Exploring Convolutional Neural Networks for Image Classification

AstroKhet @ GitHub

July 2024

Abstract

This paper explores and compares the performance of various convolutional neural network (CNN) models for image classification. Specifically, we analyze AlexNet, VGG16, ResNet, Inception, EfficientNet, and MobileNet across multiple performance metrics, including accuracy, training and inference time, and model complexity. A dataset of flower images spanning five classes is used.

1 Introduction

Convolutional Neural Networks (CNNs) are a feed-forward neural network that automatically learns features of objects in grid data like images. This is achieved through sliding a set of convolutional filters (or kernels) across the width and height of the input, which can be either the raw image itself (a 3D tensor with three color channels) or a collection of 2D previous feature maps represented as a 3D tensor. Each filter can capture local patterns such as edges, shapes, and textures (depending on its *receptive field*) and map these values onto a *feature map*. Each feature map uses one filter across the entire input, features are detected regardless of their location on the image. This is known as *weight sharing*, and hence CNNs are also known as shift-invariant neural networks as spatial translations of the focus object will produce equivalent responses. Having a collection of filters allows the model to pick up on various aspects of the image. In addition, the spatial structure of images whereby nearby pixels are usually closely related allows the filters to reduce the dimensionality of our input without significant loss in useful information. This makes CNNs particularly effective for image classification tasks compared to traditional Deep Neural Networks.

The dataset used in this paper consists of 4,317 images of flowers [source], spanning the classes 'Daisy', 'Dandelion', 'Rose', 'Sunflower', and 'Tulip'. To compare and assess different models, certain hyperparameters, such as batch size, are standardized, while others, like learning rate, are tuned to optimize each model's performance. Performance metrics such as top-1 accuracy determine how well a model performs after convergence. Resource metrics, including model size, training speed, number of epochs to convergence, and inference time, assess the model's scalability and feasibility.

(a) Daisy (b) Dandelion (c) Rose (d) Sunflower (e) Tulip

Figure 1: Flower Dataset

2 Convolutional Neural Networks (CNNs)

2.1 AlexNet

A good example for us to look at is AlexNet, a pioneering CNN architecture that significantly advanced the field of deep learning and computer vision. Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, AlexNet gained widespread recognition after winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. The figure below illustrates the AlexNet structure.

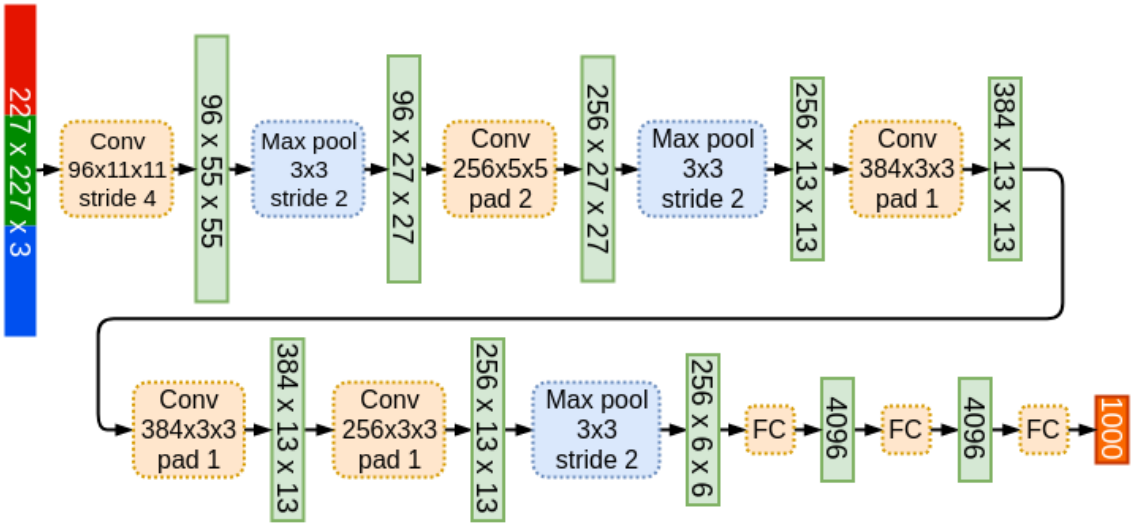


Figure 2: AlexNet architecture (Source: O'Reilly)

2.2 Layers

A simple CNN such as AlexNet consists of multiple convolutional layers with decreasing kernel sizes, usually with pooling layers between them. At the top of the model lies a network of dense layers, which learn how to use the features extracted from the convolutional network and produce an output through a softmax function.

2.2.1 Convolutional Layers

Each convolutional layer contains a set of filters with trainable values. These filters are 3D tensors with spatial dimensions $k \times k$ (k = filter size), and a depth of C_{in} input channels. They map outputs onto C_{out} feature maps. Hence, filters are written as a 4D tensor with dimension $C_{in} \times C_{out} \times k \times k$. For example, the first convolutional layer takes in an image of size $227 \times 277 (k = 277)$, an input RGB channel $C_{in} = 3$. Since there are 96 distinct trainable filters, the output channel has depth $C_{out} = 96$. Hence, the dimension of our filter tensor for this layer is $3 \times 96 \times 227 \times 227$.

During convolutional, a 2D sub-filter is taken from each 3D filter with the same input channel index. This sub-filter of size $k \times k$ is slid over the input channel and dotted with the region it covers. It is slid bidirectionally with a stride s and padding p. Stride describes the units shifted row-wise and column-wise for each window, and padding fits a 'frame' of zeros of size p around the input image. Intuitively, greater stride reduces the feature map size, while greater padding increases it. Given a filter of height and width k, let h and w be the height and width of the input, and h' and w' be the height and width of the feature map. Then,

$$h' = \left\lfloor \frac{h - k + 2p}{s} \right\rfloor + 1$$

$$w' = \left\lfloor \frac{w - k + 2p}{s} \right\rfloor + 1$$

For example, Conv-5 has 256 kernels of size 13×13 and is passed through a max-pooling filter of size $k = 3$ and stride $s = 1$. Hence, each of the final feature maps produced after max pooling with $k = 2$ and $s = 3$ will have dimensions:

$$h' = \left\lfloor \frac{13-3}{2} \right\rfloor + 1 = 6$$
$$w' = \left\lfloor \frac{13-3}{2} \right\rfloor + 1 = 6$$

This means that we will have $h \times w \times$ number of feature maps $= 6 \times 6 \times 256 = 9216$ features in total, before connecting to two dense layers of size 4096.

Since only one filter is applied to the input at once, all parts of the input will be filtered similarly and features are detected anywhere This is known as weight sharing, where a tulip is still a tulip no matter where it is in the image.

