NYCU Pattern Recognition, Homework 4

312551810, Alexandre Pauly

Part. 1, Kaggle (70% [50% comes from the competition]):

(10%) Implementation Details

In this implementation, I used the VGG16 pre-trained model to extract ca-features from images for use in classification. Here's a summary of the main steps and choices made:

1. Loading images

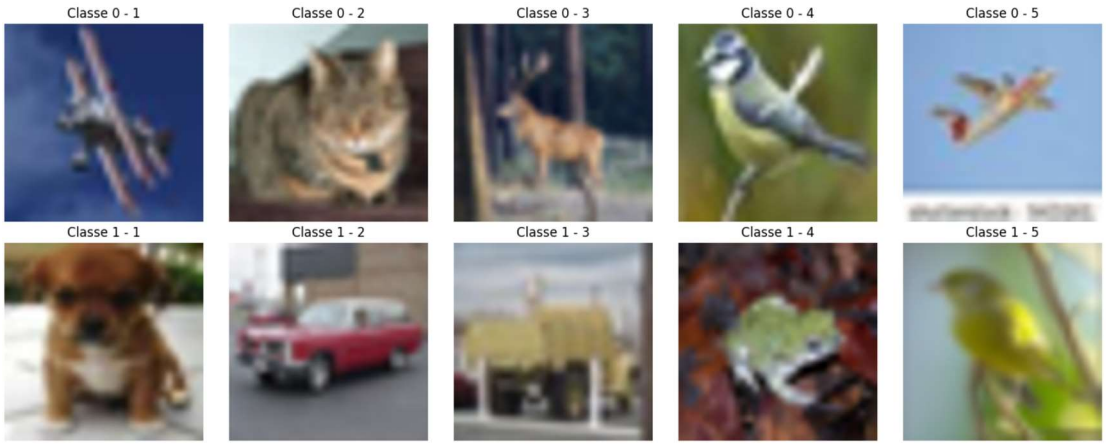
```python
def load_train_images(folder):
    # Initialization of variables
    images = []  # List to store images
    labels = []  # List to store labels

    class_folders = [os.path.join(folder, class_folder) for class_folder in sorted(os.listdir(folder))]

    # For each folder
    for label, class_folder in enumerate(class_folders):
        for filename in sorted(os.listdir(class_folder)):
            if filename.endswith('.pkl'):
                file_path = os.path.join(class_folder, filename)
                try:
                    with open(file_path, 'rb') as f:
                        image = pickle.load(f)
                        images.append(image)
                        labels.append(label)
                except (EOFError, pickle.UnpicklingError):
                    print(f"Error loading {file_path}, skipping this file.")

    # Transform list to array
    images = np.array(images)
    labels = np.array(labels)

    return images, labels
```

```python
def load_test_images(test_folder):
    images = []      # List to store images
    filenames = []   # List to store filenames

    # For each element
    for filename in sorted(os.listdir(test_folder)):
        if filename.endswith('.pkl'):
            file_path = os.path.join(test_folder, filename)
            try:
                with open(file_path, 'rb') as f:
                    image = pickle.load(f)
                    images.append(image)
                    filenames.append(filename)
            except (EOFError, pickle.UnpicklingError):
                print(f"Error loading {file_path}, skipping this file.")

    # Transform list to array
    images = np.array(images)
    filenames = np.array(filenames)

    return images, filenames
```

```python
# Initialization of variables
train_folder = '/kaggle/input/nycu-ml-pattern-recognition-hw-4/released/train'  # Train data
test_folder = '/kaggle/input/nycu-ml-pattern-recognition-hw-4/released/test'    # Test data

# Loading data
train_images, train_labels = load_train_images(train_folder)
test_images, test_filenames = load_test_images(test_folder)

# Messages display
print("Number of training images :", len(train_images))
print("Number of training labels :", len(train_labels))
print("Number of test images :", len(test_images))
print()
print("Shape of training images :", train_images.shape)
print("Shape of training labels :", train_labels.shape)
print("Shape of test images :", test_images.shape)
```

```
Number of training images : 300
Number of training labels : 300
Number of test images : 200

Shape of training images : (300, 256, 128, 128, 3)
Shape of training labels : (300,)
Shape of test images : (200, 256, 128, 128, 3)
```

2. Images visualisation



3. Caracteristics extraction by using pre-trained models

- The *extract_features* function processes images and extracts features using the model, then flattens them to make them usable by traditional classification models.

```python
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
model = Model(inputs=base_model.input, outputs=base_model.output)

def extract_features(images):
    processed_images = preprocess_input(images)
    features = model.predict(processed_images)
    flattened_features = features.reshape((features.shape[0], -1))
    return flattened_features

train_features = [extract_features(bag) for bag in train_images]
train_features = np.array(train_features)

# Flatten the features
flattened_train_features = train_features.reshape((train_features.shape[0], -1))

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(flattened_train_features, train_labels, test_size=0.2, random_state=42)
```

4. Training and evaluation of classification models

- The extracted features are divided into training and validation sets.
- Several classification models are tested: Random Forest, SVM, KNN, Gradient Boosting and Logistic Regression. Random is fixed to get the same result.
- Each model is trained on the training set and evaluated on the validation set. Results are compared to select the best model based on validation accuracy.

