Classification, Segmentation and GAN implementation on Concrete Crack Images

CS 490 : Research and Development project

by

Hrithik Mhatre

(Roll No. 210040092)

Under the Supervision of

Prof. Abir De and Prof. Alankar Alankar

Department of Civil Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Mumbai - 400076, India

April, 2024

Github : https://github.com/hrithikM86/CS-490-RND

Chapter 1

Introduction

In the realm of civil engineering, detecting concrete cracks is a pivotal task with far-reaching implications for structural integrity and safety. The diversity of crack types coupled with the challenges posed by varying environmental conditions necessitates robust detection methods. This project endeavors to address these challenges through a multifaceted approach encompassing classification, segmentation, and Generative Adversarial Network (GAN) implementation on concrete crack images.

The primary objective of this project is to develop an efficient crack detection algorithm capable of identifying diverse crack patterns encountered in real-world scenarios. While existing datasets provide a foundation, they often fail to encompass the full spectrum of cracks observed in laboratory settings. Consequently, a secondary aim of this project is to leverage GAN technology to generate a more comprehensive dataset reflective of the nuanced crack types encountered in practical applications.

Beyond mere crack detection, precise localization of cracks within concrete structures is essential for targeted remediation efforts. To this end, the project also focuses on developing a segmentation algorithm capable of accurately delineating the extent and location of cracks within images. By providing detailed spatial information, this segmentation approach enhances the utility of crack detection systems for engineers and maintenance personnel.

By integrating classification, segmentation, and GAN technologies, this project aims to advance the state-of-the-art in concrete crack detection.

Chapter 2

Concrete Crack Detection Utilizing ResNet50

2.1 Dataset

The dataset utilized for this analysis was sourced from oluwaseunad's repository, specifically the ‘concrete-and-pavement-crack-images’ dataset. This dataset comprises a total of 30,000 images categorized into two distinct classes: cracked and non-cracked concrete. Each image in the dataset has a resolution of 227 x 227 pixels and is formatted in RGB JPEG.

2.2 Data Generator and Image Preprocessing:

1. Data Augmentation : The function `gen` is designed to generate data generators for training and testing datasets. For the training dataset, an ImageDataGenerator object (`train_datagen`) is created with data augmentation capabilities. Data augmentation helps in increasing the diversity of training samples, thereby improving the robustness and generalization of the model.
2. The preprocess_input function from tensorflow.keras.applications.resnet50 is then applied to the dataset within this function.
3. This preprocessing standardizes the array, converting images from RGB to BGR and centering each color channel with respect to the ImageNet dataset without scaling, resulting in float32 type arrays.
4. Validation Split : Within the training data generator, a validation split of 20% (`validation_split=0.2`) is specified. This means that 20% of the training data will be used for validation during model training.
5. Testing Data Generator : Another ImageDataGenerator object (`test_datagen`) is created for the testing dataset. This generator is used solely for testing and does not include data augmentation since it is not necessary for testing purposes.
6. Flow from DataFrame : The training, validation, and testing data generators (`train_gen`, `valid_gen`, and `test_gen`, respectively) are generated using the `flow_from_dataframe` method. This method allows data to be loaded from a DataFrame and provides flexibility in specifying various parameters.
7. DataFrame Configuration : The DataFrame (`train` and `test`) contains information about the file paths (`x_col`) and corresponding labels (`y_col`). This information is used by the data generators to load images and their respective labels.
8. Image Configuration : The images are resized to a target size of (100, 100) pixels (`target_size=(100,100)`) to ensure uniformity. For the testing generator, the color mode is explicitly set to RGB (`color_mode='rgb'`) to ensure consistency.
9. Class Mode : Since this is a categorical classification task (two classes: cracked and non-cracked), the class mode is set to 'categorical' for all data generators.
10. Batch Size and Shuffling : The batch size is set to 64 images per batch (`batch_size=64`). For the training generator, shuffling is enabled (`shuffle=True`) to ensure randomness in each batch during training, while for the validation and testing generators, shuffling is disabled (`shuffle=False`) to maintain consistency and enable proper evaluation.
11. Return : The function returns three data generators: `train_gen`, `valid_gen`, and `test_gen`, which encapsulate the training, validation, and testing datasets, respectively. These generators can then be used for training and evaluating machine learning models.

