ANNA UNIVERSITY

MIT CAMPUS

PROJECT-REPORT

SURFACE CRACK DETECTION

SUB CODE:**AZ5311**

SUB NAME: **DATA SCIENCE LABORATORY**

TEAM MEMBERS:

NIVETHA T (2022510007)

JHANAVI.S (2022510020)

JESSICA KATHERINE.F (2022510021)

SURFACE CRACK DETECTION

The datasets contains images of various concrete surfaces with and without crack. The image data are divided into two as negative (without crack) and positive (with crack) in separate folder for image classification. Each class has 20000 images with a total of 40000 images with 227 x 227 pixels with RGB channels. The dataset is generated from 458 high-resolution images (4032x3024 pixel) with the method proposed by Zhang et al (2016). High resolution images found out to have high variance in terms of surface finish and illumination condition. No data augmentation in terms of random rotation or flipping or tilting is applied.

MARKET ANALYSIS

The market for surface crack detection datasets is primarily driven by industries such as manufacturing, automotive, aerospace, and infrastructure. These industries heavily rely on ensuring the integrity and safety of materials and components, making surface crack detection a critical aspect of quality control and maintenance processes. Here's an overview of the market and its key aspects:

Industry Demand:

Manufacturing Quality Control: Industries like automotive, electronics, and heavy machinery use surface crack detection datasets to ensure the quality of products and prevent faulty items from reaching the market.

Infrastructure and Construction: Inspection of structural components like bridges, buildings, pipelines, and railways requires surface crack detection for safety assessments and maintenance.

Aerospace and Defense: Ensuring the integrity of aircraft parts and critical components is crucial for safety and reliability in the aerospace sector.

Dataset Uses:

Algorithm Development: Machine learning and computer vision algorithms utilize these datasets for training models to automatically detect and classify cracks.

Research and Development: Researchers use these datasets to innovate and improve crack detection techniques, exploring new methods for accuracy and efficiency.

Testing and Validation: Companies use these datasets to test the effectiveness of their crack detection systems and algorithms.

Market Dynamics:

Quality Assurance Emphasis: As industries increasingly prioritize quality and safety, the demand for accurate and reliable crack detection datasets rises.

Advancements in Technology: The integration of AI/ML and computer vision in crack detection systems amplifies the need for high-quality datasets for training and validation.

Regulatory Compliance: Compliance with stringent safety standards and regulations drives the adoption of advanced crack detection methods, increasing the need for relevant datasets.

Challenges and Opportunities:

Data Variability: Datasets with diverse surface types, crack sizes, and environmental conditions are crucial for robust algorithm development.

Privacy and Ethical Considerations: Ensuring the responsible use of datasets and addressing privacy concerns in handling sensitive data.

Customization: Tailoring datasets to specific industry needs and emerging technologies offers opportunities for niche market segments.

Market Participants:

Data Providers: Companies specializing in curating and providing high-quality crack detection datasets to industries and research institutions.

Tech Companies: AI/ML solution providers offering crack detection systems, using these datasets as part of their solutions.

Research Institutions: Academia contributing to the development and enhancement of crack detection algorithms.

EXISTING SYSTEM CASE STUDY

Integration of an advanced dataset: High-resolution images and sensor data are seamlessly integrated into the system.

Real-time crack detection: The dataset, combined with computer vision algorithms and machine learning models, allows for the instantaneous identification and classification of structural cracks.

Practical scenario: The system demonstrates its efficacy by promptly detecting a potential crack in a critical bridge support, triggering immediate alerts for timely intervention by engineers.

Ongoing improvements: Continuous efforts are directed towards refining algorithms for nuanced crack classifications and expanding the dataset's coverage to include diverse environmental conditions.

Data-driven decision-making: The dataset contributes to informed decision-making in structural interventions, optimizing resource allocation and minimizing downtime for critical infrastructure.

Challenges and research efforts: Research and development initiatives are ongoing to address challenges in algorithm refinement and expand the dataset to enhance overall system effectiveness.

Here in our project we've used CNN classifier so let us know about that:

A **CNN (Convolutional Neural Network)** classifier is a type of deep learning model specifically designed for image recognition and computer vision tasks. CNNs have proven to be highly effective in tasks such as image classification, object detection, and segmentation. They are inspired by the visual processing that occurs in the human brain and leverage convolutional layers to automatically and adaptively learn spatial hierarchies of features from input images.

