BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC PHENIKAA

----------oOo----------

FINAL REPORT

DEEP LEARNING

CLASSIFY ANIMALS BASED ON IMAGES

Student - ID: Nguyễn Thế Tường - 21011079

Vũ Hoài Nam - 21012556

Nguyễn Ngọc Hiếu - 21011956

Advisor: **Dr. Le Minh Huy**

Hanoi, 24th June 2024

MỤC LỤC

Danh mục hình ảnh

- Here is the link to my presentation video: https://youtu.be/isaP7OMmbN8?si=Uo5lWMV-BrkrOKZ1
- Link to the datasets: https://drive.google.com/file/d/1-avZ32ej9QdNSdCjjpFRfJQ-KYWti4ah/view?usp=sharing
- After training, the model is saved to file h5:
  - Retnet50: https://drive.google.com/file/d/1X60jYbvnm-xRuay4Ep87SGi1VxvI3UH3/view?usp=drive_link
  - EfficientNetB7: https://drive.google.com/file/d/1USLxqnAsEkqpTNnew1XFIY-4aGg9iXEp/view?usp=sharing

CHAPTER 1: INTRODUCTION

1.1 Project Definition

We introduces convolutional neural networks (CNNs), a powerful family of neural networks that are designed for precisely this purpose. CNN-based architectures are now ubiquitous in the field of computer vision. For instance, on the Imagnet collection it was only the use of convolutional neural networks, in short Convnets, that provided significant performance improvements.

Modern CNNs, as they are called colloquially, owe their design to inspirations from biology, group theory, and a healthy dose of experimental tinkering. In addition to their sample efficiency in achieving accurate models, CNNs tend to be computationally efficient, both because they require fewer parameters than fully connected architectures and because convolutions are easy to parallelize across GPU cores. Consequently, practitioners often apply CNNs whenever possible, and increasingly they have emerged as credible competitors even on tasks with a one-dimensional sequence structure, such as audio, text, and time series analysis, where recurrent neural networks are conventionally used. Some clever adaptations of CNNs have also brought them to bear on graph-structured data and in recommender systems.

1.2 Project Objectives

The primary objectives of this project are:

- To understand the fundamental principles and components of Convolutional Neural Networks.
- To review and analyze significant CNN architectures and their contributions to the field.
- To examine the role of activation functions in CNNs and compare their impact on network performance.
- To evaluate different optimization functions and their effectiveness in training CNN models.
- To implement and test CNN models on a chosen dataset, using various preprocessing and optimization techniques.

- To provide recommendations based on experimental results regarding the best practices in CNN model training and optimization.

CHAPTER 2: LITERATURE OVERVIEW

2.1 Convolutional Neural Network (CNN)

Convolutional Neural Networks are specialized types of artificial neural networks designed to process structured grid data, such as images. They consist of several layers, including convolutional layers, pooling layers, and fully connected layers, each playing a distinct role in extracting and processing features from the input data.

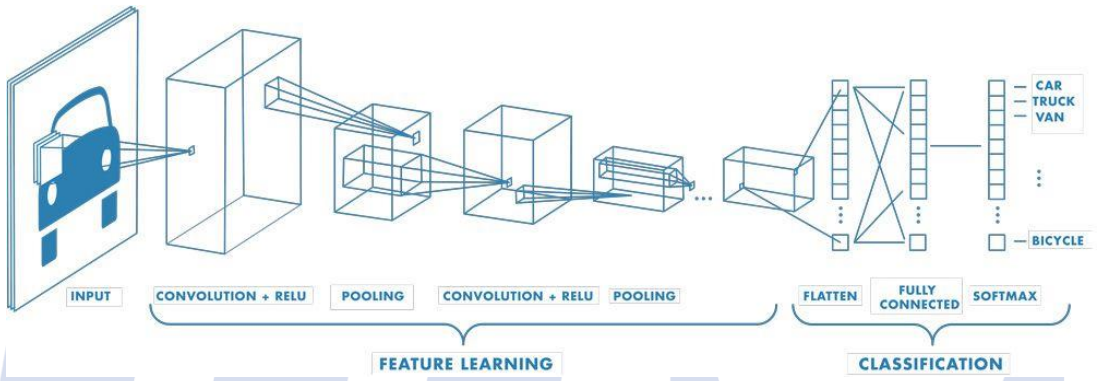


Fig 1. Generalizing the CNN model

2.1.1 Convolutions for Images

In a convolutional layer, an input array and an array of correlation kernels are combined to produce an output array using the cross-correlation operation. In the image below, the input is a two-dimensional array with length **3** and width **3**. We denote the size of the array as **(3×3)** or **(3, 3)**. The length and width of the kernel are both **2**. This array can also be called a convolution kernel, a filter, or simply a layer weight. The size of the nuclear window is the length and width of the nucleus (here, **2×2**).

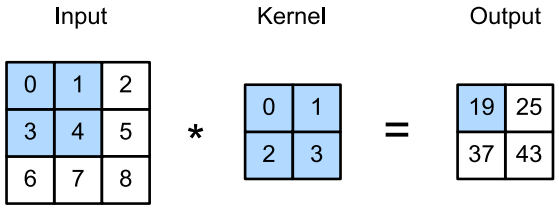


Fig 2. Convolution multiplication for 1 layer

In two-dimensional cross-correlation, we start with a convolution window located at the upper left corner of the input array and move this window from left to right and from top to bottom. When the convolution window is pushed to a certain position, the input subarray is in that window and the kernel array is multiplied element by element, and then we sum the elements in the result array to get a single scalar numeric value. This value is written to the output array at the corresponding location.

Note that along each axis, the output size is slightly smaller than the input. Because the kernel has length and width greater than one, we can only calculate cross-correlation for locations where the kernel lies completely inside the image, the output size is calculated by taking the input **(H×W)** minus the size of the convolution filter **h×w** equals **(H−h+1)×(W−w+1)**. This happens because we need enough space to 'move' the convolution kernel across the image (we'll see later how we can preserve the same size by padding zeros around the edges of the image image such that there is enough space to move the nucleus).

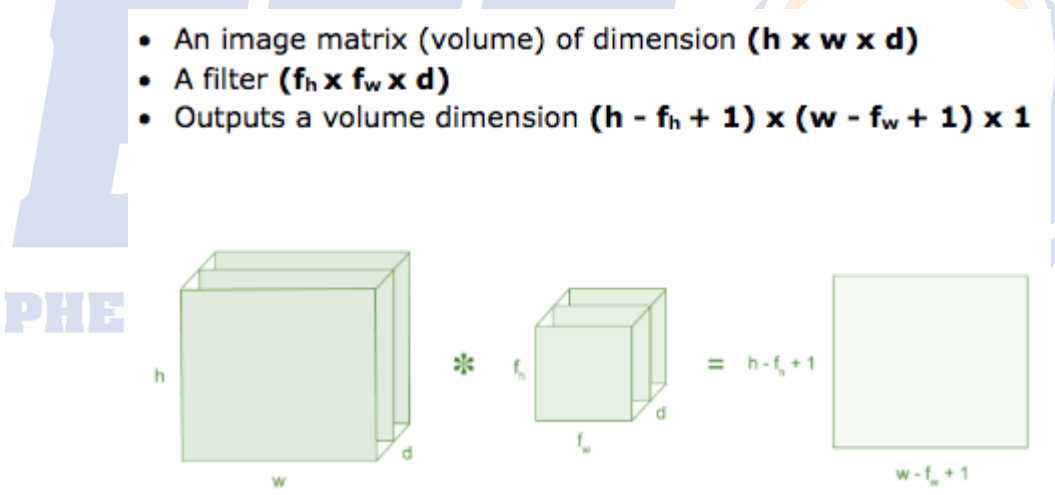


Fig 3. Convolution multiplication for multiple layers

2.1.2 Padding and Stride

In the previous example, the input had both length and width equal to **3**, the convolution kernel window had both length and width equal to **2**, so we obtained an output representation of size **2×2**. In general, assuming the size of the input is $\mathbf{n_h \times n_w}$ and the size of the convolution kernel window is $\mathbf{k_h \times k_w}$, the size of the output will be:

$$\mathbf{(n_h - k_h + 1) \times (n_w - k_w + 1).}$$

Therefore, the size of the convolution layer output is determined by the input size and the convolution kernel window size.

- Padding:

One tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. Consider that depicts the pixel utilization as a function of the convolution kernel size and the position within the image. The pixels in the corners are hardly used at all.

