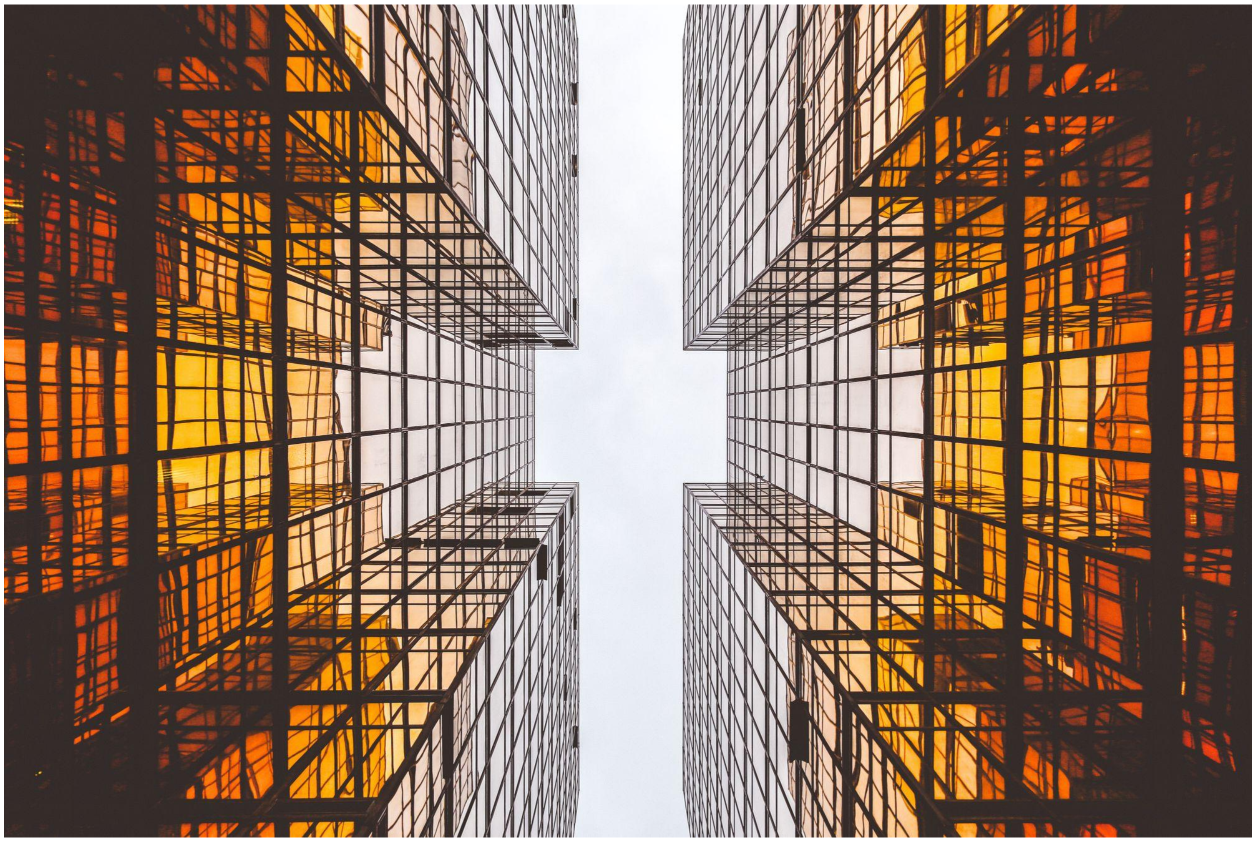
Machine Learning Course

Experimental Project

Neural Networks for Image Classification



Nicolò Cesa-Bianchi

Author:

Hamed Davoodi

11267A

Introduction

In the context of Machine Learning, the task is to classify two seemingly distinct yet occasionally perplexing categories: muffins and Chihuahuas. In the pursuit of this objective, I am presented with a dataset partitioned into training and testing subsets, adhering to a classic 80-20 split. This corpus comprises a total of 5,917 images, each holding the potential to belong to either of the two classes.

The aim of this endeavor is to construct a robust and accurate image classifier that can distinguish with precision between these two classes of images. Through the careful examination of these data points, my aspiration is to harness the power of data science and machine learning to unveil the subtle visual cues that differentiate these categories. This classification task not only poses an interesting challenge but also has practical applications in a variety of domains, from automated content tagging to enhancing image recognition systems.

In the following sections, I will present the details of the Preprocessing, the methodologies employed, and the results obtained, all in pursuit of our objective to master the classification between muffins and Chihuahuas.

CNN, ANN, RNN, etc. Which one should I choose?

Convolutional Neural Networks (CNNs) have emerged as the go-to choice for image-related tasks for several compelling reasons. Firstly, CNNs are inherently designed to handle grid-like data, making them perfectly suited for image data, where the spatial relationships among pixels or features are crucial. The use of convolutional layers allows CNNs to efficiently capture hierarchical patterns and features, starting from simple edges and shapes to more complex visual structures. Moreover, CNNs are capable of automatically learning and extracting meaningful features from the data, reducing the need for hand-crafted feature engineering. Additionally, CNNs often exhibit superior performance on image classification tasks compared to other types of neural networks, thanks to their ability to scale effectively with deeper architectures and larger datasets.

The Architecture of CNN

At its core, a Convolutional Neural Network (CNN) is a specialized type of neural network architecture meticulously designed for the purpose of handling image data. What sets CNNs apart is their distinct layer structure. They consist of convolutional layers, pooling layers, and fully connected layers. Convolutional layers are the workhorses of CNNs, where small filters slide over the input data to extract local features and patterns. These filters learn to detect various features like edges, corners, and textures. Pooling layers reduce the spatial dimensions of the data, which helps in retaining important information while reducing computational complexity. Finally, fully connected layers process the high-level features extracted by the previous layers and make the final classifications. This hierarchical architecture allows CNNs to automatically learn and discern intricate patterns in images, making them exceptionally effective in tasks such as image recognition, object detection, and more.

