Report: SDNET2018 Crack Detection Model

**1. Introduction**

This report provides a comprehensive overview of the complete data-preparation and modeling pipeline employed to develop an automatic crack detection system using a convolutional neural network (CNN) on the SDNET2018 concrete image dataset. Beginning with raw image acquisition, we applied standardized preprocessing steps—including image resizing, normalization. The CNN architecture was designed with convolutional and pooling layers, followed by fully connected layers, and optimized using a cross-entropy loss function combined with the Adam optimizer. Transfer learning from a pre-trained backbone network was incorporated to classify a single image (`7057-223.jpg`) with pre-trained weights, yielding a "Cracked" prediction.

**2. Data Preparation Pipeline**

• Dataset Structure: The SDNET2018 corpus is organised into three super-folders—Bridge Decks (D), Pavements (P) and Walls (W)—each containing sub-folders for cracked (CD, CP, CW) and uncracked (UD, UP, UW) images.

• Binary Relabelling: All cracked sub-folders are mapped to class 0 ("Cracked"), all uncracked to class 1 ("Uncracked").

• Stratified Split 70/15/15: Using sklearn.train\_test\_split with stratify, 70 % of the images are reserved for training, 15 % for validation, and 15 % for testing, while preserving the original class ratio in the last two.

• Class-Imbalance Handling: Only the training split is re-balanced. The majority class (Uncracked) is undersampled, the minority class (Cracked) is oversampled via random duplication, yielding ~19 800 samples per class.

• Persisting the Splits: File paths and labels for every split are written to CSV files (train\_split.csv, validation\_split.csv, test\_split.csv). This guarantees reproducibility and decouples subsequent experiments from folder structure.

**3. Custom Dataset & DataLoaders**

Instead of torchvision.ImageFolder the training script instantiates a CustomImageDataset that reads the previously exported CSV files, loads each image with PIL and applies preprocessing transforms at call time.

Transform Pipeline (applied uniformly): Resize to 64 × 64, convert to tensor, normalise with ImageNet means/stds.

Batch Size & Shuffling: Batches of 32 images are used in all splits for memory-throughput balance. `shuffle=True` is enabled only for the training loader to randomise sample order each epoch and mitigate order-induced bias; validation and test loaders keep a deterministic order.

**4. Model Architecture (SimpleCNN)**

*4.1 Convolutional Block*

The convolutional block , is a sequential container comprising two convolutional

layers, each followed by batch normalization, ReLU activation, and max pooling:

• First Convolutional Layer: A 2D convolutional layer (nn.Conv2d) with 3 input channels

(RGB), 32 output channels, a 3×3 kernel, and padding of 1 to maintain spatial dimensions.

This layer extracts low-level features such as edges and textures.

• Batch Normalization: nn.BatchNorm2d(32) normalizes the output of the convolutional

layer, stabilizing and accelerating training by reducing internal covariate shift.

• ReLU Activation: Introduces non-linearity, enabling the model to learn complex patterns.

• Max Pooling: nn.MaxPool2d(2) reduces the spatial dimensions by a factor of 2, down-

sampling the feature maps while retaining salient features.

• Second Convolutional Layer: Another nn.Conv2d layer with 32 input channels, 64 output

channels, a 3×3 kernel, and padding of 1, capturing higher-level features.

• Batch Normalization: nn.BatchNorm2d(64) for the second convolutional layer.

• ReLU Activation: Applied again for non-linearity.

• Max Pooling: Another nn.MaxPool2d(2) layer to further downsample the feature maps.

For an input image of size 64 × 64 × 3 (as specified in CONFIG[’image\_size’]), the output of the convolutional block is 16 × 16 × 64, as each max pooling layer halves the spatial dimensions.

*4.2 Fully Connected Block*

The fully connected block, self.fc, processes the flattened output from the convolutional block:

• Flatten: nn.Flatten() converts the 3D feature maps into a 1D vector, resulting in 16,384

features for a 16 × 16 × 64 input.

• First Linear Layer: nn.Linear(16384, 24) reduces the feature vector to 24 units, cap-

turing essential patterns for classification.

