, mlp_ratio=4, patch_size=2, embed_dim=48, num_heads=[2, 4, 8, 16].

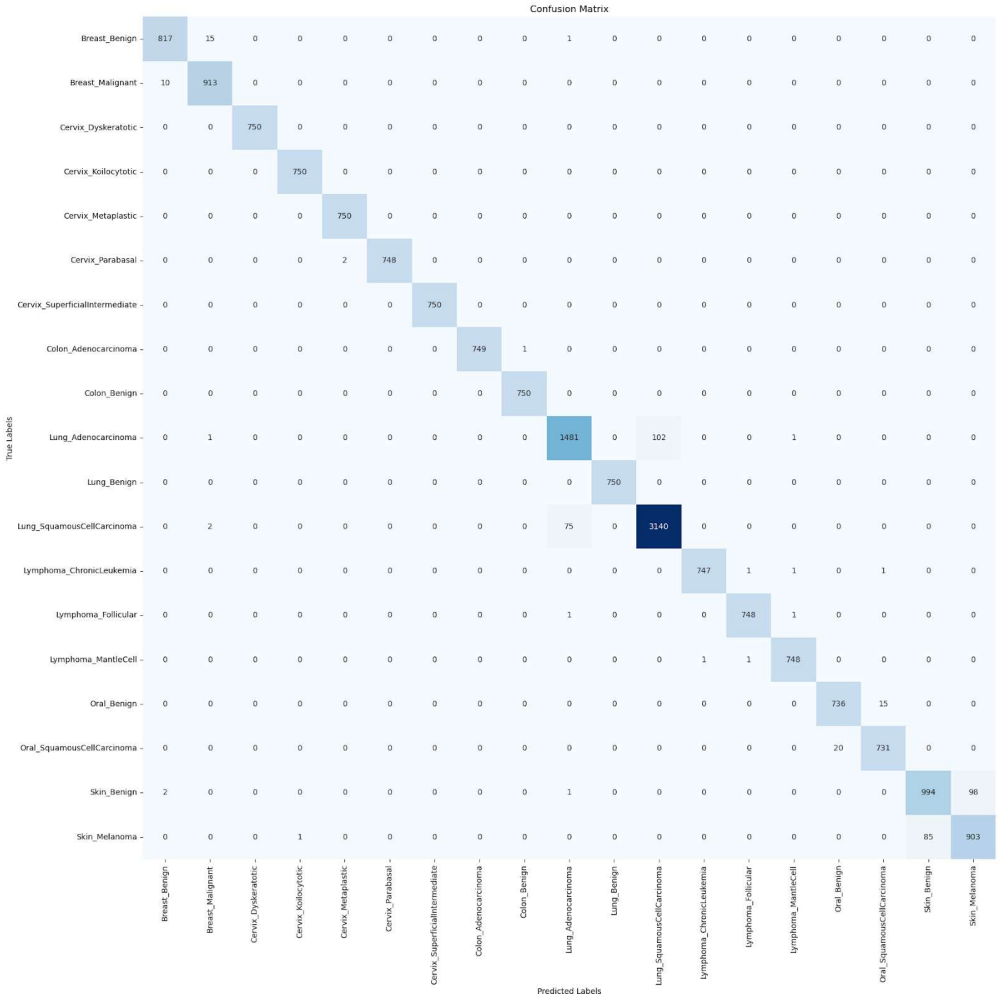
The model was then trained using the full training set with these new parameters, and saw improvements from the initial run with the default parameters that had a test accuracy of 0.9499 and a weighted F1 score of 0.9493. With the new parameters and training for 50 epochs, the model had a test accuracy of 0.97613 and a weighted F1 score of 0.97611. The training time also greatly decreased from the original 19 hours for 5 epochs (3.8 hours per epoch) to 34.13 hours for 50 epochs (40.95 minutes per epoch).



The training and validation loss was saved through training to see if there was overfitting and a dropout value should be added. Even though the training loss decreases at a steeper rate, the graph above shows a steady decrease in both training and validation loss. This combined with the fact that the validation and test accuracy are very high, shows that the model is not overfitting.



Shown in the graph above, the training and validation accuracy both increase at a consistent rate, and reach high values above 95%. This shows that the model learned the data well and could also perform well on images outside of the training data. The confusion matrix below depicts a similar outcome, with many zeros off the diagonal and a dark diagonal. This shows that of the predictions made on the test data, the majority of them were predicted correctly and contributed to the counts on the diagonal. Any false positives or false negatives are found in the lighter sections off the diagonal, showing that the model still mixed up some cells.