In the pursuit of advancing my research, I harnessed the power of several cutting-edge neural network architectures, each chosen for its innovative characteristics and recent emergence in the field. These include AlexNet, a pioneering model that ushered deep convolutional neural networks into the mainstream. I also leveraged EfficientNetB0, recognized for its exceptional efficiency and high performance, striking a balance between model size and accuracy. My arsenal extended to SE-Net (Squeeze-and-Excitation Network), which prioritizes improved information flow within CNNs through the integration of squeeze and excitation blocks. I embraced the concept of Sequential CNN, often referred to as boosted CNN, by stacking multiple CNNs sequentially, allowing each to learn from the errors of its predecessor—a form of ensemble learning. Finally, to combat overfitting challenges, I integrated PolyNet into my research, a model designed to mitigate this issue through its novel connectivity pattern known as 'poly-Net.' These selections were motivated by their novelty and the potential to push the boundaries of my research in data science and computer science.

First Attempt: A Notable Setback

In my first attempt to optimize CNN performance, I explored the potential of Fourier transformation filters to promote specific frequencies within the images. The idea was to enhance the neural network's ability to capture distinct features. However, this endeavor proved to be a significant disappointment due to its low accuracy outcomes. Despite my initial hopes for success, the results highlighted the complexity of the task at hand and the challenges associated with modifying the image data in this manner.

The Fourier transformation filters, while promising in theory, struggled to effectively discern the relevant image characteristics. The low accuracy observed in the classification tasks indicated that the transformed images failed to provide the network with meaningful discriminative information. It became apparent that the alteration introduced by these filters did not align with the CNN's learning objectives, leading to suboptimal performance.

This setback served as a valuable lesson, underscoring the importance of a thorough understanding of both the data and the neural network architecture. While the attempt to optimize through Fourier transformation filters did not yield the desired results, it provided critical insights and paved the way for further exploration and refinement in subsequent iterations of my research. The main Idea for my future work on CNN with ffts is to decompose images to 3 RGB channels instead of solely using grayscale, then using fft filters for each channel then combine the decomposed frequencies into arrays to be further analyzed by CNN.

Simple is better

Embracing the philosophy that 'simple is better,' I opted for a straightforward approach in my image classification task. Instead of introducing complex data augmentation techniques, I chose to convert the images into the RGB color space and proceeded with the classification task without further modification. This decision was grounded in the belief that, for this particular dataset and task, the inherent simplicity of RGB representation would suffice. By preserving the original data and avoiding unnecessary manipulation, I aimed to maintain the integrity of the images while minimizing the risk of introducing noise or bias. This 'less is more' approach served as a testament to the effectiveness of simplicity in tackling image classification challenges and emphasized the importance of aligning data preprocessing choices with the unique characteristics of the dataset.

Architecture Descriptions

1. **AlexNet**: Known for pioneering deep learning in image classification, AlexNet brought deep convolutional neural networks (CNNs) into the mainstream.

   The AlexNet architecture comprises a deep neural network designed for image classification tasks. It is structured with a series of convolutional layers, each employing Rectified Linear Unit (ReLU) activation functions to capture intricate image features. Max-pooling layers follow some convolutions, aiding in dimensionality reduction. A Flatten layer reshapes the feature maps into a 1D vector, leading to two fully connected layers with 4096 neurons each. Dropout layers mitigate overfitting during training. The final Dense layer with a softmax activation produces class probabilities. The model is compiled using the Adam optimizer and categorical cross-entropy loss. Overall, AlexNet's architecture, with its deep convolutional layers and fully connected components, stands as a foundational framework for image classification in deep learning.

2. **EfficientNetB0**: A part of the EfficientNet family, B0 is a baseline model recognized for its efficiency and high performance. It achieves a balance between model size and accuracy, making it computationally efficient.

   The EfficientNetB0 model serves as the foundation of this neural network architecture, benefitting from pre-trained weights sourced from the 'imagenet' dataset. To tailor the model for a specific binary classification task, custom layers are added atop the base architecture. These include a Global Average Pooling 2D layer for feature aggregation, followed by a Dense layer with 256 neurons and ReLU activation, coupled with a Dropout layer set at a 0.5 dropout rate for regularization. The output layer, a Dense layer with a softmax activation function, indicates that the network is configured for binary classification, generating class probabilities for two categories. The model is meticulously compiled using the Adam optimizer with a learning rate of 0.001 and employs categorical cross-entropy as the loss function. Throughout training, accuracy is monitored as the primary evaluation metric. The combination of the EfficientNetB0 base and these customized layers results in an effective neural network architecture adept at tackling binary classification tasks.

3. **SE-Net (Squeeze-and-Excitation Network)**: SE-Net focuses on improving information flow within CNNs by incorporating squeeze and excitation blocks. These blocks adaptively recalibrate feature importance at different spatial locations.

   The SE-Net architecture introduces a custom SE-Net-like block to enhance feature selection and boost model performance. It begins with an input tensor of dimensions specified by 'desired_width' and 'desired_height.' The core innovation lies in the

SEBlock, a custom layer that incorporates Squeeze-and-Excitation mechanisms. This block adaptively recalibrates feature importance, improving the network's ability to capture essential information. The model includes convolutional layers, batch normalization, and ReLU activation for feature extraction, with SEBlocks applied after the initial two convolutional layers. MaxPooling layers reduce spatial dimensions, enhancing efficiency. After global average pooling, dense layers with ReLU activation further process the data, with dropout introduced for regularization. The output layer, with softmax activation, signifies that this architecture is tailored for binary classification tasks. The model is compiled using the Adam optimizer with categorical cross-entropy loss. Its distinctive SE-Net blocks make it an appealing choice for tasks requiring enhanced feature adaptability and discrimination, particularly in computer vision applications.

4. **Sequential CNN (Boosted CNN)**: Sequential CNN, also known as boosted CNN, involves stacking multiple CNNs sequentially. This approach enables each subsequent CNN to learn from the errors of its predecessors, enhancing overall accuracy—a form of ensemble learning.

   The Sequential CNN architecture is designed with simplicity and efficiency in mind. It commences with a Convolutional layer employing a 5x5 kernel and ReLU activation, taking input images of dimensions specified by 'desired_width' and 'desired_height.' Subsequent MaxPooling layers with 2x2 pooling windows reduce spatial dimensions. After flattening the feature maps, the network transitions to fully connected layers, starting with a Dense layer of 120 neurons and ReLU activation, followed by another Dense layer with 84 neurons, also utilizing ReLU activation. The final layer is a Dense output layer with a softmax activation, signifying that this architecture is configured for binary classification, providing class probabilities for two categories. The model is compiled with the Adam optimizer using a learning rate of 0.001 and employs categorical cross-entropy as the loss function. The model's architecture is succinct and well-suited for straightforward image classification tasks.