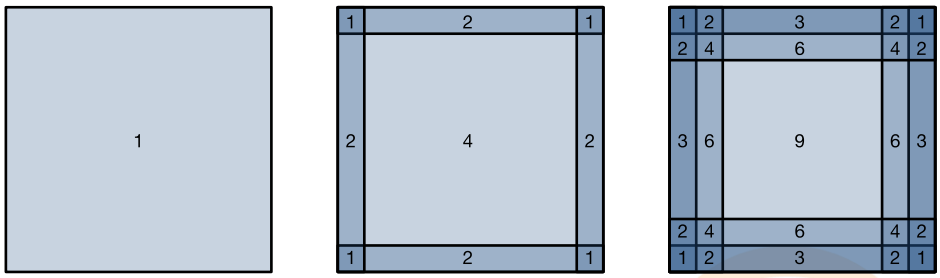


Fig 4. Borders for images

Since we typically use small kernels, for any given convolution we might only lose a few pixels but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero. We pad a **3×3** input, increasing its size to **5×5**. The corresponding output then increases to a **4×4** matrix.

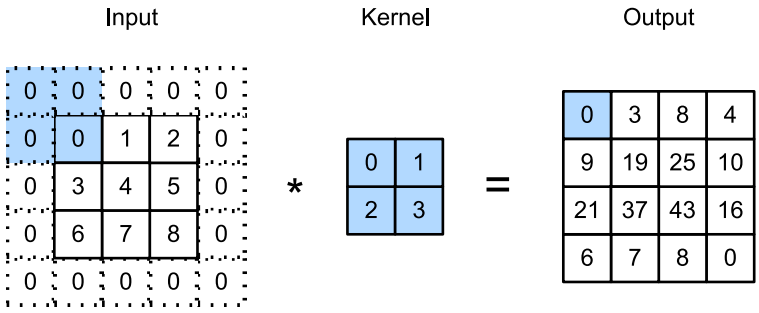


Fig 5. Padding for matrix

In general, if we add a total of $\mathbf{p_h}$ rows of padding (roughly half on top and half on bottom) and a total of $\mathbf{p_w}$ columns of padding (roughly half on the left and half on the right), the output shape will be:

$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$.

- Stride:

When computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right. Sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations. This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image.

We refer to the number of rows and columns traversed per slide as stride. So far, we have used strides of **1**, both for height and width. Sometimes, we may want to use a larger stride. We shows a two-dimensional cross-correlation operation with a stride of **3** vertically and **2** horizontally.

We can see that when the second element of the first column is generated, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is generated. When the convolution window continues to slide two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

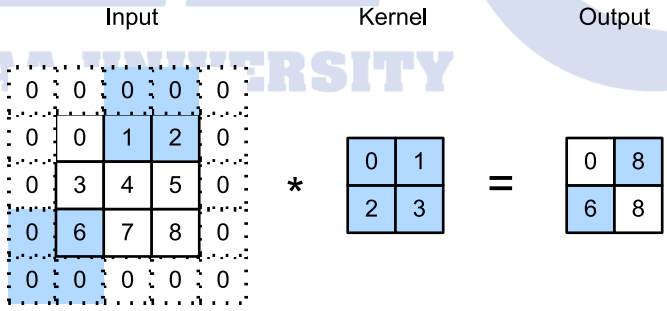


Fig 6. Stride for matrix

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is:

$[(n_h - k_h + p_h + s_h)/s_h] \times [(n_w - k_w + p_w + s_w)/s_w]$.

2.1.3 Multiple Input and Multiple Output Channels

When the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-

correlation with the input data. Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i. If our convolution kernel's window shape is $k_h \times k_w$, then, when $c_i=1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_h \times k_w$.

However, when $c_i>1$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for every input channel. Concatenating these c_i tensors together yields a convolution kernel of shape $c_i \times k_h \times k_w$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional tensor. This is the result of a two-dimensional cross-correlation between a multi-channel input and a multi-input-channel convolution kernel.

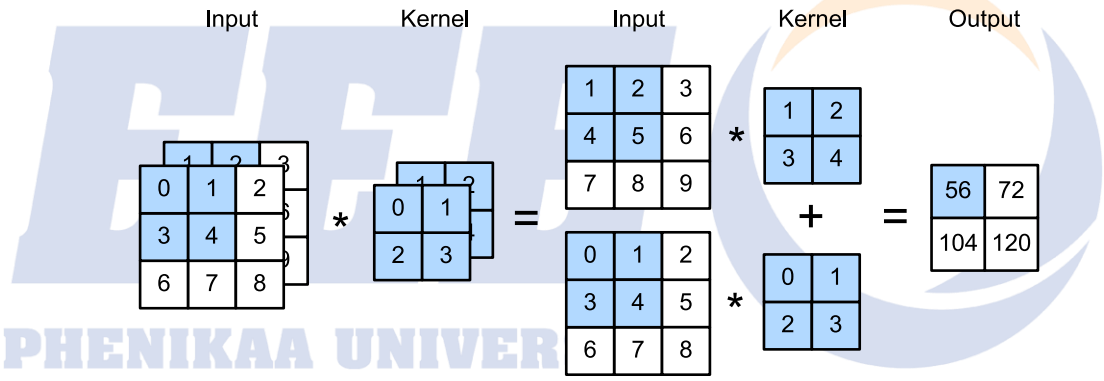


Fig 7. Input and output convolutional layers

2.1.4 Pooling

Like convolutional layers, pooling operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the pooling window). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no kernel). Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called maximum pooling (max-pooling for short) and average pooling, respectively.

In both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the upper-left of the input tensor and sliding across it from left to right and top to bottom. At each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window, depending on whether max or average pooling is employed.

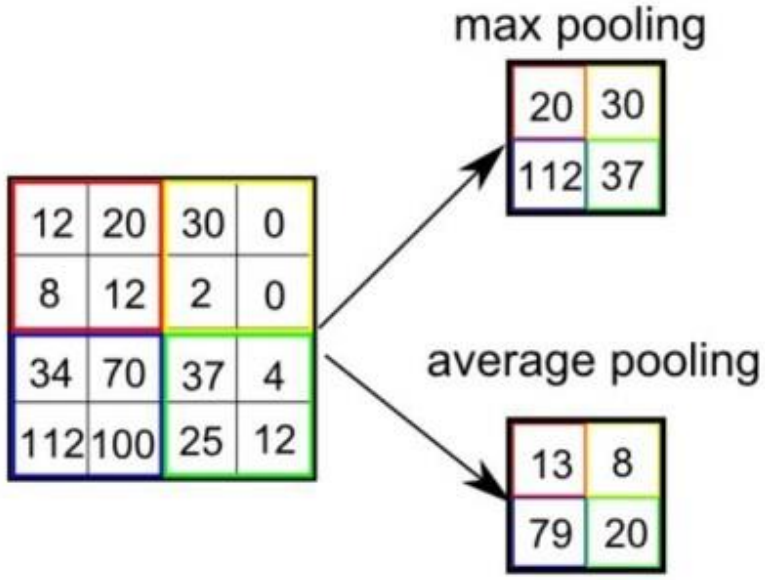


Fig 8. Pooling for matrix

2.2 Convolutional Neural Network Models (CNNs)

Several CNN architectures have been developed to address various challenges in image processing:

2.2.1 AlexNet:

AlexNet, which employed an 8-layer CNN, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a large margin (Russakovsky et al., 2013). This network showed, for the first time, that the features obtained by learning can transcend manually-designed features, breaking the previous paradigm in computer vision.

The architectures of AlexNet and LeNet are strikingly similar, as figure illustrates below.

