


AutoSave Model_DL_ESA_QP - Protected View • Saved Search Rajesh Narayanan Gopalan

File Home Insert Draw Design Layout References Mailings Review View Help

PROTECTED VIEW Be careful—files from the Internet can contain viruses. Unless you need to edit, it's safer to stay in Protected View. Enable Editing

		PES University, Bengaluru (Established under Karnataka Act No. 16 of 2013)	UE20CS935
MODEL QP: END SEMESTER ASSESSMENT (ESA) M TECH DATA SCIENCE AND MACHINE LEARNING_ SEMESTER II UE20CS935- INTRODUCTION TO DEEP LEARNING & ITS APPLICATIONS			
Time: 3 Hrs		Answer All Questions	Max Marks: 80

SECTION-A (20 marks)			
1	a)	Explain the convolutional neural network architecture in detail.	4
	b)	Explain overfitting in neural networks? How to overcome the problem?	4
	c)	What are the activation functions in neural networks? What is the use of these activation functions?	4
	d)	What is the difference between single stage and multi stage object detection models?	4
	e)	Briefly explain GANs? What are their advantages?	4
SECTION-B (30 marks)			
2			10

Page 1 of 3 475 words Text Protected View

Explain the convolutional neural network architecture in detail.

convolutional neural networks (ConvNets or CNNs) are more often utilized for classification and computer vision tasks. Prior to CNNs, manual, time-consuming feature extraction methods were used to identify objects in images. However, convolutional neural networks now provide a more scalable approach to image classification and object recognition tasks, leveraging principles from linear algebra, specifically matrix multiplication, to identify patterns within an image. That said, they can be computationally demanding, requiring graphical processing units (GPUs) to train models

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

The convolutional layer is the first layer of a convolutional network. While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object.

Convolutional Layer

The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map. Let's assume that the input will be a color image, which is made up of a matrix of pixels in 3D. This means that the input will have three dimensions—a height, width, and depth—which correspond to RGB in an image. We also have a feature detector, also known as a kernel or a filter, which will move across the receptive fields of the image, checking if the feature is present. This process is known as a convolution.

The feature detector is a two-dimensional (2-D) array of weights, which represents part of the image. While they can vary in size, the filter size is typically a 3x3 matrix; this also determines the size of the receptive field. The filter is then applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterwards, the filter shifts by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products from the input and the filter is known as a feature map, activation map, or a convolved feature.

After each convolution operation, a CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map, introducing nonlinearity to the model.

As we mentioned earlier, another convolution layer can follow the initial convolution layer. When this happens, the structure of the CNN can become hierarchical as the later layers can see the pixels within the receptive fields of prior layers. As an example, let's assume that we're trying to determine if an image contains a bicycle. You can think of the bicycle as a sum of parts. It is comprised of a frame, handlebars, wheels, pedals, et cetera. Each individual part of the bicycle makes up a lower-level pattern in the neural net, and the combination of its parts represents a higher-level pattern, creating a feature hierarchy within the CNN.

Pooling Layer

Pooling layers, also known as downsampling, conducts dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the

difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field, populating the output array. There are two main types of pooling:

- **Max pooling:** As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. As an aside, this approach tends to be used more often compared to average pooling.
- **Average pooling:** As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

While a lot of information is lost in the pooling layer, it also has a number of benefits to the CNN. They help to reduce complexity, improve efficiency, and limit risk of overfitting.

Fully-Connected Layer

The name of the full-connected layer aptly describes itself. As mentioned earlier, the pixel values of the input image are not directly connected to the output layer in partially connected layers. However, in the fully-connected layer, each node in the output layer connects directly to a node in the previous layer.

This layer performs the task of classification based on the features extracted through the previous layers and their different filters. While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.

Types of convolutional neural networks

Kunihiko Fukushima and Yann LeCun laid the foundation of research around convolutional neural networks in their work in [1980](#) (PDF, 1.1 MB) (link resides outside IBM) and "Backpropagation Applied to Handwritten Zip Code Recognition" in 1989, respectively. More famously, Yann LeCun successfully applied backpropagation to train neural networks to identify and recognize patterns within a series of handwritten zip codes. He would continue his research with his team throughout the 1990s, culminating with "LeNet-5", which applied the same principles of prior research to document recognition. Since then, a number of variant CNN architectures have emerged with the introduction of new datasets, such as MNIST and CIFAR-10, and competitions, like ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Some of these other architectures include:

- [AlexNet](#) (PDF, 1.4 MB) (link resides outside IBM)
- [VGGNet](#) (PDF, 195 KB) (link resides outside IBM)
- [GoogLeNet](#) (PDF, 1.3 MB) (link resides outside IBM)
- [ResNet](#) (PDF, 800 KB) (link resides outside IBM)
- ZFNet

However, LeNet-5 is known as the classic CNN architecture.