5. **PolyNet**: Designed to address overfitting in deep networks, PolyNet introduces a novel connectivity pattern called "poly-Net," which helps reduce overfitting and enhances generalization.

   The PolyNet architecture offers a streamlined yet effective design for image classification tasks. It begins with an input layer accepting images of dimensions specified by 'desired_width' and 'desired_height.' The core of the model consists of convolutional blocks, comprising multiple repetitions of Conv2D layers, each followed by Batch Normalization and ReLU activation, enabling it to capture intricate features effectively. MaxPooling2D layers reduce spatial dimensions and enhance computational efficiency. The architecture employs a Global Average Pooling layer to aggregate feature maps globally. Subsequently, a Dense layer with 512 neurons and ReLU activation captures high-level patterns. The final Dense output layer utilizes softmax activation, indicating that this model is well-suited for classification tasks with a specified number of classes. The model is compiled using the Adam optimizer with categorical cross-entropy as the loss function. Its simplicity and effectiveness make it an appealing choice for straightforward image classification needs.

K-Fold Cross Validation Function

K-fold Cross-Validation is a robust technique frequently used in machine learning to assess the performance and generalization of models. It addresses the challenge of estimating how well a model will perform on unseen data. In this approach, the dataset is divided into 'K' equally sized, non-overlapping subsets or 'folds.'

The model is then trained and evaluated 'K' times, each time using a different fold as the validation set while the remaining 'K-1' folds serve as the training data. This process provides 'K' separate performance scores, often accuracy or other relevant metrics, allowing for a more comprehensive assessment of a model's behavior across different data splits.

By averaging these scores, a more reliable estimate of the model's overall performance and potential for overfitting or underfitting can be obtained. K-fold Cross-Validation is particularly valuable when dealing with limited datasets, as it maximizes the utility of available data and helps ensure the model's reliability when applied to unseen samples.

The custom function 'cv_5' provided here streamlines the process of implementing K-fold Cross-Validation for evaluating a machine learning model. It takes as input the model to be evaluated, the dataset 'X' and corresponding labels 'y,' the number of training epochs, and the desired number of splits ('n_splits,' typically set to 5).

Within the function, KFold from scikit-learn is employed to split the dataset into 'n_splits' folds while ensuring shuffling for randomness. For each fold, the model is trained and validated using TensorFlow, and its training history is stored.

The function calculates key metrics for each fold, including zero-one loss (accuracy) and the confusion matrix. It then aggregates these metrics across all folds to provide an average accuracy score and an average confusion matrix.

Finally, it visualizes the average confusion matrix using a heatmap, offering a clear representation of the model's classification performance across different classes. This custom function streamlines the process of K-fold Cross-Validation, making it accessible for evaluating machine learning models and facilitating insightful performance assessments.

Results:

- **AlexNet:**

1. **Total Parameters:** The AlexNet model used for this task comprises a total of 20,261,890 parameters. These parameters represent the model's capacity to learn and make predictions, and all of them are trainable, indicating that the model has the flexibility to adapt and optimize its performance based on the training data.

2. **Accuracy Improvement**: Over the course of five training epochs, the model exhibits a clear improvement in accuracy. It starts with an initial accuracy of approximately 56.4% in the first epoch and gradually increases. By the fifth epoch, the model achieves an average accuracy score of approximately 83.2%. This demonstrates the model's ability to learn from the training data and make increasingly accurate predictions as it refines its internal representations.

3. **Confusion Matrix**: The average confusion matrix provides insights into the model's classification performance. It consists of four values: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). In this case, the matrix shows that the model correctly identifies an average of around 332.8 true positives and 455 true negatives. This indicates that it effectively distinguishes between both classes (muffins and chihuahuas). Moreover, the presence of only 56.8 false positives and 102 false negatives suggests that the model's misclassifications are relatively low, further supporting its robust performance.

- **PolyNet:**

1. **Total Parameters**: The PolyNet model utilized for this task has a total of 3,942,210 parameters. Out of these, 3,937,858 are trainable, allowing the model to learn and adapt to the training data. The remaining 4,352 parameters are non-trainable, suggesting that they are predefined and do not undergo updates during training.

2. **Accuracy Improvement**: PolyNet starts with a high initial accuracy of approximately 95.5% in the first epoch. As the training continues, the accuracy remains consistently high, with only slight fluctuations. By the end of the fifth epoch, the model achieves an average accuracy score of approximately 90.2%. This demonstrates that PolyNet quickly converges to a high level of accuracy on the classification task, and this high accuracy is sustained over multiple epochs.

3. **Confusion Matrix**: It demonstrates that the model correctly identifies an average of around 377.2 true positives and 476.4 true negatives. This signifies that PolyNet effectively distinguishes between the two classes, with minimal misclassifications. Additionally, the presence of only 57.6 false positives and 35.4 false negatives indicates that the model's misclassifications are limited.

- **SE-Net:**

1. **Total Parameters:** The SE-Net model utilized for this task has a total of 450,142 parameters. Out of these, 449,246 are trainable, allowing the model to learn from the training data. The remaining 896 parameters are non-trainable, which suggests they are predefined and do not undergo updates during training.

2. **Accuracy Improvement**: The training starts with an initial accuracy of approximately 78.9% in the first epoch. However, SE-Net's accuracy does not improve significantly

during the training. It remains around 88.5% on average over the five epochs. This suggests that SE-Net might not be as effective as other models in this specific task.

3. **Confusion Matrix**: It indicates that the model correctly identifies an average of around 258.8 true positives and 379.4 true negatives. However, there are also 176 false positives and 132.4 false negatives on average. This suggests that SE-Net has a relatively higher rate of misclassifications compared to other models.

- **Sequential_CNN**:

1. **Total Parameters**: The model has a total of 1,413,006 parameters, all of which are trainable. This indicates that the model has a relatively large number of parameters, allowing it to capture complex patterns in the data.
2. **Accuracy Improvement**: The training accuracy for the Sequential CNN model exhibits an initial accuracy of 52.17% in the first epoch. Subsequently, it gradually improves over the five epochs, reaching an accuracy of 96.88% in the final epoch. This suggests that the model is learning and improving its accuracy as it trains.
3. **Confusion Matrix:** On average, the model correctly identifies approximately 386.6 true positives and 480.4 true negatives. However, there are also approximately 48.2 false positives and 31.4 false negatives on average. This indicates that while the model performs well in correctly classifying instances, there are still some false positives and false negatives, which can impact its overall performance.