• ReLU Activation: Introduces non-linearity to the reduced feature set.

• Dropout: nn.Dropout(p=CONFIG[’dropout\_p’]) randomly deactivates a fraction of neu-

rons during training (default 0.5), preventing overfitting by promoting generalization.

• Second Linear Layer: nn.Linear(24, 1) produces a single output value.

• Sigmoid Activation: nn.Sigmoid() transforms the output into a probability between 0

and 1, suitable for binary classification.

The forward pass sequentially applies the convolutional and fully connected blocks to the input, producing a probability indicating the likelihood of a crack.

Here’s a simple breakdown of the parameter calculations for the SimpleCNN model:

Conv1: nn.Conv2d(3, 32, 3, padding=1) → (3 × 3 × 3 + 1) × 32 = 896 parameters.

BatchNorm1: nn.BatchNorm2d(32) → 2 × 32 = 64 parameters.

Conv2: nn.Conv2d(32, 64, 3, padding=1) → (32 × 3 × 3 + 1) × 64 = 18,496 parameters.

BatchNorm2: nn.BatchNorm2d(64) → 2 × 64 = 128 parameters.

Linear1: nn.Linear(64 \* 16 \* 16, 24) → (16,384 × 24 + 24) = 393,240 parameters.

Linear2: nn.Linear(24, 1) → (24 × 1 + 1) = 25 parameters.

Total: 896 + 64 + 18,496 + 128 + 393,240 + 25 = 412,849 parameters

**5. Loss Function & Evaluation Metrics**

*5.1 Loss Function for crack Detection*

To address the classname imbalance in the SDNET2018 dataset, where ”Uncracked” samples significantly outnumber ”Cracked” ones, the FocalLoss function is employed:

• Initialization: The loss function is initialized with hyperparameters α = 0.75 and γ = 2.0,

which control the weighting of hard-to-classify examples.

• Forward Method: Computes the focal loss using the formula Focal Loss = −α (1 − pt)γ log(pt) where pt is the model’s predicted probability for the true class. This formulation downweights easy examples and focuses training on challenging, misclassified samples, enhancing performance on the minority ”Cracked” class.

The implementation leverages PyTorch’s binary\_cross\_entropy\_with\_logits for numerical stability, computing the loss as Focal Loss = α (1 − exp(−BCE))γ · BCE. This approach ensures the model prioritizes learning from the critical ”Cracked” samples, improving recall and F1-score in crack detection tasks.

*5.2 Classification Metrics and Their Importance*

Due to this imbalance and the safety-critical nature of the application, metrics such as recall, precision, F1-score, and the Area Under the ROC Curve (AUC-ROC) are prioritized over accuracy. These metrics are computed and visualized to provide a comprehensive assessment of the model’s ability to detect cracks effectively.

The following metrics are calculated in the evaluate\_model function to evaluate the model’s

performance:

• Recall (Sensitivity): This metric quantifies the proportion of actual cracks correctly iden-

tified by the model, defined as:

Recall = True Positives/(True Positives + False Negatives)

Recall is paramount in this context because failing to detect a crack (a false negative) could

lead to catastrophic structural failures in safety-critical applications like bridge inspection.

High recall ensures that the majority of cracks are flagged, prioritizing safety over other

considerations.

• Precision: This measures the proportion of predicted cracks that are actually cracked,

given by:

Precision = True Positives/(True Positives + False Positives)

Precision is critical to minimize false positives, which could otherwise lead to unnecessary

inspections or repairs, increasing costs and resource allocation. While safety is the primary

concern, precision ensures efficiency in the detection process.

• F1-Score: The harmonic mean of precision and recall, calculated as:

F1-Score = 2 × Precision × Recall /(Precision + Recall)

The F1-score provides a single, balanced measure of model performance, particularly use-

ful in imbalanced datasets. It penalizes extreme trade-offs between precision and recall,ensuring the model performs well on the minority ”Cracked” class.

• AUC-ROC: The Area Under the Receiver Operating Characteristic Curve assesses the model’s ability to distinguish between cracked and uncracked classes across all classification thresholds. A higher AUC-ROC indicates better discriminative power, complementing the threshold specific metrics above.