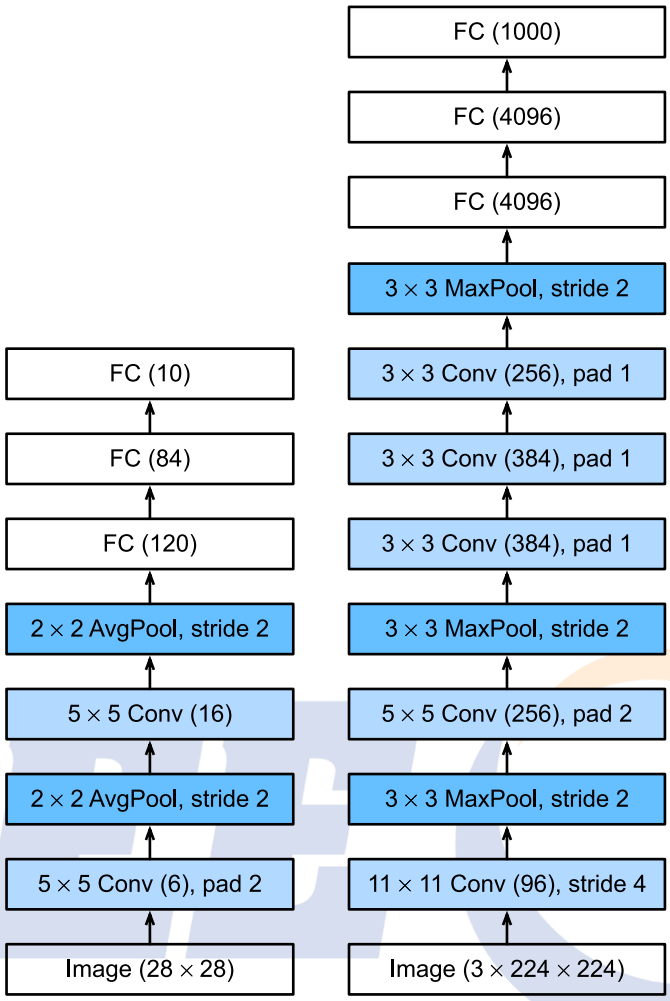


Fig 9. LeNet and AlexNet model

There are also significant differences between AlexNet and LeNet. First, AlexNet is much deeper than the comparatively small LeNet-5. AlexNet consists of eight layers: five convolutional layers, two fully connected hidden layers, and one fully connected output layer. Second, AlexNet used the ReLU instead of the sigmoid as its activation function.

In AlexNet's first layer, the convolution window shape is **11x11**. Since the images in ImageNet are eight times taller and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels with more visual detail. Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to **5x5**, followed by **3x3**. In addition, after the first, second, and fifth convolutional layers, the network adds max-pooling layers with a window shape of **3x3** and a stride of **2**.

Moreover, AlexNet has ten times more convolution channels than LeNet. After the final convolutional layer, there are two huge fully connected layers with 4096 outputs.

2.2.2 VGG:

Like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers and the second consisting of fully connected layers that are identical to those in AlexNet. The key difference is that the convolutional layers are grouped in nonlinear transformations that leave the dimensonality unchanged, followed by a resolution-reduction step, as depicted in figure below.

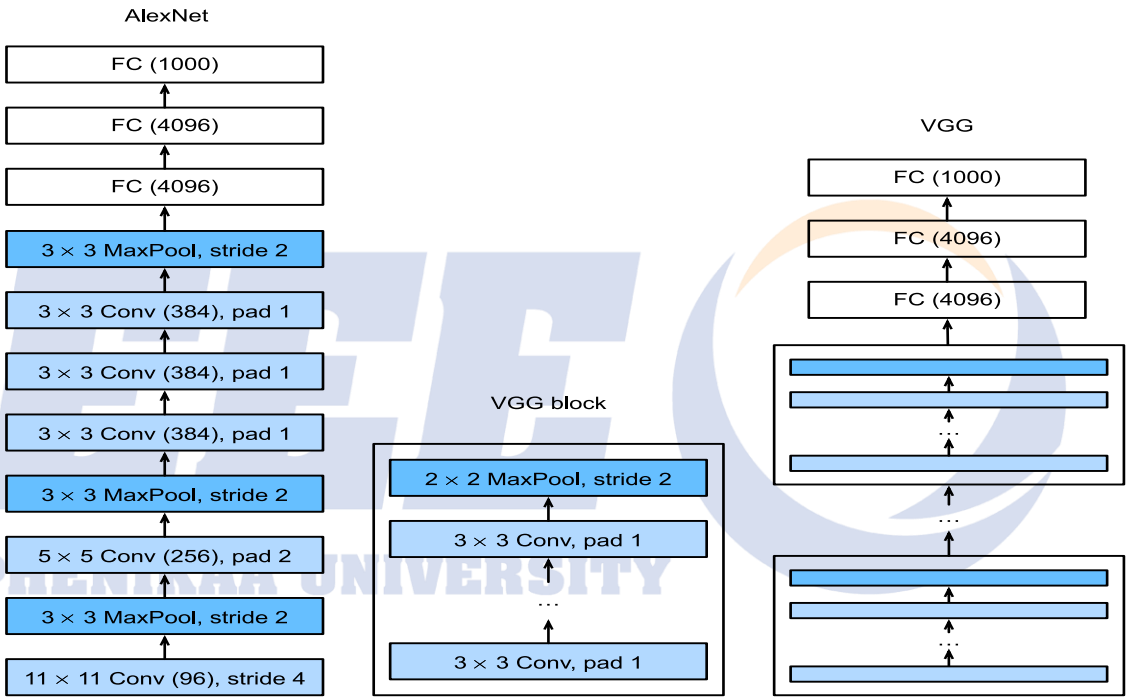


Fig 10. VGG model

The key difference is that VGG consists of blocks of layers, whereas AlexNet's layers are all designed individually. The convolutional part of the network connects several VGG blocks from figure (also defined in the vgg_block function) in succession. This grouping of convolutions is a pattern that has remained almost unchanged over the past decade, although the specific choice of operations has undergone considerable modifications. The variable arch consists of a list of tuples (one per block), where each contains two values: the number of convolutional layers and the number of output channels, which are precisely the arguments required to call the vgg_block function. As such, VGG defines a family of networks rather

than just a specific manifestation. To build a specific network we simply iterate over arch to compose the blocks.

2.2.3 Network in Network (NiN):

We said that the inputs and outputs of convolutional layers consist of four-dimensional tensors with axes corresponding to the example, channel, height, and width. Also recall that the inputs and outputs of fully connected layers are typically two-dimensional tensors corresponding to the example and feature. The idea behind NiN is to apply a fully connected layer at each pixel location (for each height and width). The resulting **1x1** convolution can be thought of as a fully connected layer acting independently on each pixel location.

Figure illustrates below the main structural differences between VGG and NiN, and their blocks. Note both the difference in the NiN blocks (the initial convolution is followed by **1x1** convolutions, whereas VGG retains **3x3** convolutions) and at the end where we no longer require a giant fully connected layer.

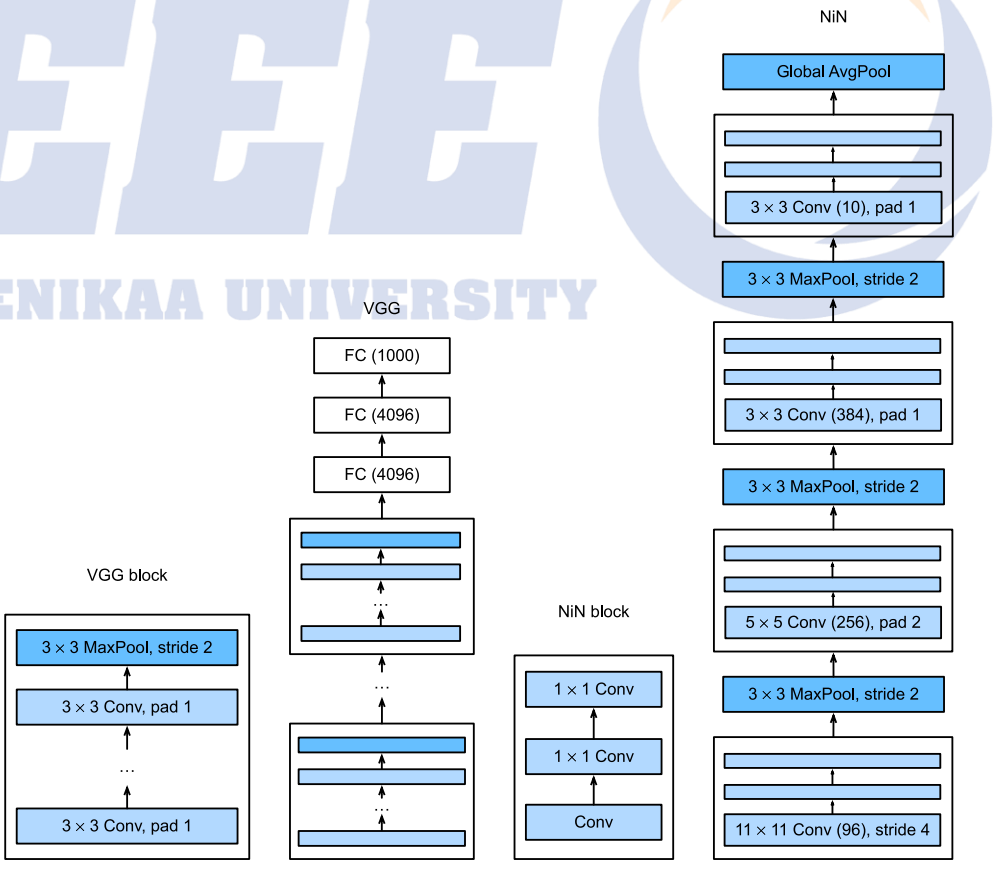


Fig 11. VGG vs NiN model

NiN uses the same initial convolution sizes as AlexNet (it was proposed shortly thereafter). The kernel sizes are **11x11**, **5x5**, and **3x3**, respectively, and the numbers of output channels match those of AlexNet. Each NiN block is followed by a max-pooling layer with a stride of **2** and a window shape of **3x3**.