Convolutional neural networks and computer vision

Convolutional neural networks power image recognition and computer vision tasks. [Computer vision](#) is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos and other visual inputs, and based on those inputs, it can take action. This ability to provide recommendations distinguishes it from image recognition tasks. Some common applications of this computer vision today can be seen in:

- **Marketing:** Social media platforms provide suggestions on who might be in photograph that has been posted on a profile, making it easier to tag friends in photo albums.
- **Healthcare:** Computer vision has been incorporated into radiology technology, enabling doctors to better identify cancerous tumors in healthy anatomy.
- **Retail:** Visual search has been incorporated into some e-commerce platforms, allowing brands to recommend items that would complement an existing wardrobe.
- **Automotive:** While the age of driverless cars hasn't quite emerged, the underlying technology has started to make its way into automobiles, improving driver and passenger safety through features like lane line detection.

Explain overfitting in neural networks? How to overcome the problem?

Prime Video: The Shield, Sea... x Olympus LMS x Inbox (4,190) - grajeshn1972 x otpdelivery.cvindia.com:808... x

https://pesedu.olympuslms.com/mentorship_recordings/2007396

PES-MTech-DS... Dashboard Courses Hackathons

Search any topic

Documents/Python Scripts x DL LI - Jupyter Notebook x GA_DI_Solution_binary_class x GA_T_DI_SOLUTIONS_gsmc x about:blank x

DL LI Last Checkpoint Last Wednesday at 12:41 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 (ipykernel)

Overfitting

```
1 # high train accuracy and low test accuracy
2 # why? Reason
3 Overfitting is another name of memorize
4 too many learning parameters
5 too many w/b in DL model
6 unscaled data, multico-linearity
7 #How
8 Regularization
9 L1, L2
10 Dropouts
11 Callbacks (model performance)
12 Early stopping
13 Increase the training data
14 Data augmentation
15 Reduce the learning parameters
16 Reduce the hidden layer Weights
17 Transfer learning
```

Under fitting

```
1 # both train and test accuracy low
```

In []:

GA_DI_Solution_...html GA_T_DI_SOLUTION_...html

Show all

Search

ENG IN 06:33 14-06-2023

Prime Video: The Shield, Sea... x Olympus LMS x Inbox (4,190) - grajeshn1972 x otpdelivery.cvindia.com:808... x

https://pesedu.olympuslms.com/mentorship_recordings/2007396

PES-MTech-DS... Dashboard Courses Hackathons

Search any topic

Documents/Python Scripts x DL LI - Jupyter Notebook x GA_DI_Solution_binary_class x GA_T_DI_SOLUTIONS_gsmc x about:blank x

DL LI Last Checkpoint Last Wednesday at 12:41 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3 (ipykernel)

```
4 too many learning parameters
5 too many w/b in DL model
6 unscaled data, multico-linearity
7 #How
8 Regularization
9 L1, L2
10 Dropouts
11 Callbacks (model performance)
12 Early stopping
13 Increase the training data
14 Data augmentation
15 Reduce the learning parameters
16 Reduce the hidden layer Weights
17 Transfer learning
```

Under fitting

```
In [ ]: 1 # both train and test accuracy low
2 # why?
3 Data may not be scaled
4 Less or not correct features
5 gradients are not learned (vanishing or exploding)
6 model not able to learn (req. amount of hidden layers)
7 Outliers (Deep learning models will not get effect)
```

GA_DI_Solution_...html GA_T_DI_SOLUTION_...html

Show all

Search

ENG IN 06:41 14-06-2023

Prime Video: The Shield, Sea... x Olympus LMS x Inbox (4,190) - grajeshn1972 x otpdelivery.cvindia.com:808... x

https://pesedu.olympuslms.com/mentorship_recordings/2007396

PES-MTech-DS... Dashboard Courses Hackathons Search any topic

jupyter DL LI Last Checkpoint Last Wednesday at 12:41 (unsaved changes)

```
In [ ]: 1 # both train and test accuracy low
2 # why?
3 Data may not be scaled
4 Standard scaler/ Batch normalization
5
6 Less or not correct features
7 Increase no of layers, no of kernels, manual feature extraction
8 gradients are not learned (vanishing or exploding)
9 activations, hyperparameter
10 model not able to learn (req amount of hidden layers)
11 change the model
12 Outliers (Deep learning models will not get effect)
13 transformation
14
```

Learning parameters

In []: 1

Activation function

In []: 1

GA_DL_Solution_...html GA_LI_DL_SOLUTION_...html Show all

Search

ENG IN 06:44 14-06-2023

What are the activation functions in neural networks? What is the use of these activation functions?

c) List and brief about the activation functions in neural networks? 4

Activation functions are mathematical equations that determine the output of a neural network model. Activation functions also have a major effect on the [neural network's](#) ability to converge and the convergence speed, or in some cases, activation functions might prevent neural networks from converging in the first place. Activation function also helps to normalize the output of any input in the range between 1 to -1 or 0 to 1.