2.2.2 Batch Normalization

Training is usually done using mini-batches of the dataset to reduce memory usage and speed up training. Batch normalization is a technique that is done to the input batch b before non-linearity to help the model learn better and quicker. It changes the input set to fit a normal distribution with mean $\mu = 0$ and variance $\sigma = 1$. Mathematically,

$$\hat{x_i} = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}$$

where:

- x_i is the i^{th} **input value**
- $\hat{x_i}$ is the i^{th} **normalized input value**
- M is the **batch size**
- **Batch mean** $\mu_b = \frac{1}{M} \sum_{j=1}^{M} x_j$
- **Batch variance** $\sigma_b^2 = \frac{1}{M} \sum_{j=1}^{M} (x_j - \mu_b)^2$
- ϵ is a small positive constant (e.g., 10^{-8} to prevent division by zero.

This scales each feature of the input to within the same range and centered about the same value. Intuitively, this works as the model will initially treat all features with equal importance, and then learn which ones are more important during training. Consider a small network trying to predict if a person is a good basketball player using the features height (meters) and age (years). Comparatively, the values of height will range somewhere around 1.5 to 2.5 and pale in comparison to age, which can be up to 100. The model then fine-tunes its parameters more towards the age of a person, as they have a greater influence on its output, overshadowing the height feature which is also important. Normalizing also helps shape the landscape of the loss function to be more uniform and increases the training speed and accuracy of the model with gradient descent.

After that, scaling and shifting are done before feeding the data into our non-linear activation function.

$$y_i = \gamma \hat{x}_i + \beta$$

Where γ and β are learnable parameters adjusted during training.

However, at inference time, input data is fed in one by one. Hence, the mean and variance used in normalization are produced through moving averages.

$$\hat{\mu}_t = (1 - \beta)\hat{\mu}_{t-1} + \beta \mu_t$$
$$\hat{\sigma}_t^2 = (1 - \beta)\hat{\sigma}_{t-1}^2 + \beta \sigma_t^2$$

where

- β is the momentum term, a hyperparameter that determines the weight given to the current mini-batch statistics versus the previous moving averages.
- $\hat{\mu}_t$ and $\hat{\sigma}_t^2$ is the moving average for the mean and variance at time step t
- μ_t and σ_t^2 is the mean and variance of the current input at time step t

2.2.3 Pooling layers

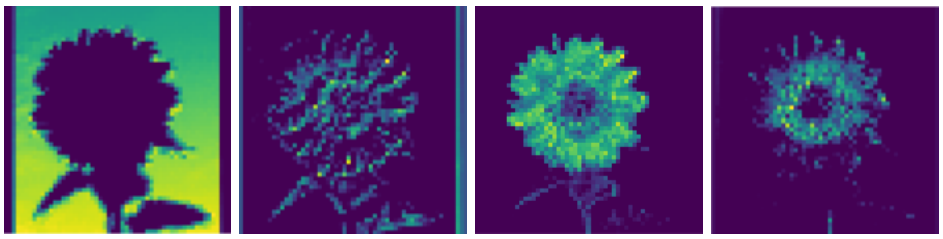
Pooling layers such as max/mean (returns max/mean value in a $k \times k$ region) reduce the spatial dimensions of the input, making the model invariant to minor distortions (e.g. an additional leaf or cloud should not produce different responses). These pooling layers operate similarly to convolutional filters and can be similarly defined with a size k and stride s (no padding of course). Pooling produces an output tensor with the same depth as the input tensor.

2.2.4 Feature Maps

Consider an input image of a sunflower:

Figure 3: A Sunflower (Source: Wikipedia)

We pass it through a pre-trained AlexNet model trained on the flower dataset. The first convolutional layer produces 96 different feature maps as described in the model. Each filter map focuses on different features. A snapshot of 4 of the filters is shown below:



(a) Outline (b) Depth (c) Petals (d) Seeds

Figure 4: Conv1 feature maps

At the first convolutional layer, the model captures only general information about the sunflower. This is due to the limited region that each 3x3 convolutional filter observes during each operation, known as the receptive field. As we progress through the network, the addition of pooling and convolutional layers increases the receptive field. Intuitively, this is because these layers take multiple inputs (e.g., a 2x2 window for max pooling) and produce a single output, causing the number of features in the original input images to cascade and stack as they are passed through these layers. Therefore, we expect the first few layers to perceive mainly general features of the images, and the later layers to perceive more low-level features such as individual shapes and edges.

Now, let us examine the 5^{th} and final convolutional layer, which contains 256 filters of size 13×13 before pooling. A snippet of 25 filters is shown below:

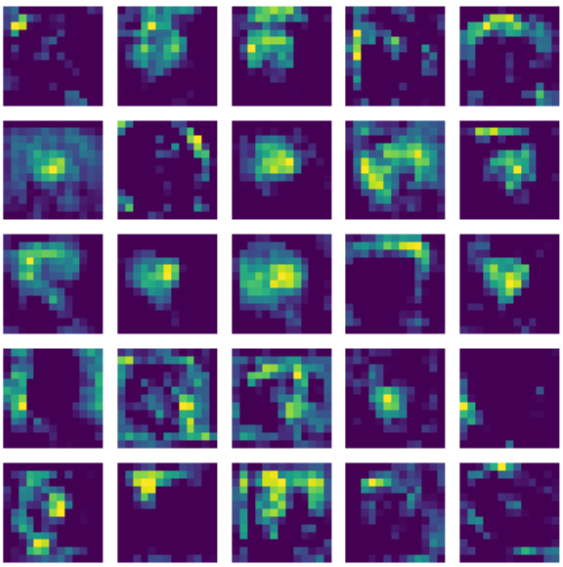
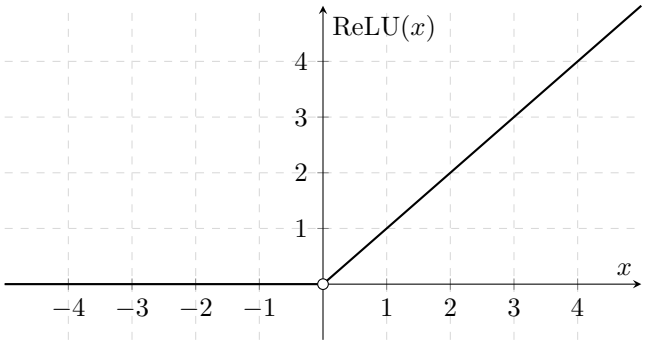


Figure 5: Conv5 feature maps

These filters have the largest receptive field of all the convolutional layers in the AlexNet model.

2.2.5 Activation Functions

After each convolutional layer, a non-linear activation function must be applied since the convolutional operation is linear. Linearity means that the effect of two or more consecutive convolutional layers can be represented in a single layer, which defeats the purpose of having multiple convolutional layers. Non-linearity also introduces complexity to the model and allows it to better learn more complex features. The most popular choice is the Rectified Linear Unit (ReLU):



$$\text{ReLU}(x) = \max(0, x)$$

ReLU is computationally efficient and mitigates the vanishing gradient problem, unlike other non-linear activation functions such as sigmoid or tanh functions.