Here are some key components and concepts related to CNN classifiers:

Convolutional Layers:

CNNs use convolutional layers to apply filters (also called kernels) to input images. These filters help capture spatial hierarchies of features in the input data.

Convolutional layers enable the network to automatically learn and detect patterns, textures, and complex features.

Pooling Layers:

Pooling layers are used to reduce the spatial dimensions of the input volume, helping to decrease the computational load and control overfitting.

Common pooling operations include max pooling and average pooling, where the maximum or average value is taken from a set of values in the input.

Activation Functions:

Non-linear activation functions, such as ReLU (Rectified Linear Unit), are applied to introduce non-linearity into the model. This allows the network to learn complex relationships between features.

Fully Connected Layers:

After several convolutional and pooling layers, CNNs often include one or more fully connected layers for final classification. These layers connect every neuron to every neuron in the previous and subsequent layers.

Softmax Activation:

For multi-class classification tasks, the softmax activation function is often used in the output layer. It converts the network's raw output scores into probability distributions, making it suitable for assigning class labels.

Loss Function:

Cross-entropy loss is commonly used as the objective (or loss) function for training CNN classifiers. It measures the difference between predicted and true class distributions.

Training:

CNNs are trained using backpropagation and optimization algorithms like stochastic gradient descent (SGD) to minimize the loss function.

Training often involves feeding labeled training data through the network, adjusting the weights and biases during each iteration until the model converges.

Transfer Learning:

Transfer learning is a common technique in which a pre-trained CNN on a large dataset (e.g., ImageNet) is fine-tuned on a specific task or dataset. This helps leverage knowledge gained from a broader context.

CNNs have demonstrated remarkable success in various computer vision applications and have become a standard architecture for image classification tasks.

CODE EXPLANATION

[6]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

from pathlib import Path
from sklearn.model_selection import train_test_split

import tensorflow as tf

from sklearn.metrics import confusion_matrix, classification_report

import warnings
warnings.simplefilter("ignore")
```

This code includes imports for popular Python libraries such as NumPy, Pandas, Matplotlib, Seaborn, Plotly Express, pathlib, scikit-learn (specifically **train_test_split**, **confusion_matrix**, and **classification_report**), TensorFlow, and a configuration to ignore simple warnings during runtime. Keep in mind that this snippet is a setup and import section, and the actual implementation, such as data loading, model training, and visualization, is not included.

[7]:
```
positive_dir = Path('C:\\Users\\Admin\\Desktop\\nive\\data␣
↪sets\\crack\\Positive')
negative_dir = Path('C:\\Users\\Admin\\Desktop\\nive\\data␣
↪sets\\crack\\Negative')
```

This code defines two file paths using the **Path** class from the **pathlib** module in Python. The paths, stored in variables **positive_dir** and **negative_dir**, point to directories containing positive and negative examples of a dataset related to cracks. The paths are specified as absolute paths, indicating the location on the **C:** drive. The directory structure suggests that within the **Desktop/nive/data sets/crack** directory, there are subdirectories named **Positive** and **Negative**, representing the positive and negative subsets of the crack dataset.

[8]:
```
def generate_df(image_dir, label):
    """
    Create the DataFrame of the associated directory and label.
    """

    filepaths = pd.Series(list(image_dir.glob(r'*.jpg')), name='Filepath').
↪astype(str)
    labels = pd.Series(label, name='Label', index=filepaths.index)
    df = pd.concat([filepaths, labels], axis=1)

    return df
```

- **image_dir**: The input parameter representing the directory containing image files.
- **label**: The label assigned to all the images in the specified directory.
- **pd.Series(list(image_dir.glob(r'*.jpg')), name='Filepath').astype(str)**: Generates a Pandas Series of file paths by listing all .jpg files in the specified directory. The series is converted to strings.
- **pd.Series(label, name='Label', index=filepaths.index)**: Creates a Pandas Series for labels with the given label, and the index is aligned with the file paths.
- **pd.concat([filepaths, labels], axis=1)**: Concatenates the file paths and labels along the columns to create a DataFrame.

The resulting DataFrame (**df**) contains two columns: 'Filepath' with the paths to image files and 'Label' with the assigned label.

Compile the model

[20]:
```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