2.3 Model creation

1. Model Initialization : The function `func` initializes a neural network model using a pre-trained architecture specified by the parameter `name_model`, which is assumed to be ResNet50 in this context. The model is configured to accept input images of size (100, 100, 3), which corresponds to a height and width of 100 pixels and three color channels (RGB).
2. Pre-trained Weights : The pre-trained weights from the ImageNet dataset are utilized by setting `weights='imagenet'`. This allows the model to leverage the knowledge learned from millions of images in the ImageNet database, which helps in feature extraction.
3. Feature Extraction : The top layer of the pre-trained model is removed (`include_top=False`) since we are adding our custom classification layers on top. This enables feature extraction from the input images.
4. Freezing Layers : The layers of the pre-trained ResNet50 model are frozen (`pre_model.trainable = False`) to prevent their weights from being updated during training. This is beneficial for two reasons: it helps in retaining the learned features from ImageNet, and it speeds up the training process by reducing the number of parameters that need to be updated.
5. Custom Classification Layers : Two fully connected (Dense) layers with ReLU activation functions are added on top of the ResNet50 model. These layers serve as the custom classifier for the specific task at hand. The output layer consists of two units with a softmax activation function, suitable for binary classification tasks.
6. Model Compilation : The model is compiled using categorical cross-entropy loss, Adam optimizer, and accuracy as the evaluation metric. Categorical cross-entropy is suitable for multi-class classification tasks, but in this case, it is used for binary classification by treating it as a two-class problem.
7. Early Stopping Callback : The function sets up an EarlyStopping callback, which monitors the validation loss during training. If the validation loss does not decrease for a certain number of epochs (`patience=1`), training will be halted early to prevent overfitting.
8. Return : The function returns two objects: the compiled model (`model`) and the EarlyStopping callback (`my_callbacks`). These objects can then be used for training the model on the provided data.