2.2.6 Dense Layer and Output

Finally, the data activations passing through are flattened into a 1D tensor and fed into a dense network. A dense network takes high-level features extracted by the aforementioned layers and interprets their patterns to make a final decision. During training, dropout with a proportion of 0.5 is used to reduce overfitting and training time. Each dense layer also uses ReLU as its activation function. The last dense layer will contain N (number of classes) neurons and a *softmax* activation function, which returns the probability of the image being labeled as that class j. The softmax function $\sigma(z)$ for the j^{th} output neuron is:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{n=1}^{N} e^{z_n}}$$

This ensures that the sum of the output probabilities is one and we can conclude that the prediction of the model is the class with the highest predicted probability. This is also known as top-1 prediction.

2.3 Training

Training enables our model to learn how to classify the five different classes of flowers. TensorFlow uses gradient descent to adjust the model's trainable parameters iteratively to improve performance. This paper employs TensorFlow (version 2.10.1) for model training, utilizing an NVIDIA GeForce RTX 3050 GPU with 8 GB of GDDR6 memory.

2.3.1 Datasets

Our dataset of 4,317 flower images is split into training and validation in an 80:20 split. This allows us to explicitly train the model on the training dataset and evaluate its performance on unseen data in the validation dataset. This provides a more realistic expectation of how the model would perform, as overfitting can happen during training which makes the model unable to adapt to unseen data.

2.3.2 Loss Function

The fundamental concept of gradient descent is to minimize a function known as the *Loss Function*, which acts as a measure of how much our model's output strays from the correct output. A large loss function implies that our model is not doing well, while a loss function of 0 implies that our model fits perfectly. A small model may suggest good performance but this is not necessarily true. For classification tasks using a softmax output, *Categorical Cross Entropy Loss* is used as the loss function. For each prediction, it is defined as:

$$L_{CCE}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

where:

- L is the loss.
- C is the number of classes.
- y_i is the ground truth label (1 if the class is the correct class, 0 otherwise).
- $\hat{y}_i$ is the predicted probability for class i.

Example:
Suppose we have a classification problem with 3 classes, and the true label is class 2. The one-hot encoded vector for the true label is $[0, 1, 0]$. Let's say our model predicts the following probabilities for each class:

Class 1	Class 2	Class 3
0.1	0.6	0.3

Figure 6: Softmax probability prediction

The predicted probability vector is $[0.1, 0.6, 0.3]$. Plugging in the values:

$$L = -\left(0 \cdot \log(0.1) + 1 \cdot \log(0.6) + 0 \cdot \log(0.3)\right) \approx 0.5108$$

The Categorical Cross Entropy Loss for this example is approximately 0.5108. The total loss is calculated by **averaging the loss for all predictions** in the validation dataset

2.3.3 Trainable Parameters

Trainable parameters include weights and biases for filters in convolutional layers, dense layers, and batch normalization. They are generally represented together with the symbol θ. One of our comparison metrics is the number of trainable parameters. This metric acts as a gauge for the model's performance and indicates the computational resources required for training and inference. We can visualize the structure of these parameters by calculating their quantity in our AlexNet model.

If we have 96 filters of size 11×11 in a convolutional layer that takes in an input of 3 color channels, we would have $96 \times (11 \times 11) \times 3 = 34,848$ total weights in total. Each filter also has a bias term that is added after the convolutional, and hence there are a total of $34,848 + 96 = 34,944$ trainable parameters in the first convolutional layer of the AlexNet.

For batch normalization, we have two parameters for scaling and shifting the normalized data which are applied to each output filter. Since there are 96 filters, this batch normalization layer has $96 \times 2 = 192$ trainable parameters.

Finally, the top of our model consists of dense (or fully connected layers). If we have two fully connected layers, each neuron in the first layer will be connected to all neurons in the second layer, scaled by some weight parameter. Each neuron in the second layer also has a bias term. If we look at the first fully connected layer in AlexNet, it consists of connecting 9216 neurons and connecting them to a layer with 4096 neurons. This amounts to $9216 \times 4096 + 4096 = 37,752,832$ trainable parameters.

Let us compile everything together to calculate the total number of trainable parameters our AlexNet contains:

Layer	**Parameters**
Conv 1	$96 \times (11 \times 11) \times 3 + 96 = 34,944$
BatchNorm 1	$96 \times 2 = 192$
Conv 2	$256 \times (5 \times 5) \times 96 + 256 = 614,656$
BatchNorm 2	$256 \times 2 = 512$
Conv 3	$384 \times (3 \times 3) \times 256 + 384 = 885,120$
BatchNorm 3	$384 \times 2 = 768$
Conv 4	$384 \times (3 \times 3) \times 384 + 384 = 1,327,488$
BatchNorm 4	$384 \times 2 = 768$
Conv 5	$256 \times (3 \times 3) \times 256 + 256 = 884,992$
BatchNorm 5	$256 \times 2 = 512$
Dense 1	$9216 \times 4096 + 4096 = 37,752,832$
Dense 2	$4096 \times 4096 + 4096 = 16,781,312$
Dense 3 (connected to 5 output classes)	$4096 \times 5 + 5 = 20,485$
Total (trainable)	58,304.581

Table 1: Parameters for Each Layer

This brings us to a total of $58,304.581$ trainable parameters, with $\approx 93.6\%$ of them being used for the dense layers. We must also include non-trainable parameters for the model such as the moving mean and moving variance used in inference for batch normalization layers. There are also 2 of these parameters for each filter, meaning we will have $2,752$ non-trainable parameters. The total number of parameters in our implementation of AlexNet, trained specifically on 5 classes of flowers amounts to $58,307,333$ parameters.

2.3.4 Hyperparameters

Hyperparameters play an integral part in our model's training process and can significantly impact its performance. These parameters, or functions of them, are defined before training and are not

updated during the training process. Properly tuning hyperparameters can mean the difference between a model that performs well and one that does not. The hyperparameters we will use are listed here:

Parameter	Description
Learning Rate (η)	Controls how much the model's weights are adjusted with respect to the loss gradient. This is initialized based on the original paper written for each model and then changed accordingly based on the fall in validation loss. This is implemented using *schedulers* in TensorFlow which allows us to change the learning rate between each epoch based on certain conditions like change in validation loss, number of epochs passed, etc.
Batch Size (M)	The number of training examples used in one iteration. We use $M = 16$ for training for all models.
Number of Epochs	The number of times the learning algorithm will work through the entire training dataset. Since we want to evaluate the performance of the model only after convergence (i.e. it stops improving), the number of epochs is set arbitrarily high.
Momentum	Helps accelerate gradient vectors in the right direction. This is used in our *optimization function*.
Dropout Rate	Used to prevent overfitting by randomly setting a fraction of input units to 0 at each update during training time.
Regularization Parameter (λ)	Used to control overfitting by adding a penalty for larger weights in the loss function. L2 regularization for model training.

