International Journal of Computer Applications Technology and Research
Volume 13–Issue 08, 115 – 122, 2024, ISSN:-2319–8656
DOI:10.7753/IJCATR1308.1011

The Performance of Convolutional Neural Network Architecture in Classification

Panca Mudjirahardjo
Department of Electrical Engineering, Faculty of Engineering,
Brawijaya University
Malang, Indonesia

Aqil Gama Rahmansyah
Department of Electrical Engineering, Faculty of Engineering
Brawijaya University
Malang, Indonesia

Alya Shafa Dianti
Department of Statistics, Faculty of Mathematics and Science,
Brawijaya University
Malang, Indonesia

Abstract: In this paper, we study the performance of convolutional neural network (CNN) architecture for object classification. We evaluate three optimizers, i.e. ADAM, SGD and RMSprop and 5 convolutional layers use CIFAR10 datasets. We conduct the experiment with python program language. We evaluate the training, validation accuracy and training execution time.

Keywords: convolutional neural network; optimizer; CIFAR10 datasets; CNN architecture; object classification

1. INTRODUCTION

In recent years, Convolutional Neural Networks (CNNs) have revolutionized the field of artificial intelligence, particularly in the areas of image and video recognition, natural language processing, and beyond. A CNN is a type of deep learning model specifically designed to process and analyze visual data. Unlike traditional neural networks, which work with vectorized inputs, CNNs are adept at recognizing patterns and spatial hierarchies in grid-like data, such as images [1][2][11][12].

Architecture of CNNs

Input Layer: The input layer accepts the raw data, typically an image represented as a matrix of pixel values. For example, a grayscale image might be a 2D matrix, while a color image is represented as a 3D matrix (height x width x channels).

Convolutional Layer: The convolutional layer is the core building block of a CNN. It applies a set of filters (or kernels) to the input image. Each filter slides across the image (a process known as convolution) and performs element-wise multiplications to produce a feature map. This feature map highlights specific patterns, such as edges or textures, at various locations in the image [6][7][8][9].

Activation Function: After convolution, an activation function like the Rectified Linear Unit (ReLU) is applied to introduce non-linearity. This step helps the network learn complex patterns by transforming the output of the convolution operation.

Pooling Layer: Pooling (or subsampling) layers reduce the spatial dimensions of the feature maps, typically through operations like max pooling or average pooling. This step helps to reduce computation, control overfitting, and make the model more invariant to small translations in the image.

Fully Connected Layer: Following several convolutional and pooling layers, the high-level reasoning is performed by fully connected layers. These layers are similar to those in traditional neural networks and are used to classify the features extracted by the convolutional layers.

Output Layer: The output layer produces the final classification or prediction. In a classification task, this layer typically uses a softmax function to output probabilities for each class.

2. THE PROPOSED STUDY

In this section we will explain briefly our proposed study. We will evaluate three optimizers, i.e. ADAM (*Adaptive Moment Estimation*), SGD (*Stochastic Gradient Descent*) and RMSprop (*Root Mean Square Propagation*). The convolutional layers are shown in Figure 1.

