Notes

1) **When working with image data**, we first scale the matrix of data. We can scale it manually, or we can use libraries. Then, we convert the matrix of image data into a 1D array by using the .reshape function of NumPy. Finally, we build a neural network with the 1D array.

**2) After the neural network is built**, we evaluate its performance on the same metrics that we use in machine learning.

3) The typical CNN architecture sequence involving convolutional layers, activation functions, and pooling layers is as follows:

Input Data -> Convolutional Layer -> Activation Function -> Pooling Layer ->

Stride

1) Stride is only applied to the pooling layer, not the convolutional layer. This is because in the convolutional layer, we want to extract as many primitive features as possible, while in the pooling layer, we only want high-level primitive features.

2) Using strides can make the deployment of a model faster, but it can also lead to information loss.

3) Using strides in convolutional neural networks (CNNs) can lead to a slight increase in the error rate due to the loss of information from the input data. However, strides can also speed up the training process and the processing speed of the application after deployment.

4) We use strides in the pooling layer only because we want to extract high-level primitive features from the input. Strides result in information loss, so we only use them in the pooling layer and not in the convolutional layer.

Batch Normalization

1) In convolutional neural networks (CNNs), batch normalization is typically applied to convolutional layers, but not to pooling layers or fully connected layers.

2) The purpose of batch normalization in CNNs is to **stabilize and speed up the training process by normalizing the input data within each mini-batch** (subset of data) during training. This helps the model converge faster (minimize loss faster) and improves overall performance.

3) **Batch normalization (BN)** is a technique used to improve the training of deep neural networks. It is **typically used after convolutional layers and before activation functions**.

4) **Batch normalization works by normalizing the activations of a layer across a batch of inputs**. This helps to stabilize the training process and prevents the activations from becoming too large or too small. **BN can also help to improve the generalization performance of the model**.

Q) Why we apply batch normalization only after convolutional layer not after pooling layer?

A) We use batch normalization after convolutional layers because it helps to stabilize and speed up the training process. Batch normalization normalizes the inputs to each layer, making the optimization process more efficient. Pooling layers are used for downsampling and do not require batch normalization for their specific task. Therefore, we apply batch normalization only after convolutional layers to improve the performance of the model during training.

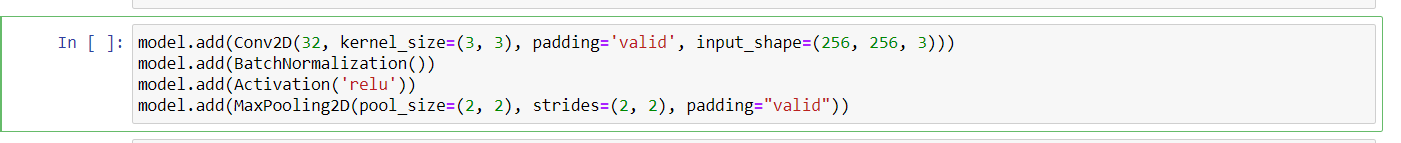
Q) Why do we apply activation function after batch normalization instead of before?

\* We apply the activation function after batch normalization to prevent the vanishing gradient problem and ensure stable and faster training. Batch normalization normalizes the inputs, making them suitable for the activation function, leading to more effective learning.

\* Applying batch normalization before the activation function scales the output of mini-batches, making the model learn efficiently. It also speeds up the training process, and the activation function performs effectively on the normalized output of batch normalization.

\* If we apply batch normalization after the activation function, it will normalize the output values to a range between 0 and 1. This can help to prevent the vanishing gradient problem, which can occur with activation functions such as the sigmoid function.

Code for apply Batch Normalization before activation function



Pooling

1) In real-time projects, image sizes are large. Therefore, we use pooling in almost every project.

Padding

Q) When to apply padding?

(I) Padding is applied to the input data before performing convolutional operations in a Convolutional Neural Network (CNN). This is done to maintain the spatial dimensions of the input and output feature maps during convolution.

(II) Padding is not applied to pooling layers because we want to extract high-level primitive features from the input data. Pooling layers downsample the input data, and padding would add unnecessary information to the output feature maps.