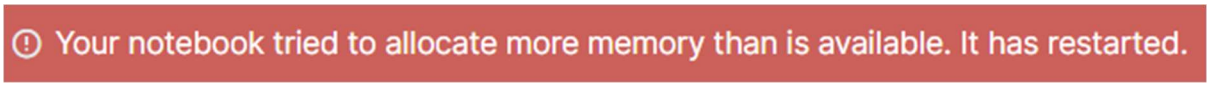
```
models = {
    "Random Forest": RandomForestClassifier(n_estimators=200, random_state=42),
    "SVM": SVC(kernel='linear', probability=True, random_state=42),
    "KNN": KNeighborsClassifier(n_neighbors=5),
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100, random_state=42),
    "Logistic Regression": LogisticRegression(max_iter=200, random_state=42)
}

def evaluate_models(X_train, y_train, X_val, y_val, models):
    results = {}

    for name, model in models.items():
        model.fit(X_train, y_train)
        accuracy = model.score(X_val, y_val)
        results[name] = accuracy
        print(f'{name} validation accuracy: {accuracy}')

    return results

results = evaluate_models(X_train, y_train, X_val, y_val, models)

# Choose the best model (for example, based on highest accuracy)
best_model_name = max(results, key=results.get)
best_model = models[best_model_name]

print(f'Best model: {best_model_name} with accuracy {results[best_model_name]}')

# Train the best model on the entire training set
best_model.fit(flattened_train_features, train_labels)
```

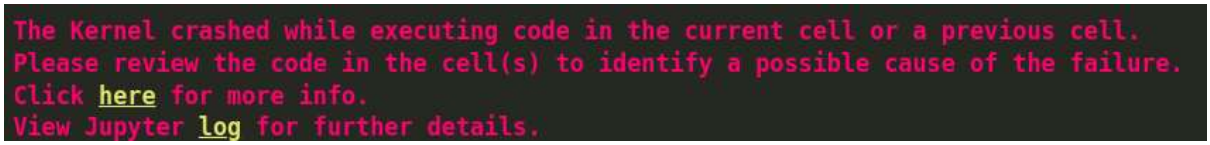
5. Prediction on test data and creation of submission file

- The best model is then used to predict the labels of the test images.
- Predictions are saved in a CSV file for submission.

```
test_features = [extract_features(bag) for bag in test_images]
test_features = np.array(test_features)
flattened_test_features = test_features.reshape((test_features.shape[0], -1))

test_predictions = best_model.predict(flattened_test_features)

def create_submission_file(test_files, predictions, output_folder):
    # Crée le dossier de sortie s'il n'existe pas
    os.makedirs(output_folder, exist_ok=True)

    output_file = os.path.join(output_folder, 'submission.csv')
    with open(output_file, 'w') as f:
        f.write('image_id,y_pred\n')
        for file, pred in zip(test_files, predictions):
            file_id = os.path.splitext(file)[0]
            f.write(f'{file_id},{pred}\n')

output_folder = "results"
create_submission_file(test_filenames, test_predictions, output_folder)
print("Submission file created successfully.")
```