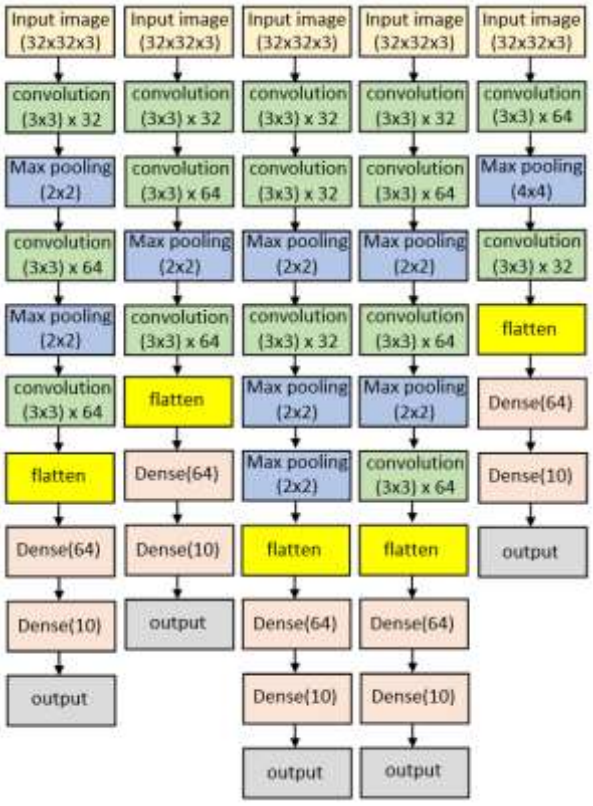


Figure 1. The 5 network architectures used in this study

Figure 1 shows the input image size of 32×32 pixels in color format. The convolution process, we use 32 filters and 64 filters with kernel size of 3×3. Max pooling, we use window

International Journal of Computer Applications Technology and Research
Volume 13–Issue 08, 115 – 122, 2024, ISSN:-2319–8656
DOI:10.7753/IJCATR1308.1011

size 2×2 pixels, except architecture 5 we use window size 4×4 pixels.

Figure 2. CIFAR10 datasets used in this study [3]

2.1 ADAM (Adaptive Moment Estimation)

The Adam optimizer is a widely used optimization algorithm that combines the ideas of momentum and adaptive learning rates to improve the training of neural networks and other machine learning models. Developed by D.P Kingma and J.Ba in 2015 [4].

Adam maintains two moving averages for each parameter:
1. First Moment (Mean): This is the moving average of the gradients, similar to momentum.
2. Second Moment (Variance): This is the moving average of the squared gradients, which adjusts the learning rate based on the variance of the gradients.

Adam Algorithm Steps:
1. Initialization:
- Initialize parameters θ with some values.
- Initialize first moment vector m and second moment vector v to zero.
- Set hyperparameters: learning rate α, β_1, β_2, and ϵ.
2. Update Rules:
For each iteration t:
- Compute the gradient g_t of the loss function with respect to the parameters θ.
- Update the first moment estimate:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

- Update the second moment estimate:

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

- Compute bias-corrected first moment estimate:

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

- Compute bias-corrected second moment estimate:

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

- Update the parameters:

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

Here, α is the learning rate, β_1 is the exponential decay rate for the first moment estimate, β_2 is the exponential decay rate for the second moment estimate, and ϵ is a small number added to prevent division by zero.

Hyperparameters
- Learning Rate α : Controls the step size for each update. Common default value: 0.001.
- β_1 : The decay rate for the first moment estimate. Common default value: 0.9.
- β_2 : The decay rate for the second moment estimate. Common default value: 0.999.
- ϵ : A small constant to prevent division by zero. Common default value: 10^{-8}.

Advantages of Adam
- Adaptive Learning Rates: Adam adjusts the learning rate for each parameter individually, which can lead to faster convergence.
- Momentum: By incorporating the moving average of the gradients, Adam can navigate ravines more effectively and smooth out the updates.
- Bias Correction: Adam's bias correction helps stabilize the estimates of the moments, especially during the initial stages of training.

Disadvantages of Adam
- Memory Usage: Adam requires storing additional vectors for the moments, which can increase memory usage compared to simpler optimizers.
- Hyperparameter Sensitivity: While Adam is often robust, the choice of hyperparameters can still affect performance, and tuning may be necessary for optimal results.

Overall, Adam is a versatile and effective optimizer that works well in many scenarios, particularly in training deep learning models.

2.2 SGD (Stochastic Gradient Descent)

SGD is one of the most fundamental and widely used optimization algorithms in machine learning and deep learning. Despite the advent of more sophisticated optimizers like Adam and RMSProp, SGD remains a popular choice due to its simplicity, effectiveness, and computational efficiency [5].