(III) In the industry, zero padding is typically used for CNNs. However, there are other padding methods as well, such as reflection padding and replicate padding.

Image Size

Q) How to specify correct input shape of images while creating dataset?

(I) The choice of the image size (image\_size) when creating a dataset of images depends on various factors, including the characteristics of the dataset, the available computational resources, and the requirements of the specific deep learning model or task you are working on.

(II) As a starting point, image sizes such as (256, 256) or (224, 224) are commonly used in various computer vision tasks. These sizes are often chosen because they strike a balance between maintaining relevant details in the images and computational efficiency.

Filters

1) The choice of the number of filters can depend on the available computational resources and the size of the dataset.

2) Larger numbers of filters increase computational cost and the risk of overfitting, while small filters can lose information.

3) A good compromise is to use (3,3) filters, which are commonly used in industry. However, you may need to use larger or smaller filters depending on the specific task.

Q) How to choose appropriate number of filters at every convolutional layer.

A common practice is to start with a small number of filters in the first layer and gradually increase the number of filters in subsequent layers. For example, in a basic CNN architecture, the number of filters might be set as follows:

(I) First Convolutional Layer: Fewer filters, e.g., 32 or 64.

(II) Intermediate Convolutional Layers: Increasing number of filters, e.g., 128, 256, etc.

(III) Final Convolutional Layer: The number of filters can be set to the number of classes in the classification task (for classification problems).

Dropout Regularization

1) You can use dropout in both convolutional layers and pooling layers in CNNs.

2) **Dropout is not typically used in pooling layers. This is because pooling layers are used to reduce the size of the feature maps, and dropout would introduce additional complexity to the model. However, some researchers have experimented with using dropout in pooling layers, and some have found that it can improve the performance of the model.**

**3) Dropout regularization can be used in both the input and hidden layers of a neural network. However, it is more commonly used in the hidden layers. This is because the input layer is typically not as prone to overfitting as the hidden layers.**

Here are some guidelines for using dropout in CNNs:

1) Pooling layers: Dropout is not typically used in pooling layers.

2) Fully connected layers: Dropout can be used in fully connected layers to prevent overfitting. A good starting point is to use a dropout percentage of 0.5 or 0. 6.

Here are some guidelines for using dropout in ANNs:

1) Smaller layers: Use a lower dropout percentage, such as 0.1 or 0.2.

2) Larger layers: Use a higher dropout percentage, such as 0.3 or 0.4.

3) More complex tasks: Use a higher dropout percentage.

4) Less training data: Use a higher dropout percentage.

Optimizers

1) Use the Adam optimizer, which is a good choice for most deep learning tasks.

Q) Which optimizer is best in which case?

A) 90% time we are using 'Adam' Optimizer in all problem statements. Perform trial and error to choose better optimizer.

Weight Initializers

1) We can use weight initializers in both pooling and convolutional layers in CNNs. In fact, it is important to use a weight initializer in all layers of a CNN, as it can help to improve the convergence of the network and prevent it from overfitting.

2) We can use weight initializers in every hidden layer of an artificial neural network (ANN). In fact, it is generally recommended to use a weight initializer in all layers of an ANN, as it can help to improve the convergence of the network and prevent it from overfitting.

3) If the network is very deep, it may be beneficial to use different weight initializers at different layers. This is because the deeper layers of the network will need to learn more complex features, and using a different initializer can help to prevent the network from becoming too unstable.

4) If the network is not very deep, it may be sufficient to use the same weight initializer in every layer. This is because the shallower layers of the network will not need to learn as complex features, and using the same initializer can help to ensure that the network converges more quickly.

5) Weight initializers can be used in both pooling and convolutional layers in CNNs. In fact, it is important to use a weight initializer in all layers of a CNN, as it can help improve the convergence of the network and prevent it from overfitting.

6) Weight initializers can be used in every hidden layer of an ANN. In fact, it is generally recommended to use a weight initializer in all layers of an ANN, as it can help improve the convergence of the network and prevent it from overfitting.