History

[21]:
```
history = model.fit(
    train_images,
    validation_data=val_images,
    epochs=100,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=3,
            restore_best_weights=True
        )
    ]
)
```

OUTPUT:

```
Epoch 1/100
105/105 [==============================] - 18s 149ms/step - loss: 0.1247 -
accuracy: 0.9649 - val_loss: 0.0931 - val_accuracy: 0.9690
Epoch 2/100
105/105 [==============================] - 15s 144ms/step - loss: 0.0805 -
accuracy: 0.9747 - val_loss: 0.0574 - val_accuracy: 0.9857
5
Epoch 3/100
105/105 [==============================] - 15s 146ms/step - loss: 0.0675 -
accuracy: 0.9798 - val_loss: 0.0789 - val_accuracy: 0.9774
Epoch 4/100
105/105 [==============================] - 16s 148ms/step - loss: 0.0636 -
accuracy: 0.9792 - val_loss: 0.0570 - val_accuracy: 0.9810
Epoch 5/100
105/105 [==============================] - 16s 150ms/step - loss: 0.0598 -
accuracy: 0.9821 - val_loss: 0.0463 - val_accuracy: 0.9833
Epoch 6/100
105/105 [==============================] - 15s 141ms/step - loss: 0.0611 -
accuracy: 0.9818 - val_loss: 0.0505 - val_accuracy: 0.9845
Epoch 7/100
105/105 [==============================] - 16s 152ms/step - loss: 0.0518 -
accuracy: 0.9821 - val_loss: 0.0426 - val_accuracy: 0.9845
Epoch 8/100
105/105 [==============================] - 15s 146ms/step - loss: 0.0466 -
accuracy: 0.9845 - val_loss: 0.0494 - val_accuracy: 0.9821
Epoch 9/100
105/105 [==============================] - 15s 143ms/step - loss: 0.0535 -
accuracy: 0.9836 - val_loss: 0.0321 - val_accuracy: 0.9893
Epoch 10/100
105/105 [==============================] - 16s 157ms/step - loss: 0.0484 -
```

accuracy: 0.9830 - val_loss: 0.0357 - val_accuracy: 0.9893

Epoch 11/100

105/105 [==============================] - 16s 156ms/step - loss: 0.0327 -

accuracy: 0.9887 - val_loss: 0.0375 - val_accuracy: 0.9857

Epoch 12/100

105/105 [==============================] - 15s 143ms/step - loss: 0.0351 -

accuracy: 0.9893 - val_loss: 0.0312 - val_accuracy: 0.9881

Epoch 13/100

105/105 [==============================] - 15s 139ms/step - loss: 0.0347 -

accuracy: 0.9893 - val_loss: 0.0511 - val_accuracy: 0.9833

Epoch 14/100

105/105 [==============================] - 15s 139ms/step - loss: 0.0344 -

accuracy: 0.9896 - val_loss: 0.0289 - val_accuracy: 0.9905

Epoch 15/100

105/105 [==============================] - 15s 139ms/step - loss: 0.0491 -

accuracy: 0.9851 - val_loss: 0.0592 - val_accuracy: 0.9857

Epoch 16/100

105/105 [==============================] - 15s 142ms/step - loss: 0.0393 -

accuracy: 0.9881 - val_loss: 0.0405 - val_accuracy: 0.9857

Epoch 17/100

105/105 [==============================] - 15s 139ms/step - loss: 0.0317 -

accuracy: 0.9887 - val_loss: 0.0218 - val_accuracy: 0.9905

Epoch 18/100

105/105 [==============================] - 15s 139ms/step - loss: 0.0292 -

accuracy: 0.9884 - val_loss: 0.0335 - val_accuracy: 0.9845

Epoch 19/100

105/105 [==============================] - 15s 140ms/step - loss: 0.0328 -

accuracy: 0.9875 - val_loss: 0.0222 - val_accuracy: 0.9929

Epoch 20/100

105/105 [==============================] - 15s 141ms/step - loss: 0.0263 -

accuracy: 0.9890 - val_loss: 0.0385 - val_accuracy: 0.9857

Explanation:

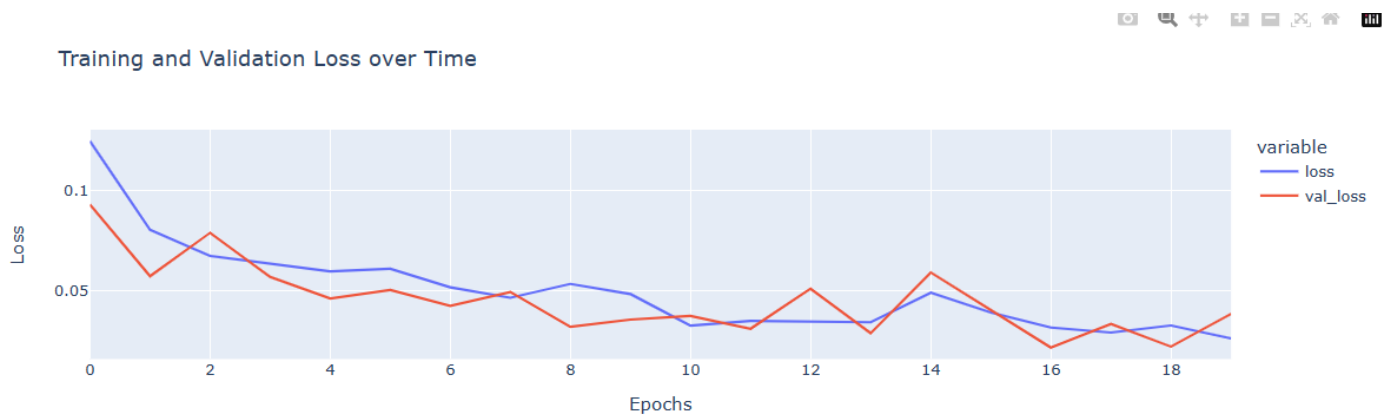
1. **Compile the Model:**
    - **optimizer='adam'**: Adam optimizer is used for model optimization. It's an adaptive learning rate optimization algorithm.
    - **loss='binary_crossentropy'**: Binary crossentropy is chosen as the loss function, suitable for binary classification tasks.
    - **metrics=['accuracy']**: The accuracy metric is chosen to evaluate model performance during training.
2. **Training the Model:**
    - **model.fit**: This method is used to train the model.
    - **train_images**: Training data (features).
    - **validation_data=val_images**: Validation data to monitor model performance during training.
    - **epochs=100**: The number of times the entire training dataset is passed through the neural network.
    - **callbacks**: A list of callbacks to apply during training. In this case, the **EarlyStopping** callback is used.
3. **Early Stopping Callback:**
    - **tf.keras.callbacks.EarlyStopping**: This callback stops training when a monitored metric (in this case, validation loss) has stopped improving.
    - **monitor='val_loss'**: Monitors the validation loss.
    - **patience=3**: Waits for three epochs with no improvement before stopping.
    - **restore_best_weights=True**: Restores the model weights from the epoch with the best value of the monitored metric.

This setup ensures that the model is compiled with specified optimization parameters and is then trained with early stopping to prevent overfitting and save the best weights based on validation loss.

[25]:
```
fig = px.line(
    history.history,
    y=['loss', 'val_loss'],
    labels={'index': "Epochs", 'value': "Loss"},
    title=("Training and Validation Loss over Time")
)

fig.show()
```

OUTPUT:



Explanation:

- **px.line**: This function from Plotly Express is used to create a line plot.
- **history.history**: The training history object usually obtained from the **fit** method of a Keras model. It contains information about training and validation loss over epochs.
- **y=['loss', 'val_loss']**: Specifies that both training loss and validation loss should be plotted on the y-axis.
- **labels={'index': "Epochs", 'value': "Loss"}**: Sets the labels for the x-axis ('Epochs') and y-axis ('Loss').
- **title="Training and Validation Loss over Time"**: Sets the title of the plot.
- **fig.show()**: Displays the created plot.

This visualization helps in understanding how the training and validation loss evolve over epochs, providing insights into the model's performance and potential issues like overfitting or underfitting

RESULTS

[26]:
```
y_pred = (model.predict(test_images).squeeze() >= 0.5).astype(int)

def evaluate_model(model):

    results = model.evaluate(test_images, verbose=0)
    loss = results[0]
    acc = results[1]

    print("Test Loss: {:.5f}".format(loss))
    print("Accuracy: {:.2f}%".format(acc * 100))

    cm = confusion_matrix(test_images.labels, y_pred)
    clr = classification_report(test_images.labels, y_pred,␣
 ↪target_names=["NEGATIVE", "POSITIVE"])

    plt.figure(figsize=(6, 6))
    sns.heatmap(cm, annot=True, fmt='g', vmin=0, cmap='Blues', cbar=False)
    plt.xticks(ticks=np.arange(2) + 0.5, labels=["NEGATIVE", "POSITIVE"])
    plt.yticks(ticks=np.arange(2) + 0.5, labels=["NEGATIVE", "POSITIVE"])
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

    print("Classification Report:\n----------------------------\n", clr)
```