Stochastic Gradient Descent is a variant of the classic Gradient Descent algorithm. While Gradient Descent updates model parameters based on the average of gradients computed over the entire training dataset (batch gradient descent), SGD uses a single or a small subset of training examples to update the parameters at each iteration. This approach has several implications for training:
1. Gradient Computation:
- In SGD, instead of computing the gradient of the loss function across the entire dataset, the gradient is computed for a randomly selected subset of the data (a mini-batch) or even a single training example. This stochastic approximation can be expressed as:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(x_i, y_i; \theta_t)$$

where θ_t represents the model parameters at time step t, α is the learning rate, (x_i, y_i) is the training example, and $\nabla_\theta L$ is the gradient of the loss function.
2. Parameter Update:
- The parameters are updated iteratively as:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(x_i, y_i; \theta_t)$$

where $\nabla_\theta L(x_i, y_i; \theta_t)$ is the gradient computed from the mini-batch or single example.

Advantages of SGD
1. Computational Efficiency: By updating the parameters more frequently (with each mini-batch or training example), SGD often requires less memory and computational resources

compared to batch gradient descent. This can be particularly advantageous for large datasets.

2. Faster Convergence: The frequent updates help the model to start improving more quickly, potentially leading to faster convergence to a good solution.

3. Escape Local Minima: The noisy updates due to stochastic gradients can help the optimization process escape from local minima and explore the parameter space more broadly, which might lead to better overall solutions.

4. Online Learning: SGD is suitable for online learning scenarios where data arrives sequentially. It allows the model to be updated continuously with new data.

Challenges of SGD

1. Hyperparameter Sensitivity: The learning rate α is a crucial hyperparameter that significantly affects the performance of SGD. If it's too high, the algorithm may converge too quickly to a suboptimal solution; if it's too low, convergence may be too slow.

2. Convergence Issues: Due to the noisy nature of the gradients, SGD can exhibit high variance in updates, which might make the convergence path erratic. It may require careful tuning of learning rates and other hyperparameters.

3. Learning Rate Scheduling: To mitigate issues with convergence, learning rate schedules (e.g., decaying learning rates) are often used. However, determining the right schedule can be complex.

4. Local Minima and Saddle Points: Despite the ability to escape local minima, SGD may still get stuck in saddle points or poor local minima, depending on the problem landscape.

Variations and Improvements

1. Mini-Batch Gradient Descent: Instead of using a single training example, mini-batch gradient descent uses small random subsets of data (mini-batches) to compute gradients. This balances the trade-off between the noisy updates of SGD and the computational cost of batch gradient descent.

2. Momentum: Momentum-based SGD helps accelerate convergence and smooth out oscillations by incorporating a moving average of past gradients. This can be expressed as:

$$v_{t+1} = \beta v_t + (1-\beta)\nabla_\theta L(x_i, y_i; \theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

where v_t is the velocity term and β is the momentum factor.

3. Nesterov Accelerated Gradient (NAG): NAG improves upon traditional momentum by looking ahead at the estimated future position of the parameters, thus providing more informed updates.

4. Learning Rate Schedulers: Various learning rate schedules, such as exponential decay, step decay, or cyclical learning rates, can help improve convergence by adjusting the learning rate during training.

Best Practices

1. Choosing the Right Learning Rate: Start with a moderate learning rate and use techniques like learning rate schedules or adaptive methods to fine-tune it.

2. Using Mini-Batches: Employ mini-batch gradient descent to achieve a balance between computational efficiency and gradient estimation accuracy.

3. Incorporating Momentum: Apply momentum to accelerate convergence and stabilize updates, especially in noisy or complex landscapes.

4. Regularization: Combine SGD with regularization techniques (like dropout or L2 regularization) to prevent overfitting and improve generalization.

Stochastic Gradient Descent remains a foundational optimization algorithm due to its simplicity and efficiency. While it can be challenging to tune and may have convergence issues, its variants and improvements have addressed many of these challenges. Understanding and effectively applying SGD can lead to significant advancements in training machine learning models.

2.3 RMSprop

RMSProp (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm designed to address some of the limitations of traditional stochastic gradient descent (SGD) and its variants. Introduced by Geoffrey Hinton in his Coursera class on neural networks, RMSProp is especially effective in handling problems associated with non-stationary objectives and noisy gradients [10].