Table 2: Training Parameters and Their Descriptions

2.3.5 Regularisation

Regularisation is a technique that reduces *overfitting*, which is when our model performs very well on training data but poorly on validation/unseen data. This happens when our model focuses too much on reducing its categorical cross entropy loss and places too much emphasis on features local to the training set (a.k.a. "statistical noise"). Two regularization techniques known as *L2 Regularization* and *Dropout* are used for our models.

- **L2 Regularization:** Also known as *weight decay*, this technique penalizes the model for having weights with large values as it incorporates a penalty term proportionate to their magnitude. L2 regularization in a layer adds the mean of the square of the layer's weights into the loss function, multiplied by some regularization parameter λ. (Bias terms are not included as regularizing them has little to no effect on tackling overfitting)

$$L_{\text{reg}} = L_{\text{CCE}}(\mathbf{y}, \hat{\mathbf{y}}) + \lambda \sum_j w_j^2$$

  where w_j are the weights of the model.

  The term $\lambda \sum_j w_j^2$ is the L2 regularization term, which penalizes large weights and helps reduce the risk of overfitting by encouraging smaller weights.

- **Dropout:** Between each dense layer, we can set a dropout rate $0 \leq p < 1$ which randomly initializes a proportion p of the nodes in the input layer to have a value equal to 0 in each iteration. In the problem of overfitting, neurons within the same layer may learn through the errors made by other neurons, where the process of fixing these mistakes to adapt to statistical noise creates *co-adaptations*, where some neurons become highly dependent on other neurons.

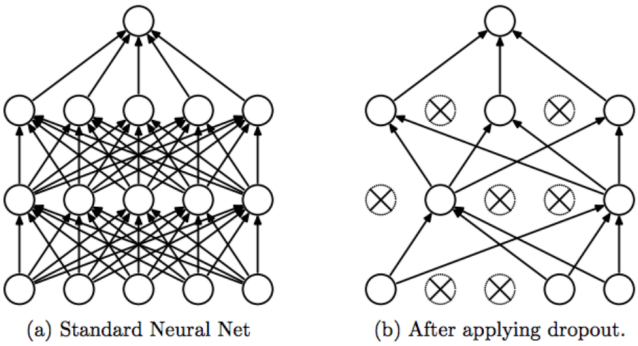


Figure 7: Visualization of Dropout layers (Source: Medium)

2.3.6 Transfer Learning

Models pre-trained on large datasets can serve as a good starting point for our parameter initialization in models with more specific functions. For instance, we can use the parameters from the VGG16 convolution layers that were trained on ImageNet, and then customize the parameters in the fully connected layers so our model specializes in classifying the 5 different flower groups. This is useful as the pre-trained model can already extract features from real-life objects competently, and training is quicker and more stable as we are only *fine-tuning* the parameters in the fully connected layers. However, since we are also interested in comparing training times across different model sizes and architectures, we will use pre-trained weights and train the entire model as well.

2.3.7 Confusion Matrix

A confusion matrix is a two-dimensional square array with rows and columns corresponding to each class in our dataset. Each row represents the true label, and each column represents the predicted label. All values in the cells ($\mathbf{y}$, $\hat{\mathbf{y}}$) will be divided by the total number of true labels corresponding to the row of that cell. If a model performs perfectly well, then the cells diagonal from the top left to bottom right (i.e. true label=predicted label) will have the value 1.

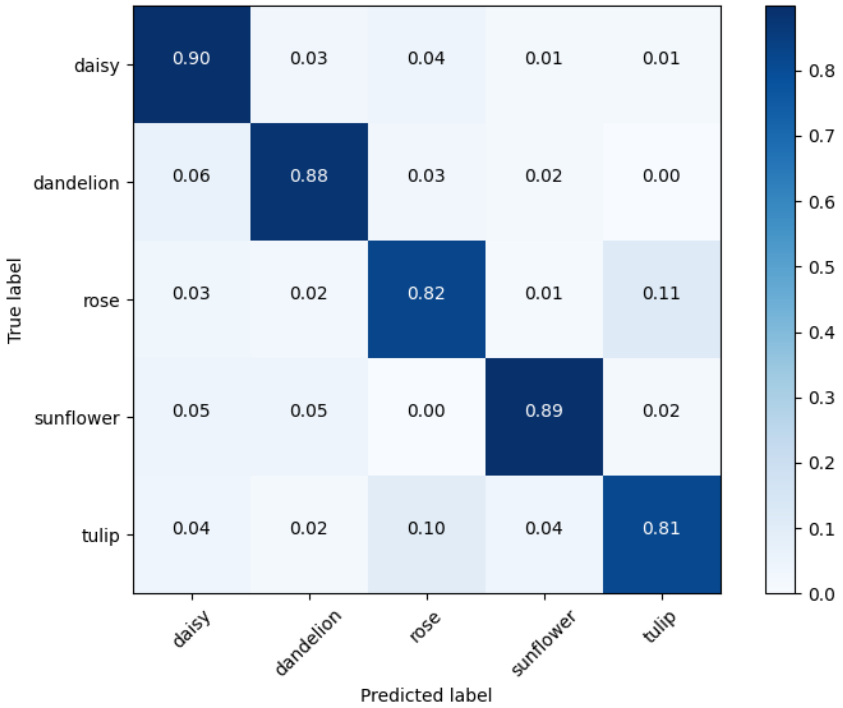


Figure 8: Confusion Matrix for VGG16

Confusion matrices are useful for identifying which labels a model performs poorly on.

2.4 Optimization

2.4.1 Stochastic Gradient Descent with Momentum

Stochastic Gradient Descent (SGD) is a technique that randomly utilizes only part of our training data to perform each step of gradient descent. SGD with momentum is an extension of the basic SGD algorithm that helps accelerate gradient vectors in the right directions, thus leading to faster converging. We can call it "SGD+" in this paper.

Let θ be the model parameters, $\nabla J(\theta)$ be the gradient of the loss function with respect to θ, and η be the learning rate. The momentum term v is introduced, which accumulates the gradient of the past steps to smooth out the updates.

The update rules for SGD with momentum are:

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

where:

- v_t is the velocity or the momentum term at iteration t.
- β is the momentum coefficient, typically set to a value between 0 and 1 (e.g., 0.9).

2.4.2 Adam Optimizer

Adam combines the benefits of two other extensions of SGD: adaptive learning rate methods and momentum. Adam computes individual adaptive learning rates for different parameters from estimates of the first and second moments of the gradients. This often results in faster convergence and more stable training.