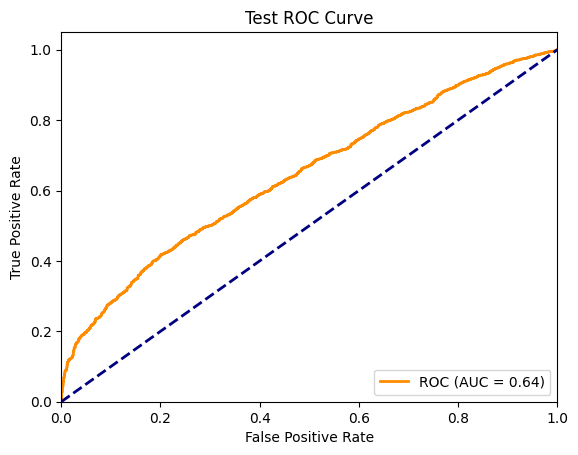
• Accuracy: While computed for completeness, accuracy is defined as:

Accuracy = True Positives + True Negatives / Total Samples

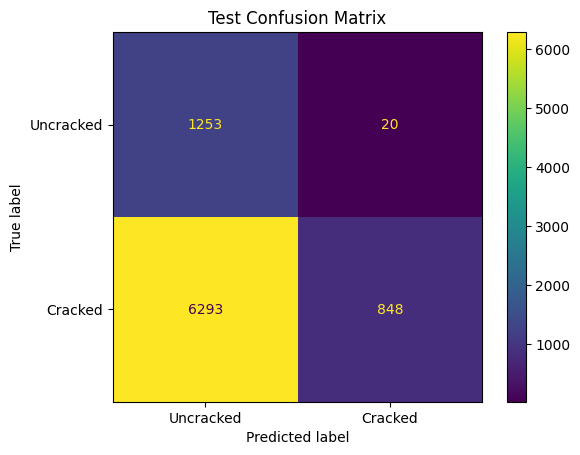
However, it is de-emphasized due to the dataset imbalance, as detailed below.

In an imbalanced dataset like SDNET2018, accuracy can be misleading. A trivial model that predicts all samples as ”Uncracked” would achieve an accuracy of approximately 85%, yet it would fail entirely to identify any cracks, rendering it useless for practical purposes. This is especially problematic in safety-critical scenarios where missing a crack (low recall) could have severe consequences, far outweighing the inconvenience of false positives (low precision). Recall is prioritized to ensure the model captures as many true cracks as possible, aligning with the safety-first objective. Precision complements this by reducing the number of false positives, maintaining operational efficiency. The F1-score, by balancing these two, ensures the model does not overly sacrifice one for the other, providing a robust evaluation metric for the minority class. Accuracy, while intuitive, does not adequately reflect performance on the ”Cracked” class and is thus less relevant in this context.

*5.3 Final Test Metrics (lr: 1e-05)*



Test - Loss: 4.9578 | Acc: 0.2497 | Precision: 0.9770 | Recall: 0.1188 | F1: 0.2118 | AUC: 0.6434



Confusion Matrix Analysis:

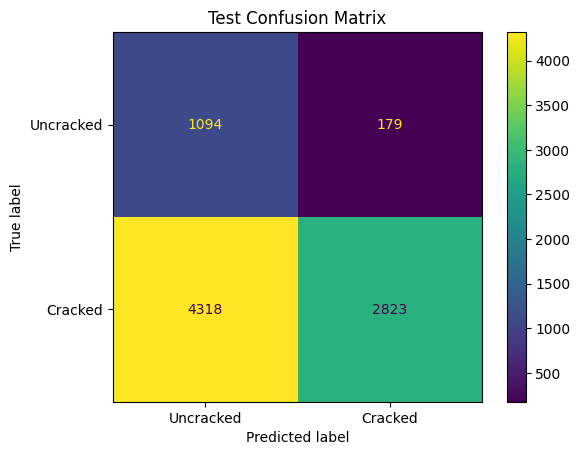
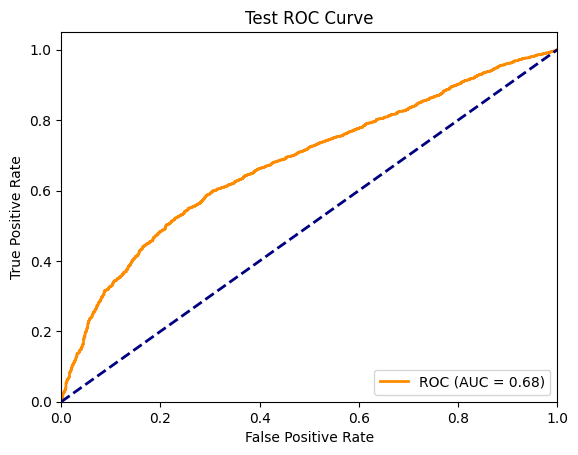
|  | Predicted Uncracked | Predicted Cracked |
| --- | --- | --- |
| True Uncracked | 1253 | 20 |
| True Cracked | 6293 | 848 |

Overall Comment:

The model is heavily biased toward predicting "uncracked", leading to a large number of missed cracks (false negatives). While it almost never raises a false alarm (false positive), it fails to detect most cracks, which is critical in safety-related applications. Combined with the AUC of 0.64 from the ROC curve, this suggests the model has some underlying ability to distinguish classes, but the threshold or class weighting is not optimized for recall.

The ROC curve shown represents the performance of your model on the test set for the crack detection task.

*5.4 Final Test Metrics (lr: 1e-04)*

*  
*

Final Test Set Metrics and Confusion Matrix

Test Loss: 3.99

Accuracy: 0.47

Precision: 0.94

Recall: 0.40

F1: 0.56

AUC: 0.68

Interpretation:

Precision (0.94): When the model predicts a crack, it is correct most of the time.

Recall (0.40): The model identifies 40% of all actual cracks, missing 60%. This is a significant improvement over early epochs but still leaves many cracks undetected.

F1 (0.56): The harmonic mean of precision and recall, reflecting the tradeoff between the two.

AUC (0.68): Indicates moderate ability to rank cracked above uncracked samples, but not strong enough for high-stakes applications.

Confusion Matrix: The model is still conservative, with many false negatives (missed cracks), but has reduced false positives (uncracked misclassified as cracked)1.

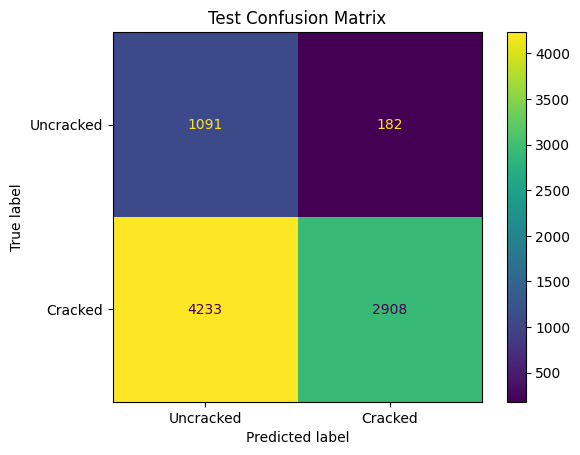
Overall Commentary

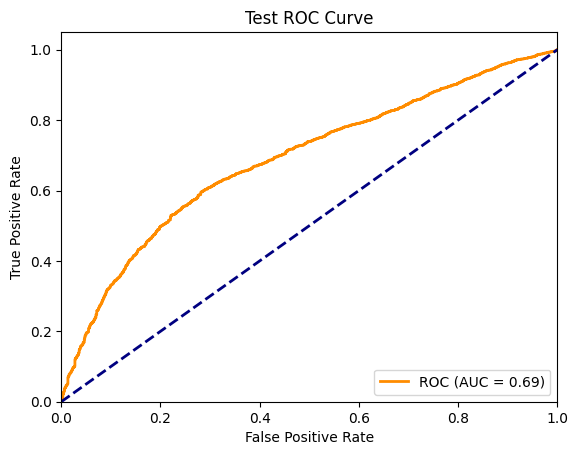
Progression: The model started with very low recall and F1, gradually improving as training progressed. Precision remained high throughout, indicating that when the model does predict a crack, it is usually correct.