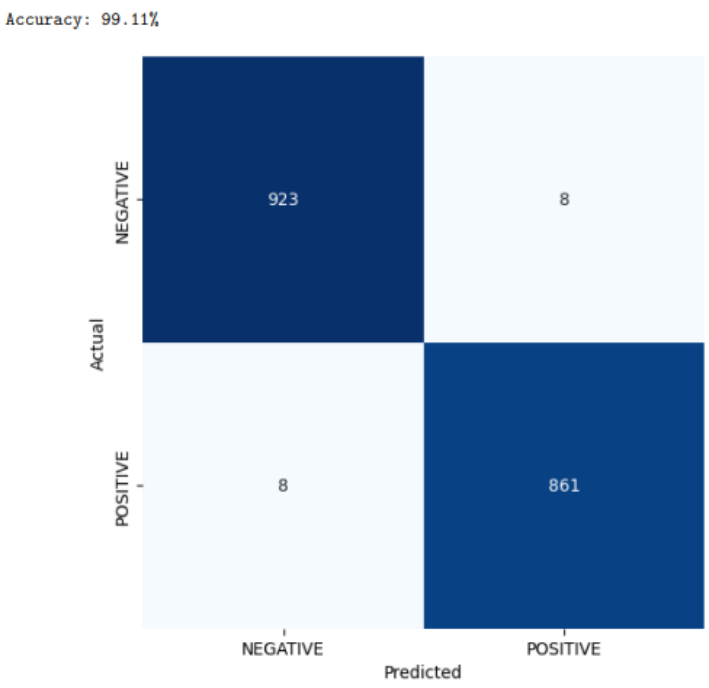
Explanation:

1. **Predictions:**
    - **y_pred = (model.predict(test_images).squeeze() >= 0.5).astype(int)**: This line generates binary predictions (0 or 1) based on the model's output probabilities. A threshold of 0.5 is used.
2. **Evaluation Function:**
    - **evaluate_model(model)**: The function takes the trained model as an argument and prints the test loss, accuracy, confusion matrix, and a classification report.
3. **Confusion Matrix and Classification Report:**
    - **confusion_matrix**: Computes the confusion matrix based on true labels (**test_images.labels**) and predicted labels (**y_pred**).
    - **classification_report**: Generates a comprehensive classification report with metrics such as precision, recall, and F1-score.
4. **Plotting Confusion Matrix:**
    - **plt.figure**, **sns.heatmap**, and related functions are used to create and display a heatmap representing the confusion matrix.
5. **Displaying Classification Report:**
    - **print("Classification Report:\n----------\n", clr)**: Outputs the classification report with detailed metrics.

This code provides a comprehensive analysis of the model's performance on the test dataset.

```
[27]: evaluate_model(model)
```

OUTPUT:



```
Classification Report:
------------------------------
              precision    recall  f1-score   support

    NEGATIVE       0.99      0.99      0.99       931
    POSITIVE       0.99      0.99      0.99       869

    accuracy                           0.99      1800
   macro avg       0.99      0.99      0.99      1800
weighted avg       0.99      0.99      0.99      1800
```

Confusion Matrix:

A confusion matrix is a table used in classification to describe the performance of a classification model on a set of data for which the true values are known. It summarizes the results of a classification problem, showing the number of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions.

- **True Positive (TP):** The number of correctly predicted positive instances.
- **True Negative (TN):** The number of correctly predicted negative instances.
- **False Positive (FP):** The number of instances predicted as positive but are actually negative.
- **False Negative (FN):** The number of instances predicted as negative but are actually positive.

The confusion matrix is particularly useful for evaluating the performance of a binary classification model and provides insights into the model's ability to make correct predictions and identify areas of improvement.

Classification Report:

A classification report is a table that presents a comprehensive set of evaluation metrics for a classification model. It includes metrics such as precision, recall, F1-score, and support for each class.