The second significant difference between NiN and both AlexNet and VGG is that NiN avoids fully connected layers altogether. Instead, NiN uses a NiN block with a number of output channels equal to the number of label classes, followed by a global average pooling layer, yielding a vector of logits. This design significantly reduces the number of required model parameters, albeit at the expense of a potential increase in training time.

2.2.4 GoogLeNet:

The basic convolutional block in GoogLeNet is called an Inception block.

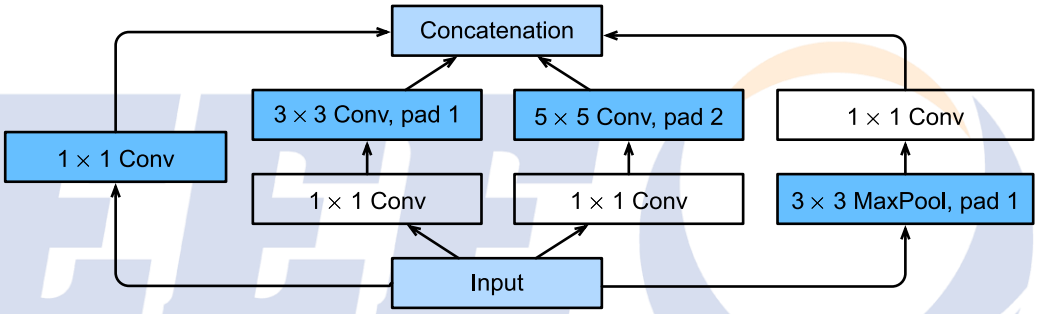


Fig 12. Inception block in GoogLeNet model

As depicted in figure, the inception block consists of four parallel branches. The first three branches use convolutional layers with window sizes of **1x1**, **3x3**, and **5x5** to extract information from different spatial sizes. The middle two branches also add a **1x1** convolution of the input to reduce the number of channels, reducing the model's complexity. The fourth branch uses a **3x3** max-pooling layer, followed by a **1x1** convolutional layer to change the number of channels. The four branches all use appropriate padding to give the input and output the same height and width. Finally, the outputs along each branch are concatenated along the channel dimension and comprise the block's output. The commonly-tuned hyperparameters of the Inception block are the number of output channels per layer, i.e., how to allocate capacity among convolutions of different size.

GoogLeNet uses a stack of a total of 9 inception blocks, arranged into three groups with max-pooling in between, and global average pooling in its head to generate its estimates.

Max-pooling between inception blocks reduces the dimensionality. At its stem, the first module is similar to AlexNet and LeNet.

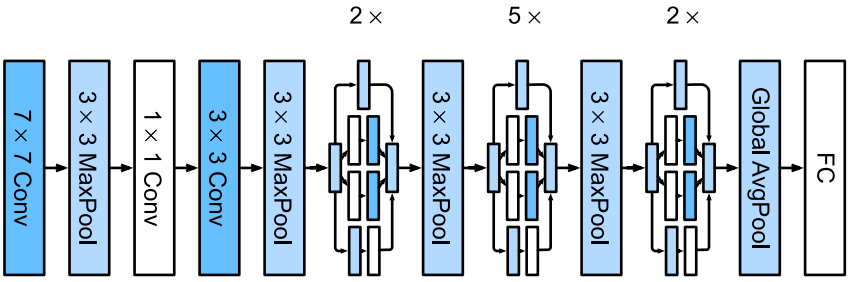


Fig 13. GoogLeNet Model

2.2.5 ResNet:

- Function Classes:

Consider **F** to be a class of functions that a particular network architecture (along with learning rate and other hyperparameters) can achieve. In other words, for any function **f∈F**, there exists some set of parameters **W** that can be found by training on a suitable data set. Suppose **f*** is the function to be found. It would be advantageous if this function belonged to the set **F**, but usually not so lucky. Instead, we will try to find the best possible functions $\mathbf{f^*_F}$ in the set **F**. For example, we can try to find $\mathbf{f^*_F}$ by solving the following optimization problem:

$$\mathbf{f^*_F := \operatorname{argmin}_f L(X,Y,f) \text{ subject to } f \in F.}$$

It is reasonable to assume that if a different, more powerful architecture **F′** is designed, better results will be achieved. In other words, we expect the function $\mathbf{f^*_{F'}}$ to be "better" than $\mathbf{f^*_F}$. However, if **F⊈F′**, then it cannot be claimed that $\mathbf{f^*_{F'}}$ is "better" than $\mathbf{f^*_F}$. In fact, $\mathbf{f^*_{F'}}$ might be worse. And this is often the case - adding layers does not always increase the performance of the network and sometimes causes very unpredictable changes. Figure below illustrates this more clearly.

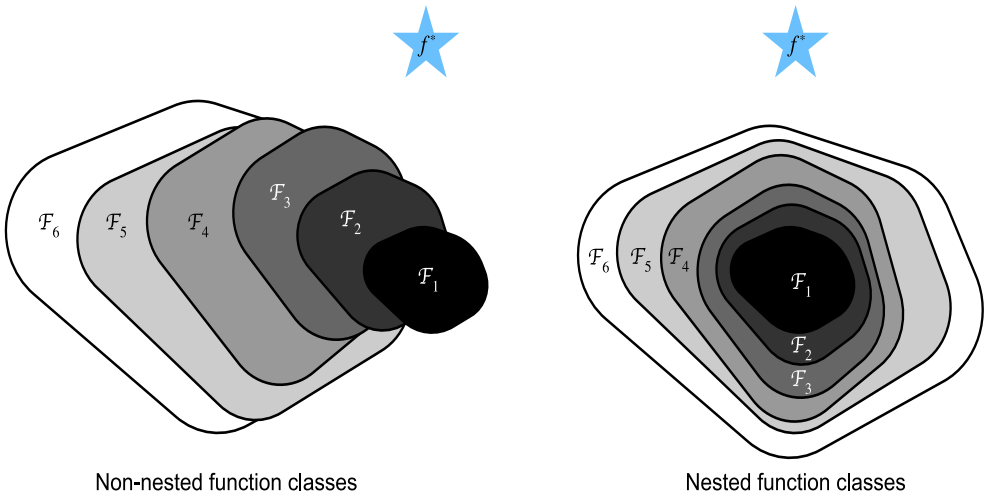


Fig 14. Function classes in ResNet

Only when larger function layers contain smaller layers is it guaranteed that adding more layers will increase the network's representational capacity. The central idea of ResNet is that each added layer should have a component that is a uniform function. This means that, if we train the newly added layer to a uniform mapping **f(x)=x**, the new model will be at least as efficient as the original model. Because the added layer can fit the training data better, the training error is also smaller. Better yet, the identity function should be the simplest function in a cascade instead of the null function **f(x)=0**.

- Residual Blocks:

Let's focus on a local part of a neural network, as depicted in figure below. Denote the input by **x**. We assume that **f(x)**, the desired underlying mapping we want to obtain by learning, is to be used as input to the activation function on the top. On the left, the portion within the dotted-line box must directly learn **f(x)**. On the right, the portion within the dotted-line box needs to learn the residual mapping **g(x)=f(x)-x**, which is how the residual block derives its name. If the identity mapping **f(x)=x** is the desired underlying mapping, the residual mapping amounts to **g(x)=0** and it is thus easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully connected layer and convolutional layer) within the dotted-line box to zero. The right figure illustrates the residual block of ResNet, where the solid line carrying the layer input x to the addition operator is called a residual connection (or shortcut connection). With residual blocks, inputs can forward propagate faster through the residual connections across layers. In fact, the residual block can

be thought of as a special case of the multi-branch Inception block: it has two branches one of which is the identity mapping.