Activation function must be efficient and it should reduce the computation time because the neural network sometimes trained on millions of data points.

Let's consider the simple neural network model without any hidden layers.

Here is the output-

$$Y = \sum (\text{weights} * \text{input} + \text{bias})$$

and it can range from -infinity to +infinity. So it is necessary to bound the output to get the desired prediction or generalized results.

$$Y = \text{Activation function}(\sum (\text{weights} * \text{input} + \text{bias}))$$

So the activation function is an important part of an artificial neural network. They decide whether a neuron should be activated or not and it is a non-linear transformation that can be done on the input before sending it to the next layer of neurons or finalizing the output.

Properties of activation functions

1. Non Linearity
2. Continuously differentiable
3. Range
4. Monotonic
5. Approximates identity near the origin

Types of Activation Functions

The activation function can be broadly classified into 2 categories.

1. Binary Step Function
2. Linear Activation Function

Binary Step Function

A binary step function is generally used in the Perceptron linear classifier. It thresholds the input values to 1 and 0, if they are greater or less than zero, respectively.

The step function is mainly used in binary classification problems and works well for linearly severable pr. It can't classify the multi-class problems.

Also Read: [3 Things to Know before deep diving into Neural Networks](#)

Linear Activation Function

The equation for Linear activation function is:

$$f(x) = a.x$$

When $a = 1$ then $f(x) = x$ and this is a special case known as identity.

Properties:

1. Range is -infinity to +infinity
2. Provides a convex error surface so optimisation can be achieved faster
3. $df(x)/dx = a$ which is constant. So cannot be optimised with gradient descent

Limitations:

1. Since the derivative is constant, the gradient has no relation with input
2. Back propagation is constant as the change is delta x

Non-Linear Activation Functions

Modern neural network models use non-linear activation functions. They allow the model to create complex mappings between the network's inputs and

outputs, such as images, video, audio, and data sets that are non-linear or have high dimensionality.

Majorly there are 3 types of Non-Linear Activation functions.

1. Sigmoid Activation Functions
2. Rectified Linear Units or ReLU
3. Complex Nonlinear Activation Functions

Sigmoid Activation Functions

Sigmoid functions are bounded, differentiable, real functions that are defined for all real input values, and have a non-negative derivative at each point.

Sigmoid or Logistic Activation Function

The sigmoid function is a logistic function and the output is ranging between 0 and 1.

The output of the activation function is always going to be in range (0,1) compared to $(-\infty, \infty)$ of linear function. It is non-linear, continuously differentiable, monotonic, and has a fixed output range. But it is not zero centred.

Hyperbolic Tangent

The function produces outputs in scale of $[-1, 1]$ and it is a continuous function. In other words, function produces output for every x value.

$$Y = \tanh(x)$$
$$\tanh(x) = (e^x - e^{-x}) / (e^x + e^{-x})$$

Inverse Hyperbolic Tangent (arctanh)

It is similar to sigmoid and tanh but the output ranges from $[-\pi/2, \pi/2]$

Softmax

The softmax function is sometimes called the soft argmax function, or multi-class logistic regression. This is because the softmax is a generalization of logistic regression that can be used for multi-class classification, and its formula is very

similar to the sigmoid function which is used for logistic regression. The softmax function can be used in a classifier only when the classes are mutually exclusive.

Gudermannian

The Gudermannian function relates circular functions and hyperbolic functions without explicitly using complex numbers.

The below is the mathematical equation for Gudermannian function:

GELU (Gaussian Error Linear Units)

An activation function used in the most recent Transformers such as Google's BERT and OpenAI's GPT-2. This activation function takes the form of this equation:

$$\text{GELU}(x) = 0.5x(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3)))$$

So it's just a combination of some functions (e.g. hyperbolic tangent \tanh) and approximated numbers.

It has a negative coefficient, which shifts to a positive coefficient. So when x is greater than zero, the output will be x , except from when $x=0$ to $x=1$, where it slightly leans to a smaller y -value.

Also Read: [What is Recurrent Neural Network / Introduction of Recurrent Neural Network](#)

Problems with Sigmoid Activation Functions

1. Vanishing Gradients Problem

The main problem with deep neural networks is that the gradient diminishes dramatically as it is propagated backward through the network. The error may be so small by the time it reaches layers close to the input of the model that it may have very little effect. As such, this problem is referred to as the "vanishing gradients" problem.

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.

2. Exploding Gradients

Exploding gradients are a problem where large error gradients accumulate and result in very large updates to neural network model weights during training. These large updates in turn results in an unstable network. At an extreme, the values of weights can become so large as to overflow and result in NaN values.