The update rules for Adam are as follows:

- Compute the first-moment estimate (mean of the gradients):

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\theta_{t-1})$$

- Compute the second-moment estimate (uncentered variance of the gradients):

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla J(\theta_{t-1}))^2$$

- Correct the bias in the first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

- Update the parameters:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

where:

- m_t and v_t are the first and second moment estimates at iteration t, respectively.
- β_1 and β_2 are hyperparameters for controlling the exponential decay rates of the moment estimates, typically set to 0.9 and 0.999, respectively.
- ϵ is a small constant (e.g., 10^{-8}) added to prevent division by zero.

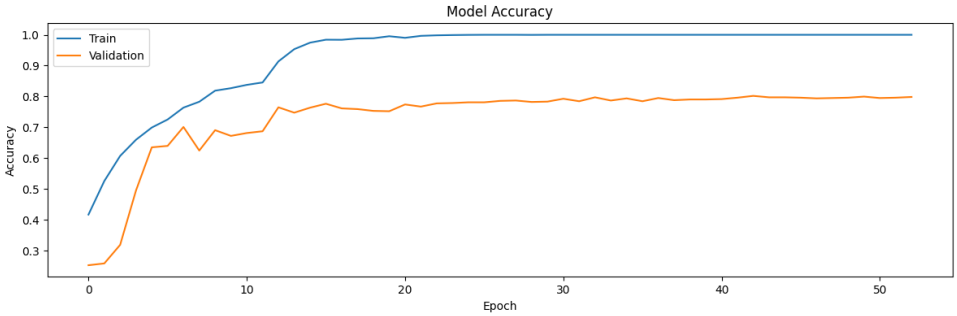


Figure 9: AlexNet trained using Adam

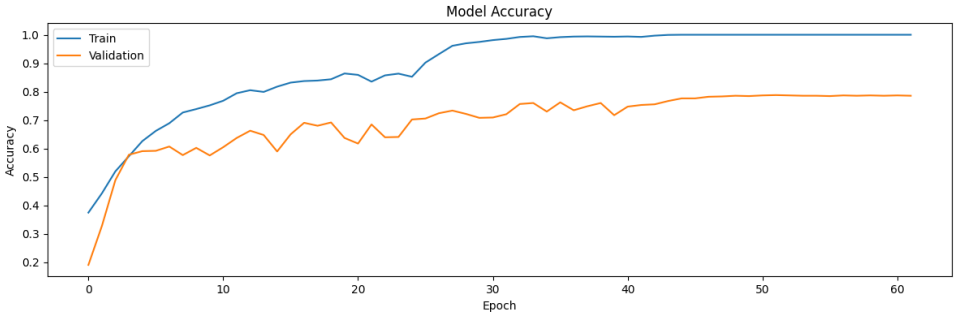


Figure 10: AlexNet trained using SGD+ Optimizer

2.4.3 Comparison

The nature of SGD means that convergence will generally take longer; the model takes more time to generalize to the dataset compared to training with Adam optimizer. As shown in the figures below, the validation accuracy curve is more volatile for the model trained with SGD+.

In essence, adaptive learning rate methods and momentum used in Adam often result in faster convergence and more stable training. On the other hand, SGD+ accelerates gradient vectors in the right directions, thus leading to faster converging, especially in the case of noisy gradients. While Adam adapts learning rates based on the gradients, which can be very beneficial in practice, SGD+ can sometimes provide better generalization by escaping shallow minima more effectively. In essence, Adam is generally more robust and faster to converge (i.e. for models with poor parameter initialization), but SGD+ can yield better final performance on certain tasks and datasets. Both optimizers can be ideal in the correct situations.

3 AlexNet vs. VGG16

In our first experiment, we will explore if a deeper CNN architecture improves image classification performance on our dataset. We hypothesize that increasing the depth of the network will allow for more complex feature extraction and thus better accuracy. Specifically, we will compare the performance of a standard CNN (AlexNet) to a much deeper architecture, such as VGG16.

3.1 VGG16

VGG16 stands for Visual Geometry Group model with 16 layers of depth and uses smaller 3x3 filters to extract more intricate features. Our VGG16 model uses 134, 289, 477 trainable parameters. The increase in depth stems from VGG16's use of smaller 3×3 convolutional filters that enhance feature extraction. The architecture of a VGG16 is shown below:

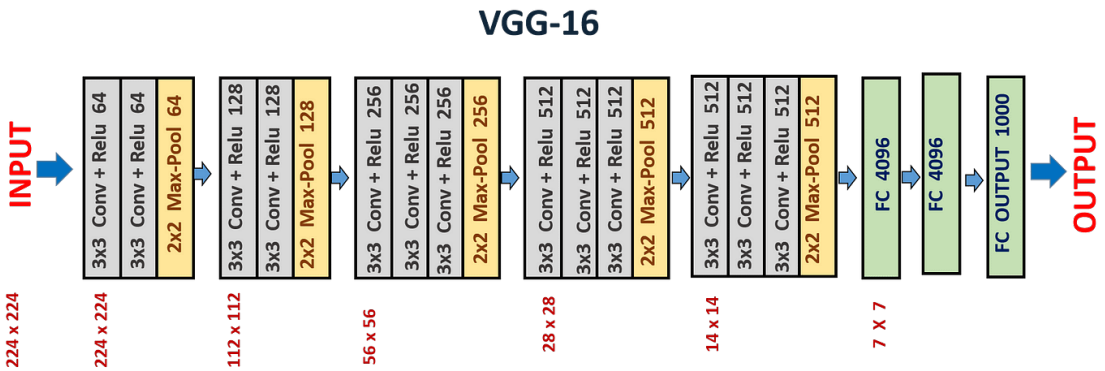


Figure 11: VGG16 architecture (Source: Medium)

3.2 Training

Beyond accuracy, we will also analyze the training and inference time for these two models to analyze the trade-offs. We hypothesize that AlexNet will perform not as well as the VGG16, but incur significantly cheaper training and inference times.

In our implementation of **all** of our CNNs in the further sections, we will pre-set the following hyperparameters:

Parameter	**Value**
Learning Rate (η)	Initially set to $1 \cdot 10^{-4}$. Reduced by a factor of 3 if validation accuracy has not improved for more than 5 iterations.
Number of Epochs	An early-stopping condition is set such that the model stops training if validation accuracy fails to improve (even after reducing learning rate), or 100 epochs (whichever comes first).
Momentum	**Adam:** $\beta_1 = 0.9$, $\beta_2 = 0.999$ **SGD+:** $\beta = 0.9$
Dropout Rate	0.5 for all fully connected layers.
Regularization Parameter	L2 Regularization; $\lambda = 0.01$ for non pre-trained convolution layers and all dense layers.

Figure 12: AlexNet and VGG16 training parameters

3.3 Comparison

The table below contains quantitative measures of the models' performances derived from training.