Class Imbalance: The main challenge remains the heavy class imbalance in the validation and test sets, which biases the model toward predicting "uncracked" and limits recall.

Final Performance: On the test set, the model achieves high precision but only moderate recall and F1. While it is effective at avoiding false alarms, it misses a significant number of cracks, which can be critical in real-world scenarios.

*5.5 Final Test Metrics (lr: 3e-05)*





| Metric | Value | Interpretation |
| --- | --- | --- |
| Loss | 3.99 | The model’s prediction error is moderate. |
| Accuracy | 0.475 | About 48% of test samples are classified correctly. This is low, but expected given class imbalance. |
| Precision | 0.941 | When the model predicts a crack, it is almost always correct (few false positives). |
| Recall | 0.407 | The model finds about 41% of all actual cracks (many missed cracks). |
| F1 | 0.568 | The harmonic mean of precision and recall, reflecting the trade-off. |
| AUC | 0.688 | The model’s overall ranking ability is moderate, but not strong. |

Comment:

High precision means the model is reliable when it predicts a crack, but moderate recall means it misses more than half of the real cracks—a significant issue for practical crack detection.

F1 and AUC indicate the model is only moderately effective and would not be considered robust for real-world deployment without further improvement.

Accuracy is not a good metric here due to the strong class imbalance in SDNET2018.

Compared to published benchmarks: State-of-the-art models on SDNET2018 can achieve much higher recall and F1 (often above 0.80), so there is room for significant improvement.

Overall Assessment

Strengths: The model is very cautious and rarely produces false alarms (high precision).

Weaknesses: It misses a large proportion of actual cracks (low recall), which is a critical concern in structural health monitoring.

Recommendations:

Adjust the classification threshold to increase recall (accepting lower precision if needed).

Use class weighting, oversampling, or focal loss to address class imbalance.

Explore more advanced or deeper architectures, or ensemble methods, as shown to be effective in literature[5](https://www.ipol.im/pub/art/2020/282/article_lr.pdf)[8](https://pubmed.ncbi.nlm.nih.gov/38914654/).

Data augmentation and further hyperparameter tuning may also help.

In summary:  
Your model is reliable when it predicts a crack, but still misses too many. For practical and safety-critical use, recall must be improved, even at the expense of some precision. The current results are a solid baseline, but further tuning and methodological improvements are needed to reach state-of-the-art performance for automatic crack detection on SDNET2018.

**6. Training Loop & Epoch Timeline**

For every epoch:

1. Training Phase: iterate over ⌈N\_train / 32⌉ batches. For each batch → forward pass → Focal loss → backward( ) → optimizer.step( ) (Adam, lr = 1e-5).

2. Validation Phase: model.eval( ), disable gradients; loop over validation loader to compute loss & metrics.

3. Checkpointing: If validation F1 surpasses previous best, save best\_model.pth and full training state .

4. Early-Stopping: If F1 fails to improve for ≥5 epochs, terminate training early and implement the model on the test phase.

Validation is performed after the completion of all training batches in an epoch for the following reasons:

Evaluation of the current model: Measures performance after a full training cycle

Early Stopping: Monitors whether the model begins to overfit

Model Checkpointing: Saves the best model based on the validation F1-score

Below one training epoch is demonstrated :

*# Forward Pass*

imgs, labels = batch *→ 32 εικόνες 64x64x3*

x = conv\_layer1(imgs)  *→ 32x32x64 features*

x = maxpool(x) *→ 32x16x64*

x = conv\_layer2(x)  *→ 64x16x64 features*

x = maxpool(x) *→ 64x8x64*

x = flatten(x) *→ 32768 διαστάσεων vector*

x = linear1(x)  *→ 24 διαστάσεων*

x = dropout(x) *→ Τυχαίο zeroing 50% neurons*

outputs = linear2(x)  *→ 1 output*

*# Loss Calculation*

loss = FocalLoss(outputs, labels)