The results demonstrate that the SWIN Transformer excels in classifying cell types and their associated cancers, with minimal misclassifications. Additionally, its scalability makes it adaptable for training even with limited computational resources.

CNN: EfficientNet

A convolutional neural network utilizing EfficientNetB0 was implemented in order to classify the medical images into different classes. The model has a base input layer with dimensions of 128x128x3. The overall model uses compound scaling with several coefficients to balance the network's depth, width, and resolution. An additional rotation parameter can be used to rotate the image, this is helpful for pattern recognition, particularly in medical imaging. Additionally, the model uses pre-trained weights available from ImageNet. The model also

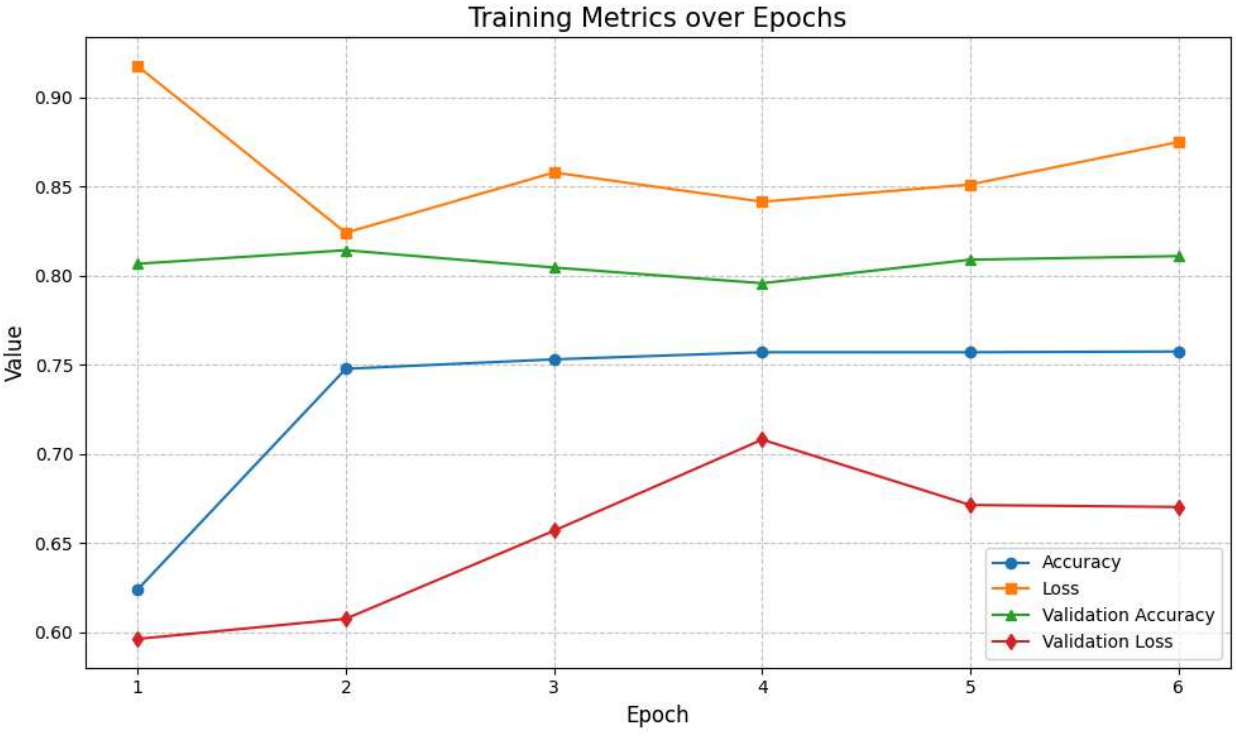
utilizes a 0.2 dropout rate, meaning that during training the model randomly will set 20% of the inputs to 0. In the classification layer, the softmax activation function was utilized as it was best suited to the multi-class classification task. Finally, loss was modelled using categorical cross entropy.

To run the optimized model, first clone both the CellNet_128 repository and the Cell_Classification repository. Detailed instructions about setting up the environment and packages can be found in the README.md for the Cell_Classification. If all packages and tools are installed according the README.md, after cloning the repository, one need only change the paths for “test_img_path”, “train_img_path”, and “val_img_path” in both the optimized.ipynb and GridSearchOnSubset.ipynb files. To view the optimization process, open GridSearchOnSubset.ipynb and select “Run All”. To train the model with the optimal hyperparameters, open optimized.ipynb and click “Run All”.

