Computer Vision

Table of Contents

Introduction 1

Purpose 1

Scope 1

Procedure 1

Brief Problem 1

CNNs (Convolutional Neural Networks) 2

1. Kernels / Filters 2

2. Channels 2

3. Depth of Layer 2

4. Stride 3

5. Padding 3

6. Dropout 3

7. Pooling 4

8. Output Size Formula 5

CNN Model Example 6

1. Importing Libraries 6

2. Setting a Random Seed 6

3. Dataset Loading 7

4. Data Augmentation 8

5. Grayscale Conversion & Normalization 9

6. Label Encoding & Train/Validation Split 9

7. CNN Model Architecture 10

8. Custom Checkpoint Callback 12

9. Training 13

10. Model Evaluation 14

11. Classification Report & Confusion Matrix 15

12. Sample Predictions Visualization 16

13. Accuracy & Loss Curves 17

Conclusion 18

Feedback & Contribution 18

Copyright & Usage 18

Acknowledgments 19

Introduction

Purpose

The purpose of this work is to demystify the field of Computer Vision (CV), transitioning from manual feature extraction to modern deep learning approaches. It aims to equip practitioners with the theoretical understanding of how machines interpret visual data and the practical skills to implement state-of-the-art architectures for tasks like image classification, object detection, and segmentation.

Scope

This document provides a detailed exploration of the algorithms that enable computers to "see." The scope progresses from the limitations of traditional Neural Networks (MLPs) in processing images to the foundational building blocks of Convolutional Neural Networks (CNNs)—including filters, pooling, and stride. It further expands into advanced modern architectures (such as VGG, ResNet, and Inception), Object Detection frameworks (YOLO, R-CNN), and techniques for handling limited data such as Data Augmentation and Transfer Learning.

Procedure

The procedure undertaken in this study includes:

1. **Theoretical Study:** Analyzing the mathematical operations behind convolutions, activation maps, and backpropagation in spatial domains.
2. **Implementation:** Writing code examples using libraries like TensorFlow/Keras or PyTorch to build and train CNNs.
3. **Visualization:** Visualizing feature maps, filters, and class activation maps (CAM) to interpret what the model learns.
4. **Evaluation:** Assessing model performance using metrics specific to vision, such as Intersection over Union (IoU) and mean Average Precision (mAP).
5. **Guidelines:** Providing best practices on selecting architectures and tuning hyperparameters for different visual tasks.

Brief Problem

Standard neural networks struggle with image data due to the "Curse of Dimensionality" and the loss of spatial structure when flattening images into vectors. Furthermore, real-world visual data is highly variable—objects appear in different orientations, lighting conditions, and scales. This document addresses these problems by introducing translation-invariant architectures and robust feature learning techniques that generalize well to unseen visual scenarios.

CNNs (Convolutional Neural Networks)

CNNs are **deep learning architectures** designed to process **grid-like data**, such as images.
They are inspired by how the **visual cortex** of the human brain detects patterns — edges, shapes, textures, etc.

A CNN learns **spatial hierarchies of features** — from simple edges in the first layers to complex structures (like faces or objects) in deeper layers.

Building Blocks of CNNs are **convolutional layers**, which perform **critical math operations** through applying **kernels,** making it great at **Feature Extraction**.

Each **neuron** in a **CNN** looks at a **certain input** and learns from it

---

1. Kernels / Filters

- A **kernel (or filter)** is a small matrix (e.g., 3×3 or 5×5) that slides across the image to detect features.
- Each filter learns to recognize a **specific feature**, like an edge, corner, or texture.

---

2. Channels

- Images usually have **multiple channels**:
  - Grayscale → 1 channel
  - RGB → 3 channels (Red, Green, Blue)
- A CNN processes all channels **simultaneously**, combining them to extract richer features.

Each filter operates **across all input channels** and produces one **output feature map**.

---

3. Depth of Layer

- The **depth** of a CNN layer = number of filters used.
- Example:
  If a convolution layer has 64 filters → output has **64 feature maps**.
- Each filter captures a **different type of feature** from the same image.

4. Stride

- **Stride** defines how many pixels the filter moves at a time.
- A larger stride → smaller output (less overlap, more downsampling).
- Example: stride = 1 (moves 1 pixel at a time), stride = 2 (skips every other pixel).

---

5. Padding

When filters reach image borders, they might “fall off” the edge.
Padding adds extra pixels (usually zeros) around the image to control output size.