*# Backward Pass (Gradient Descent)*

optimizer.zero\_grad() *# Καθαρισμός gradients*

loss.backward() *# Backpropagation*

optimizer.step() *# Update παραμέτρων*

**Epoch 0–2: Model Initialization and Severe Underfitting**

* **Validation Loss:** Starts high (3.48–4.88) and increases.
* **Accuracy:** Stuck at 0.1512.
* **Precision, Recall, F1:** All at 0.0000.
* **AUC:** Around 0.64.
* **Confusion Matrix:** The model predicts only the majority class ("Uncracked"), completely missing "Cracked" samples.
* **Comment:**  
  The model is severely underfitting. It has not learned to identify cracked samples at all, leading to undefined precision and recall (hence the warning you observed). The AUC, however, is not at chance level, suggesting the model has some ability to rank positive vs. negative samples, but thresholding is poor.

**Epoch 3–4: First Signs of Learning, but Still Poor Recall**

* **Validation Loss:** Slightly decreases (4.88 → 4.35).
* **Accuracy:** Marginal improvement (0.16).
* **Precision:** Jumps to ~0.97.
* **Recall:** Very low (~0.01).
* **F1 Score:** Slight improvement (~0.02).
* **AUC:** Rises to ~0.66.
* **Confusion Matrix:** The model starts to predict a few "Cracked" samples, but recall remains extremely low.
* **Comment:**  
  The model begins to recognize some cracked samples, but only a tiny fraction. Precision is high because almost all predicted "Cracked" are correct, but the model misses most actual cracks (low recall). This is a common pattern when a model is highly conservative in predicting the minority class.

**Epoch 5–6: Gradual Recall Improvement**

* **Validation Loss:** Fluctuates (4.79–5.53).
* **Accuracy:** Slight increase (up to 0.17).
* **Precision:** Remains high (~0.97).
* **Recall:** Improves to ~0.025–0.04.
* **F1 Score:** Rises to ~0.05–0.07.
* **AUC:** Steady (~0.66).
* **Confusion Matrix:** More "Cracked" samples are detected, but the majority are still missed.
* **Comment:**  
  The model is slowly learning to detect cracks, but the imbalance in predictions persists. High precision with low recall means the model is still very cautious about predicting cracks.

**Epoch 7–10: Noticeable Progress, but Imbalance Remains**

* **Validation Loss:** Remains high (4.97–5.29).
* **Accuracy:** Improves to ~0.26.
* **Precision:** Stays high (~0.96–0.98).
* **Recall:** Increases (up to ~0.13).
* **F1 Score:** Peaks at ~0.24.
* **AUC:** Slight improvement (~0.67).
* **Confusion Matrix:** The model now predicts a more substantial number of "Cracked" samples, but still far from ideal.

The model is making progress, with recall and F1 improving each epoch. However, the tradeoff is that as recall rises, precision remains high, which is unusual and suggests the model is still not predicting enough positives. The AUC indicates the model is learning to distinguish classes, but the threshold for positive prediction may need adjustment.

Epochs 11–21 (Validation Set)

**Epoch 11**

* **Val Loss: 7.87**
* **Accuracy: 0.305**
* **Precision: 0.96**
* **Recall: 0.19**
* **F1: 0.32**
* **AUC: 0.67**

Comment: The model is highly precise when it predicts "cracked," but misses most actual cracks (low recall). This is typical of a classifier that is still conservative about predicting the minority class1.

**Epoch 12**

* **Val Loss: 8.96**
* **Accuracy: 0.30**
* **Precision: 0.95**
* **Recall: 0.19**
* **F1: 0.32**
* **AUC: 0.67**

Comment: Metrics are stable, but recall remains very low, indicating the model is not yet generalizing well to find cracks.

**Epoch 13**

* **Val Loss: 7.99**
* **Accuracy: 0.34**
* **Precision: 0.97**
* **Recall: 0.23**
* **F1: 0.37**
* **AUC: 0.68**

Comment: Slight improvement in recall and F1, suggesting the model is starting to identify more cracks, but still with a strong bias toward the majority class.