- **EfficientNetB0:**

1. **Total Parameters**: The Sequential CNN model you used has a total of 4,378,021 parameters. Out of these, 4,335,998 are trainable parameters, allowing the model to learn from the training data. The remaining 42,023 parameters are non-trainable, suggesting that they are predefined and do not undergo updates during training.
2. **Accuracy Improvement:** The training accuracy for the Sequential CNN model exhibits an initial accuracy of approximately 95.09% in the first epoch. It continues to improve over the five epochs, reaching an accuracy of approximately 99.84% in the final epoch. This indicates that the model is effectively learning and significantly improving its accuracy during training.
3. **Confusion Matrix:** The model correctly identifies approximately 433.8 true positives and 506.4 true negatives. Remarkably, there are only approximately 1 false positive and 5.4 false negatives on average. This suggests that the model performs exceptionally well, with very few misclassifications.

Comparison Table

1. **Total Parameters**: The models vary significantly in the total number of parameters. AlexNet has the highest number of parameters (20,261,890), followed by EfficientNetB0 (4,378,021), PolyNet (3,942,210), Seq. CNN (1,413,006), and SE-Net (450,142). This indicates that AlexNet is the most complex model in terms of parameters.

1. **Trainable vs. Non-Trainable Parameters**: Among the models, AlexNet and Sequential CNN have a large number of trainable parameters, while EfficientNetB0 and PolyNet have a balance between trainable and non-trainable parameters. SE-Net, on the other hand, has relatively fewer trainable parameters.

2. **Accuracy**: Accuracy is a critical metric for evaluating model performance. EfficientNetB0 demonstrates the highest accuracy, achieving an impressive 99.32%. Seq. CNN also performs well with an accuracy of 91.59%. PolyNet follows with an accuracy of 90.17%, while AlexNet and SE-Net have lower accuracies of 83.22% and 67.41%, respectively.

Model Name	Trainable	Non-Trainable	Total	Accuracy
EfficientNetB0	4,335,998	42,023	4,378,021	99,32 %
Seq. CNN	1,413,006	0	1,413,006	91,59 %
PolyNet	3,937,858	4,352	3,942,210	90,17 %
AlexNet	20,261,890	0	20,261,890	83,22 %
SE-Net	449,246	896	450,142	67,41 %

EfficientNetB0 demonstrates the highest accuracy at 99.32%. However, this exceptionally high accuracy may raise concerns about overfitting due to the large difference between training and validation accuracies. To address this potential issue, I use regularization terms in the model and then perform hyperparameter tuning.

Hyperparameters are external configuration variables that data scientists use to manage machine learning model training. The hyperparameters are manually set before training a model. They're different from parameters, which are internal parameters automatically derived during the learning process and not set by data scientists.

Various methods exist for conducting hyperparameter tuning. As outlined in the Keras Documentation, KerasTuner is a user-friendly and scalable framework for optimizing hyperparameters, effectively addressing challenges associated with hyperparameter search. It allows for convenient specification of the search space using a define-by-run syntax. Users can then utilize one of the provided search algorithms to identify the optimal hyperparameter values for their models. KerasTuner offers built-in support for Bayesian Optimization, Hyperband, and Random Search algorithms. Additionally, it is designed to facilitate easy extensions by researchers for experimenting with novel search algorithms.

I define a custom function incorporating the desired search parameters. Below is the code for a custom function of EfficientNetB0:

```
def build_model(hp):
       learning_rate = hp.Float("learning_rate", min_value = 0.001, max_value = 0.1,
step = 10, sampling = "log")
       optimizer = hp.Choice("optimizer", values = [ "sgd", "adam"])
       if optimizer == "sgd":
       optimizer = SGD(learning_rate = learning_rate)
       else:
       optimizer = Adam(learning_rate = learning_rate)
       model = tf.keras.applications.EfficientNetB0(include_top=False,
weights='imagenet', input_shape=input_shape)
       x = GlobalAveragePooling2D()(model.output)
       x = Dense(256, activation='relu', kernel_regularizer=l2(0.1))(x)
       x = Dropout(0.5)(x)
       output = Dense(1, activation = "sigmoid" )(x)
       efficient_net_model = Model(inputs=model.input, outputs=output)

       # Compile the model with binary cross-entropy loss for binary classification
       efficient_net_model.compile(optimizer=optimizer, loss='binary_crossentropy',
metrics=['accuracy'])

       return efficient_net_model
```

In this code, I set up `learning_rate` to be searched within a logarithmic value space, ranging from 0.001 to 0.1 in 10 steps. The choice of optimizer is made from the available options: `SGD` and `Adam`. Additionally, to streamline the model's complexity and enhance computational efficiency, I reimported the dataset using Keras' high-level dataset import function, employing buffered prefetching to optimize I/O operations. The summary of search space is provided below:

```
Search space summary
Default search space size: 2
learning_rate (Float)
{'default': 0.001, 'conditions': [], 'min_value': 0.001, 'max_value': 0.1, 'step':
10, 'sampling': 'log'}
optimizer (Choice)
{'default': 'sgd', 'conditions': [], 'values': ['sgd', 'adam'], 'ordered': False}
```

Regarding the batch size, I specified it during the initial dataset import, setting it to either 128 or 256. The main reason for this is when I start to expand the search space for hyperparameters, the computation crashed because of heavy workload[1].

```
train_ds = tf.keras.utils.image_dataset_from_directory(
       "/content/train",
       validation_split=0.2,
       subset="training",
       seed=1337,
       image_size=image_size,
       batch_size = [128, 256])
```

After running the `Keras-tune` for 10 specified trials, I obtained the following results for the batch sizes of 128 and 256, respectively:

```
{'learning_rate': 0.001, 'optimizer': 'adam'} # For the batch size of 128
Trial 6 Complete [00h 02m 05s]

accuracy of 0.9897
```