Rectified Linear Units or ReLU

The sigmoid and hyperbolic tangent activation functions cannot be used in networks with many layers due to the vanishing gradient problem. The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better. The rectified linear activation is the default activation when developing multilayer Perceptron and convolutional neural networks.

Rectified Linear Units(ReLU)

ReLU is the most commonly used activation function in neural networks and The mathematical equation for ReLU is:

$$\text{ReLU}(x) = \max(0, x)$$

So if the input is negative, the output of ReLU is 0 and for positive values, it is x.

Though it looks like a linear function, it's not. ReLU has a derivative function and allows for backpropagation.

There is one problem with ReLU. Let's suppose most of the input values are negative or 0, the ReLU produces the output as 0 and the neural network can't perform the back propagation. This is called the Dying ReLU problem. Also, ReLU is an unbounded function which means there is no maximum value.

Pros:

1. Less time and space complexity
2. Avoids the vanishing gradient problem.

Cons:

1. Introduces the dead relu problem.
2. Does not avoid the exploding gradient problem.

Leaky ReLU

The dying ReLU problem is likely to occur when:

1. Learning rate is too high
2. There is a large negative bias

Leaky ReLU is the most common and effective method to solve a dying ReLU problem. It adds a slight slope in the negative range to prevent the dying ReLU issue.

Again this doesn't solve the exploding gradient problem.

Parametric ReLU

PReLU is actually not so different from Leaky ReLU.

So for negative values of x , the output of PReLU is α times x and for positive values, it is x .

Parametric ReLU is the most common and effective method to solve a dying ReLU problem but again it doesn't solve exploding gradient problem.

Exponential Linear Unit (ELU)

ELU speeds up the learning in neural networks and leads to higher classification accuracies, and it solves the vanishing gradient problem. ELUs have improved learning characteristics compared to the other activation functions. ELUs have negative values that allow them to push mean unit activations closer to zero like batch normalization but with lower computational complexity.

The mathematical expression for ELU is:

ELU is designed to combine the good parts of ReLU and leaky ReLU and it doesn't have the dying ReLU problem. It saturates for large negative values, allowing them to be essentially inactive.

Scaled Exponential Linear Unit (SELU)

SELU incorporates normalization based on the central limit theorem. SELU is a monotonically increasing function, where it has an approximately constant negative output for large negative input. SELU's are mostly commonly used in Self Normalizing Networks (SNN).

The output of a SELU is normalized, which could be called internal normalization, hence the fact that all the outputs are with a mean of zero and standard deviation of one. The main advantage of SELU is that the Vanishing and exploding gradient problem is impossible and since it is a new activation function, it requires more testing before usage.

Softplus or SmoothReLU

The derivative of the softplus function is the logistic function.

The mathematical expression is:

And the derivative of softplus is:





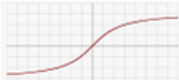




Swish function

The Swish function was developed by Google, and it has superior performance with the same level of computational efficiency as the ReLU function. ReLU still plays an important role in deep learning studies even for today. But experiments show that this new activation function overperforms ReLU for deeper networks

The mathematical expression for Swish Function is:

The modified version of swish function is:

Here, β is a parameter that must be tuned. If β gets closer to ∞ , then the function looks like ReLU. Authors of the Swish function proposed to assign β as 1 for reinforcement learning tasks.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Using Activation Functions in Neural Networks

by [Zhe Ming Chng](#) on July 4, 2022 in [Deep Learning](#)

TweetTweetShare

Last Updated on August 6, 2022

Activation functions play an integral role in neural networks by introducing nonlinearity. This nonlinearity allows neural networks to develop complex representations and functions based on the inputs that would not be possible with a simple linear regression model.

Many different nonlinear activation functions have been proposed throughout the history of neural networks. In this post, you will explore three popular ones: sigmoid, tanh, and ReLU.

To make the network represent more complex functions, you would need nonlinear activation functions.

The screenshot shows a web browser window with the URL <https://machinelearningmastery.com/using-activation-functions-in-...>. The main content area displays a code editor with the following Python code:

```
2 x = relu(x)

However, for many Keras layers, you can also use a more compact representation to add the activation on top of the layer:

1 x = Dense(units=10, activation="relu")(input_layer)

Using this more compact representation, let's build our LeNet5 model using Keras:

2 import tensorflow.keras as keras
3 from tensorflow.keras.layers import Dense, Input, Flatten, Conv2D, BatchNormalization, MaxPool2D
4 from tensorflow.keras.models import Model
5
6 (trainX, trainY), (testX, testY) = keras.datasets.cifar10.load_data()
7
8 input_layer = Input(shape=(32,32,3))
9 x = Conv2D(filters=6, kernel_size=(5,5), padding="same", activation="relu")(input_layer)
10 x = MaxPool2D(pool_size=(2,2))(x)
11 x = Conv2D(filters=16, kernel_size=(5,5), padding="same", activation="relu")(x)
12 x = MaxPool2D(pool_size=(2,2))(x)
13 x = Conv2D(filters=120, kernel_size=(5,5), padding="same", activation="relu")(x)
14 x = Flatten()(x)
15 x = Dense(units=84, activation="relu")(x)
16 x = Dense(units=10, activation="softmax")(x)
17
18 model = Model(inputs=input_layer, outputs=x)
19
20 print(model.summary())
21
22 model.compile(optimizer="adam", loss=tf.keras.losses.SparseCategoricalCrossentropy(), metrics="accuracy")
23
24 history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10, validation_data=(testX, testY))
```