RMSProp modifies the learning rate for each parameter by scaling it inversely proportional to a running average of recent magnitudes of the gradients. Here's a detailed breakdown of how RMSProp operates:

1. Gradient Computation: Similar to SGD, RMSProp computes gradients of the loss function with respect to the model parameters. However, it adjusts the learning rate for each parameter based on the historical gradient magnitudes.

2. Exponential Decay of Squared Gradients: RMSProp maintains a running average of the squared gradients, which helps to normalize the gradient updates. This is achieved through an exponentially decaying average:

$$v_t = \beta v_{t-1} + (1-\beta) g_t^2$$

where v_t represents the average of squared gradients at time step t, β is the decay factor (often set to 0.9), and g_t is the gradient at time step t.

3. Parameter Update: The parameters are updated using the following rule:

$$\theta_{t+1} = \theta_t - \frac{\alpha g_t}{\sqrt{v_t + \epsilon}}$$

where α is the learning rate, $\sqrt{v_t}$ is the root of the running average of squared gradients, and ϵ is a small constant added for numerical stability (e.g., 10^{-8}).

Advantages of RMSProp

1. Adaptive Learning Rates: RMSProp adapts the learning rate for each parameter individually, which helps to address issues like vanishing or exploding gradients and improves convergence speed.

2. Handling Non-Stationary Objectives: By normalizing the gradient updates, RMSProp can handle non-stationary objectives, where the distribution of data or the objective function changes over time.

3. Stable Updates: The running average of squared gradients smooths out the updates, making them more stable and reducing the variance caused by noisy gradients.

4. Efficient Computation: RMSProp requires only a modest amount of memory and computational resources, making it suitable for large-scale and complex models.

Challenges of RMSProp

1. Hyperparameter Tuning: Choosing appropriate values for the learning rate α and the decay factor β can be challenging. These hyperparameters can significantly impact the performance of the optimizer.

2. Sensitivity to Initial Learning Rate: The performance of RMSProp can be sensitive to the initial learning rate. Careful tuning is necessary to achieve optimal results.

3. No Global Optimal Learning Rate: While RMSProp adapts the learning rate for each parameter, it does not

guarantee a globally optimal learning rate for all parameters, which can sometimes lead to suboptimal convergence.

Variations and Improvements

1. RMSProp with Warm Restarts: Adding warm restarts (also known as cyclical learning rates) to RMSProp can improve its performance by periodically resetting the learning rate, which helps the optimizer escape local minima.

2. RMSProp with Decoupled Weight Decay (AdamW): Combining RMSProp with decoupled weight decay (as in AdamW) can improve generalization by addressing issues with weight regularization.

Best Practices

1. Learning Rate Scheduling: Experiment with different learning rates and use learning rate scheduling techniques to adjust the learning rate during training.

2. Decay Factor: Set the decay factor β to a value like 0.9, which works well in practice, but be prepared to adjust it based on the specific problem and dataset.

3. Numerical Stability: Ensure that ϵ is appropriately set to avoid division by zero and numerical instability.

4. Experimentation: As with other optimizers, empirical testing and experimentation are crucial for finding the best hyperparameters and configurations for your specific application.

RMSProp is a powerful optimization algorithm that addresses many of the limitations of traditional gradient descent methods by adapting learning rates based on the magnitude of recent gradients. Its ability to handle noisy gradients and non-stationary objectives makes it particularly useful for training deep learning models. By understanding its mechanics and applying best practices, you can leverage RMSProp to improve the performance and efficiency of your machine learning models.

3. THE EXPERIMENTAL RESULT

In this section, we explain our experimental result. As in Figure 1 show the convolutional layers we used in this study, and the datasets CIFAR10 is depicted in Figure 2. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

To conduct this study, we write code programming in python language as Program 1. The model summary of architectures are depicted in Figure 3, 6, 7, 8, 10 for architecture 1, 2, 3, 4 and 5 respectively. We evaluate each architecture and optimizer in 10, 30 and 50 epoch.

