ImageDataGenerator

Thursday, July 13, 2023 11:54 AM

Image augmentation technique which makes our training robust with many data where original image is augmented multiple ways to increase the size of training set

ImageDataGenerator: It helps to artificially increase the size and diversity of the dataset which helps to improve the ability of models to generalize.
Main goal is to increase the generalizability of the model. Perform well on testing data although perform slightly worse on our training data.

How it works:

- Accepting a batch of images used for training and taking this batch and applying a series of random transformations to each image in the batch(including random rotation, resizing, shearing)
- Replacing the original batch with the new, randomly transformed batched
- Train the CNN on this randomly transformed batch
- Accepts the original data, randomly transforms It and returns only the new transformed data.
- Why? Because our neural network is seeing new, slightly modified of the input data, the network is able to learn more robust features.
- Our data may follow perfect random distribution on the real data but, in real world its difficult to follow such a nice and neat distribution.
- TO increase the generalizability of our classifier we may first randomly jitter points along with the distribution by adding some random values.
- A model trained on modified and augmented data is more likely to generalize to example data points not included in the training set.

Three types of Data Augmentation:

- Dataset Generation and expanding the existing dataset(Less common): Generate many images from the sample we have applying different transformations and saving back to disk.
- In place/ On the fly data augmentation(most common): ImageDataGenerator does to create new transformed batch of images. Here, our network when trained will see new variations of our data at each and every epoch. It only returns the randomly transformed data. And it is on the fly as it generates batch of images in training time.
- Combining dataset generation and in-place augmentation: combine both.used in self driving car

Why VGG16?
Deep network with 16 layers, it has large capacity to learn complex features and patterns rom images. Pre trained version of Vgg16 has been trained on massive dataset like ImageNet, making them adept to learning generic features from images. It has demonstrated strong performance on classic image recognition tasks, including image classification and object detection.

Model Explanation:

CODE: vgg = VGG16(input_shape=(224,224,3), include_top= False)

When set to False, the final fully connected layers (often used for classification) are not included. This is useful when you want to use the network as a feature extractor for another task, such as fine-tuning or transfer learning. It exclude top layers.

CODE: layer.trainable = False
Inside the loop, the trainable attribute of each layer is set to False. This means that the parameters (weights and biases) of these layers will not be updated during the training process. Essentially, this makes the layers "frozen" and ensures that their pre-trained features are preserved.

CODE: x = Flatten()(vgg.output)

Here, a Flatten layer is applied to the output of the VGG16 model. The purpose of this layer is to convert the multi-dimensional feature maps from the previous convolutional layers into a one-dimensional feature vector. This is necessary when transitioning from convolutional layers to fully connected layers.

CODE: x = Dense(units=2, activation='sigmoid', name='predictions')(x)
A Dense layer, also known as a fully connected layer, is added. It will be connected to the flattened output from the previous layer.

- units=2: This specifies the number of neurons (or units) in this fully connected layer. In this case, there are 2 units, which suggests that the model is designed for a binary classification task (e.g., classifying between two categories like "covid" and "normal").
- activation='sigmoid': The activation function used for the neurons in this layer is the sigmoid activation function. The sigmoid function squashes the output between 0 and 1, which makes it suitable for binary classification where each output represents the probability of a certain class.
- name='predictions': This gives a name to this layer, which can be useful for later referencing or visualization.

CODE: Model = Model(vgg.input, x)

- vgg.input: This refers to the input layer of the original VGG16 model, which will be the input to the new extended model.
- x: This refers to the output of the layers you've added on top of the VGG16 model. This is the output of the Dense layer you defined earlier.
  The resulting model is the complete architecture that includes the pre-trained VGG16 layers along with the additional layers you've added for your specific task.
  In summary, the provided code extends the VGG16 model by adding a Flatten layer to convert the convolutional features into a one-dimensional vector, followed by a Dense layer for classification. The resulting model represents the complete architecture for your specific classification task, where the pre-trained VGG16 layers act as feature extractors and the new layers perform the final classification.

CODE: model.compile(optimizer='adam', loss='binary_crossentropy',

metrics=['accuracy'])

- Optimizer (optimizer='adam'): The optimizer is a crucial component of the training process. It determines how the model's weights are updated during training to minimize the loss function. 'adam' refers to the Adam optimizer, which is a popular choice for optimizing deep neural networks. Adam combines the benefits of two other optimization algorithms, Adagrad and RMSprop, and adapts the learning rates for each parameter based on their past gradients.
- Loss Function (loss='binary_crossentropy'): The loss function quantifies the difference between the predicted values and the actual target values. 'binary_crossentropy' is a common loss function used for binary classification problems, where the model predicts probabilities for two classes (e.g., 0 and 1). The binary cross-entropy loss measures the dissimilarity between the predicted probabilities and the true binary labels, encouraging the model to output higher probabilities for the correct class.
- Metrics (metrics=['accuracy']): Metrics are used to evaluate the performance of the model during training and validation. 'accuracy' is a metric that calculates the ratio of correctly predicted samples to the total number of samples in a batch. It's a common metric for classification problems and provides insight into the model's ability to correctly classify data points.
- When you compile the model using the compile function, you prepare the model for training. The specified optimizer, loss function, and metrics will guide the training process as the model updates its weights to minimize the loss function.
- After compiling, you can start training the model using your training data. During training, the optimizer will adjust the model's weights based on the calculated loss, with the goal of improving the model's performance on the specified metric (in this case, accuracy).