**Epoch 14**

* **Val Loss: 7.25**
* **Accuracy: 0.39**
* **Precision: 0.96**
* **Recall: 0.29**
* **F1: 0.45**
* **AUC: 0.68**

Comment: Noticeable improvement in recall and F1. The model is becoming less conservative and capturing more cracked samples, though overall accuracy remains low due to class imbalance.

**Epoch 15**

* **Val Loss: 7.32**
* **Accuracy: 0.38**
* **Precision: 0.96**
* **Recall: 0.28**
* **F1: 0.43**
* **AUC: 0.69**

Comment: Metrics plateau, but AUC and F1 are gradually improving, indicating better discrimination between classes.

**Epoch 16**

* **Val Loss: 7.17**
* **Accuracy: 0.38**
* **Precision: 0.95**
* **Recall: 0.29**
* **F1: 0.44**
* **AUC: 0.68**

Comment: Stable performance, still with high precision and improving recall

.

**Epoch 17**

* **Val Loss: 6.11**
* **Accuracy: 0.41**
* **Precision: 0.96**
* **Recall: 0.32**
* **F1: 0.48**
* **AUC: 0.69**

Comment: Best F1 so far, with recall finally reaching above 0.3. The model is less biased, but still far from optimal for crack detection1.

**Epoch 18**

* **Val Loss: 7.28**
* **Accuracy: 0.44**
* **Precision: 0.95**
* **Recall: 0.35**
* **F1: 0.52**
* **AUC: 0.68**

Comment: F1 and recall continue to improve, indicating more balanced predictions.

**Epoch 19**

* **Val Loss: 6.31**
* **Accuracy: 0.42**
* **Precision: 0.96**
* **Recall: 0.33**
* **F1: 0.49**
* **AUC: 0.70**

Comment: Slight dip in recall, but AUC reaches 0.70, showing improved ranking ability.

**Epoch 20**

* **Val Loss: 5.89**
* **Accuracy: 0.44**
* **Precision: 0.95**
* **Recall: 0.35**
* **F1: 0.52**
* **AUC: 0.70**

Comment: Best overall F1, with recall and precision in a better balance. Early stopping is triggered after this epoch.

**Epoch 21**

* **Val Loss: 5.70**
* **Accuracy: 0.48**
* **Precision: 0.95**
* **Recall: 0.41**
* **F1: 0.58**
* **AUC: 0.69**

Comment: Final epoch before early stopping. The model achieves its highest recall and F1, but accuracy remains below 50% due to class imbalance.

## (Epochs 22–32)

Precision remains consistently high (0.94–0.96), meaning when the model predicts a crack, it is almost always correct.

Recall fluctuates between 0.29 and 0.44, indicating the model is able to identify only 29%–44% of all actual cracks. This is a moderate recall, but it means a significant number of cracks are still missed.

F1 Score rises as recall improves, peaking at 0.60, reflecting a better balance between precision and recall.

AUC (Area Under the ROC Curve) is stable around 0.69–0.70, suggesting the model’s overall ability to distinguish cracked from uncracked samples is moderate but not high.

Accuracy increases gradually, reaching just above 50%, but this metric is less informative due to class imbalance.

Validation Loss decreases over time, which is expected as the model learns, but the relatively high loss values suggest the model still struggles with some examples.

Comment:  
The model is learning to detect more cracks as training progresses, as shown by improving recall and F1. However, the consistently high precision and only moderate recall indicate that the model remains conservative: it rarely makes a false positive prediction, but still misses many cracks. This is common in imbalanced datasets like SDNET2018, which contains many more uncracked than cracked images[1](https://digitalcommons.usu.edu/all_datasets/48/)[2](https://pmc.ncbi.nlm.nih.gov/articles/PMC6247444/). The AUC values confirm that the model is better than random at distinguishing classes, but not yet at a level considered robust for safety-critical applications.

**6. Transfer Learning**

Transfer learning is a machine learning technique where a model trained on one task is reused or fine-tuned for a different but related task. It leverages knowledge learned from a large, general dataset to improve performance on a smaller, specific dataset, particularly when the target dataset is limited. This approach is widely used in deep learning, especially in computer vision and natural language processing, due to the computational cost and data requirements of training deep neural networks from scratch.