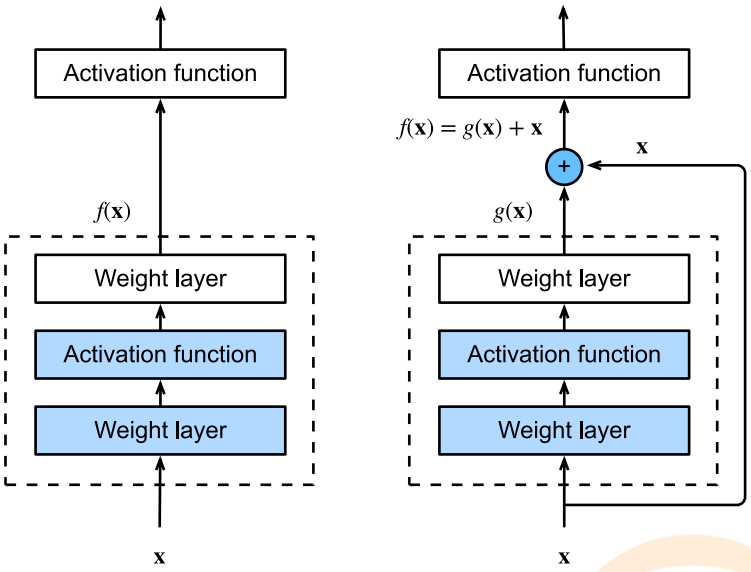


Fig 15. Residual Blocks in ResNet

ResNet has VGG's full **3x3** convolutional layer design. The residual block has two **3x3** convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers has to be of the same shape as the input, so that they can be added together. If we want to change the number of channels, we need to introduce an additional **1x1** convolutional layer to transform the input into the desired shape for the addition operation.

- Model:

The first two layers of ResNet are the same as those of the GoogLeNet we described before: the **7x7** convolutional layer with 64 output channels and a stride of 2 is followed by the **3x3** max-pooling layer with a stride of **2**. The difference is the batch normalization layer added after each convolutional layer in ResNet.

There are four convolutional layers in each module (excluding the **1x1** convolutional layer). Together with the first **7x7** convolutional layer and the final fully connected layer, there are 18 layers in total. Therefore, this model is commonly known as ResNet-18. By

configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152. Although the main architecture of ResNet is similar to that of GoogLeNet, ResNet's structure is simpler and easier to modify. All these factors have resulted in the rapid and widespread use of ResNet.

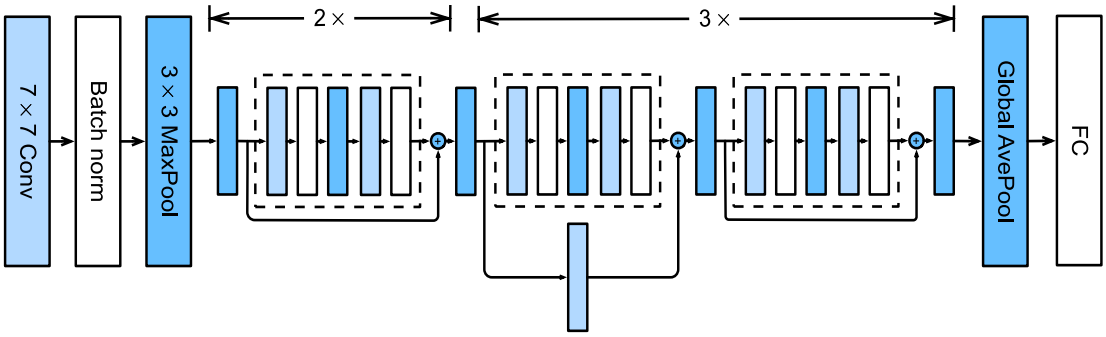


Fig 16. ResNet Model

2.2.6 EfficientNet:

EfficientNet is a convolutional neural network architecture and scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient. Unlike conventional practice that arbitrary scales these factors, the EfficientNet scaling method uniformly scales network width, depth, and resolution with a set of fixed scaling coefficients. For example, if we want to use 2^N times more computational resources, then we can simply increase the network depth by α^N, width by β^N, and image size by $У^N$, where α, β, $У$ are constant coefficients determined by a small grid search on the original small model. EfficientNet uses a compound coefficient Φ to uniformly scales network width, depth, and resolution in a principled way. The compound scaling method is justified by the intuition that if the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image.

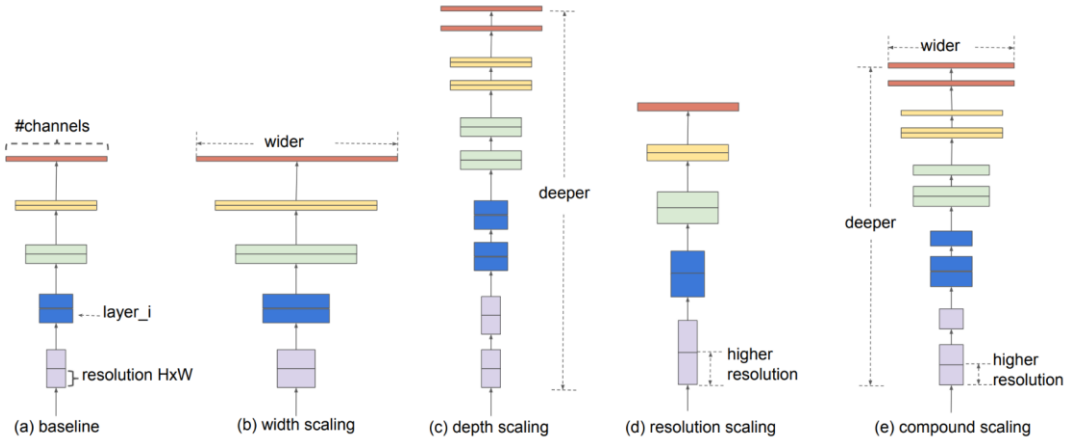


Fig 16. EfficientNet Model

2.3 Introduction to Activation Functions

An activation function is a function that is added to an artificial neural network in order to help the network learn complex patterns in the data. When comparing with a neuron-based model that is in our brains, the activation function is at the end deciding what is to be fired to the next neuron.

2.3.1 Sigmoid

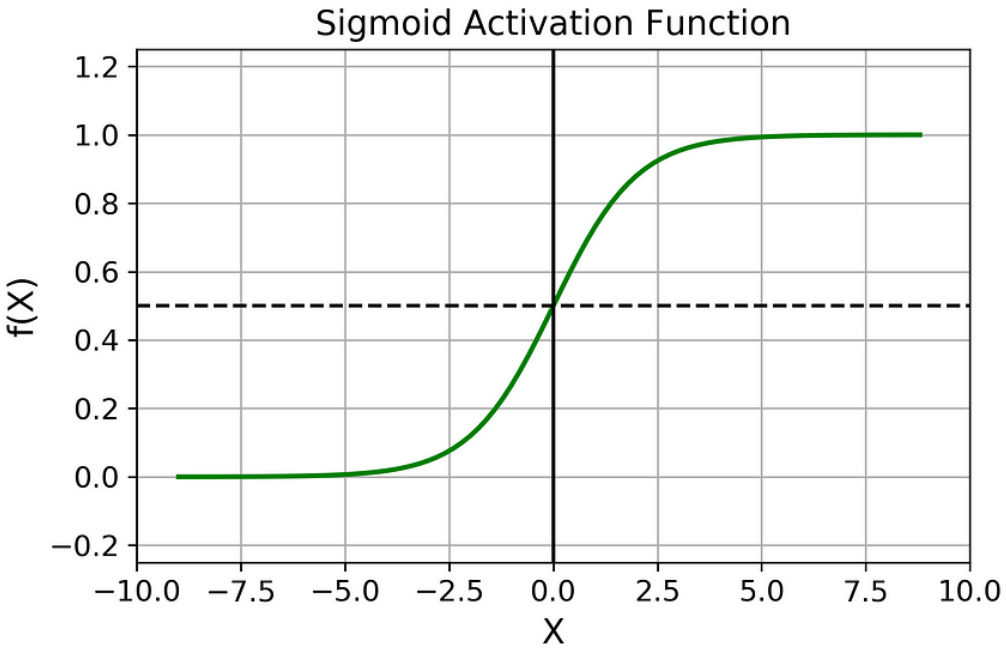


Fig 18. Sigmoid activation function graph

The Sigmoid Function looks like an S-shaped curve.

Formula : f(z) = 1/(1+ e^-z)

The output of a sigmoid function ranges between 0 and 1. Since, output values bound between 0 and 1, it normalizes the output of each neuron.

Specially used for models where we have to predict the probability as an output. The probability of anything exists only between the range of 0 and 1.