Following several substantial attempts to optimize the parameters on the raw dataset, the decision was made to downscale the images from 512x512 to 128x128 in the dataset to reduce runtime. To further reduce runtime during optimization, the dataset itself was downsampled to a subset of 5000 images. The subset images were chosen based on the ratio of images for a label to the entire dataset, such that the subset has a similar distribution of images. The images chosen for each label were selected randomly.

After downsampling both the images and the dataset, the hyperparameters could be tuned. To do so, a grid search was conducted over the following parameter settings: rotation: [0, 10, 20, 30], batch size: [16, 32, 64], dropout rate: [0.1, 0.2, 0.3], learning rate: [0.0001, 0.001, 0.01], and epochs: [3, 5, 7]. Without downsampling, the model was allowed to optimize over the grid for 6564 minutes (4.55 days) without passing the halfway mark of the grid search. Following the downsampling, optimization was able to complete in 4097 minutes (2.845 days). The grid search yielded the following hyperparameters as optimal: batch size: 64, dropout rate: 0.2, epochs: 5, learning rate: 0.01, and image rotation: 10. The accuracy for this downsampled model was 0.81199. A second experiment was conducted for epochs, holding other parameters constant at the optimal value. Epoch count varied from 1 to 15, resulting in an optimal epoch count of 6.

Following the selection of the optimal hyperparameters for the downsampled set, the model was trained on the full dataset using these parameters. Training took 4481 seconds, or 74.683 minutes. The model’s metrics versus the epoch are illustrated on the graph below.