Type	Description
Valid Padding	No padding → output smaller than input.
Same Padding	Pads so output size = input size.

---

6. Dropout

Dropout is a **regularization technique** to prevent overfitting.
During training, it randomly "turns off" some neurons (sets them to 0) in each iteration.

- Dropout rate (e.g., 0.5) = probability of dropping a neuron.
- Helps the network generalize better and not memorize training data.

7. Pooling

Pooling (or **subsampling**) is a **downsampling operation** used in CNNs to **reduce the spatial size** of feature maps (width × height) while **preserving important information**.

It's applied **after convolutional layers** to make the network:

- Smaller and faster,
- More robust to small shifts, rotations, and distortions in the input.

After a convolutional layer, you get a **feature map** — which can still be quite large.
Pooling helps by:

1. **Reducing computation** → fewer parameters and operations.
2. **Preventing overfitting** → by compressing information.
3. **Retaining important features** → only the strongest or average responses are kept.

Types of Pooling

Type	Operation	Use Case
Max Pooling	Takes the **maximum value** in each window	Most common — keeps strongest features
Average Pooling	Takes the **average value**	Smooths the feature map
Global Average Pooling (GAP)	Averages the entire feature map into one value per channel	Used before fully connected layers to reduce dimensions
Global Max Pooling	Takes the max of entire feature map	Captures the most activated feature per channel

8. Output Size Formula

After a convolution operation, the **output dimensions** are calculated as:

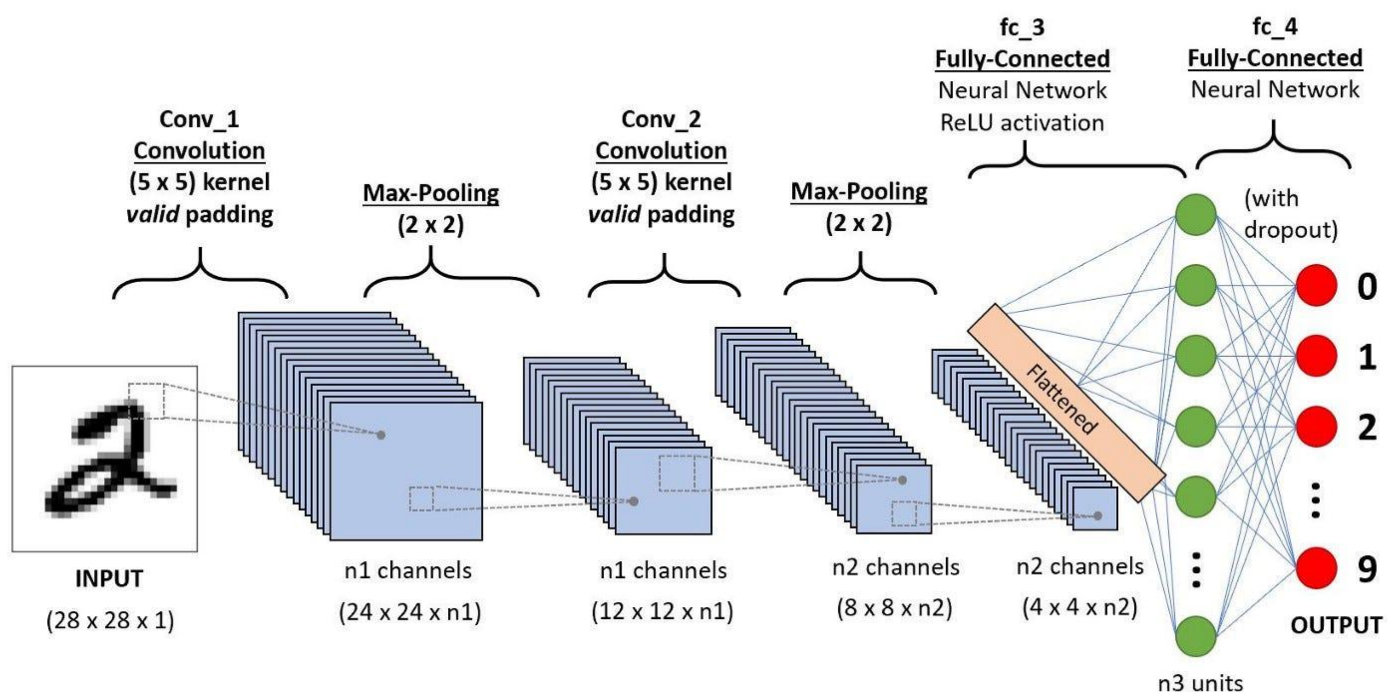
$$\frac{W-F+2P}{S}+1$$

Where:

- O: output size (height or width)
- W: input size
- F: filter (kernel) size
- P: padding
- S: stride

Example:
Input = 32×32,
Filter = 5×5,
Padding = 2,
Stride = 1

$$O=\frac{32-5+2(2)}{1}+1=32$$



Made by Youssef Amgad Elkhatib

CNN Model Example

1. Importing Libraries

```
The first section loads all the necessary dependencies for the model:
import os, warnings, tensorflow as tf, seaborn as sns, random, numpy as np
from tensorflow.keras.callbacks import Callback
from keras import models, layers
from matplotlib import pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from tensorflow.keras import regularizers
from tensorflow.keras.callbacks import ModelCheckpoint
```

Purpose:

- tensorflow / keras: For building and training the CNN.
- numpy: For numerical operations.
- matplotlib, seaborn: For visualization (accuracy/loss curves, confusion matrix, predictions).
- sklearn: For metrics, dataset splitting, and evaluation.
- warnings: To suppress unnecessary warnings.
- os, random: For managing environment and reproducibility.

---

2. Setting a Random Seed

```
SEED = 42
os.environ['PYTHONHASHSEED'] = str(SEED)
os.environ['TF_DETERMINISTIC_OPS'] = '1'
tf.config.threading.set_intra_op_parallelism_threads(1)
tf.config.threading.set_inter_op_parallelism_threads(1)
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

Why this matters:
Machine learning models have inherent randomness (in weight initialization, shuffling, etc.).
Setting a **fixed random seed** ensures reproducibility — i.e., the same model trained multiple times gives identical results.

3. Dataset Loading

```
train_path = '/kaggle/input/brain-tumor-mri-dataset/Training'
test_path = '/kaggle/input/brain-tumor-mri-dataset/Testing'

train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    train_path,
    image_size=(299,299),
    batch_size=32,
    shuffle=True,
    seed=SEED
)

test_ds = tf.keras.preprocessing.image_dataset_from_directory(
    test_path,
    image_size=(299,299),
    batch_size=32,
    shuffle=True,
    seed=SEED
)
```

Explanation:

- Loads images from folders, where each subfolder is a class label (e.g., glioma, pituitary, etc.).
- Resizes all images to (299, 299) for model consistency.
- Shuffles and batches them.

4. Data Augmentation

```
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
    tf.keras.layers.RandomContrast(0.1),
])
data_augmentation_2 = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomZoom(0.1),
    tf.keras.layers.RandomContrast(0.1),
])

augmented_train_ds = train_ds.map(lambda x, y: (data_augmentation(x, training=True), y))
augmented_train_ds_2 = train_ds.map(lambda x, y: (data_augmentation_2(x, training=True), y))

# Combine original + augmented

train_ds = train_ds.concatenate(augmented_train_ds)
train_ds = train_ds.concatenate(augmented_train_ds_2)
```

Purpose:

- Artificially increases the diversity of the training data by flipping, rotating, zooming, and changing contrast.
- Helps prevent **overfitting**, making the model generalize better.

They created **two augmentations** (data_augmentation and data_augmentation_2) and concatenated both with the original dataset — effectively tripling the dataset size.

5. Grayscale Conversion & Normalization

```
def dataset_to_numpy(dataset):
  x, y = [], []
  for images, labels in dataset:
    images = tf.image.rgb_to_grayscale(images)
    images = tf.cast(images, tf.float32) / 255.0
    x.append(images.numpy())
    y.append(labels.numpy())
  return np.concatenate(x), np.concatenate(y)
```

Explanation:

- Converts RGB MRI scans into grayscale, since color is not informative here.
- Normalizes pixel values from [0, 255] → [0, 1], making training numerically stable.

Then it converts the TensorFlow dataset into NumPy arrays (x_train, y_train, x_test, y_test).

---

6. Label Encoding & Train/Validation Split

```
x_train, y_train = dataset_to_numpy(train_ds)
x_test, y_test   = dataset_to_numpy(test_ds)

y_train = to_categorical(y_train, num_classes=4)
y_test  = to_categorical(y_test, num_classes=4)