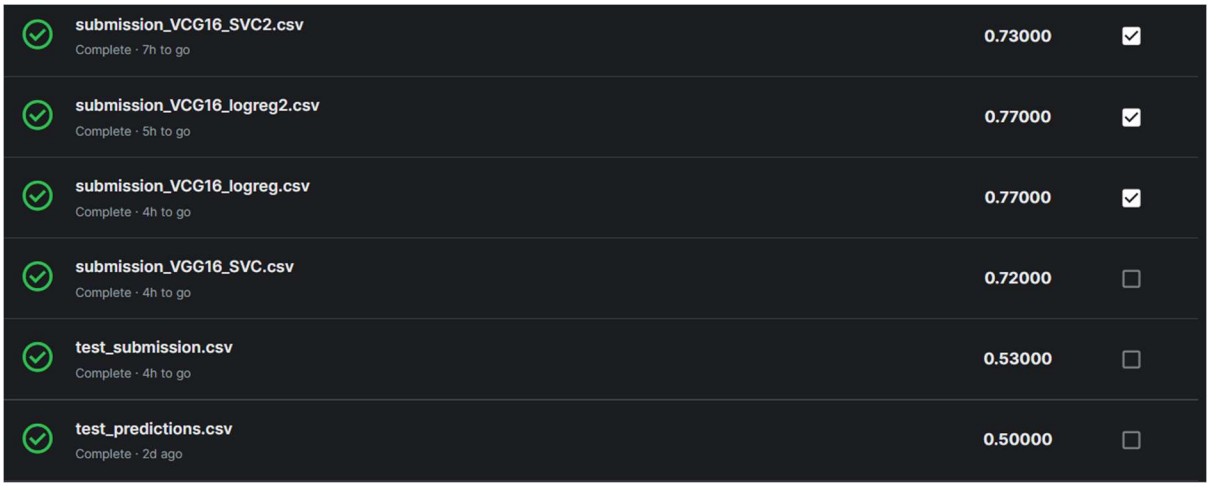
6. Issues: By using kaggle, some crash or limitations like this one (look picture). So, it's the reason why I'm using pre-trained model to reduce this kind issue because when I'm using handcrafted models, it's crashing everytime.





(10%) Experimental Results

The image below show the different results with 300 images (80% training and 20% validation)



Part. 2, Questions (30%):

1. (10%) Why Sigmoid or Tanh is not preferred to be used as the activation function in the hidden layer of the neural network? Please answer in detail.

Sigmoid and Tanh functions are not widely used today for hidden layers due to their characteristics at the extremes of the function. The shape of these functions near their upper and lower limits poses several problems:

1. **Vanishing gradient problem**:

- **Sigmoid**: The Sigmoid function returns input values between 0 and 1. When the input values are very large or very small, the derivatives of the Sigmoid function become very close to zero. This means that during back-propagation, the gradients can become extremely small, slowing down the learning of the weights in the previous layers.
- **Tanh**: Although the Tanh function compresses input values between -1 and 1 and is zero-centered, it also suffers from the vanishing gradient problem for very large or very small input

values. The gradients become very weak, which can prevent the model from learning effectively.

2. **Saturation problem**:
   - **Sigmoid and Tanh**: Both functions tend to saturate for very large or very small input values, meaning the output of the function becomes flat and the derivatives become almost zero. When this happens, weight updates become negligible, making the training of the network inefficient.
3. **Non-zero-centered output (for Sigmoid)**:
   - **Sigmoid**: The output of the Sigmoid function is always positive, which can lead to imbalances in activation. This can cause biased gradients during back-propagation, making optimization more difficult and less efficient.

As a result, alternative activation functions such as ReLU or its variants are often preferred for the hidden layers of neural networks:

- **ReLU**: The ReLU function is defined as f(x)=max(0,x). It is simple, allows for faster learning, and largely avoids the vanishing gradient problem since its derivatives are constant for positive inputs.
- **Leaky ReLU**: This variant of ReLU allows a small slope for negative input values, which helps to avoid the problem of "dead neurons" (neurons that stop activating during training).

2. (10%) What is over fitting? Please provide at least three techniques with explanations to overcome this issue.

Over fitting is an undesirable behavior in machine learning that occurs when the model makes accurate predictions for the training data but not for new data. When using machine learning models for forecasting, they first train the model on a known dataset. Based on this information, the model then attempts to predict outcomes for new datasets. An over fitted model may provide inaccurate forecasts and may not perform well on all types of new data.

It is possible to avoid over fitting by diversifying and adapting the training dataset or by using more advanced techniques:

1. **Regularization**: Regularization is a set of training and optimization techniques aimed at reducing over fitting. These methods aim to eliminate factors that do not impact the prediction results by evaluating features based on their importance. Regularization imposes a penalty on the complexity of the model, helping to prevent it from fitting too closely to the training data. The two common regularization techniques are:
   - **L1 (Lasso)**: This technique adds a penalty proportional to the absolute value of the parameter coefficients to the loss term, which can lead to some coefficients being set to zero, effectively performing feature selection.
   - **L2 (Ridge)**: This technique adds a penalty proportional to the square of the parameter coefficients, reducing their values without setting them to zero.

The effect of regularization is to limit the model's weights, preventing it from becoming too complex.

2. **Ensemble Methods**: Ensemble methods combine predictions from several distinct machine learning algorithms to improve overall performance and reduce overfitting, based on the "wisdom of the crowd" principle. Some models are called weak learners because their individual results are often inaccurate. Ensemble methods combine these weak learners to achieve more accurate results. The two main ensemble methods are:

   - **Bagging (Bootstrap Aggregating)**: This method trains multiple models in parallel on random subsets of the training data and then aggregates their predictions. An example is Random Forest, which uses multiple decision trees to enhance prediction robustness.
   - **Boosting**: This method trains models sequentially, with each new model focusing on correcting the errors of the previous models. An example is Gradient Boosting, which builds decision trees sequentially and weights their contributions to minimize overall errors.