- **Precision:** Precision is the ratio of correctly predicted positive observations to the total predicted positives (TP / (TP + FP)). It measures the accuracy of the positive predictions.
- **Recall (Sensitivity or True Positive Rate):** Recall is the ratio of correctly predicted positive observations to the total actual positives (TP / (TP + FN)). It measures the ability of the model to capture all the positive instances.
- **F1-score:** The F1-score is the harmonic mean of precision and recall, providing a balance between the two metrics. It is calculated as 2 * (Precision * Recall) / (Precision + Recall).
- **Support:** Support is the number of actual occurrences of the class in the specified dataset.

The classification report is a valuable tool for understanding the overall performance of a model across different classes, especially in multiclass classification problems.

In the context of the provided code, the confusion matrix and classification report are used to assess the performance of a neural network model on a test dataset, providing insights into how well the model is classifying instances and where it may be making errors.

[28]:
```
plt.figure(figsize=(18, 12))

for i in range(15):

    plt.subplot(3, 5, i+1)
    plt.imshow(test_images[0][0][i])
    plt.title("No crack detected" if y_pred[i] == 0 else "Crack detected",
              color='blue' if y_pred[i] == test_images.labels[i] else 'red')
    plt.axis('off')

plt.show()
```

Explanation:

- **plt.figure(figsize=(18, 12))**: Creates a figure with a specified size (18 inches by 12 inches) for the grid of images.
- **for i in range(15):**: Iterates over the first 15 images in the test dataset.
- **plt.subplot(3, 5, i+1)**: Creates a subplot in a 3x5 grid, with the current iteration index **i+1** specifying the position of the subplot.
- **plt.imshow(test_images[0][0][i])**: Displays the i-th image from the test dataset.
- **plt.title("No crack detected" if y_pred[i] == 0 else "Crack detected", color='blue' if y_pred[i] == test_images.labels[i] else 'red')**: Sets the title of the subplot based on the model's prediction. If the model predicts no crack (0), the title is "No crack detected"; otherwise, it's "Crack detected". The title text is displayed in blue if the prediction matches the actual label and in red if there is a mismatch.
- **plt.axis('off')**: Removes the axis labels and ticks for better visualization.
- **plt.show()**: Displays the entire grid of subplots.

This code visually compares the model's predictions with the actual labels for the first 15 images in the test dataset. The titles of each subplot indicate whether the model correctly or incorrectly detected cracks in the images.

MISTAKES

[29]:
```
mistake_idx = (y_pred != test_images.labels).nonzero()[0]
print(len(mistake_idx), "mistakes.")
print("Indices:", mistake_idx)
```

```
16 mistakes.
Indices: [ 411  422  450  459  493  655  899  922 1051 1201 1312 1367 1453 1462
 1697 1709]
```

Explanation:

- **(y_pred != test_images.labels)**: This creates a boolean array where **True** indicates a mismatch between the model's predictions and the actual labels.
- **.nonzero()**: This method returns the indices of the elements that are non-zero (in this case, where there is a mismatch).
- **[0]**: This extracts the indices from the result of **nonzero()**.
- **mistake_idx**: This variable stores the indices where the model made mistakes.
- **print(len(mistake_idx), "mistakes.")**: This prints the total number of mistakes made by the model.
- **print("Indices:", mistake_idx)**: This prints the indices of the instances where mistakes occurred.