# Split train into (train + validation)

x_train_new, x_val, y_train_new, y_val = train_test_split(
  x_train, y_train, test_size=0.2, random_state=SEED
)
```

Explanation:

- to_categorical() → one-hot encodes labels (e.g., [1,0,0,0] for "glioma").
- Dataset is further split into **training** and **validation** (80%-20%) for tuning hyperparameters.

7. CNN Model Architecture

```
def create_model(input_shape):
    model = models.Sequential()

    # Convolutional layer 1
    model.add(layers.Conv2D(64, (5,5), activation="relu", input_shape=input_shape))
    model.add(layers.MaxPooling2D(pool_size=(3,3)))
    model.add(layers.Dropout(0.2))

    # Convolutional layer 2
    model.add(layers.Conv2D(64, (5,5), activation="relu"))
    model.add(layers.MaxPooling2D(pool_size=(3,3)))
    model.add(layers.Dropout(0.2))

    # Convolutional layer 3
    model.add(layers.Conv2D(128, (4,4), activation="relu"))
    model.add(layers.MaxPooling2D(pool_size=(2,2)))
    model.add(layers.Dropout(0.3))

    # Convolutional layer 4
    model.add(layers.Conv2D(128, (4,4), activation="relu"))
    model.add(layers.MaxPooling2D(pool_size=(2,2)))
    model.add(layers.Dropout(0.3))


    # Flatten
    model.add(layers.Flatten())

    # Dense layers
    model.add(layers.Dense(512, activation="relu", kernel_regularizer=regularizers.l2(1e-4)))
    model.add(layers.Dropout(0.6))
    model.add(layers.Dense(4, activation="softmax"))

    # Compile model with Adam optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.85, beta_2=0.9925)
    loss = tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.1)
    model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])


    return model
```

Layers Overview

Layer	Type	Purpose
Conv2D(64, (5,5))	Feature extraction	Detects patterns (edges, shapes)
MaxPooling2D(3,3)	Downsampling	Reduces image size, keeps key features
Dropout(0.2–0.3)	Regularization	Prevents overfitting by randomly dropping neurons
Flatten()	Flatten 2D → 1D	Prepares data for Dense layers
Dense(512) + L2	Fully connected	Learns high-level features
Dense(4, softmax)	Output	Predicts one of 4 tumor classes

Optimizer:

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.85, beta_2=0.9925)

loss = tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.1)

- **Adam** is adaptive and fast.
- **Label smoothing** prevents overconfidence by softening labels slightly.

8. Custom Checkpoint Callback

```
# Class that saves the best model based on validation accuracy, then validation loss

class DualMetricCheckpoint(Callback):
    def __init__(self, filepath):
        super(DualMetricCheckpoint, self).__init__()
        self.filepath = filepath
        self.best_acc = -np.Inf
        self.best_loss = np.Inf

    def on_epoch_end(self, epoch, logs=None):
        val_acc = logs.get("val_accuracy")
        val_loss = logs.get("val_loss")

        # If accuracy improves or (same acc within 1e-4 AND lower loss), save the model
        if (val_acc > self.best_acc) or \
           (np.isclose(val_acc, self.best_acc, atol=1e-4) and val_loss < self.best_loss):

            print(f"\nEpoch {epoch+1}: val_accuracy={val_acc:.4f}, val_loss={val_loss:.4f} "
                  f"--> saving best model")

            self.best_acc = val_acc
            self.best_loss = val_loss
            self.model.save(self.filepath)
```

Purpose:

- Custom callback that saves the **best model** based on:
    - Highest validation accuracy.
    - If accuracy is same, then lowest validation loss.

This is smarter than the usual ModelCheckpoint, as it prioritizes model generalization.

9. Training

```
# Train with checkpoint to save the best model

checkpoint = DualMetricCheckpoint("best_model.h5")

# Create the model

input_shape = x_train_new.shape[1:]
model = create_model(input_shape)

# Train the model

history = model.fit(
   x_train_new, y_train_new,
   epochs=55,
   batch_size=32,
   verbose=1,
   validation_data=(x_val, y_val),
   callbacks=[checkpoint]
)
```

Trains the model for **55 epochs** with:

- Training data: (x_train_new, y_train_new)
- Validation data: (x_val, y_val)
- Batch size: 32

History keeps a record of accuracy and loss across epochs for plotting later.

10. Model Evaluation