Program 1:

```
print('CNN with CIFAR10 ..\n'*5)
print("SB_Montag,29.07.2024; 07:24")
print("Panca" +
      " Mudjirahardjo")
print("")
print("=========================================")

print("")
print('')
modelKE = input('What model (1 .. 5) : ')

print('')
optim = input('Optimizer ((1)SGD, (2)ADAM, (3)RMSprop) : ')
if optim=='1':
   optim='SGD'
   print('-- optimizer: SGD')
elif optim=='2':
   optim='ADAM'
   print('-- optimizer: ADAM')
elif optim=='3':
   optim='RMSprop'
   print('-- optimizer: RMSprop')

print('')
ep = input('Jumlah epoch (1 epoch 42 sec !): ')
ep = int(ep)

print('')
print('---- import library & tools dimulai --- ')

# ------------------------------------------------------------------------------
import datetime
import numpy as np
import os
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'

import tensorflow as tf
import matplotlib.pyplot as plt
from tqdm import tqdm

from tensorflow.keras import datasets, layers, models
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.layers import Dense, Flatten, Conv2D,
MaxPooling2D, Dropout

# ------------------------------------------------------------------------------
def create_model_1():
   model = models.Sequential()
   model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)))
   model.add(layers.MaxPooling2D((2, 2)))
   model.add(layers.Conv2D(64, (3, 3), activation='relu'))
   model.add(layers.MaxPooling2D((2, 2)))
   model.add(layers.Conv2D(64, (3, 3), activation='relu'))

   print('')
   model.summary()

   model.add(layers.Flatten())
   model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10))

   print('')
   model.summary()

   return model

def create_model_2():
   model = models.Sequential()
   model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)))
   model.add(layers.Conv2D(64, (3, 3), activation='relu'))
   model.add(layers.MaxPooling2D((2, 2)))
   model.add(layers.Conv2D(64, (3, 3), activation='relu'))

   print('')
   model.summary()
```

```
    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))

    print('')
    model.summary()

    return model

def create_model_3():
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)))
    model.add(layers.Conv2D(32, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(32, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.MaxPooling2D((2, 2)))

    print('')
    model.summary()

    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))

    print('')
    model.summary()

    return model

def create_model_4():
    model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))

    print('')
    model.summary()

    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))

    print('')
    model.summary()
    return model

def create_model_5():
    model = models.Sequential()
    model.add(layers.Conv2D(64, (3, 3), activation='relu',
input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((4, 4)))
    model.add(layers.Conv2D(32, (3, 3), activation='relu'))

    print('')
    model.summary()

    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))

    print('')
    model.summary()

    return model

# ------------------------------------------------------------------------------

print('')
print('--- download datasets --')
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images /
255.0

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()

print('')
print('')
if modelKE == '1':
    model = create_model_1()
    print('---- model ke 1 (satu) ---')
elif modelKE == '2':
    model = create_model_2()
    print('---- model ke 2 (dua) ---')
elif modelKE == '3':
    model = create_model_3()
    print('---- model ke 3 (tiga) ---')
elif modelKE == '4':
    model = create_model_4()
    print('---- model ke 4 (empat) ---')
elif modelKE == '5':
    model = create_model_5()
    print('---- model ke 5 (lima) ---')

print('')
print('---- training dimulai --- ')
print('---- Optimizer: ' +optim)
print('')

# -- optimizer: SGD, RMSprop, dan ADAM
model.compile(optimizer=optim,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logit
s=True),
              metrics=['accuracy'])

# Restore the weights
history = model.fit(train_images, train_labels, epochs=ep,
                validation_data=(test_images, test_labels))

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
```

International Journal of Computer Applications Technology and Research
Volume 13–Issue 08, 115 – 122, 2024, ISSN:-2319–8656
DOI:10.7753/IJCATR1308.1011

```
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()

print('')
print(' ----- model evaluate ---- ')
test_loss, test_acc = model.evaluate(test_images,  test_labels,
verbose=2)

print("")
print('.. press ENTER to quit ..')
input()

# ----------------------------------------------------------------------------
```