7) Whether to use the same weight initializer in every layer of a CNN or ANN, or to use different initializers at each layer, is a matter of debate. There is no single answer that is universally correct, as the best approach will depend on the specific dataset and task that the network is being trained on.

Q) Whether to use the same weight initializer in every layer of a CNN or ANN, or to use different initializers at each layer?

A) Whether to use the same weight initializer in every layer of a CNN or ANN, or to use different initializers at each layer, is a matter of debate. There is no single correct answer that is universally applicable, as the best approach will depend on the specific dataset and task that the network is being trained on.

Loss Function

1) In CNNs, loss is typically calculated using the binary cross entropy, categorical cross entropy, or sparse categorical cross entropy loss functions.

2) Use a binary cross entropy loss function, which is a good choice for binary classification tasks and categorical Crosse trophy for multi-class classification tasks.

Q) Which loss function is used for which task?

(I) Regression: MSE, MAE, Huber Loss

(II) Binary Classification: Binary Cross Entropy

(III) Multi-Class Classification: Categorical Loss Entropy

(IV) Image Data: Sparse Categorical Entropy

Metrics

1) Use the accuracy metric, which is a good choice for binary classification tasks.

Activation Function

Q) Why activation function is not applied on pooling layer?

(I) Pooling layers are used to downsample (extract primitive features) from feature map’s output by the convolutional layers, while activation functions are used to introduce non-linearity into the network. Non-linearity is important for deep learning models because it allows them to learn more complex patterns. However, pooling layers are inherently non-linear, so there is no need to add an activation function after them.

(II) Pooling layers are used to **downsample** the feature maps output by the convolutional layers. This reduces the dimensionality of the feature maps, which can help to improve the performance of the network. Activation functions are used to **introduce non-linearity** into the network. Non-linearity is important for deep learning models because it allows them to learn more complex patterns. However, pooling layers are already non-linear, so there is no need to add an activation function after them.

Q) Activations functions are only applied in convolutional layer not in pooling layer, why?

Activation Function after Convolutional Layer

After applying convolution to the input data using a set of filters (kernels), the resulting feature maps **undergo** an activation function element-wise. The activation function introduces non-linearity to the output of the convolutional layer, which allows the network to learn and model complex relationships in the data. Common activation functions used in CNNs include ReLU (Rectified Linear Unit), Leaky ReLU, ELU (Exponential Linear Unit), and others.

No Activation Function after Pooling Layer

After applying pooling (e.g., MaxPooling or AveragePooling), **no activation function is applied directly to the pooled feature maps**. Pooling layers are purely for down-sampling and spatial dimension reduction, and they do not introduce non-linearity.

Convolutional Layer

Q) What will happen if we increase the number of convolutional layers?

(I) Adding more convolutional layers can make the model more complex and difficult to train, but it can also improve the model's performance by extracting more features and reducing overfitting.

(II) Increased complexity: Adding more convolutional layers will increase the complexity of the CNN model. This can make the model more difficult to train, but it can also improve the model's performance.

(III) Improved feature extraction: More convolutional layers can extract more features from the input data. This can improve the model's ability to recognize patterns in the data and make more accurate predictions.

(IV) Reduced overfitting: Adding more convolutional layers can help to reduce overfitting. This is because the model will be able to learn more features from the data, which will make it less likely to memorize the training data and generalize better to new data.

Data Augmentation

1) Data augmentation is a technique that can be used to artificially increase the size of a training dataset by creating modified versions of images in the dataset. This can help to prevent overfitting, which is a problem that can occur when a model is trained on a small dataset.

2) Data augmentation is a technique that can be used to increase the size of a training dataset, reduce overfitting, and improve the performance of a model.

3) Data augmentation is a technique used to artificially increase the size and diversity of a dataset by applying various transformations to the existing images, such as rotations, flips, translations, zooming, etc.

4) However, it's important to note that data augmentation does not create new images. The augmented images are generated on-the-fly during training and do not add to the total number of unique images in the dataset.

5) Instead, they are applied dynamically during each training epoch, allowing the model to see different variations of the original images and improve its generalization.