And running this code gives the following output:

```
1 Model: "model"
```

The sidebar on the right contains social media links (LinkedIn, Twitter, Facebook, Email, RSS) and a section titled "Picked for you:" with several tutorial links:

- Your First Deep Learning Project in Python with Keras Step-by-Step
- How to Grid Search Hyperparameters for Deep Learning Models in Python with Keras
- Regression Tutorial with the Keras Deep Learning Library in Python
- Multi-Class Classification Tutorial with the Keras Deep Learning Library
- How to Save and Load Your Keras Deep Learning Model

At the bottom, there is a Prime Video advertisement for "OnePlus Bullets Z2 Bluetooth Wireless in Ear Earphones with Mic, Bombastic Bass - 12.4 mm Drivers, 10 Mins Charge - 20 Hrs Music, 30 Hrs Battery Life, IP55 Dust and Water Resistant (Magico Black)" priced at ₹1,695.00.

Prime V x IntroUC x Explain x What a x Why are x Using A x tds Activati x What a x

https://machinelearningmastery.com/using-activation-functions-in-...

24 history = model.fit(x=trainX, y=trainY, batch_size=256, epochs=10, validation_data=(testX, testY))

And running this code gives the following output:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 32, 32, 3)]	0
conv2d (Conv2D)	(None, 32, 32, 6)	456
max_pooling2d (MaxPooling2D)	(None, 16, 16, 6)	0
conv2d_1 (Conv2D)	(None, 16, 16, 16)	2416
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 16)	0
conv2d_2 (Conv2D)	(None, 8, 8, 128)	48128
flatten (Flatten)	(None, 7680)	0
dense (Dense)	(None, 84)	645204
dense_1 (Dense)	(None, 10)	850

Total params: 697,046
Trainable params: 697,046
Non-trainable params: 0

Epoch 1/10
196/196 [=====] - 14s 11ms/step - loss: 2.9758 acc: 0.3390 - val_loss: 1
Epoch 2/10
196/196 [=====] - 2s 8ms/step - loss: 1.4319 - acc: 0.4927 - val_loss: 1
Epoch 3/10
196/196 [=====] - 2s 8ms/step - loss: 1.2505 acc: 0.5582 - val_loss: 1
Epoch 4/10
196/196 [=====] - 2s 8ms/step - loss: 1.1755 acc: 0.6152 - val_loss: 1
Epoch 5/10
196/196 [=====] - 2s 8ms/step - loss: 1.1005 acc: 0.6722 - val_loss: 1
Epoch 6/10
196/196 [=====] - 2s 8ms/step - loss: 1.0255 acc: 0.7292 - val_loss: 1
Epoch 7/10
196/196 [=====] - 2s 8ms/step - loss: 0.9505 acc: 0.7862 - val_loss: 1
Epoch 8/10
196/196 [=====] - 2s 8ms/step - loss: 0.8755 acc: 0.8432 - val_loss: 1
Epoch 9/10
196/196 [=====] - 2s 8ms/step - loss: 0.8005 acc: 0.9002 - val_loss: 1
Epoch 10/10
196/196 [=====] - 2s 8ms/step - loss: 0.7255 acc: 0.9572 - val_loss: 1

Never miss a tutorial:

[Your First Deep Learning Project in Python with Keras Step-by-Step](#)

[How to Grid Search Hyperparameters for Deep Learning Models in Python with Keras](#)

[Regression Tutorial with the Keras Deep Learning Library in Python](#)

[Multi-Class Classification Tutorial with the Keras Deep Learning Library](#)

[How to Save and Load Your Keras Deep Learning Model](#)

Loving the Tutorials?

OnePlus Bullets Z2 Bluetooth Wireless in Ear Earphones with Mic, Bombastic Bass - 12.4 mm Drivers, 10 Mins Charge - 20 Hrs Music, 30 Hrs Battery Life, IP55 Dust and Water Resistant (Magico Black) ₹1,695.00 prime Shop now

Waiting for machinelearningmastery.com...

What is the difference between single stage and multi stage object detection models?