es = EarlyStopping(monitor= "val_accuracy" , min_delta= 0.01, patience= 3, verbose=1)
mc = ModelCheckpoint(filepath="bestmodel.h5", monitor="val_accuracy", verbose=1, save_best_only= True)
EarlyStopping Callback: EarlyStopping is a callback that monitors a specified metric during training and stops training early if the monitored metric stops improving. Here's what each parameter does:

- monitor="val_accuracy": The metric to monitor during training. In this case, it's monitoring the validation accuracy.
- min_delta=0.01: The minimum change in the monitored metric that qualifies as an improvement. If the change is smaller than this value, it's not considered an improvement.
- patience=3: The number of epochs with no improvement after which training will be stopped. In this case, if the validation accuracy does not improve for 3 consecutive epochs, training will be halted.
- verbose=1: It controls the verbosity of the printed output. A value of 1 means that it will print messages when early stopping is triggered.

1. ModelCheckpoint Callback: ModelCheckpoint is a callback that saves the model's weights during training based on a specified condition. Here's what each parameter does:
    - filepath="bestmodel.h5": The file path to which the model weights will be saved. In this case, the model weights will be saved in a file named "bestmodel.h5".

- monitor="val_accuracy": The metric to monitor for determining whether to save the model. It will save the model weights if the validation accuracy improves.
- verbose=1: It controls the verbosity of the printed output. A value of 1 means that it will print messages when the model weights are saved.
- save_best_only=True: It specifies that only the best model weights (based on the monitored metric) will be saved. If set to True, the model weights will be saved only if the validation accuracy improves compared to the previous best.

These callbacks are very useful tools during the training process. EarlyStopping helps prevent overfitting and saves training time by stopping when the model's performance plateaus, and ModelCheckpoint saves the best model weights based on the validation accuracy, allowing you to restore the best model later. These callbacks can improve the efficiency and effectiveness of your model training process.

Heatmap:

- Inside the function, a new model (grad_model) is created to map the input image to both the activations of the specified last convolutional layer and the output predictions of the original model.
- Using TensorFlow's GradientTape, the gradient of the top predicted class (or the specified class using pred_index) with respect to the activations of the last convolutional layer is computed. This is essentially calculating how the predicted class output changes with respect to changes in the convolutional feature maps.
- The gradient is calculated for each channel (feature map) in the last convolutional layer output, and then the mean intensity of the gradient is computed across all channels. This results in a vector pooled_grads containing the importance of each channel.
- The importance values (pooled gradients) are used to create a heatmap by element-wise multiplication of each channel in the last convolutional layer output with the corresponding importance value. The heatmap represents the regions of the image that are most relevant to the predicted class.
- The heatmap is then normalized to have values between 0 and 1 by taking the element-wise maximum with 0 and dividing by the maximum value in the heatmap.
- Finally, the normalized heatmap is converted to a NumPy array and returned from the function.

About Grad CAM:
Gradient-weighted Class Activation Mapping (Grad-CAM) is a technique used for visualizing the regions of an image that are important for a neural network's prediction. It helps us understand which parts of the input image were influential in the model's decision-making process.

Here's an overview of the mathematics behind Grad-CAM:

- Select a Target Class: To generate a Grad-CAM heatmap for a specific class, we first need to choose the target class for which we want to understand the importance.
- Create a Saliency Map (Gradient): We calculate the gradient of the target class score with respect to the feature maps of the last convolutional layer. This gradient reflects how the output of the target class changes with respect to changes in the feature maps.
- Global Average Pooling: We compute the average gradient for each feature map by performing global average pooling. This gives us a vector of importance scores, where each score represents the average gradient intensity for a specific feature map.
- Weight the Feature Maps: We create a weighted combination of the feature maps by multiplying each feature map by its corresponding importance score. This highlights the regions of the image that had the strongest impact on the target class prediction.

- Summation: The weighted feature maps are summed to generate the Grad-CAM heatmap. This heatmap indicates which regions of the image contributed the most to the target class prediction.
- Rectification and Normalization: The values in the heatmap can be rectified (ReLU) to ensure that we only focus on positive contributions. Then, the heatmap is normalized to have values between 0 and 1 for visualization purposes.