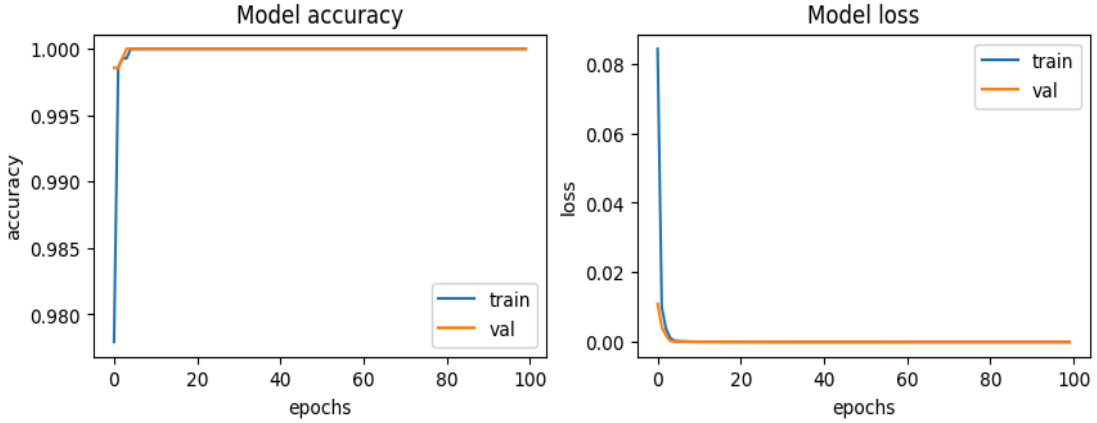
2.4 Training the model

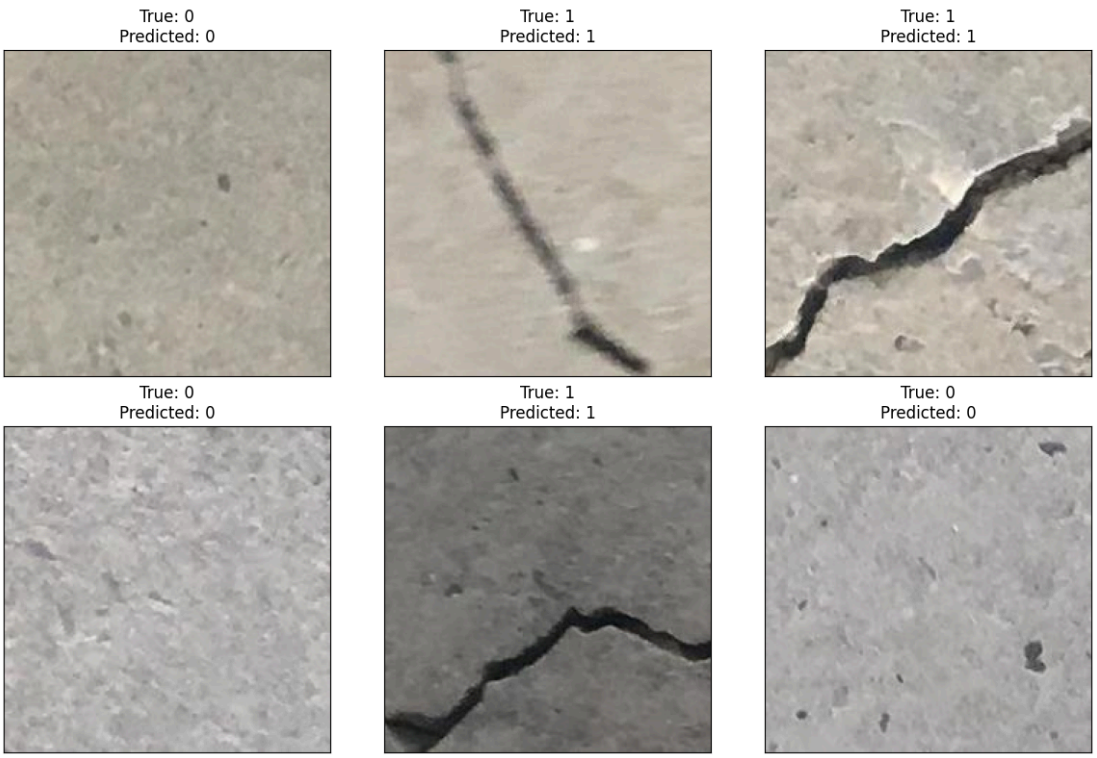
1. The model is trained for 100 epoch using model.fit() function with early stopping criteria as mentioned above

2.5 Results

2.5.1 Performance Evaluation on the Same Dataset

	precision	recall	f1-score	support
0	0.99	1.00	1.00	295
1	1.00	0.99	1.00	307
accuracy			1.00	602
macro avg	1.00	1.00	1.00	602
weighted avg	1.00	1.00	1.00	602