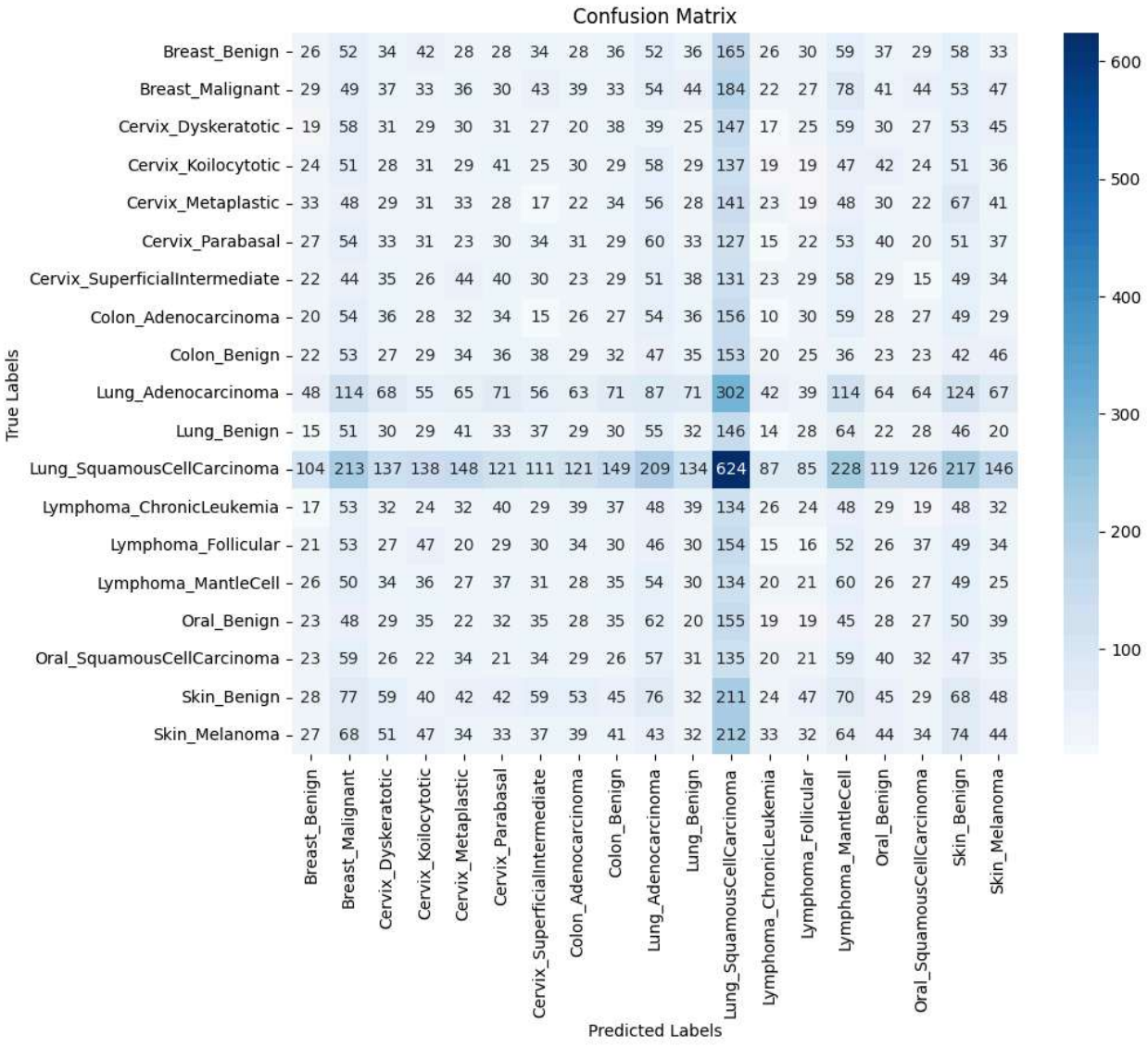


After training, the model was evaluated for the training, test, and validation sets. Results are shown in the table below.

	Accuracy	Loss
Train	0.7954	0.6188
Test	0.8024	0.6043
Validation	0.8067	0.5963

While the metrics presented in the table above appear promising, they are undercut by the model's poor F1 score. The F1 score for the EfficientNet was 0.0694, which is startlingly low, suggesting that the model is not performing as well as the accuracy and loss may make it seem.

The confusion matrix pictured below indicates that the model has high true positive and true negative rates for several of the cancer types. The model appears to be particularly well suited to classify Lung_Squamous_Cell_Carcinoma, while it appears to struggle very much with the other types, which sheds light on why the F1 score may be so low.



CNN: ResNet

The ResNet50 model uses residual connections in order to bypass layers to fix the vanishing gradient problem and build deeper architectures while keeping performance. ResNet50 is a deeper neural network and will be useful in finding more complex patterns in such a large dataset like the one we have, making the model more accurate. The 50 stands for 50 layers so there are more layers than a ResNet18 model which will hopefully improve performance.

The ResNet model utilizes the ReLU activation function throughout. This function will convert negative values to positive or zero and leave positive values untouched. This helps prevent the vanishing gradient problem. The equation for this activation function is below. As you can see the function keeps all values from a range of 0 to infinity so there are no negative values while also not shrinking the gradients. The second equation below ReLU is max pooling which is also used by the ResNet model in order to reduce the dimensions of the feature maps by taking the max value of each region of the image.

$f(x) = max(0, x) \rightarrow$ ReLU

$y_{i,j,k} = max\, x_{(i+m),(j+n),k} \rightarrow$ Max pooling

I made many decisions when deciding on optimization and loss functions. I first decided on the Cross Entropy Loss function as it is very effective when it comes to multi-classification problems. It will minimize the loss by finding the dissimilarity between the predicted and actual outcomes penalizing incorrect predictions. The equation is below with y as ground truth and p as predicted probabilities for class c and n as the number of samples.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y\, log(p)$$

I used Stochastic Gradient Descent for optimizing my model. In order to tune the model I ran my code on only 4 labels or cell image folders. Therefore, there was a lot less data for my model to train on as it was only around 15 thousand images. My model was able to run at around 1 to 2 hours each time and I altered my chosen parameters. I ended up with a learning rate of 0.01. I also chose weight decay as it helps with preventing overfitting and that was also optimized to 0.01. The final parameter was momentum which I adjusted to 0.8. Momentum is used to speed up convergence by considering past updates in direction for the SGD optimization. The equations are below where the first portion incorporates a fraction of the previous velocity to remember the direction of the previous update. The gradient term is the second portion of the first equation scaled by the learning rate and helps the update minimize the loss. The second equation updates the parameters by adding the computed velocity.

$$v_{t+1} = \mu v_t - \eta \nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

The final layer of my model uses the Softmax activation function. Softmax is used because it is useful for multi-class classification at the end when it comes to deciding on a label or outcome based on the data. It converts the raw scores into probabilities where it will give a probability for each class based on the given data. The sum of the probabilities will add up to 1. It will then choose the class with the highest predicted probability. The equation is listed below where z is the raw data, C is the total number of classes, and the base of the log is e.