```
# Load best saved model

model.load_weights("best_model.h5")

# Evaluate the model

train_loss_best, train_acc_best = model.evaluate(x_train, y_train, verbose=0)
val_loss_best, val_acc_best = model.evaluate(x_val, y_val, verbose=0)
test_loss_best, test_acc_best = model.evaluate(x_test, y_test, verbose=0)

# Save the model

model.save("/kaggle/working/OncoVision.h5")

# Display accuracies and losses

print("\n=== Best Model Evaluation ===\n")
print(f"Train Accuracy:      {train_acc_best:.4f}   | Training loss:    {train_loss_best:.4}")
print(f"Validation Accuracy:  {val_acc_best:.4f}   | Validation loss:  {val_loss_best:.4}")
print(f"Test Accuracy:       {test_acc_best:.4f}   | Test loss:       {test_loss_best:.4}")
```

- Evaluates the model performance on all three sets.
  Then prints out the scores to check for overfitting or underfitting.

Made by Youssef Amgad Elkhatib

11. Classification Report & Confusion Matrix

```
# Convert one-hot encoded labels to class indices

y_true = np.argmax(y_test, axis=1)

# Convert predicted probabilities to class indices

y_pred = model.predict(x_test, verbose=0)
y_pred = np.argmax(y_pred, axis=1)

# Generate classification report

report = classification_report(y_true, y_pred, target_names=class_names)
print("\n\n=== Classification Report ===\n\n")
print(report)
# Display Confusion matrix

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()
```

- Converts predictions and true labels back from one-hot vectors.
- **Classification Report:** Precision, Recall, F1-score.
- **Confusion Matrix:** Shows which tumor types are most often confused.

Visualized using **Seaborn heatmap**.

12. Sample Predictions Visualization

```
# Display 20 Sample predictions with images

print("\n=== Sample Predictions (with images) ===\n\n")
fig, axes = plt.subplots(4, 5, figsize=(15, 12))
fig.subplots_adjust(hspace=0.7)

for i, ax in enumerate(axes.flat):
    true_label = class_names[y_true[i]]
    pred_label = class_names[y_pred[i]]
    img = x_test[i].squeeze()
    color = "green" if true_label == pred_label else "red"
    ax.imshow(img, cmap='gray')
    ax.set_title(f"True: {true_label}\nPred: {pred_label}", fontsize=12,color=color)
    ax.axis("off")

plt.show()
```

Displays 20 test images with:

- True label.
- Predicted label (colored green if correct, red if wrong).

This gives a **qualitative understanding** of how well the model performs visually.

13. Accuracy & Loss Curves

```
# Display Accuracies graph

plt.figure(figsize=(8,5))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.title("Model Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()

# Display Losses graph

plt.figure(figsize=(8,5))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title("Model Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

These graphs are crucial:

- **Accuracy curve:** Shows how well the model is learning.
- **Loss curve:** Helps identify overfitting if validation loss diverges from training loss.

Conclusion

In conclusion, Computer Vision represents a paradigm shift from defining rules to learning features. While basic CNNs provide the foundation for feature extraction, the field has evolved into specialized architectures designed for speed (MobileNet) and accuracy (EfficientNet). The study highlights that success in CV requires not just choosing a model, but understanding the hierarchy of features—from edges and textures to complex objects. By mastering these concepts, practitioners can build systems capable of automating tasks in healthcare, autonomous driving, and security, effectively bridging the gap between human perception and digital interpretation.

Feedback & Contribution

This is a living document. If you have feedback, suggestions, or additional insights that could improve it, I welcome your input. Sharing ideas helps keep this material accurate, relevant, and valuable for everyone.

Copyright & Usage

© **2025 [Youssef Amgad Elkhatib]**.
This document is intended **for learning and reference purposes only**.
You are free to read, use, and share the content for personal or educational use, but:

- **Do not copy or republish this document as your own.**
- Always provide proper credit when referencing.
- Commercial use or redistribution without permission is not allowed.

You may not reproduce this document in whole or in part without attribution. Suggestions and feedback are welcome, and contributors will be acknowledged, but copyright remains with the author.

Acknowledgments

This document is authored and copyrighted by **[Youssef Amgad Elkhatib]**.
Special thanks to the contributors who provided valuable feedback and suggestions for improvements:

Name	Contribution