Object Detection is one of the most famous and vastly researched topics in the field of Computer Vision and Machine Learning

The algorithms designed to do object detection are based on two approaches - one-stage object detection and two-stage object detection. One-stage detectors have high inference speeds and two-stage detectors have high localization and recognition accuracy. The two stages of a two-stage detector can be divided by a RoI (Region of Interest) Pooling layer. One of the prominent two-stage object detectors is Faster R-CNN. It has the first stage called RPN, a Region Proposal Network to predict candidate bounding boxes. In the second stage, features are by RoI pooling operation from each candidate box for the following classification and bounding box regression tasks [1]. In contrast, a one-stage detector predicts bounding boxes in a single-step without using region proposals. It leverages the help of a grid box and anchors to localize the region of detection in the image and constraint the shape of the object.

Excellent read for Object Detection

[Object Detection in 2023: The Definitive Guide - viso.ai](https://viso.ai/object-detection-in-2023/)

One-stage vs. two-stage deep learning object detectors As you can see in the list above, state-of-the-art object detection methods can be categorized into two main types: One-stage vs. two-stage object detectors. In general, deep learning based object detectors extract features from the input image or video frame. An object detector solves two subsequent tasks: Task #1: Find an arbitrary number of objects (possibly even zero), and Task #2: Classify every single object and estimate its size with a bounding box. To simplify the process, you can separate those tasks into two stages. Other methods combine both tasks into one step (single-stage detectors) to achieve higher performance at the cost of accuracy. Two-stage detectors: In two-stage object detectors, the approximate object regions are proposed using deep features before these features are used for the image classification as well as bounding box regression for the object candidate. The two-stage architecture involves (1) object region proposal with conventional Computer Vision methods or deep networks, followed by (2) object classification based on features extracted from the proposed region with bounding-box regression. Two-stage methods achieve the highest detection accuracy but are typically slower. Because of the many inference steps per image, the performance (frames per second) is not as good as one-stage detectors. Various two-stage detectors include region convolutional neural network (RCNN), with evolutions Faster R-CNN or Mask R-CNN. The latest evolution is the granulated RCNN (G-RCNN). Two-stage object detectors first find a region of interest and use this cropped region for classification. However, such multi-stage detectors are usually not end-to-end trainable because cropping is a non-differentiable operation. One-stage detectors: One-stage detectors predict bounding boxes over the images

without the region proposal step. This process consumes less time and can therefore be used in real-time applications. One-stage object detectors prioritize inference speed and are super fast but not as good at recognizing irregularly shaped objects or a group of small objects. The most popular one-stage detectors include the YOLO, SSD, and RetinaNet. The latest real-time detectors are YOLOv7 (2022), YOLOR (2021) and YOLOv4-Scaled (2020). View the benchmark comparisons below. The main advantages of object detection with single-stage algorithms include a generally faster detection speed and greater structural simplicity and efficiency compared to multi-stage detectors.

Read more at: <https://viso.ai/deep-learning/object-detection/>

A Gentle Introduction to Generative Adversarial Networks (GANs)

by [Jason Brownlee](#) on June 17, 2019 in [Generative Adversarial Networks](#)

TweetTweetShare

Last Updated on July 19, 2019

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks.

Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake.

In this post, you will discover a gentle introduction to Generative Adversarial Networks, or GANs.

After reading this post, you will know:

- Context for GANs, including supervised vs. unsupervised learning and discriminative vs. generative modeling.
- GANs are an architecture for automatically training a generative model by treating the unsupervised problem as supervised and using both a generative and a discriminative model.
- GANs provide a path to sophisticated domain-specific data augmentation and a solution to problems that require a generative solution, such as image-to-image translation.

Kick-start your project with my new book [Generative Adversarial Networks with Python](#), including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.



A Gentle Introduction to Generative Adversarial Networks (GANs)

Photo by [Barney Moss](#), some rights reserved.

Overview

This tutorial is divided into three parts; they are:

1. What Are Generative Models?
2. What Are Generative Adversarial Networks?
3. Why Generative Adversarial Networks?

What Are Generative Models?

In this section, we will review the idea of generative models, stepping over the supervised vs. unsupervised learning paradigms and discriminative vs. generative modeling.