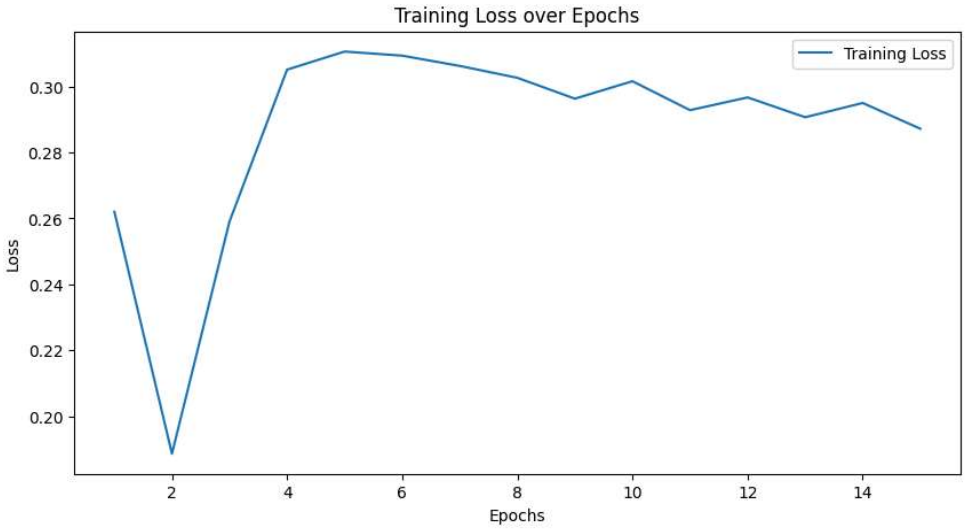
$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}, \text{ for i=1,2,3,...,C}$$

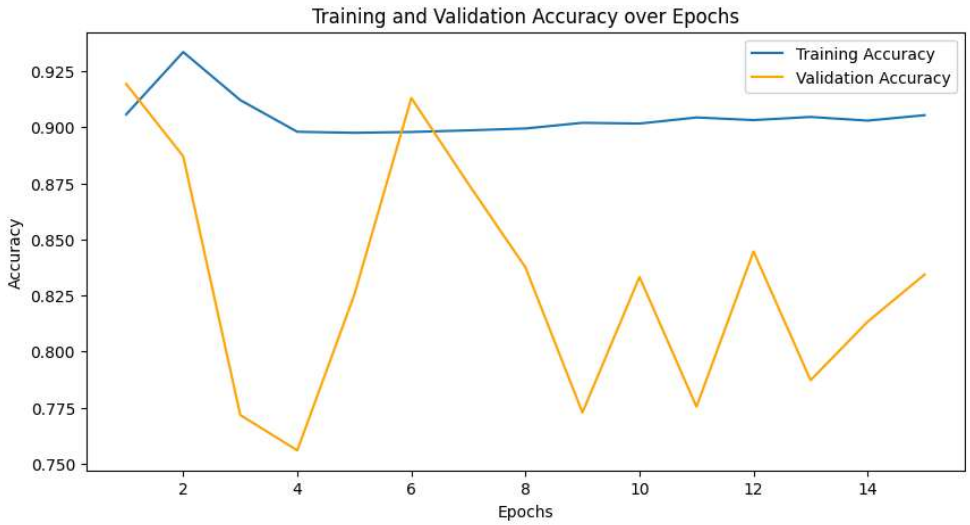
In order to run the model first you should clone the Cell_Classification repository and go to the link in the read me for the kaggle dataset. Download the dataset from kaggle and download the CellNet directory containing another CellNet folder and all the cell folders. Move this downloaded CellNet folder to the same folder as the ResNet50.ipynb notebook. Detailed instructions about setting up the environment and packages can be found in the README.md for the Cell_Classification. Then navigate over to the ResNet50 folder named Cell_Classification/ResNet50. From there you can open the ResNet50.ipynb file in VSCode and click run all. This will then run the model and provide all the outputs and graphs needed to analyze its performance.

When analyzing the data we can see that my model does pretty well as the testing accuracy when the model was tested on 20% of the original unseen data is 84 percent. The graphs below show the loss and training accuracy as I trained my model over 15 epochs. Over the epochs the model tends to learn a little bit each time as the loss is gradually decreasing and the accuracy is gradually increasing.