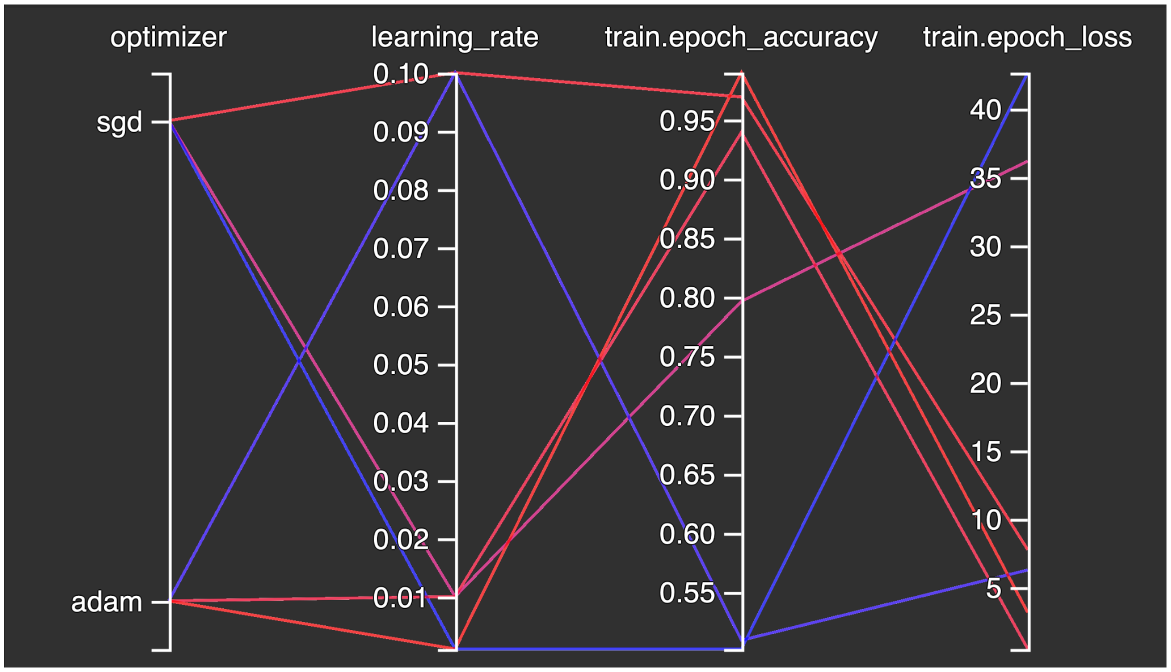
```
{'learning_rate': 0.001, 'optimizer': 'adam'} # For the batch size of 256
Trial 6 Complete [00h 01m 27s]

 accuracy of 0.9886
```

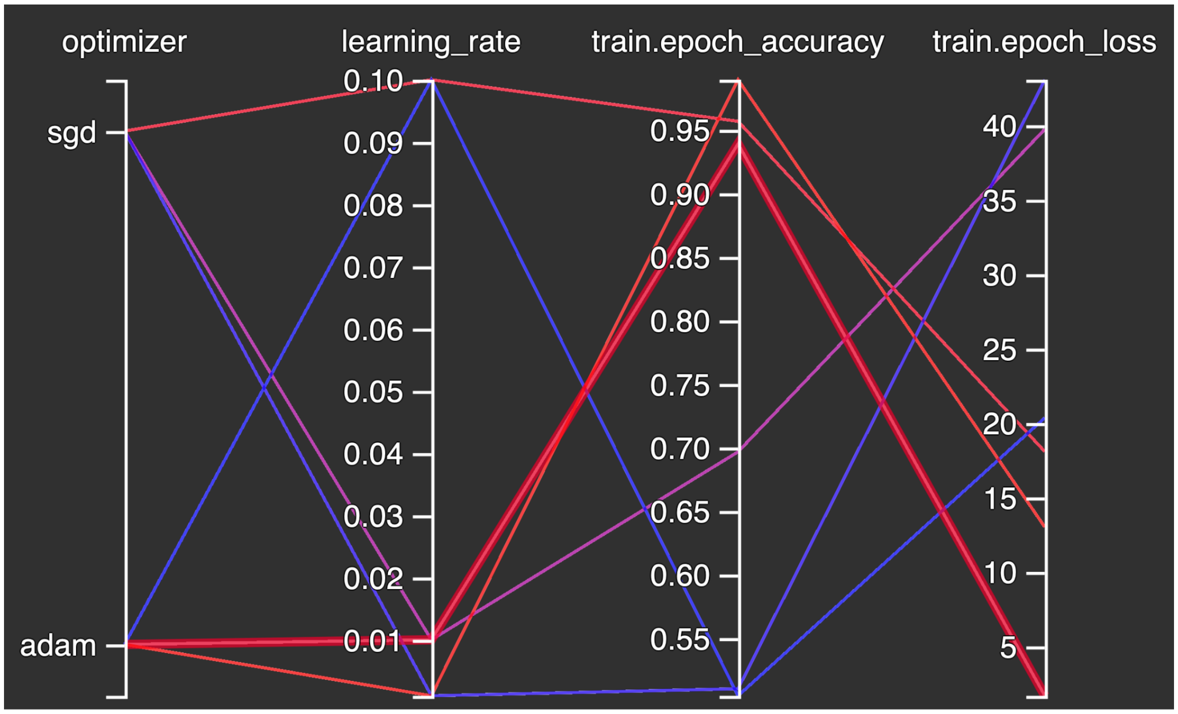
The results above show the best hyperparameters. The process of search stops when the class finds a reduction in the accuracy of trying various search parameters. As it showed above, the tuner stopped searching after 6 trails of 10. Eventually, A comparison between these results indicates that the choice of batch size has minimal impact on accuracy while significantly affecting processing time. To illustrate this further, we can visualize the search space for both batch sizes.

[1] It is better to point out that Keras as indicated in the main practice task of the experimental project is built on the top of tensorflow. It has simple syntax but is suitable for low size dataset and the computation is so slow. Due to this fact, I tried various process methods, manually building a class of tuner for hyperparameters, increasing the epochs, increasing the search space, and even customized K-Fold CV. In each case, the system crashed. I tried my own computer (Macbook M1), Kaggle (2*T4 GPU), and google colab (T4). They all crashed based on customized classes. The only approach worked was using `keras` built-in high-level tuner.