2.5.2 Evaluating Model Performance on a New Dataset Using Pre-Trained Weights

Dataset : hesighsrikar/concrete-crack-images-for-classification

Test Accuracy : 99.33%
F1 Score : 0.99
ROC AUC : 0.99

Dataset : aniruddhsharma/structural-defects-network-concrete-crack-images

Test Accuracy : 82.02%
F1 Score : 0.77
ROC AUC : 0.59

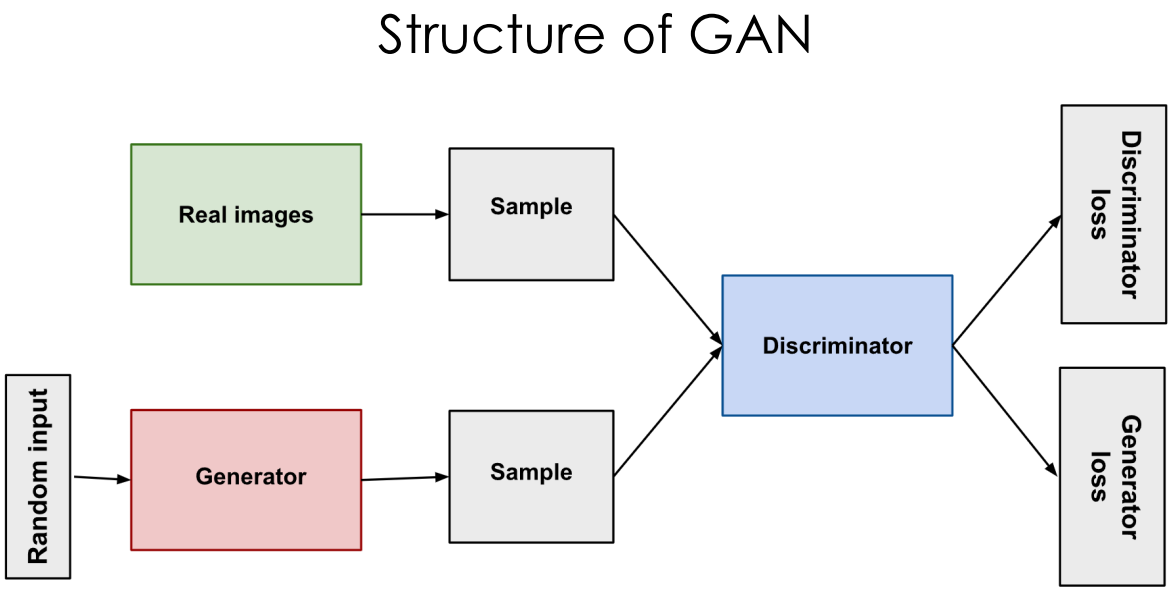
Given the presence of numerous misclassified images within this dataset, it is anticipated that the model's performance on this particular dataset may be adversely affected.

Chapter 3

Generative Adversarial Network (GAN)

3.1 Introduction to GAN

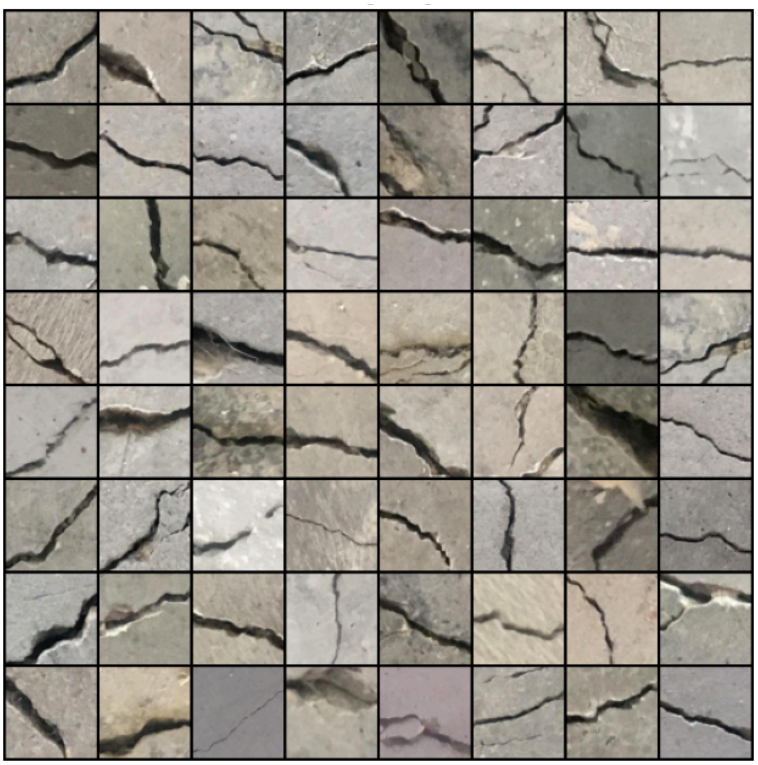
1. A generative adversarial network (GAN) is a machine learning (ML) model in which two neural networks compete with each other by using deep learning methods to become more accurate in their predictions
2. The two neural networks that make up a GAN are referred to as the generator and the discriminator
3. The generator is a convolutional neural network and the discriminator is a deconvolutional neural network
4. The goal of the generator is to artificially manufacture outputs that could easily be mistaken for real data
5. The goal of the discriminator is to identify which of the outputs it receives have been artificially created



Source : https://developers.google.com/machine-learning/gan/gan_structure

3.2 Dataset

1. The dataset used was thesighsrikar/concrete-crack-images-for-classification
2. The dataset comprises 40,000 images, evenly split between cracked and uncracked concrete images
3. However, due to computational constraints, only 10,000 cracked concrete images were utilized for training the model
4. The dataset contains 227 x 227 pixels images with RGB channels