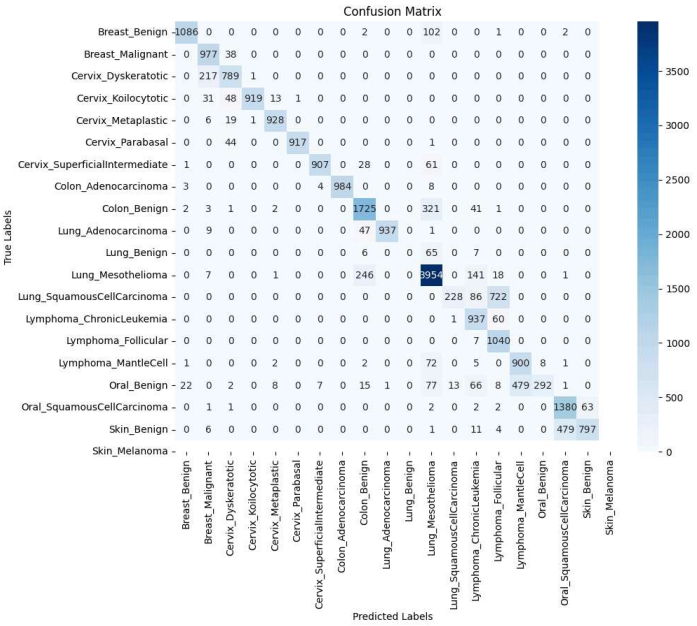
The time that it took for this run code was 5 hours, 22 minutes, and 38 seconds.

Final Test Accuracy on unseen data: 0.8386





I also plotted the confusion matrix and it is pretty reflective of a 84 percent prediction accuracy. As you can see most of the cell images are correctly predicted when the model was tested as there is a pretty solid diagonal line. There are only a few mistakes that are visible off of the line where the model mistakes a certain cell for another. The F1 score is also pretty good at 0.8260. This means that there is a pretty good balance between precision and recall but there is still some room for improvement of my model. This F1 score will need to increase as it is for medical images when identifying cells and usually they want around 0.9 F1 or above to make sure there are few false positives or negatives.



Weighted F1 Score: 0.8260

To finish off the report on my ResNet50 model I think there could be a few changes for sure. First off I think my model would benefit greatly from an increase in the number of epochs. I would try to increase the epochs from 15 to 40. This would allow my model more time to adjust

its parameters and test out different weights to be able to find the best configuration and hopefully learn enough without overfitting. The loss was continually going down in the training graph so I think that an increase in epochs would allow for an even lower loss and better performance in testing.

Multilayer-Perceptron

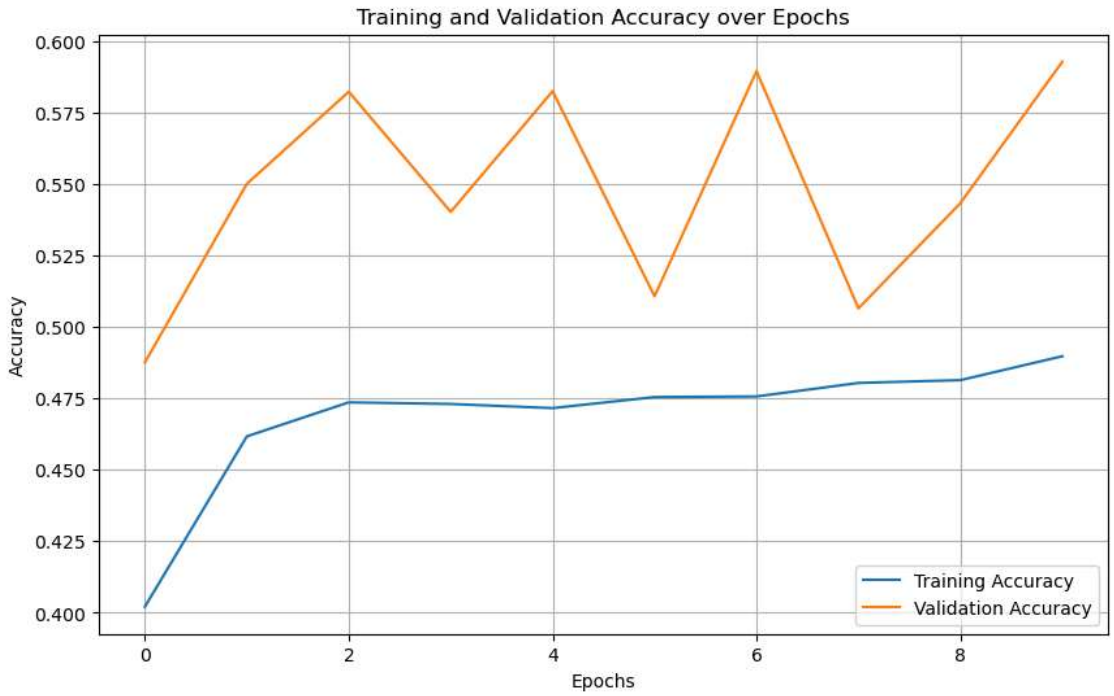
The Multilayer Perceptron can be explained via a basic structure. There are three layers, an input layer. It has an input layer that receives data input, then goes to the hidden layers that process this data. This model can have multiple layers, but the typical model has around two-three. Lastly there is an output layer that outputs the final result, and can have either one neuron for binary output, or multiple if there's multi class classification.