2.3.2 tanh (Hyperbolic Tangent)

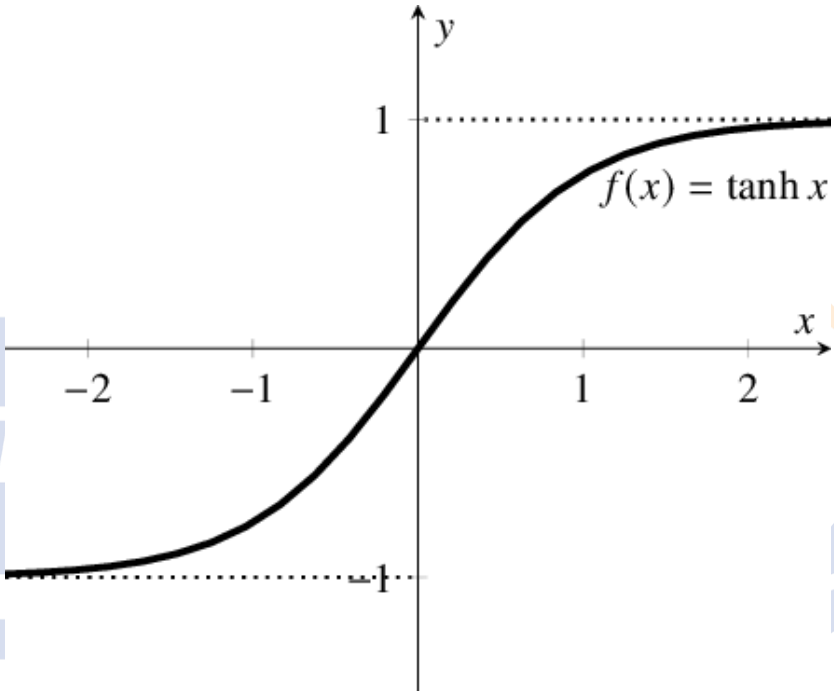


Fig 19. Tanh activation function graph

The tanh activation function is also sort of sigmoidal (S-shaped).

Formula: $f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$

Tanh is a hyperbolic tangent function. The curves of tanh function and sigmoid function are relatively similar.

2.3.3 ReLU (Rectified Linear Unit)

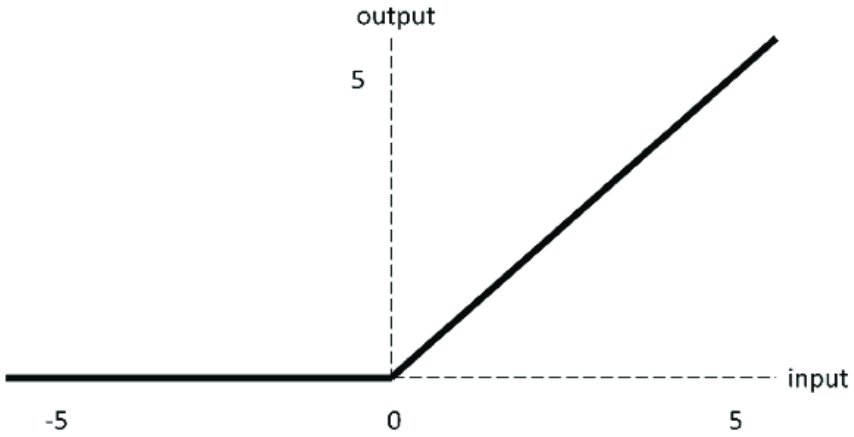


Fig 20. ReLU activation function graph

The ReLU is half rectified (from the bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

$$\sigma(x) = \begin{cases} \max(0, x), x \geq 0 \\ 0, x < 0 \end{cases} \quad \text{Range } 0 \text{ to } \infty$$

2.3.4 Softmax

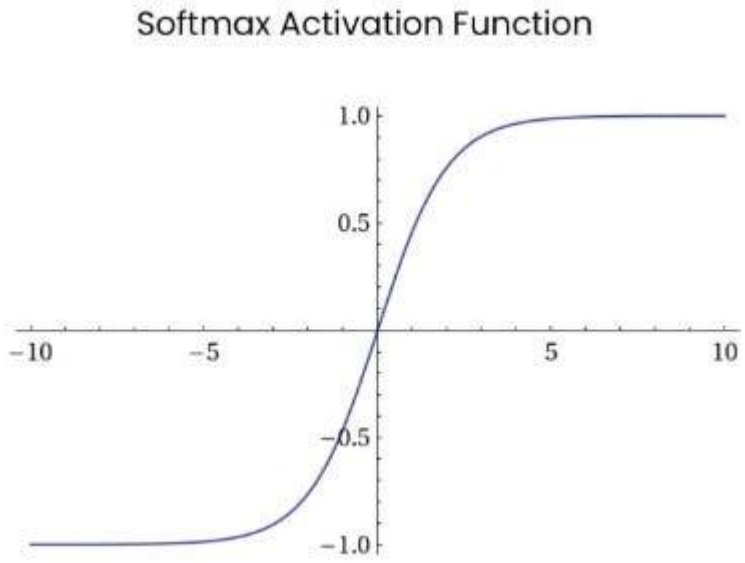


Fig 21. Softmax activation function graph

Softmax is used as the activation function for multi-class classification problems where class membership is required on more than two class labels. For an arbitrary real vector of length K, Softmax can compress it into a real vector of length K with a value in the range (0, 1), and the sum of the elements in the vector is 1.

2.4 Optimization Functions

Optimization functions are crucial in training CNNs as they adjust the weights to minimize the loss function. Various algorithms are used to optimize the training process:

2.4.1 SGD (Stochastic Gradient Descent)

Rather than going for batch processing, this optimizer focuses on performing one update at a time. It is therefore usually much faster, also the cost function minimizes after each iteration (EPOCH). It performs frequent updates with a high variance that causes the objective function(cost function) to fluctuate heavily. Due to which it makes the gradient to jump to a potential Global Minima.

- Objective function:

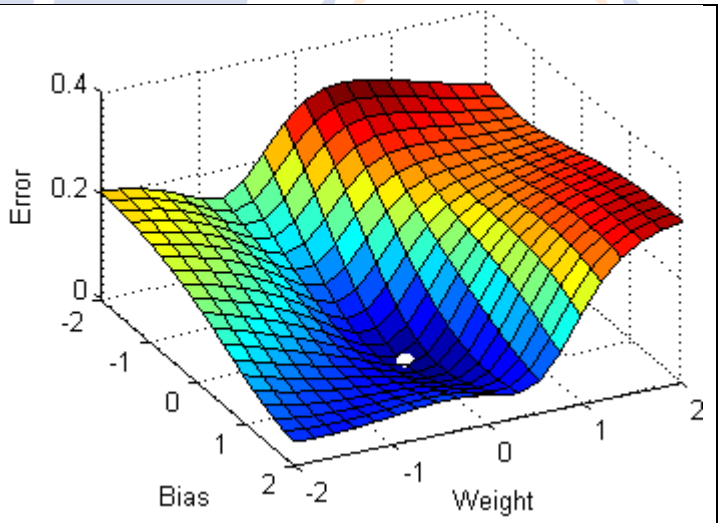
$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x)$$

- Gradient:

$$\nabla f(x) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x)$$

- Update:

$$x \leftarrow x - \eta \nabla f_i(x)$$



2.4.2 Adagrad:

This optimizer uses a different learning rate for each iteration (EPOCH) rather than using the same learning rate for determining all the parameters. Thus it performs smaller updates(lower learning rates) for the weights corresponding to the high-frequency features and bigger updates(higher learning rates) for the weights corresponding to the low-frequency features.

$$\begin{aligned}\mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),\\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2,\\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.\end{aligned}$$

2.4.3 Adadelta

This is an extension of the Adagrad optimizer, taking care of its aggressive nature of reducing the learning rate infinitesimally. Here instead of using the previous squared gradients, the sum of gradients is defined as a reducing weighted average of all past squared gradients(weighted averages) this restricts the learning rate to reduce to a very small value.

$$\begin{aligned}\mathbf{s}_t &= \rho \mathbf{s}_{t-1} + (1 - \rho)\mathbf{g}_t^2,\\ \mathbf{g}_t' &= \sqrt{\frac{\Delta\mathbf{x}_{t-1} + \epsilon}{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t,\\ \mathbf{x}_t &= \mathbf{x}_{t-1} - \mathbf{g}_t',\\ \Delta\mathbf{x}_t &= \rho\Delta\mathbf{x}_{t-1} + (1 - \rho)\mathbf{x}_t^2.\end{aligned}$$

2.4.4 RMSprop:

Both the optimizing algorithms, RMSprop(Root Mean Square Propagation) and Adadelta were developed around the same time, use the same method which utilizes an Exponential Weighted Average to determine the learning rate at time t for each iteration. RMSprop appropriately divides the learning rate by an exponentially weighted average of squared gradients.

$$\begin{aligned}\mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma)\mathbf{g}_t^2,\\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.\end{aligned}$$

2.4.5 Adam (Adaptive Moment Estimation):

This is the Adaptive Moment Estimation algorithm which also works on the method of computing adaptive learning rates for each parameter at every iteration. It comes with several parameters, which are β1, β2, and ε (epsilon). Where β1 and β2 are the initial restricting

parameters for Momentum and RMSprop respectively. Here, β1 corresponds to the first moment and β2 corresponds to the second moment.