6) So, the number of unique images in your dataset remains 5000, but the model effectively sees a more diverse and varied dataset due to data augmentation, which can help improve the model's performance and robustness.

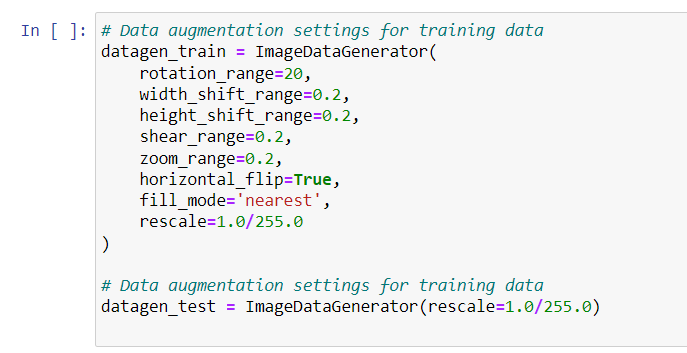
**7) Data augmentation is a technique that can be used to artificially increase the size and diversity of a training dataset by creating modified copies of images in the dataset. This can help to prevent overfitting, which is a problem that can occur when a model is trained on a small dataset.**

**8) Data augmentation increases the training data size, reduces overfitting, and improves the performance of the model.**

**9) Data augmentation is performed on the training dataset after it has been created.**

**Details About Parameters**

**(I) Parameters of Data Augmentation using: (ImageDataGenerator)**



|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Description | Default value | Possible values |
| **rescale** | The factor by which the images are rescaled. | 1 | A float. |
| **featurewise\_center** | Whether to subtract the mean of the training set from each image. | FALSE | Boolean. |
| **featurewise\_std\_normalization** | Whether to divide each image by the standard deviation of the training set. | FALSE | Boolean. |
| **samplewise\_center** | Whether to subtract the mean of each image from itself. | FALSE | Boolean. |
| **samplewise\_std\_normalization** | Whether to divide each image by the standard deviation of itself. | FALSE | Boolean. |
| **zca\_whitening** | Whether to apply ZCA whitening to the images. | FALSE | Boolean. |
| **zca\_epsilon** | The epsilon value to use for ZCA whitening. | 1.00E-06 | A float. |
| **rotation\_range** | The range of degrees by which to randomly rotate the images. | 0 | An integer between -90 and 90. |
| **width\_shift\_range** | The range of pixels to randomly shift the images horizontally. | 0.1 | A float between 0 and 1. |
| **height\_shift\_range** | The range of pixels to randomly shift the images vertically. | 0.1 | A float between 0 and 1. |
| **shear\_range** | The range of degrees by which to randomly shear the images. | 0 | A float between 0 and 0.3. |
| **zoom\_range** | The range of zoom factors to randomly apply to the images. | (0.9, 1.1) | A tuple of floats where the first value is the minimum zoom factor and the second value is the maximum zoom factor. |
| **horizontal\_flip** | Whether to randomly flip the images horizontally. | FALSE | Boolean. |
| **vertical\_flip** | Whether to randomly flip the images vertically. | FALSE | Boolean. |
| **fill\_mode** | The strategy used to fill in the pixels that are outside the image after a transformation is applied. | "nearest" | One of "nearest", "reflect", or "wrap". |
| **cval** | The value used to fill in the pixels that are outside the image after a transformation is applied. | 0 | A float or integer. |
| **validation\_split** | The fraction of the data to use for validation. | 0 | A float between 0 and 1. |
| **subset** | The subset of data to load. Can be one of "training", "validation", or "test". | "training" | One of "training", "validation", or "test". |
| **interpolation** | The interpolation method to use when resizing the images. Can be one of "nearest", "bilinear", "bicubic", "lanczos3", or "lanczos4". | "bilinear" | One of "nearest", "bilinear", "bicubic", "lanczos3", or "lanczos4". |
| **preprocessing\_function** | A function to apply to each image before it is loaded. | None | A function that takes an image as input and returns an image as output. |
| **\*\*kwargs** | Keyword arguments to pass to the tf.keras.preprocessing.image.Generator constructor. | None | Any keyword arguments that are supported by the tf.keras.preprocessing.image.Generator constructor. |

Fill\_mode

**(I) Nearest:** This is like copying the colours from nearby pixels to fill in the empty spots. It's quick, but it might not always look perfect.