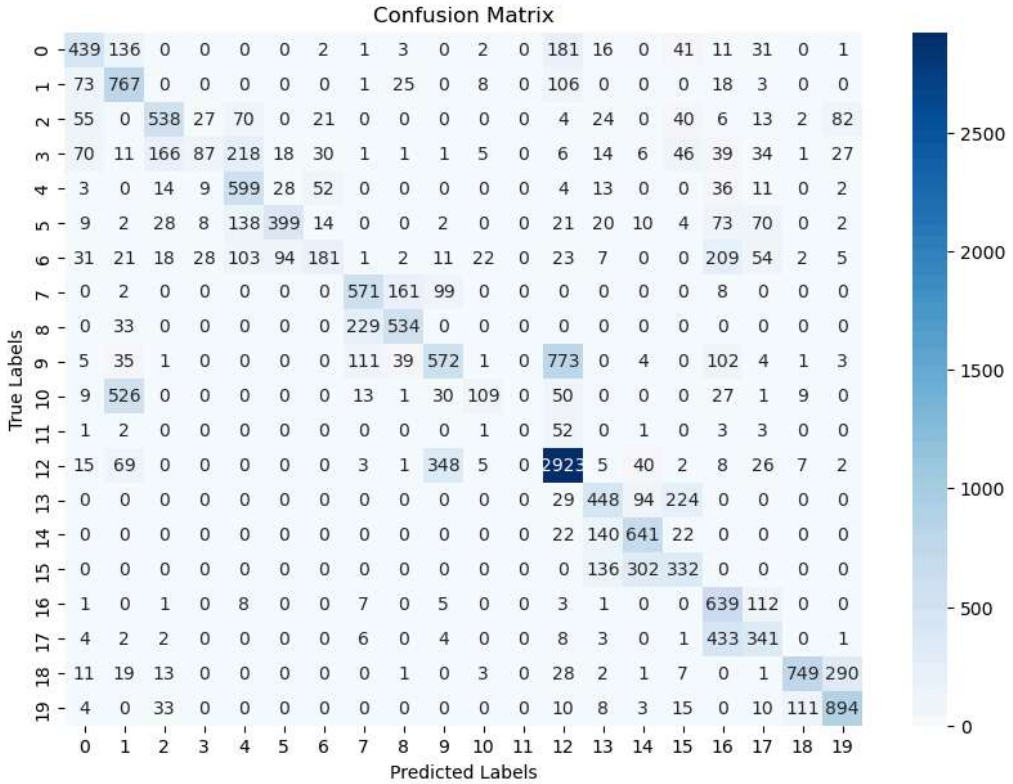
Going into more of a mathematical scope, the input layer can be defined as $x = [x_1, x_2, .. x_n]$. This is the input vector of size n, n being the number of features. Afterwards, we move onto the hidden layers, and all the hidden layers are connected with their associated weights and biases. $W^l, l, b^{(l)}$ represent the weight for that layer, layer, and bias for that layer respectively. The data is moved throughout the network via passing the data from one neuron and layer to another and this is represented mathematically as $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$. $a^{(l-1)}$ represents the activation from the previous layer, and $z^{(l)}$ represents the weighted sum of inputs into the layer. Then this output is passed through the ReLu activation function. The output layer is computed the same as the hidden layer but instead replaced l with L, the final layer.

Next, we compare that output to a loss function, which measures the differences between the trained output versus a target output. For this model we used a categorical cross-entropy function which is denoted as $L = \sum_{i=1}^{C} y_i log(p_i)$, L being the loss, C being the number of classes, y_i being the label, and p_i being the predicted probability. Lastly, we have backpropagation which can be categorized as forward and backward passes. This is denoted by $\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(l)}} * \frac{\partial a^{(l)}}{\partial z^{(l)}} * \frac{\partial z^{(l)}}{\partial W^{(l)}}$, where L is the loss function.

To run this software, you are going to want to make sure you have the CellNet dataset downloaded either locally or on your cloud computing software. Afterwards you want to take the “MLP Implementation.ipynb” file and paste it into a folder that can access it. Next, you want to modify the “img_path” variable to the file path that has the folder in it. Afterwards, just run the python notebook and it will run. If you are running locally ensure that you have enough disk space to temporarily store all the data, as that was an issue that ruined some of my overnight training sessions due to lack of memory. To free up some disk space from training, just close out your local editor and it should clear out the data stored temporarily.