Supervised vs. Unsupervised Learning

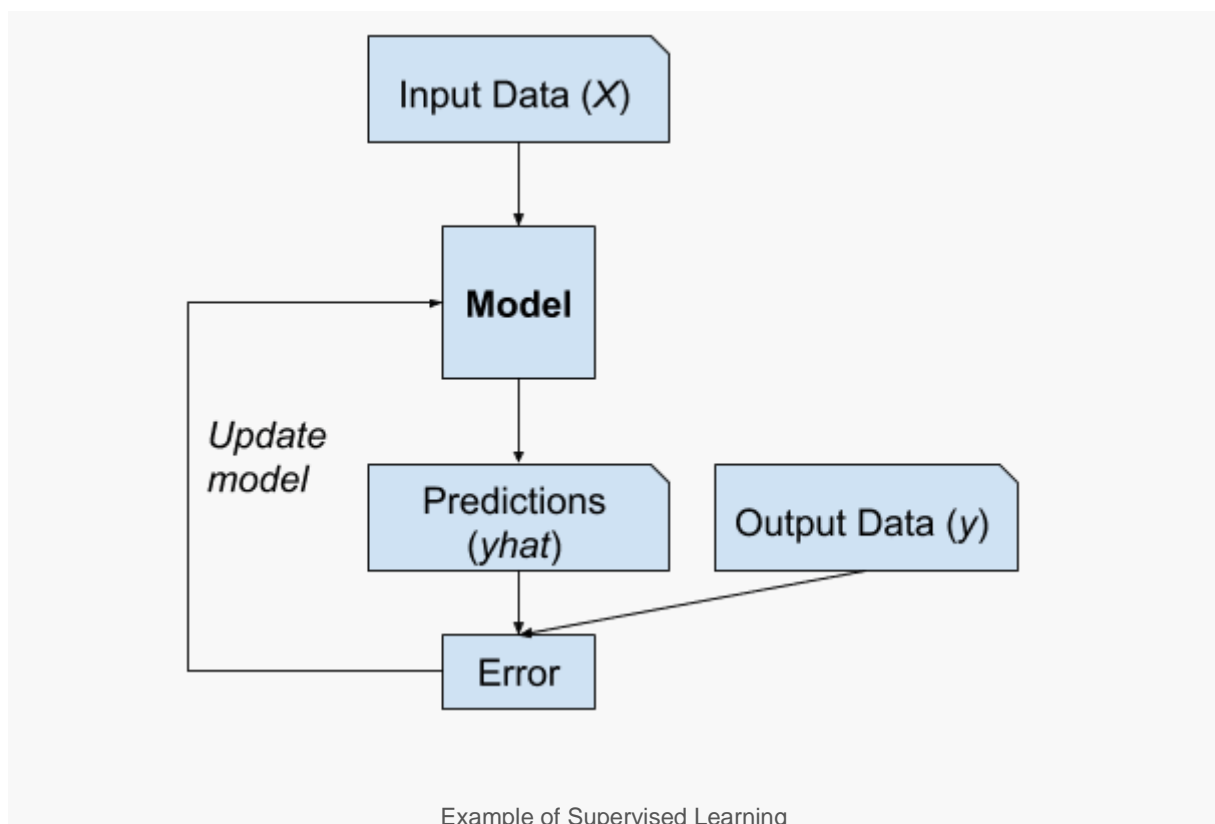
A typical machine learning problem involves using a model to make a prediction, e.g. [predictive modeling](#).

This requires a training dataset that is used to train a model, comprised of multiple examples, called samples, each with input variables (X) and output class labels (y). A model is trained by showing examples of inputs, having it predict outputs, and correcting the model to make the outputs more like the expected outputs.

In the predictive or supervised learning approach, the goal is to learn a mapping from inputs x to outputs y , given a labeled set of input-output pairs ...

— Page 2, [Machine Learning: A Probabilistic Perspective](#), 2012.

This correction of the model is generally referred to as a supervised form of learning, or supervised learning.



Examples of supervised learning problems include classification and regression, and examples of supervised learning algorithms include logistic regression and random forest.

There is another paradigm of learning where the model is only given the input variables (X) and the problem does not have any output variables (y).

A model is constructed by extracting or summarizing the patterns in the input data.

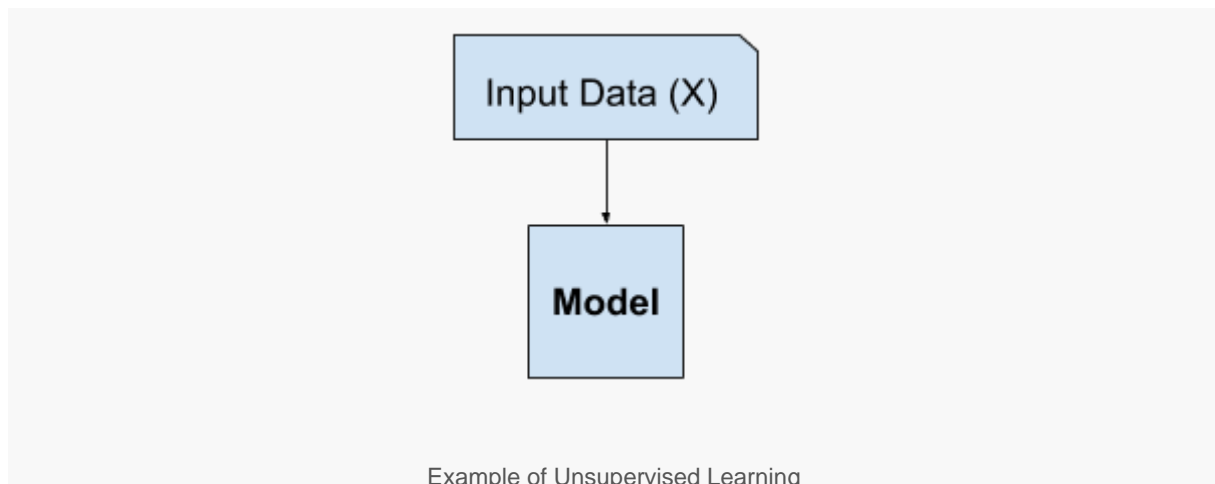
There is no correction of the model, as the model is not predicting anything.