3.1 Architecture 1

The model summary

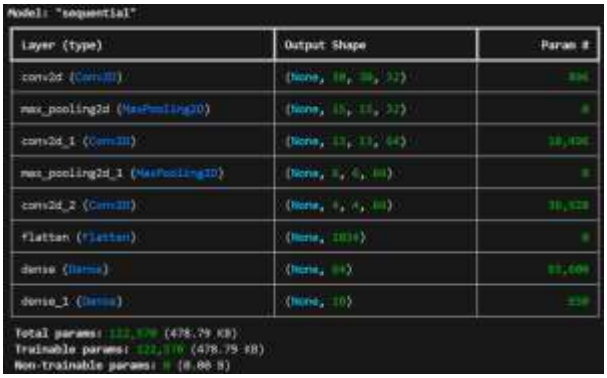
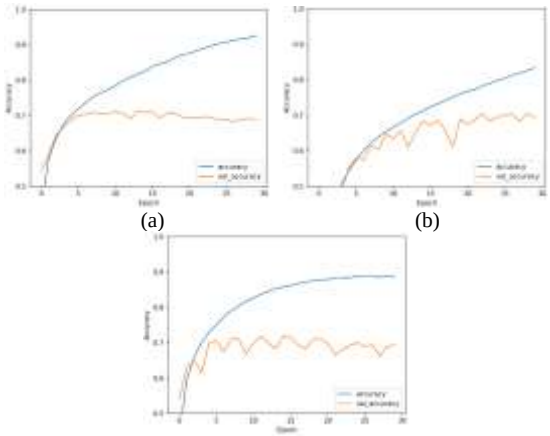


Figure 3. Model summary of architecture #1

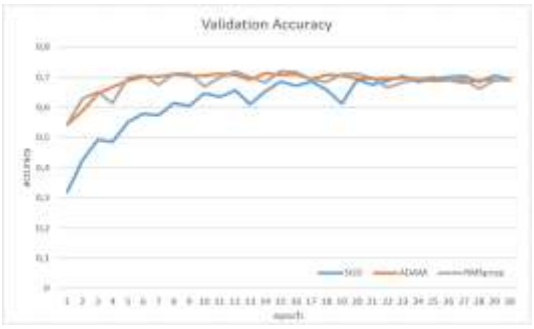


(a) (b)

(c)

Figure 4. Architecture #1 with optimizer (a) adam (b) SGD (c) RMSprop

(a)



(b)

Figure 5. Architecture #1 (a) training accuracy (b) validation accuracy

3.2 Architecture 2

The model summary

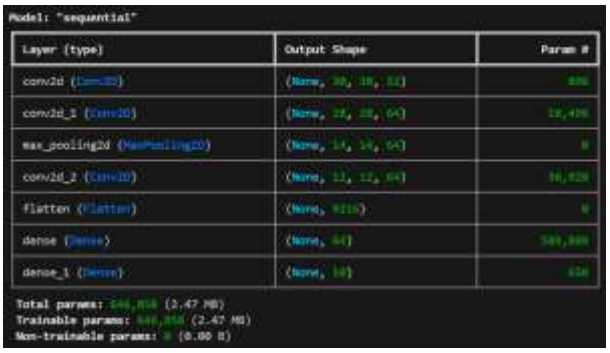


Figure 6. Model summary of architecture #2

3.3 Architecture 3

The model summary

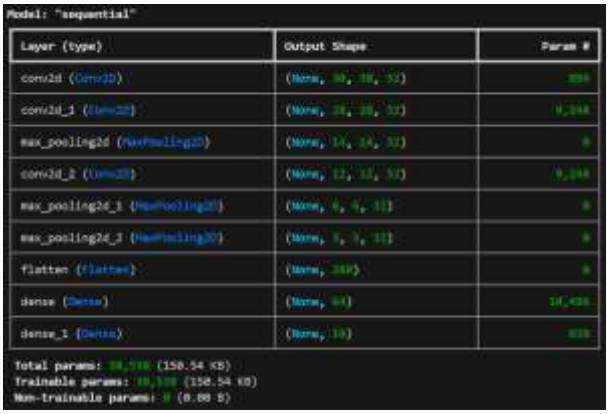


Figure 7. Model summary of architecture #3

3.4 Architecture 4

The model summary

International Journal of Computer Applications Technology and Research
Volume 13–Issue 08, 115 – 122, 2024, ISSN:-2319–8656
DOI:10.7753/IJCATR1308.1011