**(II) Constant:** You can choose a specific colour to fill in the empty parts. It's like using a paintbrush to colour those areas with the same colour.

**(III) Reflect:** Imagine the image is mirrored at its edges. This means the empty spots are filled with the colours from the nearest parts of the image, reflected across the edge.

**(IV) Wrap**: Think of the image being wrapped around like a gift. The colours from the opposite edge are used to fill in the empty areas.

(II) Parameters to Create Data Augmented Training and Test Data using: (flow\_from\_directory)

**1) directory (Not applicable, no default value as it's required):** Path to the target directory containing subdirectories of images. Each subdirectory should represent a class.

**2) target\_size (256, 256):** Tuple of integers (height, width) specifying the dimensions to which all images will be resized during loading.

**3) batch\_size (32):** Number of images in each batch.

**4) class\_mode (categorical):** Mode of the class labelling. It can take values like:

**(I) 'categorical':** For one-hot encoded labels (multi-class classification).

**(II) 'binary':** For binary classification.

**(III) 'sparse':** For integer labels.

**(IV) 'None':** If no labels are provided (useful for inference).

**5) shuffle (True):** Whether to shuffle the data after each epoch.

**6) seed (None):** Random seed for shuffling and transformations.

**7) color\_mode (rgb):** The colour mode used for loading images.

**(I) 'rgb':** Loads images in RGB format.

**(II) 'grayscale':** Loads images in grayscale format.

**8) save\_to\_dir (None):** Directory to save augmented images (useful for visualizing augmentation).

**9) save\_prefix (''):** Prefix to use for filenames of saved augmented images.

**10) save\_format (png):** Format for saved augmented images ('png', 'jpeg', etc.).

**11) subset (None):** Specifies which subset of the data to load ('training' or 'validation').

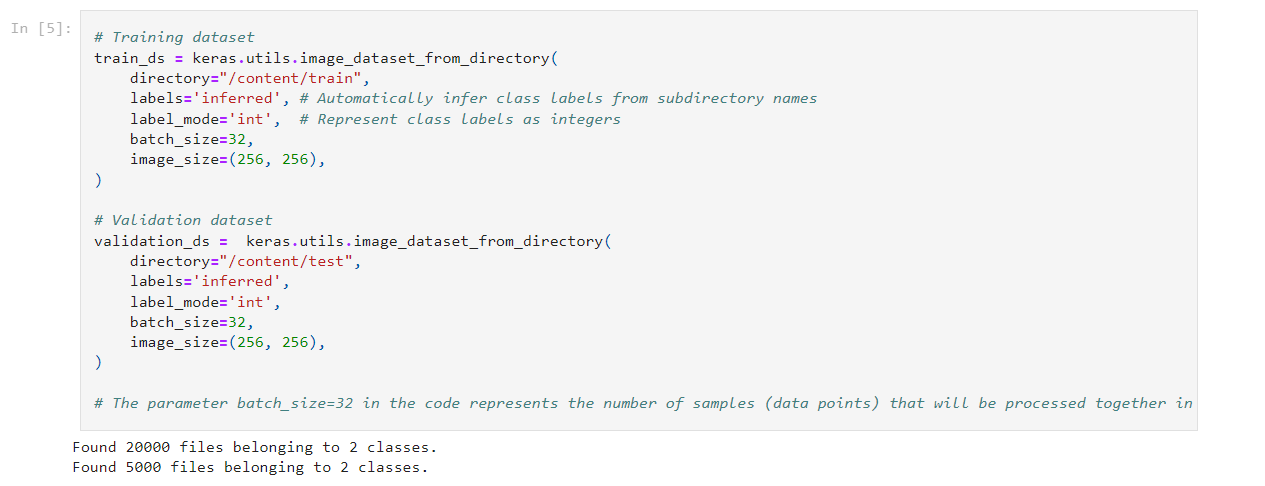
**12) interpolation (nearest):** Interpolation method used for resizing images.