- Optimizer: The type of optimizer being used in conjunction with the model
- Epochs: Number of training iterations until convergence condition
- T_{train}/s: is the total training time until convergence
- t_{epoch}/s: is the training time per training iteration (216 batches of 16 images with a remainder batch of 13 images)
- t_{infer}/ms: inference time per batch (16 images)
- Train Acc.: Observed training accuracy at the final epoch
- Val Acc.: Observed validation accuracy at the final epoch (as compared to the largest observed accuracy). This is used to benchmark the model's performance.

Model	No. Trainable Params	Optimizer
AlexNet	58,304,581	Adam
VGG16	134,281,029	Adam

Figure 13: Model Parameters

Model	Optimizer	Epochs	T_{train}/s	t_{epoch}/s	t_{infer}/ms	Train. Acc.	Val. Acc.
AlexNet	SGD+	70	803	11.47	22	99.25%	81.97%
AlexNet	Adam	43	490	11.40	22	100%	84.17%
VGG16	Adam	25	2160	86.40	87	99.88%	89.92%

Figure 14: Comparison Between AlexNet and VGG16

3.4 Conclusion

We observe that VGG16 (Adam) yields the best validation accuracy, followed by AlexNet (Adam) and AlexNet (SGD+). As expected, due to the larger number of parameters, the training time per epoch and inference time are significantly higher for VGG16 models. although these performance times do not scale directly proportionately likely due to resource constraints and how tensorflow scales larger models. AlexNet (SGD+) took the most number of iterations (epochs) before converging, hence we will stick to Adam optimizer for the other models ahead. Both models perform exceptionally well on the training set, signifying a certain degree of overfitting despite regularization.

We can conclude that deeper models will perform slightly better, although the more than double the number of parameters in VGG16 do not translate to an equally significant boost in accuracy. Hence, AlexNet would be more suitable for most classification tasks, while VGG16 can be used for less intensive and higher accuracy tasks.

4 Improved CNN models

Many other network architectures are imbued with the core concept of CNNs, such as Residual Networks (e.g. ResNet50), Inception, Efficient, and MobileNet

4.1 ResNet

ResNet, short for Residual Networks, was introduced by Kaiming He et al. in the paper "Deep Residual Learning for Image Recognition" in 2015, and it addresses the problem of vanishing gradients, which often occur when training deep neural networks.

4.1.1 Skip Connections

ResNet introduces a novel architecture with *skip connections*, or *shortcuts*, which enable the model to skip one or more layers. The main idea is to let these layers fit a residual mapping instead of directly fitting a desired underlying mapping. Formally, if the desired underlying mapping is $H(x)$, we let the stacked nonlinear layers fit another mapping of $F(x) = H(x) - x$. The original mapping is then recast into $F(x) + x$.

$$H(x) = F(x) + x \tag{1}$$

Skip connections are the core of ResNet's architecture, and they allow gradients to flow through the network more easily, addressing the issue of vanishing gradients. The presence of skip connections enables the training of much deeper networks than was previously possible.

4.1.2 ResNet Architecture

The ResNet architecture consists of multiple blocks, each containing several layers. The building block for ResNet is a *residual block*, as shown in Figure 15. Each block consists of a few convolutional layers, followed by batch normalization and ReLU non-linearization. The output of these layers is added to the input of the block through the skip connection.

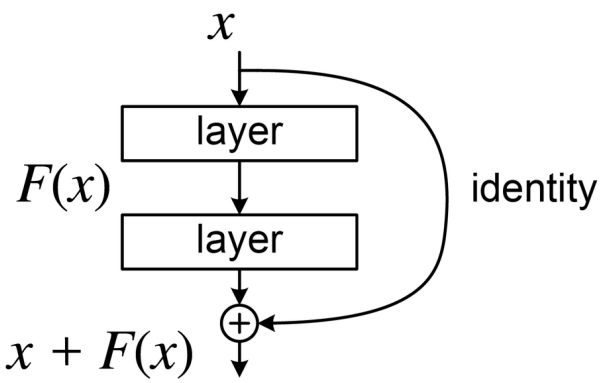


Figure 15: Residual Block (Source: Wikipedia)

These blocks are combined to form very deep networks, such as ResNet-50, ResNet-101, and ResNet-152, where the numbers represent the number of layers in the network. Deeper networks require *bottleneck blocks* which utilizes 1×1 filters to reduce the number of parameters and operations required later on.

4.2 Inception

Inception (also known as GoogLeNet), was introduced by Szegedy et al. in the paper "Going Deeper with Convolutions" in 2014. The Inception architecture allows increased model depth and width without a significant rise in computational cost, through optimizing the use of computation resources in the network.

4.2.1 Inception Modules

The core idea of Inception is the Inception module, which allows the network to use multiple convolutional filter sizes within the same module. Instead of deciding the filter size in advance, the Inception module performs convolutions with multiple filter sizes (e.g., 1×1, 3×3, 5×5) and then concatenates the results across the channels. This allows the network to capture features at different scales in each inception module.

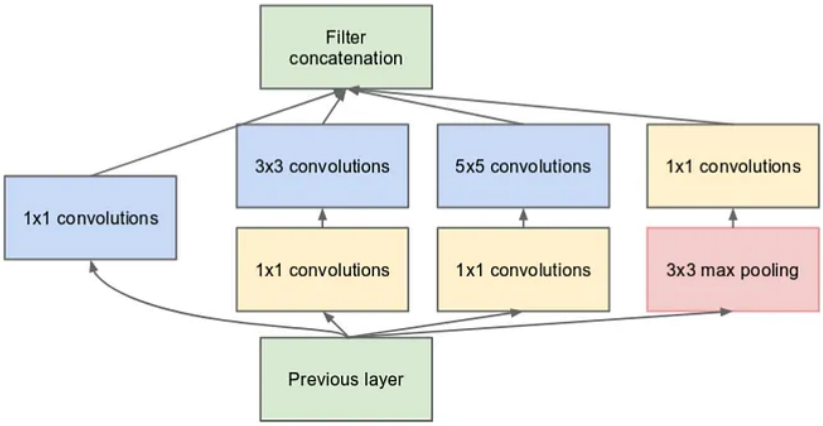


Figure 16: Inception Module (Source: Inception V1)

4.2.2 Inception Architecture

The Inception architecture consists of multiple Inception modules stacked upon each other. Each module performs convolutions with different filter sizes and concatenates the results. Additionally, 1×1 convolutions are used to reduce the dimensionality and computational cost. This is done before the more computationally expensive 3×3 and 5×5 convolutions. Recall the dimensions of a convolutional 4D tensor $C_{in} \times C_{out} \times k \times k$; 1×1 convolutions compresses the number of channels $C_{out} < C_{in}$(i.e. depthwise reduction) so that fewer filters of higher dimensions are required later on.