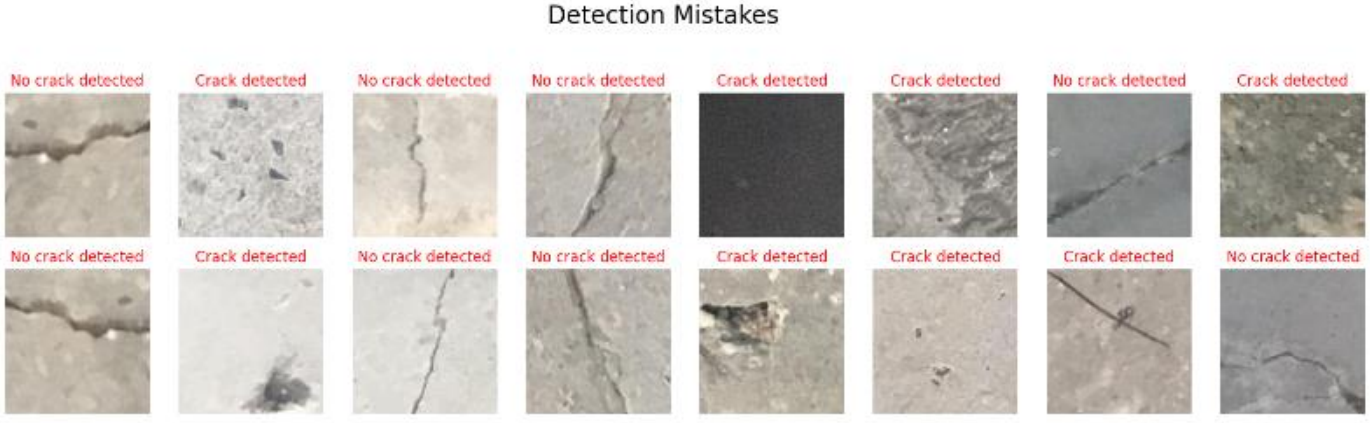
[30]:
```
plt.figure(figsize=(20, 10))

for i, idx in enumerate(mistake_idx):

    # Get batch number and image number (batch of 32 images)
    batch = idx // 32
    image = idx % 32

    plt.subplot(4, 8, i+1)
    plt.imshow(test_images[batch][0][image])
    plt.title("No crack detected" if y_pred[idx] == 0 else "Crack detected", color='red')
    plt.axis('off')

plt.suptitle("Detection Mistakes", fontsize=20)
plt.show()
```



Explanation:

- **plt.figure(figsize=(20, 10))**: Creates a figure with a specified size (20 inches by 10 inches) for the grid of images.
- **for i, idx in enumerate(mistake_idx):**: Iterates over the indices of instances where mistakes were made.
- **batch = idx // 32**: Calculates the batch number from the index.
- **image = idx % 32**: Calculates the image number within the batch.
- **plt.subplot(4, 8, i+1)**: Creates a subplot in a 4x8 grid, with the current iteration index **i+1** specifying the position of the subplot.
- **plt.imshow(test_images[batch][0][image])**: Displays the image corresponding to the mistake.
- **plt.title("No crack detected" if y_pred[idx] == 0 else "Crack detected", color='red')**: Sets the title of the subplot based on the model's prediction. If the model predicts no crack (0), the title is "No crack detected" in red; otherwise, it's "Crack detected" in red.
- **plt.axis('off')**: Removes the axis labels and ticks for better visualization.
- **plt.suptitle("Detection Mistakes", fontsize=20)**: Sets the super title of the entire grid.
- **plt.show()**: Displays the entire grid of subplots showing instances where the model made mistakes in its predictions. This can be useful for understanding the nature of the errors and potential areas for improvement.

TOPICS IN SYLLABUS

1. NumPy:

- NumPy is a powerful numerical computing library in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is a fundamental package for scientific computing in Python and is widely used in various fields, including machine learning, data science, and engineering.

2. Pandas:

- Pandas is a data manipulation library in Python that provides high-performance, easy-to-use data structures, namely Series and DataFrame. A DataFrame is a two-dimensional table with labeled axes (rows and columns), allowing for efficient manipulation and analysis of structured data. Pandas is extensively used for tasks such as data cleaning, preprocessing, and exploration in data science and machine learning workflows.

3. Confusion Matrix:

- A confusion matrix is a table used to evaluate the performance of a classification model. It summarizes the results of a classification problem by presenting counts of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions. These metrics are useful for understanding the strengths and weaknesses of a model, particularly in binary or multiclass classification scenarios.

4. Accuracy Score:

- Accuracy is a metric that measures the overall correctness of a classification model. It is calculated as the ratio of correctly predicted instances (sum of TP and TN) to the total number of instances.

The accuracy score is expressed as a percentage, providing a quick and intuitive assessment of how well a model is performing across all classes. While accuracy is a commonly used metric, it may not be suitable for imbalanced datasets, where other metrics like precision, recall, or the F1 score may be more informative.

5.Classification Report:

- A classification report is a summary of a classification model's performance on a dataset. It includes metrics like precision (accuracy of positive predictions), recall (sensitivity or true positive rate), F1-score (harmonic mean of precision and recall), and support (number of actual occurrences). The report is generated for each class in a multi-class classification problem and helps assess the model's effectiveness in distinguishing between different classes.