**13) follow\_links (False):** Whether to follow symbolic links to subdirectories.

**14) dtype (float32):** Data type to use for loaded images.

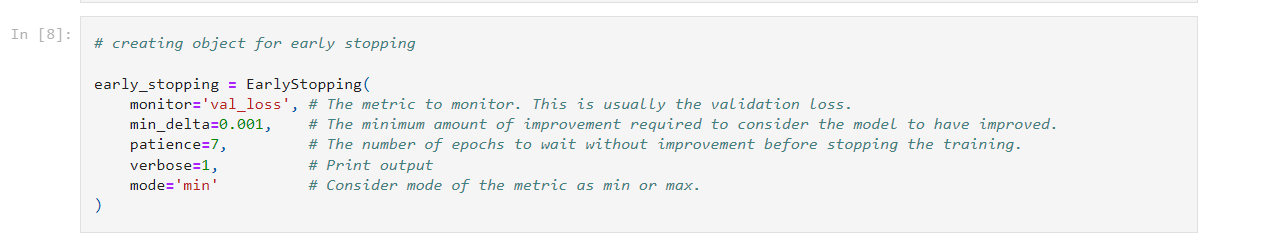
**15) validation\_split (0.0):** Fraction of data to reserve for validation (if not using separate validation data).

(III) Parameters to Create Training and Test Data using: (**tf.keras.preprocessing.image\_dataset\_from\_directory**)



|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Description | Default value | Possible values |
| **directory** | The directory where the images are located. | None | A string. |
| **labels** | The labels for the images. If this is not specified, the images will be labelled with the names of their subdirectories. | "inferred" | A string or a list of strings. |
| **label\_mode** | The mode for handling the labels. Can be one of "int", "categorical", or "binary". | "categorical" | One of "int", "categorical", or "binary". |
| **image\_size** | The size of the images to resize to. | (256, 256) | A tuple of integers. |
| **batch\_size** | The batch size to use. | 32 | An integer. |
| **shuffle** | Whether to shuffle the data. | TRUE | Boolean. |
| **seed** | The random seed to use. | None | An integer. |
| **validate\_filenames** | Whether to validate the filenames. | TRUE | Boolean. |
| **class\_names** | The list of class names. If this is not specified, the class names will be inferred from the directory structure. | None | A list of strings. |
| **color\_mode** | The colour mode of the images. Can be one of "rgb", "grayscale", or "rgb". | "rgb" | One of "rgb", "grayscale", or "rgb". |
| **subset** | The subset of data to load. Can be one of "training", "validation", or "test". | "training" | One of "training", "validation", or "test". |
| **interpolation** | The interpolation method to use when resizing the images. Can be one of "nearest", "bilinear", "bicubic", "lanczos3", or "lanczos4". | "bilinear" | One of "nearest", "bilinear", "bicubic", "lanczos3", or "lanczos4". |
| **preprocessing\_function** | A function to apply to each image before it is loaded. | None | A function that takes an image as input and returns an image as output. |
| **augment** | Whether to apply data augmentation to the images. | FALSE | Boolean. |
| **\*\*kwargs** | Keyword arguments to pass to the ImageDataGenerator constructor. | None | Any keyword arguments that are supported by the ImageDataGenerator constructor. |

(IV) Parameters of Early Stopping



|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Default Value | Possible Values | Description |
| **monitor** | 'val\_loss' | String. E.g., 'val\_loss', 'val\_accuracy' | Quantity to be monitored for early stopping. Typically, a validation metric such as validation loss or validation accuracy. |
| **min\_delta** | 0 | Float | Minimum change in the monitored quantity to qualify as an improvement. For example, if min\_delta=0.1, then the monitored quantity must improve by at least 0.1 to be considered an improvement. |
| **patience** | 0 | Integer | Number of epochs with no improvement after which training will be stopped. If patience=5, training will stop after 5 consecutive epochs with no improvement. |
| **verbose** | 0 | Integer (0, 1, or 2) | Verbosity mode. 0: Silent, 1: Update messages, 2: Epoch-wise updates. |
| **mode** | 'auto' | String. E.g., 'auto', 'min', 'max' | Direction of improvement. 'auto' automatically determines the mode based on the monitored quantity. 'min' looks for decreasing monitored quantity, and 'max' looks for increasing monitored quantity. |
| **baseline** | None | Float | Baseline value for the monitored quantity. If the monitored quantity doesn't improve beyond this value, training will stop. |
| **restore\_best\_weights** | FALSE | Boolean | Whether to restore model weights from the epoch with the best value of the monitored quantity. If True, the model's weights will be set to the weights from the best epoch. |