4.2.3 Auxillary output

To allow a strong error signal to be sent through the network, auxiliary classifiers are placed along the depth of the network and make predictions purely based on layers up to that point. The total loss of the model is then computed with both the loss from the main classifier and a portion of the auxiliary classifiers, e.g.

$$L_{Total} = L_{main} + 0.3 \cdot L_{aux1} + 0.3 \cdot L_{aux2}$$

Figure 17: Auxiliary Classifier

4.3 EfficientNet

EfficientNet is a CNN that achieves impressive accuracy while being computationally efficient. Introduced by Tan and Le in the paper "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks" in 2019, EfficientNet uses a systematic approach to scaling up neural networks in a balanced manner.

4.3.1 Compound Scaling

Traditional methods for scaling up neural networks, such as increasing depth (more layers) or width (more channels), are often unbalanced and inefficient. *Compound scaling*, however, uniformly scales network depth, width, and resolution using a set of fixed scaling coefficients. This approach ensures that the network maintains a balance between these dimensions, leading to better performance and efficiency.

$$\text{depth: } d = \alpha^k, \quad \text{width: } w = \beta^k, \quad \text{resolution: } r = \gamma^k \tag{2}$$

where α, β, and γ are constants determined through a *grid search*, and k is some scaling coefficient.

EfficientNet starts with a baseline network (EfficientNet-B0), which is optimized for both accuracy and efficiency. This baseline is then scaled up to create a family of models (EfficientNet-B1 to B7) using the compound scaling method.

The architecture of EfficientNet is based on the MobileNetV2 architecture but includes several modifications for improved performance. Key components include:

- **Mobile Inverted Bottleneck Convolution (MBConv)**: EfficientNet uses MBConv blocks, which are efficient building blocks that combine depthwise separable convolutions with inverted residuals.
- **Squeeze-and-Excitation Optimization (SE)**: SE blocks are incorporated to improve the representational power of the network by adaptively recalibrating channel-wise feature responses.

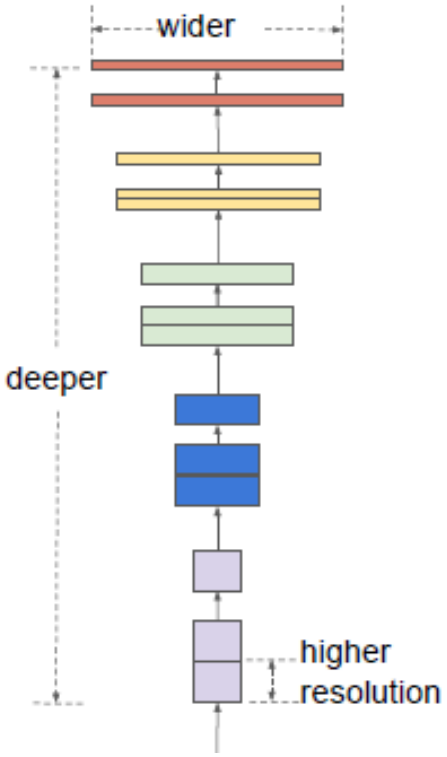


Figure 18: EfficientNet Scaling (Source: M. Tan, Quoc V. Le)

4.4 MobileNet

MobileNet is a family of convolutional neural networks designed specifically for efficient execution on mobile and embedded vision applications. Introduced by Howard et al. in the paper "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications" in 2017, MobileNet models are based on a streamlined architecture that uses depthwise separable convolutions to reduce the number of parameters and computational cost.

4.4.1 Depthwise Separable Convolutions

The core idea of MobileNet is the use of depthwise separable convolutions, which factorize a standard convolution into two separate layers: a depthwise convolution and a pointwise convolution.

- **Depthwise Convolution**: This performs a single convolution on each input channel (input depth), thus filtering the input channels.
- **Pointwise Convolution**: This uses 1×1 convolutions to combine the outputs of the depthwise convolution.

By separating the convolution into these two layers, MobileNet drastically reduces the computational cost and the number of parameters compared to traditional convolutions.

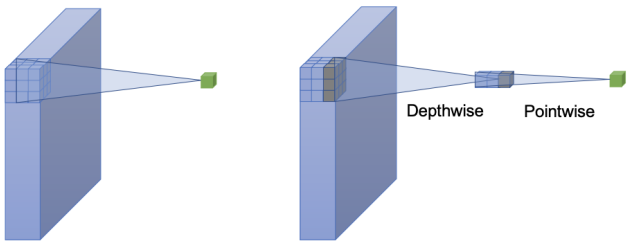


Figure 19: Depthwise Separable Convolution (Source: Xception)

The architecture of MobileNet is built on depthwise separable convolutions. Each layer consists of a depthwise convolution followed by a pointwise convolution. The architecture includes several key components:

- **Depthwise Separable Convolutions**: As described, these form the basic building blocks of the network.
- **Width Multiplier (α)**: This is a hyperparameter that adjusts the number of channels in each layer. It reduces the width of the network uniformly at each layer.
- **Resolution Multiplier (ρ)**: This is a hyperparameter that reduces the input image resolution, further reducing the computational cost.

4.5 Observations

After training each model for 3 times, their performance metrics are averaged and shown in the figure below.

Model	No. Trainable Params	Optimizer
AlexNet	58,304,581	Adam
VGG16	134,281,029	Adam
ResNet50	23,544,837	Adam
InceptionV3	23,871,653	Adam
EfficientNet	5,366,440	Adam
MobileNetV2	3,540,741	Adam

Figure 20: Model Parameters

Model	Epochs	T_{train}/s	t_{epoch}/s	t_{infer}/ms	Train. Acc.	Val. Acc.
AlexNet	43	490.9	11.40	22	100%	84.17%
VGG16	25	2160	86.40	87	99.88%	89.92%
ResNet50	14	707.1	50.46	60	100%	93.40%
InceptionV3	11	427.6	38.87	57	100%	91.43%
EfficientNet	14	524.2	37.31	43	99.88%	92.12%
MobileNet	19	410.7	21.62	22	100%	92.24%

Figure 21: Comparison Between ResNet50, InceptionV3, EfficientNet and MobileNet

5 Conclusions

Below we compare the number of parameters with validation accuracy, training time per epoch, and inference time.