3.3 Hyper Parameters

1. Number of workers (workers) : 2
2. Batch size (image_size) : 64
3. Image spatial size (image_size) : 64x64
4. Number of channels (nc) : 3
5. Latent vector size (nz) : 100
6. Generator feature map size (ngf) : 64
7. Discriminator feature map size (ndf) : 64
8. Number of epochs (num_epochs) : 60
9. Learning rate (lr) : 0.0002
10. Adam optimizer hyperparameter (beta1) : 0.5
11. Number of GPUs available (ngpu) :1

3.4 Image Preprocessing

Images underwent the following transformation steps :

1. Resized to a square of dimension ‘image_size’
2. Center-cropped to ensure uniformity at ‘image_size’
3. Converted to tensor format for compatibility with PyTorch
4. Normalized using mean (0.5, 0.5, 0.5) and standard deviation (0.5, 0.5, 0.5) to standardize pixel values

3.5 Data loader

A Data Loader is a utility that helps manage the process of loading and processing data for training or inference tasks. It's particularly useful when dealing with large datasets that can't fit entirely into memory

1. torch.utils.data.DataLoader was used to create a Data Loader
2. Hyper Parameters of torch.utils.data.DataLoader
    a. Dataset
    b. Batch size
    c. Shuffle = True
    d. Num_workers

3.6 Weights

1. The function ‘weights_init’ initializes the weights of convolutional layers (‘Conv’) with values drawn from a normal distribution with mean 0 and standard deviation 0.02
2. For batch normalization layers (BatchNorm), it sets the weights to 1 and biases to 0, also using normal distribution for weights

3.7 Generator architecture

1. Initialization: The ‘__init__’ method initializes the Generator class. It takes `ngpu` as input, which represents the number of GPUs available for training. `ngpu` is stored as an attribute of the class (‘self.ngpu’). Inside the initialization, a sequence of convolutional layers and batch normalization layers are defined within ‘self.main’
2. Layers :
    a. ConvTranspose2d Layers: Convolutional transpose layers are used for upsampling the input. They perform the opposite operation of convolutional layers. The parameters of these layers determine the size of the output feature maps
    b. The first ‘ConvTranspose2d’ layer takes an input ‘nz’ (size of the input noise vector ‘Z’) and outputs ‘ngf x 8’ feature maps. It has a kernel size of 4, a stride of 1, and no padding
    c. The subsequent ‘ConvTranspose2d’ layers gradually decrease the number of feature maps while increasing the spatial dimensions of the output.
    d. The second layer upsamples to (ngf x 4) feature maps with a kernel size of 4, a stride of 2, and padding of 1
    e. The third layer upsamples to (ngf x 2) feature maps with similar parameters
    f. The fourth layer upsamples to `(ngf feature maps with similar parameters
    g. The fifth and final layer upsamples to the number of channels ‘nc’ (size of the image's channel) using a kernel size of 4, a stride of 2, and padding of 1.
3. Batch Normalization Layers : Batch normalization layers are added after each convolutional transpose layer to stabilize and accelerate the training process. They normalize the input to a layer over a mini-batch
4. Activation Function : ReLU (Rectified Linear Unit) activation functions are applied after each batch normalization layer, introducing non-linearity to the network
5. Tanh Activation Function : The last layer uses a `Tanh` activation function to squash the pixel values to the range [-1, 1], which is typical for image generation tasks
6. Forward Pass :
    a. The ‘forward’ method defines the forward pass of the network
    b. It takes an `input` tensor and passes it through the layers defined in ‘self.main’
    c. The output of the last layer is returned
7. Output Size :

a. The final output size after passing through all layers is ‘(nc) x 64 x 64’, where ‘nc’ represents the number of channels in the generated image, and ‘64 x 64’ represents the spatial dimensions of the generated image