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1)\mathbf{g}_t,$$
$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2.$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}.$$

$$\mathbf{g}'_t = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}.$$

2.4.6 Nadam (Nesterov-accelerated Adam):

Integrates Nesterov momentum into Adam, providing more informative updates and potentially faster convergence.

CHAPTER 3: SOLUTIONS/METHODS

3.1 Data Set

For this project, we utilized the ANIMALS10 dataset, a well-known benchmark in the field of machine learning. The dataset consists of 26,179 color images in 10 different classes. We use 8563 training images and 2167 test images, offering a balanced and comprehensive dataset for evaluating the performance of various CNN models.

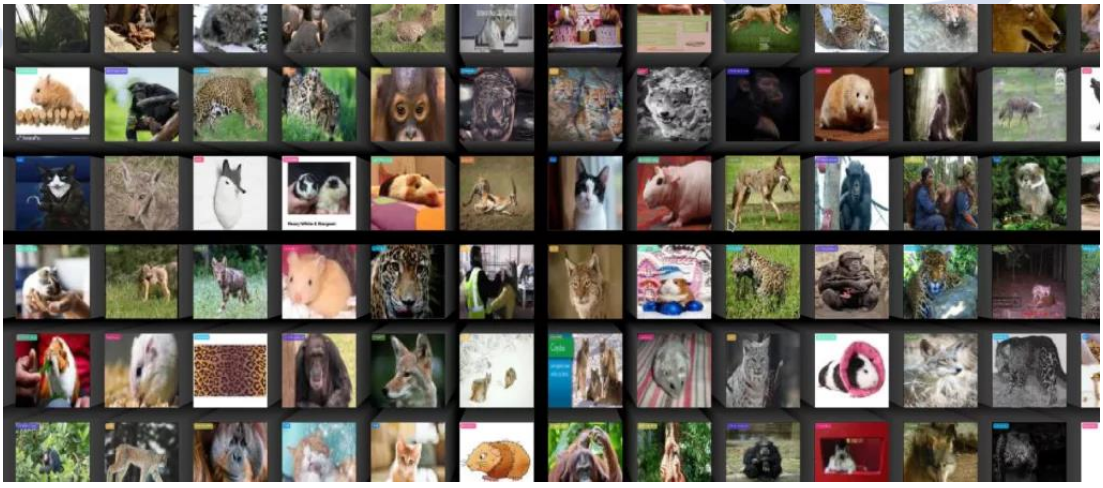


Fig 22. Dataset about 10 animals

3.2 Data Preprocessing

Preprocessing steps are critical in preparing the data for training and enhancing the performance of the CNN models. The following preprocessing techniques were applied:

- **Normalization:** The pixel values of the images were scaled to the range [0, 1] by dividing by 255. This helps in stabilizing the learning process and speeding up convergence.

- **Data Augmentation:** Techniques such as random cropping, horizontal flipping, and rotation were employed to artificially increase the diversity of the training data, reducing the risk of overfitting.

- **Standardization:** Mean subtraction and division by the standard deviation were performed to ensure that each feature has a mean of zero and a standard deviation of one, improving model training.

```
# Thiết lập ImageDataGenerator cho train
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)
# Chuẩn hóa giá trị điểm ảnh của test
validation_datagen = ImageDataGenerator(rescale=1./255)
```

Fig 23. Code setup Image Data Generator

3.3 Recommended Models

Based on the literature review and the requirements of the project, the following CNN models were selected for implementation and evaluation:

- EfficientNet (EfficientNetB7)

+ High Performance: It significantly improves accuracy compared to previous models without significantly increasing complexity.

+ Resource Optimization: EfficientNetB7 uses a method called Compound Scaling to optimize model size. Instead of linearly increasing model size, it balances increases in width,

depth, and image resolution, improving performance without consuming too much resources. calculation principle.

+ Modern Architecture: EfficientNetB7 uses the Swish activation function instead of ReLU, helping to improve model accuracy. EfficientNetB7's modular architecture uses MBConv (Mobile Inverted Bottleneck Conv) blocks, optimized for mobile and embedded deep learning models.

- ResNet (ResNet50)

+ Solve the Vanishing Gradient Problem: ResNet (Residual Network) is known for its ability to solve the vanishing gradient problem that deep networks often encounter. This is done through the use of shortcut connections that allow the gradient to pass through multiple layers without degradation.

+ Outstanding Performance: ResNet has won many competitions and benchmarks thanks to its ability to learn complex features from data with great network depth. ResNet50, with 50 layers, is one of the most popular variants, balancing complexity and performance.

+ Better Generalization with Greater Depth: Studies show that deeper networks are better able to extract high-level features, and ResNet, thanks to its special techniques, can Take advantage of this without gradient problems.

3.4 Proposed Optimization Functions

The following optimization functions were tested to determine their effectiveness in training the selected CNN models:

- Adam (Adaptive Moment Estimation)

+ Combining the Advantages of RMSprop and SGD with Momentum: Adam combines two important concepts from other methods: adaptive learning rate from RMSprop and momentum from SGD with Momentum. In particular, Adam uses both the weighted average of the gradient (speed) and the squared gradient (state).

+ Fast Convergence Ability: Adam has the ability to converge quickly, often with less time and resources than many other optimizers. This is important in reducing training time without affecting the final accuracy.

+ Stable and Robust: Adam adapts well to diverse training conditions and can maintain high performance on a variety of data types, including image classification problems. It helps maintain stability and performance in complex training scenarios.

+ No Need to Adjust Many Parameters: An important advantage of Adam is that it generally works well with default values of parameters such as learning rate, making it easy to use and optimize in reality.

- Nadam (Nesterov-accelerated Adaptive Moment Estimation)

+ Nesterov's Momentum Integration: Nadam is a variant of Adam with the integration of Nesterov's momentum, allowing update steps to be performed slightly in advance of the next step. This helps reduce the repetition of inefficient updates and improves convergence.

+ Faster and More Efficient Convergence: Thanks to Nesterov momentum's next step prediction feature, Nadam is able to converge faster than Adam in many cases, especially in problems with complex gradients. like image classification.

+ High Stability During Training: Nadam provides greater stability during long training sessions and can better handle gradient fluctuation problems compared to Adam. This is especially useful in training deep and complex CNN models.

3.5 Other Optimization Tools

In addition to the main optimization functions, other tools and techniques were employed to enhance the training process:

- Learning rate: Usually chosen 0.01 or 0.001.

- Epochs: Usually chosen from 30 - 100 epochs.

- Dropout: Reduce overfitting, improve generalization ability. The dropout ratio is usually chosen from 0.2 to 0.5

- Callback function:

+ EarlyStopping: Prevent overfitting, save resources:

+ ReduceLROnPlateau: Optimize learning speed, avoid overfitting

=> Set appropriate patience and learning rate reduction rate, for example reduce learning rate to 0.5 when there is no improvement after a number of epochs.

- Batch Normalization: Speeds up training, improves performance and stability, reduces reliance on Hyperparameters tuning

3.6 Experimental Results

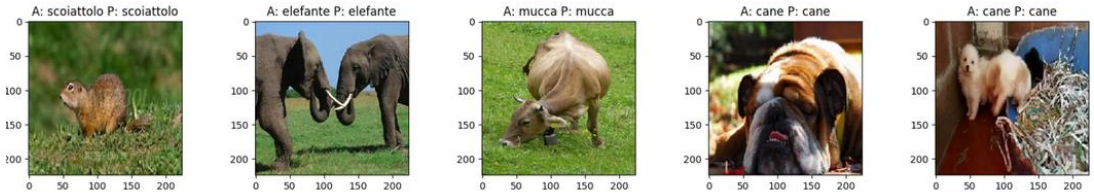


Fig 24. Correct labels with original images

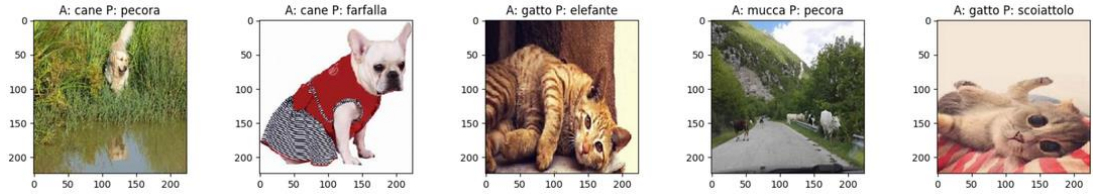


Fig 25. Wrong labels with original images

- Calculation table of accuracy parameters of CNNs (based on learning_rate and epochs):