After tuning and testing my code with different number of layers, neurons per layers, optimizers, learning rates, activation functions, dropout functions, batch normalization functions, and early stopping functions I have come to the conclusion that MLP transformers, while easy to setup and initialize, provide a poor performance for the complex dataset that we are using. The CellNet dataset has thousands of images and twenty classes. Multiclass Image classification is a problem that MLP does not excel at, rather they excel at smaller datasets that are more numerical or binary. When I was training and testing initially, I noticed that the increase in the amount of classes drastically negatively affected my performance. When it was only two classes I was averaging out around 80-85 percent accuracy, but with the full dataset these are the metrics I have been achieving. At ten epochs with the full dataset my best accuracy is 0.4897, val_accuracy is 0.5927, and weighted F1 score is 0.5755.





Overall, it performed rather poorly with the whole dataset. While I was achieving success with the smaller dataset I was working with, this huge dataset seemed to be more than it can handle. I also noticed that as I added the increasing amounts of Dense hidden layers, I was experiencing diminishing returns at the cost of extremely higher compute time. I found that adding BatchNormalization and Dropout functions generally improved the accuracy from 5-10

percent, as well as the adding the Adam optimizer and reLu activation functions leading to more consistencies with the accuracies. I also noticed when I tried implementing an l2 kernel regularizer, my overall accuracy plummeted. I am unsure why, but that is what I noticed. I also noticed that in general, while my training accuracy was high, my validation accuracy was generally low. Overall it just seems that the Multilayer Perceptron, while easy to implement, struggles with large scale tasks such as classifying our dataset. This makes sense due to sources such as GeeksforGeeks, Medium, Datacamp, and Artificial Intelligence Stack Exchange only really use MLP for smaller scale tasks (such as identifying digits 0-9) and recommend using CNN's for any task remotely more complicated that the basic examples that they give. While I do believe that the work can be further improved by scaling down the dataset and getting into the nitty gritty when it comes to tuning the hyperparameters, it will just end up with diminishing returns as well as not really solve the problem our team is facing.

Conclusion

The model that performed the greatest was the SWIN transformer, with the ResNet trailing close behind. This makes sense and aligns with our background research that we did at the start of the project. The SWIN transformer improves best on large scale datasets such as the one we chose. Due to its hierarchical approach with shifted windows, it made the model highly accurate. Due to the environment of our problem, it played to the SWIN transformers strengths of requiring large datasets to reach its peak performance, having versatile scalability, and handling a large load of images. Over its epochs, it had the highest and most consistent accuracies between its training and validation set as seen in the graphs as well as test accuracy of 0.97613. It also has the highest weighted F1 score of 0.97611, as well as having the cleanest confusion matrix out of the group.

Future Work

We believe our work can be further improved by taking the SWIN transformer and making that model our sole focus. If we can continue to filter through the data, be able to use more pixels of the images, and have access to more powerful computing processes we would be able to improve our classification software.

Our first issue was the data. Our dataset is around ten gigabytes of images, and taking the time to figure out which classes we wanted to test, which images we were going to train on, and how many pixels to use was a lengthy process. Our team having more time to do a deeper dive into the data will allow us to further feed our model better and more efficient data.

Our hardest challenge that we have come across is computation time and understanding which resources we can use to solve this problem. Initially we used Kaggle's cloud services, but as we expanded our training and testing to more fields, Kaggle ran into issues either timing out or our team running out of computation time on their servers. After experiencing these issues for days, we all resolved to run the model on our local machines. While it led to more consistent and uninterrupted runs, it made our computation time quite long.

We also believe that more epochs can allow our performance to be better, however as previously stated, this will increase our already lengthy computation time with the restricted resources that we have.

Contributions of team members

Rylee Albrecht prepared the dataset by resizing images and organizing them into training, testing, and validation subsets. She also implemented the SWIN-transformer model and wrote part of the reports, readme, and presentation slides.

Evan Chang implemented the EfficientNet CNN and wrote parts of the report, readme, and slides.

Jaden Barnwell implemented a ResNet50 CNN model and wrote parts of the report. He also contributed to the readme document and the presentation slides as well.

Anthony Banaag implemented the Multilayer Perceptron Model, wrote parts of the report, initialized the slides, edited the video submissions, and was the team's spokesperson.