3.8 Discriminator Architecture

1. Initialization :
    a. The ‘__init__’ method initializes the ‘Discriminator’ class. Like the ‘Generator’ it takes ‘`ngpu’ as input, representing the number of GPUs available for training
    b. Inside the initialization, a sequence of convolutional layers, batch normalization layers, and leaky ReLU activation functions are defined within ‘self.main’
2. Layers :
    a. Conv2d Layers : These are standard 2D convolutional layers responsible for extracting features from the input images
    b. The first ‘Conv2d’ layer takes input images of size ‘(nc) x 64 x 64’ (where ‘nc’ is the number of channels) and outputs ‘ndf’ feature maps. It uses a kernel size of 4, a stride of 2, and padding of 1
    c. Subsequent ‘Conv2d’ layers gradually increase the number of feature maps while decreasing the spatial dimensions of the output
    d. The second layer outputs ‘ndf x 2’ feature maps with similar parameters
    e. The third layer outputs ‘ndf x 4’ feature maps with similar parameters
    f. The fourth layer outputs ‘ndf x 8’ feature maps with similar parameters
    g. The fifth and final convolutional layer reduces the spatial dimensions to ‘4 x 4’ while producing a single output channel, aiming to discriminate between real and fake images
3. Batch Normalization Layers : These layers stabilize and accelerate the training process by normalizing the inputs to each layer over a mini-batch
4. LeakyReLU Activation Function : LeakyReLU activation functions introduce non-linearity to the network while preventing the vanishing gradient problem. They are applied after each convolutional layer
5. Sigmoid Activation Function : The final layer uses a sigmoid activation function to output a probability indicating the authenticity of the input image
6. Forward Pass :
    a. The ‘forward’ method defines the forward pass of the network.
    b. It takes an ‘input’ tensor representing an image and passes it through the layers defined in ‘self.main’
    c. The output is the discriminator's prediction regarding the authenticity of the input image
7. Output Size :
    a. The output size after passing through all layers is a single scalar value, indicating the probability that the input image is real

3.9 Optimiser

1. ADAM : Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problems involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the ‘gradient descent with momentum’ algorithm and the ‘RMSP’ algorithm.
2. Hyper Parameters
    a. Parameters of model

b. Learning rate
c. Beta1
d. Beta2

3.10 Loss Function

3.10.1 Discriminator loss

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

1. While the discriminator is trained, it classifies both the real data and the fake data from the generator
2. It penalizes itself for misclassifying a real instance as fake, or a fake instance (created by the generator) as real, by maximizing the below function
3. log(D(x)) refers to the probability that the generator is correctly classifying the real image, maximizing log(1-D(G(z))) would help it to correctly label the fake image that comes from the generator

3.10.2 Generator Loss

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

1. While the generator is trained, it samples random noise and produces an output from that noise.
2. The output then goes through the discriminator and gets classified as either “Real” or “Fake” based on the ability of the discriminator to tell one from the other
3. The generator loss is then calculated from the discriminator’s classification – it gets rewarded if it successfully fools the discriminator, and gets penalized otherwise

3.11 Training Loop

1. Initialization : The training loop is run for a certain number of epochs (‘num_epochs’). For each epoch, the loop iterates over batches of data loaded by the data loader (‘dataloader’)
2. Discriminator Training :
    a. Real Data: For each batch of real data, the discriminator (‘netD’) is trained to correctly classify them as real. The discriminator's weights are updated using the gradients obtained from the loss calculated based on its classification of real data.
    b. Fake Data: Next, the discriminator is trained on a batch of fake data generated by the generator (‘netG’). The generator's output is detached from the graph to prevent gradients from flowing back into it during this step. The discriminator's weights are updated using the gradients obtained from the loss calculated based on its classification of fake data.
    c. Overall Discriminator Loss: The total discriminator loss is calculated as the

sum of the losses on real and fake data.

3. Generator Training :
    a. The generator (‘netG’) is trained to generate data that fools the discriminator into classifying it as real. Its weights are updated based on the gradients of the loss obtained from the discriminator's classification of the generated data.
4. Output and Logging :
    a. Training statistics such as discriminator and generator losses, the average output of the discriminator for real data (‘D(x)’), and for fake data (‘D(G(z))’) are printed periodically.
    b. Losses for both the discriminator and generator are saved for later plotting.
    c. Optionally, at certain intervals, generated images from a fixed noise vector are saved for visualization.

3.12 Results