(V) Parameters of model.add() method in keras

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter | Default Value | Possible Values | Description |
| **layer** | None (required) | Keras Layer object | The layer to be added to the model. |
| **Arguments specific to certain layer types** |  |  |  |
| **input\_shape** | None | Tuple or list of integers | Defines the shape of the input data. Required for the first layer, but optional for subsequent layers. |
| **batch\_input\_shape** | None | Tuple or list of integers | Similar to input\_shape, but includes the batch size dimension. |
| **dtype** | None | Data type (string) | Data type expected by the layer's inputs. |
| **activation** | None | String (e.g., 'relu', 'sigmoid', etc.) | Activation function applied to the layer's output. |
| **units** | None | Integer | Number of neurons or units in the layer. |
| **Regularization and Constraints** |  |  |  |
| **kernel\_regularizer** | None | Keras Regularizer object | Regularizer applied to the layer's kernel (weight) matrix. |
| **bias\_regularizer** | None | Keras Regularizer object | Regularizer applied to the layer's bias vector. |
| **kernel\_constraint** | None | Keras Constraint object | Constraint applied to the layer's kernel matrix elements. |
| **bias\_constraint** | None | Keras Constraint object | Constraint applied to the layer's bias vector elements. |
| **Initializers** |  |  |  |
| **kernel\_initializer** | 'glorot\_uniform' | Keras Initializer object or string | Initializes the layer's kernel (weight) matrix. |
| **bias\_initializer** | 'zeros' | Keras Initializer object or string | Initializes the layer's bias vector. |
| **Other** |  |  |  |
| **name** | None | String | Name of the layer. |

Frequently Used Parameter

1) batch\_size

(I) Batch size is the number of training examples used in one iteration of the training process in a CNN. It is important because it determines how many samples are processed together in each step during training. Using a proper batch size can lead to efficient utilization of computational resources and can result in more stable and faster convergence during training.

(II) The batch size parameter in deep learning represents the number of training examples that are used in one forward/backward pass.

(III) A larger batch size can improve the performance of the neural network, but it can also make the training process slower.

(IV) A smaller batch size can make the training process faster, but it can also lead to less accurate results.

Small Batch Size

Advantages: Smaller batch sizes tend to converge faster as they update the model's parameters more frequently. They also require less memory, which can be beneficial when working with limited resources.

Disadvantages: Smaller batch sizes can introduce more noise to the parameter updates, potentially leading to less stable convergence and noisy gradients.

Large Batch Size

Advantages: Larger batch sizes can provide smoother gradient updates, potentially leading to more stable convergence. They can also benefit from vectorized operations, improving computational efficiency.

Disadvantages: Larger batch sizes require more memory and may take longer per epoch due to fewer parameter updates. They might not generalize as well to the validation set due to their potential to get stuck in local minima.

2) mini\_batch

(I) The mini-batch size is a hyperparameter that can be tuned to improve the performance of the model. A smaller mini-batch size can help to prevent overfitting, but it can also make the training process slower. A larger mini-batch size can speed up the training process, but it can also increase the risk of overfitting.

(II) The optimal mini-batch size depends on the specific problem and the dataset. However, a good starting point is a mini-batch size of 32 to 128.

Perfect Code

1. We can add multiple convolutional layers in one convolutional layer before a pooling layer.
2. We can use batch normalization in dense layers as well.
3. We can use data augmentation while training the model to prevent overfitting.
4. If you want to add more data to a pretrained model and do not want to rebuild their fully connected architecture, build a model on the same architecture of fully-connected layers.