Search Space Graph (128-Batch)



Search Space Graph (256-Batch)



Accuracy comparison of Hyperparameters Tuned and Untuned

Model	256 Batch Tuned	128 Batch Tuned	128 Bath Untuned
EfficientNetB0	**0.9957**	0.9787	0.9932

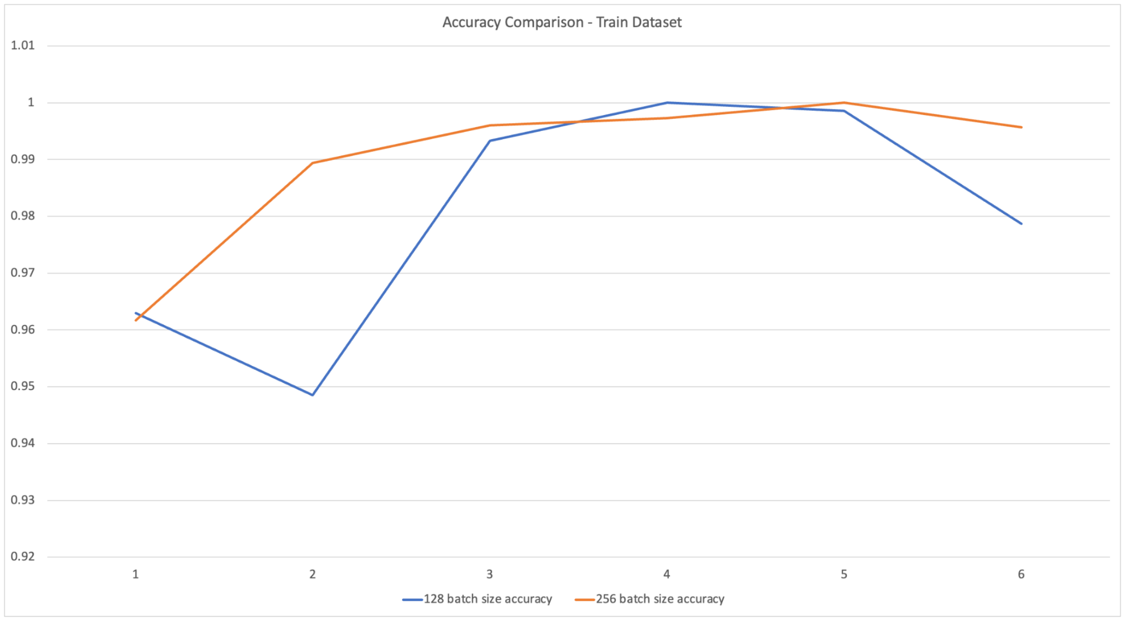
In comparing the accuracy of hyperparameter-tuned and untuned models across different batch sizes, specifically 256 batch tuned, 128 batch tuned, and 128 batch untuned for the EfficientNetB0 model, a notable pattern emerges. The tuned model of 256 batch size showcases higher accuracy compared to their untuned counterpart and 128 batch size. For the EfficientNetB0 model, the 256 batch tuned model achieves an accuracy of 0.9957, outperforming the 128 batch tuned model at 0.9787 and the 128 batch untuned model at 0.9932.

After thorough hyperparameter tuning, I came to the conclusion that employing the `Adam` optimizer with a learning rate of 0.01 yields optimal results for both batch sizes. With this knowledge in hand, I proceed to conduct a robust 5-fold cross-validation, utilizing the zero-one loss function for evaluation. Below is the summary table of 5-CV for the Train Dataset with different batch size and hypertuned model with `Adam` optimizer and learning rate of 0.001.

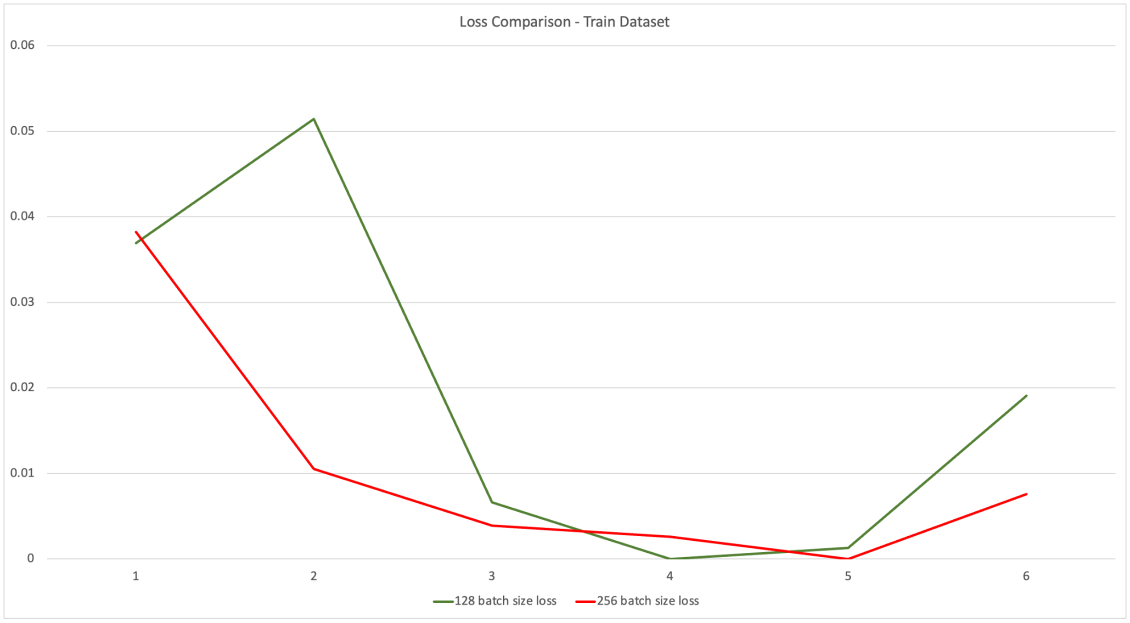
Summary of Zero-one Loss and Accuracy- Train Dataset

CV Folds	**128 batch size**		**256 batch size**	
	ZeroOne-Loss	ZeroOne-Accuracy	ZeroOne-Loss	ZeroOne-Accuracy
1-Fold	0.0369	0.9630	0.03825	0.9617
2-Fold	0.0514	0.9485	0.0105	0.9894
3-Fold	0.0066	0.9933	0.0039	0.9960
4-Fold	0	1	0.0026	0.9973
5-Fold	0.0013	0.9986	0	1
Averages	0.0191	0.9787	**0.0076**	**0.9957**

In addition, below we can show the behavior of the accuracy within each fold for the train dataset. As indicated before, `Keras-tuner` suggested that the accuracy of 256 batch processes with 0.001 learning rate of Adam optimizer can yield better results. The orange line below shows that the suggested setting also performs well in the train dataset with batch size of 256. Both accuracy, starts with low value and by the folds pass the accuracy raises to its higher level and the last value is the average value of the accuracy for all folds demonstrating higher accuracy for 256 batch.