   Ensemble methods reduce variance and bias, improving performance on test data.

3. **Data Augmentation**: Another method to limit over fitting is to increase the dataset. This can be done by collecting more real data or by using data augmentation techniques. These techniques create new data from existing data by applying various transformations, such as:

   - **Image Rotations**: Slightly rotating the images by a few degrees.
   - **Cropping**: Slightly modifying the dimensions of the images.
   - **Brightness Changes**: Adjusting color intensity.

   For example, in image classification, a picture of a cat can be slightly rotated, cropped, or its color intensity can be modified to create new versions of this image. This increases the diversity of the training data without needing to collect more real data. By increasing the quantity and diversity of the training data, data augmentation helps the model to generalize better.

3. (15%) Given a valid $k_1(x,x')$ kernel, prove that the following proposed functions are or are not valid kernels. If one is not a valid kernel, give $k(x,x')$ an example of that the corresponding K is not positive semi-definite and show its eigenvalues.

   a. $k(x, x') = k_1(x, x') + \|x\|^2$
   b. $k(x, x') = k_1(x, x') - 1$
   c. $k(x, x') = k_1(x, x') + \exp(x^T x')$
   d. $k(x, x') = \exp\big(k_1(x, x')\big) - 1$

   a. $k(x, x') = k_1(x, x') + \|x\|^2$

   Let's consider the Gramm matrix for two points x_1 and x_2 :

$$K = \begin{pmatrix} k_1(x_1, x_1) + \|x_1\|^2 & k_1(x_1, x_2) + \|x_1\|^2 \\ k_1(x_2, x_1) + \|x_2\|^2 & k_1(x_2, x_2) + \|x_2\|^2 \end{pmatrix}$$

The off-diagonal terms are not symmetric unless $\|x_1\|^2 = \|x_2\|^2$. Moreover, the positive semid efiniteness can be violated. Consider $k_1(x_1, x_2) = k_1(x_2, x_1) = 0$ and $\|x_1\|^2 \neq \|x_2\|^2$:

$$K = \begin{pmatrix} \|x_1\|^2 & \|x_1\|^2 \\ \|x_2\|^2 & \|x_2\|^2 \end{pmatrix}$$

The eigenvalues of this matrix are $\frac{(\|x_1\|^2 + \|x_2\|^2 \pm \sqrt{((\|x_1\|^2 - \|x_2\|^2)^2)})}{2}$. This can yield negative values depending of $\|x_1\|^2$ and $\|x_2\|^2$, indicating that K is not positive semi-definite. So, this function is not a valid kernel.

b. $k(x, x') = k_1(x, x') - 1$

Consider $k_1(x, x) = 1$ for all x. Then, for two points $x_1$and x_2 :

$$K = \begin{pmatrix} k_1(x_1, x_1) - 1 & k_1(x_1, x_2) - 1 \\ k_1(x_2, x_1) - 1 & k_1(x_2, x_2) - 1 \end{pmatrix} = \begin{pmatrix} 0 & k_1(x_1, x_2) - 1 \\ k_1(x_2, x_1) - 1 & 0 \end{pmatrix}$$

If $k_1(x_1, x_1) = 0 : K = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$

In this case, eigenvalues of are -1 and +1, including a negative value. So K is not positive semi-definite and not a valid kernel.

c. $k(x, x') = k_1(x, x') + e^{x^T x}$

Consider the kernel matrix K for this function:

$$K = k_1(x_1, x_2) + e^{x_1^T x_2}$$

This is the sum of the positive semi-definite kernel k_1 and $e^{x_i^T x_j}$, which is also positive semi-definite because $e^{x_i^T x_j}$ can be interpreted as a valid kernel itself (specifically, it can be derived from the RBF kernel).

Since the sum of positive semi-definite matrices is positive semi-definite, so this kernel is a valid kernel.

d. $k(x, x') = e^{k_1(x, x')} - 1$

Consider $k_1(x, x') = 0$:

$$k(x, x') = e^0 - 1 = 0$$

But, if we consider $k_1(x, x') = \ln(2)$:

$$k(x, x') = e^{\ln(2)} - 1 = 1$$

For Gramm matrix, with $x_1 = x_2$:

$$K = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Like the b., eigenvalues of are -1 and +1, including a negative value. So K is not positive semi-definite and not a valid kernel.