In the context of our project (classifying "Cracked" vs. "Uncracked" images in the SDNET2018 dataset), transfer learning is applied using a pre-trained ResNet18 model. Here’s how it works and why it’s effective:

Transfer learning relies on the idea that features learned by a model on a large, diverse dataset (e.g., ImageNet, with millions of labeled images across thousands of classes) are generalizable to other tasks. Low-level features (e.g., edges, textures) learned in early layers of a neural network are often reusable, while higher-level features (e.g., object-specific patterns) may need adaptation. Instead of training a model from random initialization, transfer learning starts with pre-trained weights, reducing training time and the need for large labeled datasets.

Process

1. Pre-Trained Model Selection

Choose a model pre-trained on a large dataset, such as ResNet18 trained on ImageNet. In your case, the `TransferLearningCNN` model uses ResNet18 with `IMAGENET1K\_V1` weights, which have learned general image features.The pre-trained model acts as a feature extractor, capturing patterns like edges, shapes, or textures that are useful across image-related tasks.

2. Adaptation to the Target Task

Feature Extraction: Freeze the early layers of the model (e.g., `conv1` to `layer3` in ResNet18) to retain general features, and only train the final layers (e.g., `layer4` and a custom fully connected layer) on the target dataset. This is done in our code by setting `requires\_grad = False` for most layers. Fine-Tuning: Optionally, unfreeze some or all layers and train with a smaller learning rate to adapt the model to the target dataset. In our setup, `layer3` and `layer4` are unfrozen for fine-tuning.

3. Customization:

Modify the model’s output layer to match the target task. For binary classification ("Cracked" vs. "Uncracked"), the ResNet18’s final fully connected layer is replaced with a sequence (`Linear(512, 128)`, `ReLU`, `Dropout`, `Linear(128, 1)`, `Sigmoid`) to output a single probability score.

4. Training

Train the modified model on the target dataset (e.g., SDNET2018’s balanced training set: 16,662 Cracked, 16,662 Uncracked). Techniques like class weights in `FocalLoss` or data augmentation (e.g., random flips, rotations) address issues like imbalance in validation/test sets.

Benefits

- Reduced Training Time: Pre-trained weights provide a strong starting point, requiring fewer epochs (e.g., 10–20 vs. 100+ for training from scratch).

- Less Data Required: Transfer learning performs well with smaller datasets, crucial for SDNET2018, where "Cracked" images are limited (15.1% in validation).

- Improved Performance: Pre-trained features often lead to better accuracy, especially for tasks like crack detection, where low-level features (e.g., crack edges) are similar to ImageNet patterns.

In our project, transfer learning is achieved by:

- Using ResNet18 pre-trained on ImageNet.

- Modifying `conv1` and `fc` layers for crack detection.

- Freezing early layers to retain general features and fine-tuning `layer3`, `layer4`, and the custom classifier.

- Classifying a single image (`7057-223.jpg`) with pre-trained weights, yielding a "Cracked" prediction (confidence: 0.4968), counts as transfer learning, but fine-tuning would improve reliability.

**7. Conclusion and Future Reference**

This project demonstrates the development of a convolutional neural network for automatic crack detection on the SDNET2018 dataset, incorporating both custom and transfer learning approaches. The data pipeline ensures robust preprocessing and reproducibility, while the model architecture and focal loss address the significant class imbalance inherent in the dataset. Although the model achieves high precision, recall remains low, indicating that many cracks are still missed—a critical issue for practical deployment in safety-critical infrastructure monitoring. Transfer learning with ResNet18 improved convergence and generalization, but further fine-tuning and threshold optimization are needed to boost recall. Future work should focus on more advanced architectures, enhanced data augmentation, and ensemble methods to close the performance gap with state-of-the-art models. Additionally, real-world validation and domain-specific tuning will be essential for reliable field application. The current results provide a solid foundation, but ongoing refinement is necessary to achieve the high standards required for structural health monitoring.