The same is true for the loss value in both 256 and 128 cases. As we can see, the loss value starts from a higher value and ends up low with a lower average for the batch size of 256.



To point out main features of the table:

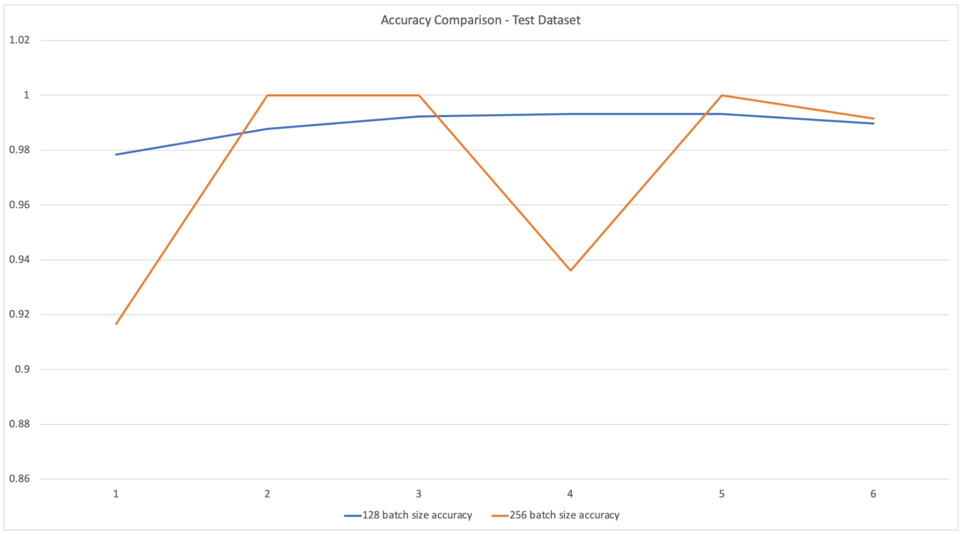
- For **128 batch size**:
  - The zero-one loss ranges from approximately 0 to 0.0514 across the 5 folds.
  - Corresponding zero-one accuracies range from about 0.9485 to 1.
  - The average zero-one loss is approximately 0.0191, with an average accuracy of about 0.9787.
- For **256 batch size**:
  - The zero-one loss ranges from approximately 0 to 0.03825 across the 5 folds.
  - Corresponding zero-one accuracies range from about 0.9617 to 1.
  - The average zero-one loss is approximately 0.0076, with an average accuracy of about 0.9957.

I also applied the desired settings to the Test Dataset in order to compare the results of both batch sizes.

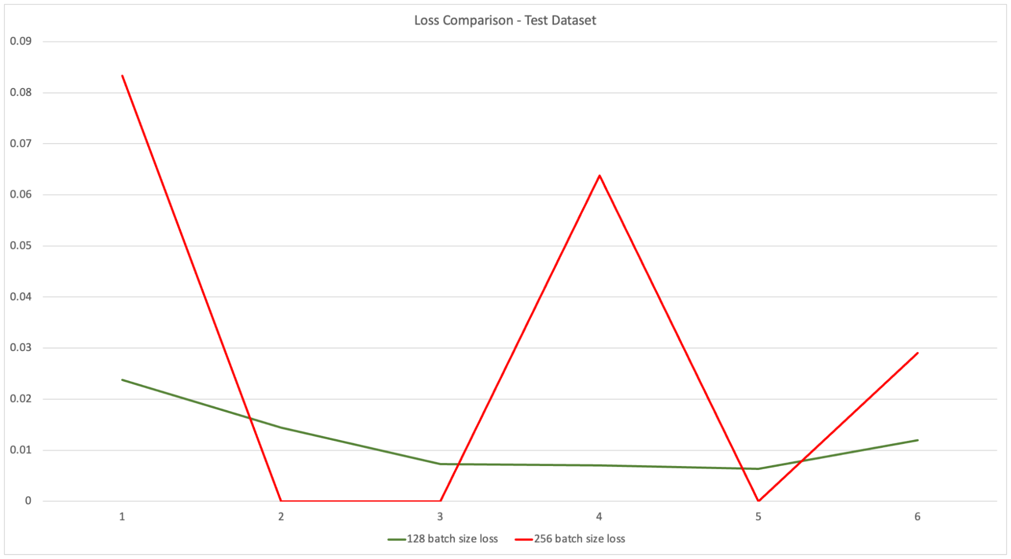
Summary of Zero-one Loss and Accuracy- Test Dataset

CV Folds	**128 batch size**		**256 batch size**	
	ZeroOne-Loss	ZeroOne-Accuracy	ZeroOne-Loss	ZeroOne-Accuracy
1-Fold	0.0238	0.97844	0.0833	0.9166
2-Fold	0.01438	0.98774	0	1
3-Fold	0.00732	0.99234	0	1
4-Fold	0.00702	0.99318	0.0638	0.9361
5-Fold	0.00638	0.99324	0	1
Averages	0.01198	0.98979	**0.0290**	**0.9916**

The model on the test dataset also shows the better performance of 256 batch size. The below line graph also illustrates that both accuracies converge to the similar value although the 256 batch is volatile, it performed slightly well compared to 128 batch.