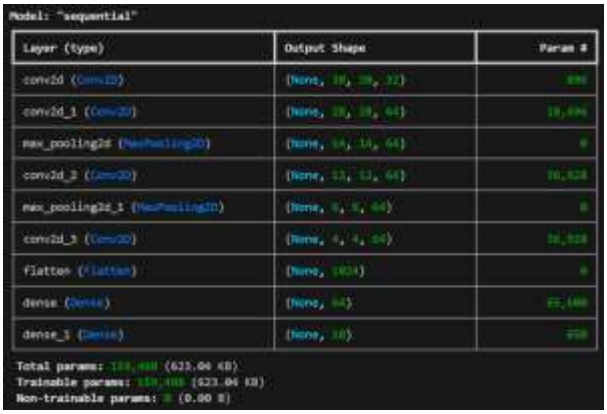
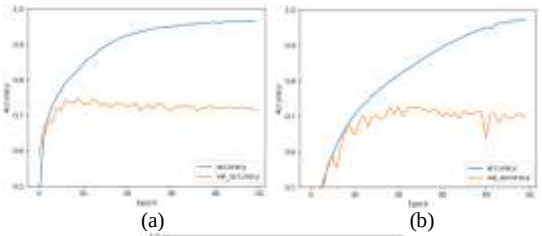
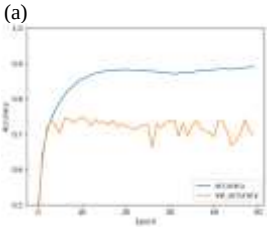


Figure 8. Model summary of architecture #4



(a) (b)



(c)

Figure 9. Validation Accuracy of architecture #4 (a) **adam of 0,7145** (b) SGD of 0,6940 (c) RMSprop, of 0,7005

3.5 Architecture 5

The model summary

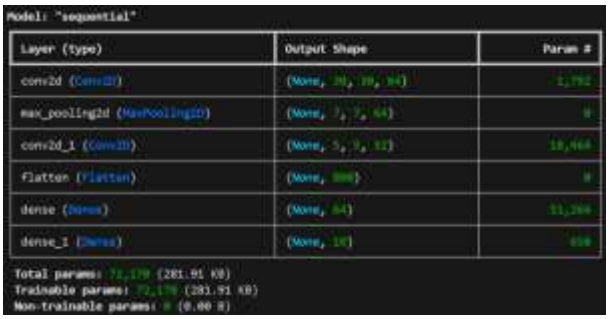
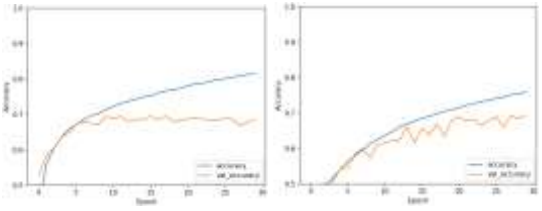
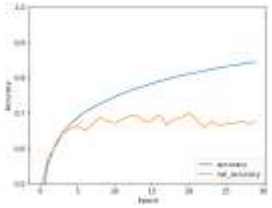


Figure 10. Model summary of architecture #5



(a) (b)



(c)

Figure 11. Validation Accuracy of architecture #5 (a) adam of 0,6867 (b) **SGD of 0,6951** (c) RMSprop, of 0,6770