Epochs=30

	AlexNet					
	SGD	**Adagrad**	**Adadelta**	**RMSprop**	**Adam**	**Nadam**
0.01	**52,38%**	**53,24%**	**65,53%**	**61,48%**	**64,93%**	**66,33%**
0.001	**54,08%**	**37,17%**	**66,23%**	**62,83%**	**62,41%**	**65,49%**
0,0001	**38,29%**	**21,38%**	**66,18%**	**46,90%**	**40,75%**	**49,09%**
	VGG19					
	SGD	**Adagrad**	**Adadelta**	**RMSprop**	**Adam**	**Nadam**
0.01	**53,38%**	**58,69%**	**53,61%**	**18,40%**	**18,54%**	**18,58%**
0.001	**51,79%**	**38,66%**	**53,61%**	**63,20%**	**44,43%**	**63,62%**
0,0001	**43,88%**	**24,22%**	**53,61%**	**53,80%**	**56,96%**	**53,61%**

Epochs=30

Models	EfficientNetB7					
learning_rate	SGD	Adagrad	Adadelta	RMSprop	Adam	Nadam
0.01	62.38%	63.24%	75.53%	71.48%	88.35%	76.33%
0.001	64.08%	47.17%	76.23%	72.83%	90.53%	75.49%
0,00001	48.29%	31.38%	76.18%	56.90%	95.65%	69.09%

Epochs=30

Models	ResNet50					
learning_rate	SGD	Adagrad	Adadelta	RMSprop	Adam	Nadam
0.01	69.80%	63.56%	70.18%	65.15%	85.67%	75.63%
0.001	71.85%	38.27%	77.26%	65.99%	90.08%	68.22%
0,00001	71.48%	60.30%	72.23%	76.79%	93.22%	72.08%

Overall, based on the graphs of the results we ran, we have drawn 2 graphs with the best and worst results.

As seen in Figure 25, in the training set, accuracy tends to increase steadily through each epochs in a stable manner, while in the test set, the graph will change strongly in the first 10 epochs, then it tends to change. slightly in the next 20 epochs and gradually stabilizes until the 50th epoch. The loss of the training set is similar, decreasing steadily through the epochs. With the test set, we can see that the loss fluctuates strongly in the first 15 epochs. However, there is a decreasing trend in subsequent epochs.

=> We can expect the model to run more epochs with a larger amount of data to significantly increase accuracy and reduce loss.

Fig 26. Good assessment graph for Accuracy and Loss

With figure 26, we can see that both the training and test sets for accuracy fluctuate very strongly. From there, it shows that the generalization ability of the model is not good, so is the learning rate and the number of epochs. Regarding loss, both the train and test sets have too high loss, there is not much change in loss between train and test, there is not even a decreasing trend.

=> The optimal model or function is not suitable, too many parameters need to be modified.

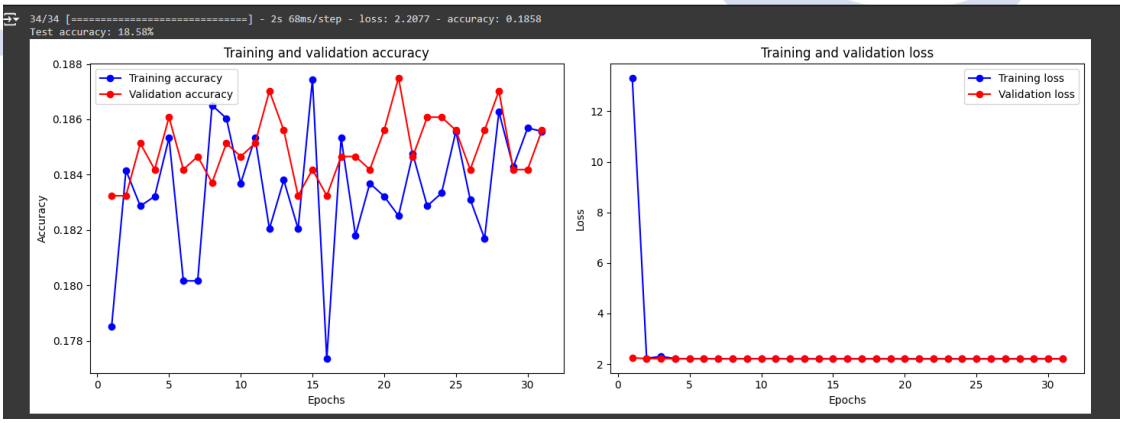


Fig 27. Bad assessment graph for Accuracy and Loss

With the confusion matrix, we can see that the accuracy is very low in the bad model and better in the good model. This also comes mainly from the data set. However, we can see that in a good model, the confusion matrix has a better and more accurate classification. We expect a better data set to generalize and significantly increase the

model's accuracy when we have built the model in such an optimal way.

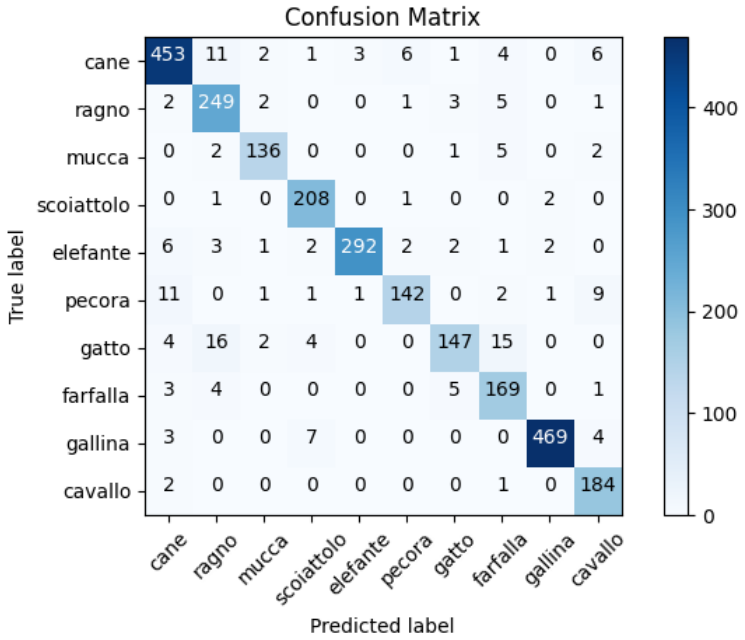


Fig 28. Good Confusion Matrix

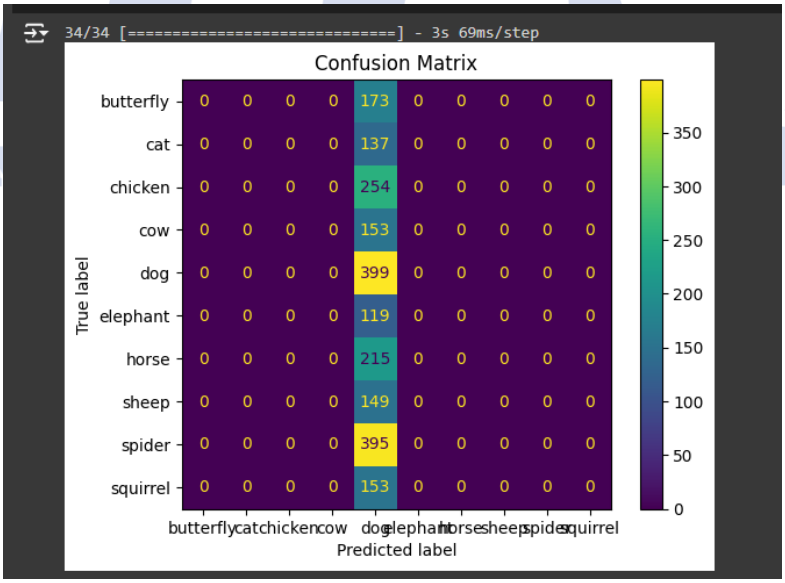


Fig 29. Bad Confusion Matrix

REFFERENCES

1. https://www.mathworks.com/discovery/convolutional-neural-network.html
2. https://nttuan8.com/bai-6-convolutional-neural-network/
3. https://d2l.aivivn.com/
4. https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a
5. https://medium.com/analytics-vidhya/activation-functions-all-you-need-to-know-355a850d025e
6. https://medium.com/analytics-vidhya/this-blog-post-aims-at-explaining-the-behavior-of-different-algorithms-for-optimizing-gradient-46159a97a8c1