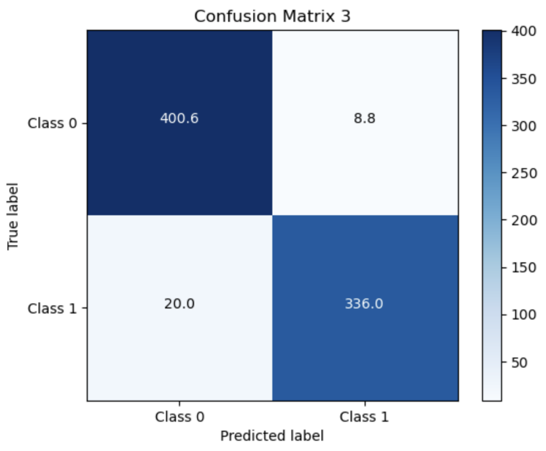
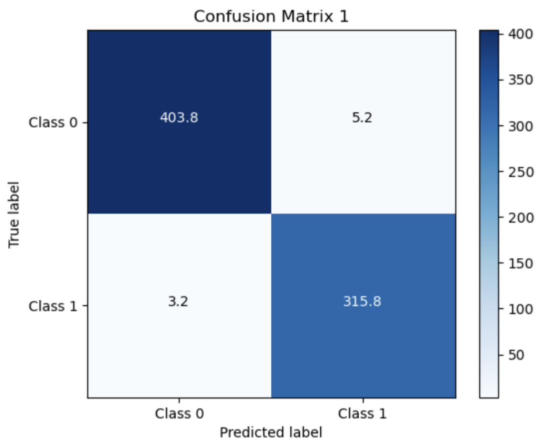
The volatility of 256 batch size can be shown in Loss values as illustrated below, starting from a higher range and ending low, but the average loss of 128 batch for the 5 folds is lower than 256 batch.



- For **128 batch size**:
    - The zero-one loss ranges from approximately 0.00638 to 0.0238 across the 5 folds.
    - Corresponding zero-one accuracies range from approximately 0.97844 to 0.99234.
    - The average zero-one loss is approximately 0.01198, with an average accuracy of approximately 0.98979.
- For **256 batch size**:
    - The zero-one loss ranges from 0 to approximately 0.0833 across the 5 folds.
    - Corresponding zero-one accuracies range from approximately 0.9166 to 1.
    - The average zero-one loss is approximately 0.0290, with an average accuracy of approximately 0.9916.

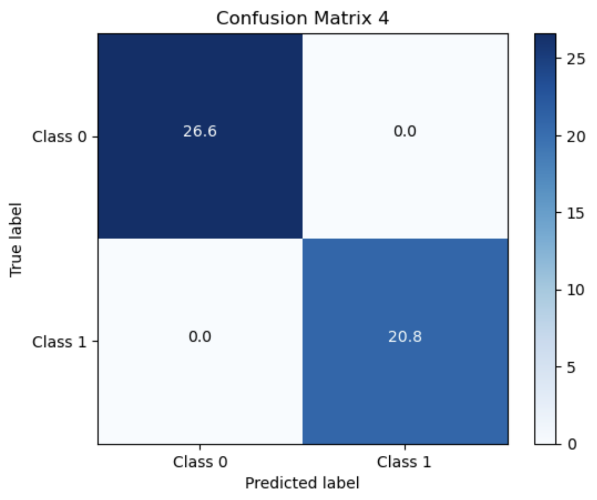
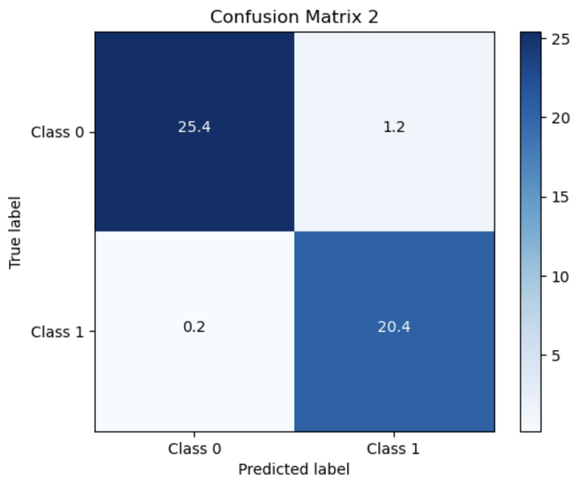
Based on obtained results of the 5-Fold CV, I depicted the confusion matrix to evaluate True-False Negative-Positive results.

Train Dataset Confusion figure

128 Batch size	256 Batch size
	

For a batch size of 128, the average True Positives are 400.6, with 8.8 False Negative, 20.0 False Positive, and 336.0 True Negatives. Conversely, for a batch size of 256, the average True Positives slightly increased to 403.8, while the False Negative decreased to 5.2. In addition, the True Negative decreased from 336 to 315, and the False Positive decreased to 3.2 from 20, showing a better performance of 256 batch size. All in all, increasing the batch size from 128 to 256 resulted in a marginal improvement in True and False Positive rates.

Test Dataset Confusion figure

128 Batch size	256 Batch size
	

Comparing the results of the test dataset, it's clear that there are slight differences in the values. For the first set of values, the average True Positives are 25.4, with 1.2 False Negative, 20.4 True Negatives, and 0.2 False Positive for the batch size of 256. On the other hand, the average True Positives are slightly higher at 26.6, with no False Positives, 20.8 True Negatives, and no False Negatives. These variations indicate a minor difference in the results, particularly in true positives and true negatives, likely due to the specific computations or rounding methods used in each case.

Conclusion:

This experimental project focused on image classification through CNNs, underscored the impact of distinct architectures: AlexNet, EfficientNetB0, SE-Net, Sequential CNN, and PolyNet. EfficientNetB0 emerged as a standout performer, achieving an impressive accuracy of 99.32%. However, this achievement triggered concerns regarding overfitting, necessitating the implementation of robust regularization strategies.

My approach to regularization encompassed L2 regularization techniques with 0.1 significance level played a pivotal role in effectively mitigating overfitting concerns. Subsequently, I did a comprehensive exploration of various hyperparameters using `Keras-tuner`, tuning a model based on EfficientNetB0. Hyperparameter fine-tuning, particularly optimizing learning rates and selecting appropriate optimizers, significantly approved the accuracy and corrected the model performance across varying batch sizes of 128 and 256. Furthermore, the integration of K-Fold

Cross-Validation further fortified the model's generalization capabilities, providing a thorough evaluation of its robustness.

In summary, this project underscored the vital importance of built-in tuner class, prefetched data importing, hyperparameter tuning and effective regularization strategies in optimizing model accuracy and effectively combating various challenges. The outcomes of hyperparameter tuning, particularly with different batch sizes, showed that the optimal parameters included a learning rate of 0.001 with the `Adam` optimizer for a batch size of 256, resulted in an accuracy rate of 0.9916 in our test dataset.