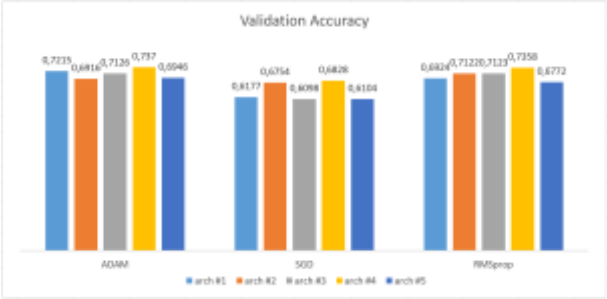


Figure 12. Comparison of validation accuracy based on architecture and optimizer

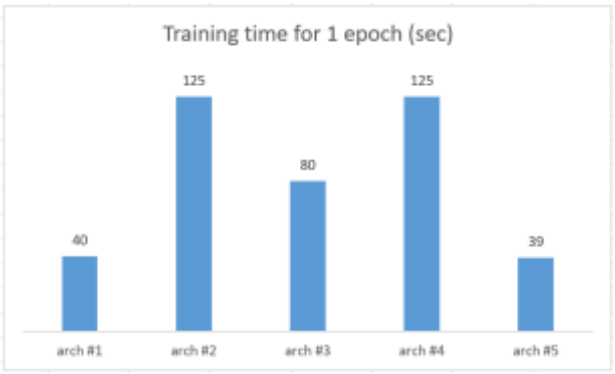


Figure 13. Average training time for 1 epoch

4. CONCLUSION

In this paper, we study the performance of CNN in classification based on the different architecture and optimizer. In the Figure 12, it is shown the highest validation accuracy belongs to architecture 4. Figure 13 shows the average training time for one epoch.

5. REFERENCES

[1] Alzubaidi, L., Zhang, J., Humaidi, A.J., et.al. 2021. *Review of deep learning: concepts, CNN architectures, challenges, applications, future directions*. Journal of Big Data.

[2] Yamashita, R., Nishio, M., Do, R.K.G. *et al.* (2018). *Convolutional neural networks: an overview and application in radiology*. Insights Imaging **9**, 611–629 https://doi.org/10.1007/s13244-018-0639-9

[3] Will Cukierski. (2013). CIFAR-10 - *Object Recognition in Images. Kaggle*. https://kaggle.com/competitions/cifar-10

[4] Kingma, Diederik & Ba, Jimmy. (2014). *Adam: A Method for Stochastic Optimization*. International Conference on Learning Representations.

[5] Gower RM, Loizou N, Qian X, Sailanbayev A, Shulgin E, Richtárik P (2019). *SGD: general analysis and improved rates*. In international conference on machine learning (pp. 5200-5209). PMLR.

[6] Mudjirahardjo, P. (2024). *The comparison of convolution and Max Pooling Process in real-time*. International Journal of Advanced Multidisciplinary Research and Studies (IJAMRS). Vol. 4, Issue 3. pp. 1039-1044. ISSN: 2583-049x.

[7] Mudjirahardjo, P. (2024). *Real-Time 2-D Convolution Layer for Feature Extraction*. International Journal of Modern Engineering Research (IJMER). Vol. 14, Issue 03. pp. 211-216. ISSN: 2249-6645.

[8] Mudjirahardjo, P. (2024). *Real-Time 2D Convolution and Max Pooling Process*. International Journal of Computer Applications Technology and Research (IJCATR). Vol. 13, Issue 06. pp. 18-23. ISSN: 2319-8656. DOI: 10.7753/IJCATR1306.1003.

[9] Mudjirahardjo, P. (2024). *The Effect of Grayscale, CLAHE Image and Filter Images in Convolution Process*. International Journal of Advanced Multidisciplinary Research and Studies (IJAMRS). Vol. 4, issue 3. Pp. 943-946. ISSN: 2583-049x.

[10] Elshamy, Reham & Abu Elnasr, Osama & Elhoseny, Mohamed & Elmougy, Samir. (2023). *Improving the efficiency of RMSProp optimizer by utilizing Nestrove in deep learning*. Scientific Reports. 13. 10.1038/s41598-023-35663-x.

[11] Hassan, E., Shams, M.Y., Hikal, N.A. *et al.* (2023). The effect of choosing optimizer algorithms to improve computer vision tasks: a comparative study. *Multimed Tools Appl* **82**, 16591–16633. https://doi.org/10.1007/s11042-022-13820-0

[12] Abdulkadirov, R.; Lyakhov, P.; Nagornov, N. (2023) *Survey of Optimization Algorithms in Modern Neural Networks*. Mathematics, *11*, 2466. https://doi.org/10.3390/math11112466