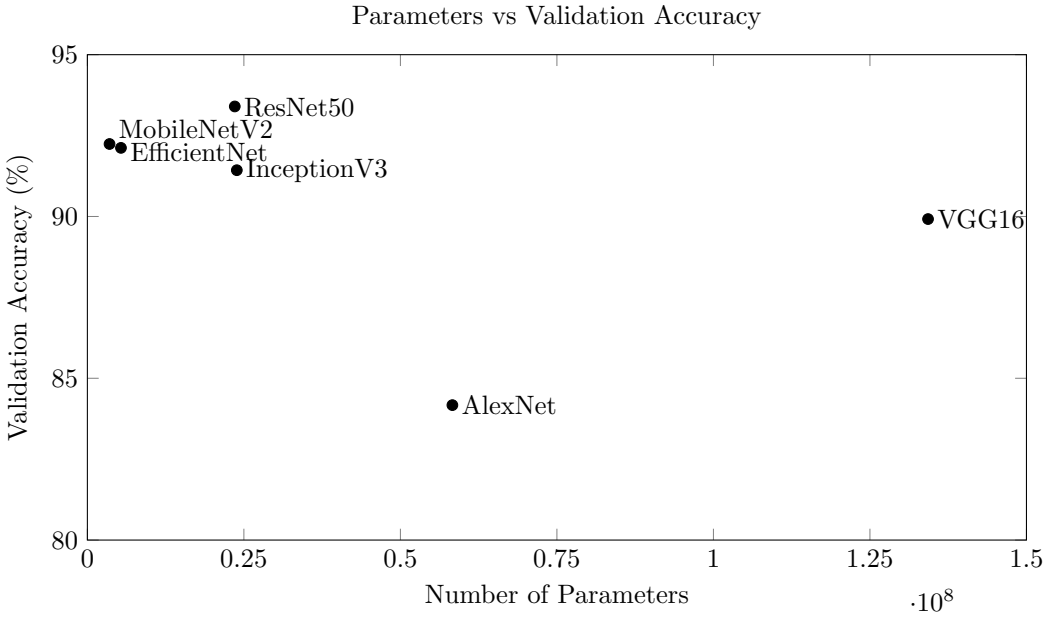


Figure 22: Parameters vs Validation Accuracy

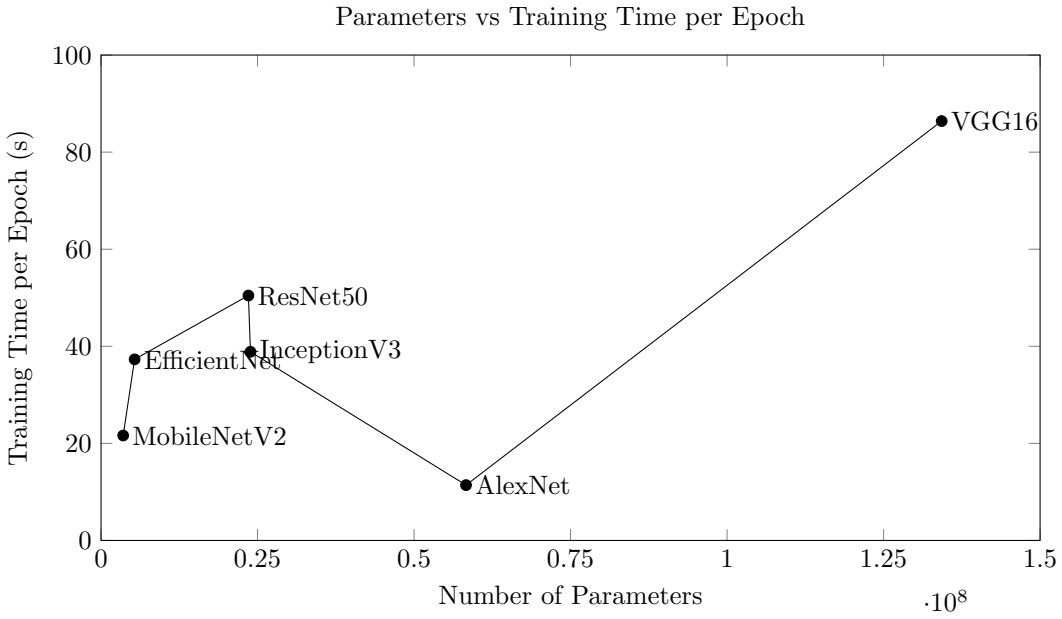


Figure 23: Parameters vs Training Time per Epoch

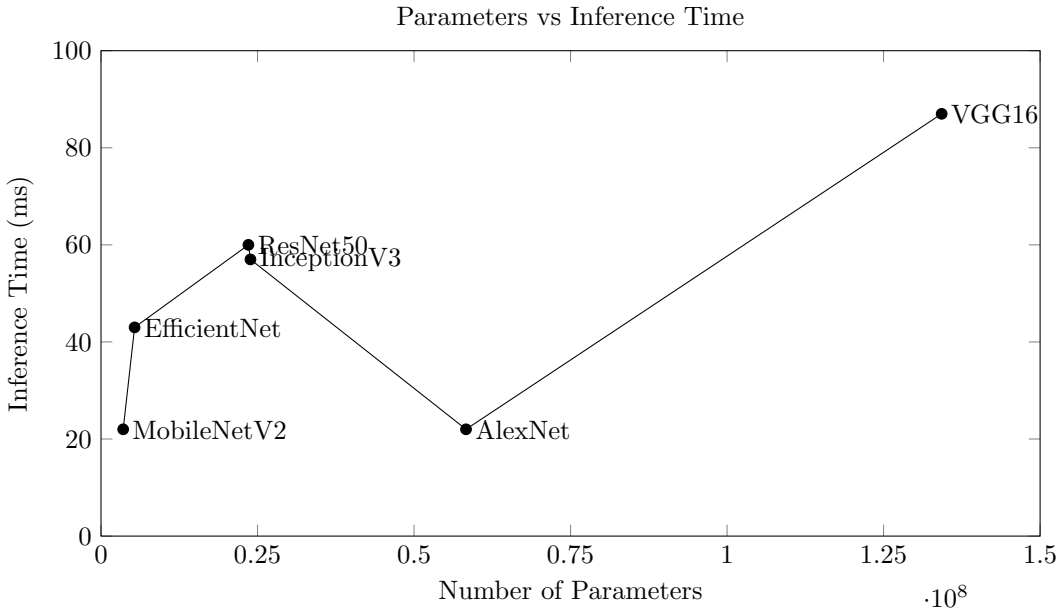


Figure 24: Parameters vs Inference Time

To conclude, the number of parameters does not necessarily imply greater accuracy, instead, the architecture of the model plays a greater role. The inference time and training time per epoch exhibit similar patterns, although they do not have a clear trend upward with respect to parameter count. This is due to different CNN architectures using vastly different operations such as skip networks and depthwise convolutions. We observe that the more modern CNNs (Resnet50, InceptionV3, EfficientNet, MobileNet) cluster about the 90% mark for validation accuracy and have significantly faster inference and training times compared to AlexNet and VGG16. MobileNet produces the quickest inference times and second-highest accuracy despite only having 3.5 million parameters, making it our best candidate for flower classification. It is possible to use grid search to find more favorable hyperparameters about each model, but due to resource constraints, we chose to fix them for all models.