The second main type of machine learning is the descriptive or unsupervised learning approach. Here we are only given inputs, and the goal is to find “interesting patterns” in

the data. [...] This is a much less well-defined problem, since we are not told what kinds of patterns to look for, and there is no obvious error metric to use (unlike supervised learning, where we can compare our prediction of y for a given x to the observed value).

— Page 2, [Machine Learning: A Probabilistic Perspective](#), 2012.

This lack of correction is generally referred to as an unsupervised form of learning, or unsupervised learning.



Examples of unsupervised learning problems include clustering and generative modeling, and examples of unsupervised learning algorithms are K-means and Generative Adversarial Networks.

Want to Develop GANs from Scratch?

Take my free 7-day email crash course now (with sample code).

Click to sign-up and also get a free PDF Ebook version of the course.

Download Your FREE Mini-Course

Discriminative vs. Generative Modeling

In supervised learning, we may be interested in developing a model to predict a class label given an example of input variables.

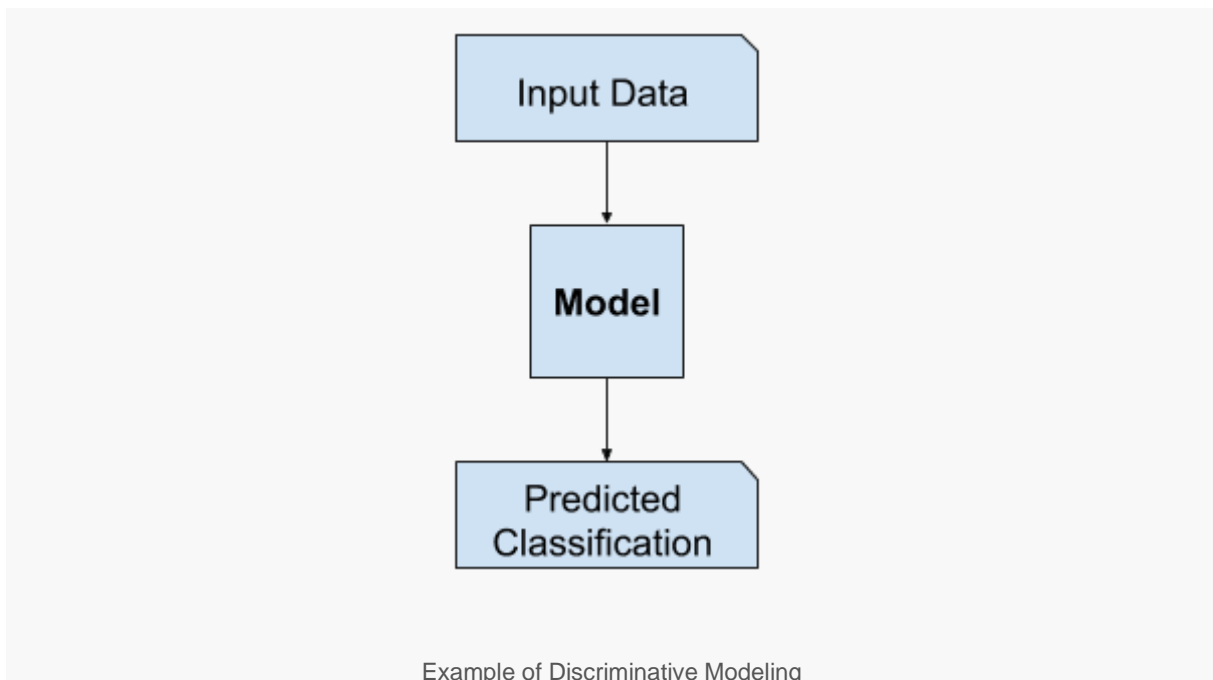
This predictive modeling task is called classification.

Classification is also traditionally referred to as discriminative modeling.

... we use the training data to find a discriminant function $f(x)$ that maps each x directly onto a class label, thereby combining the inference and decision stages into a single learning problem.

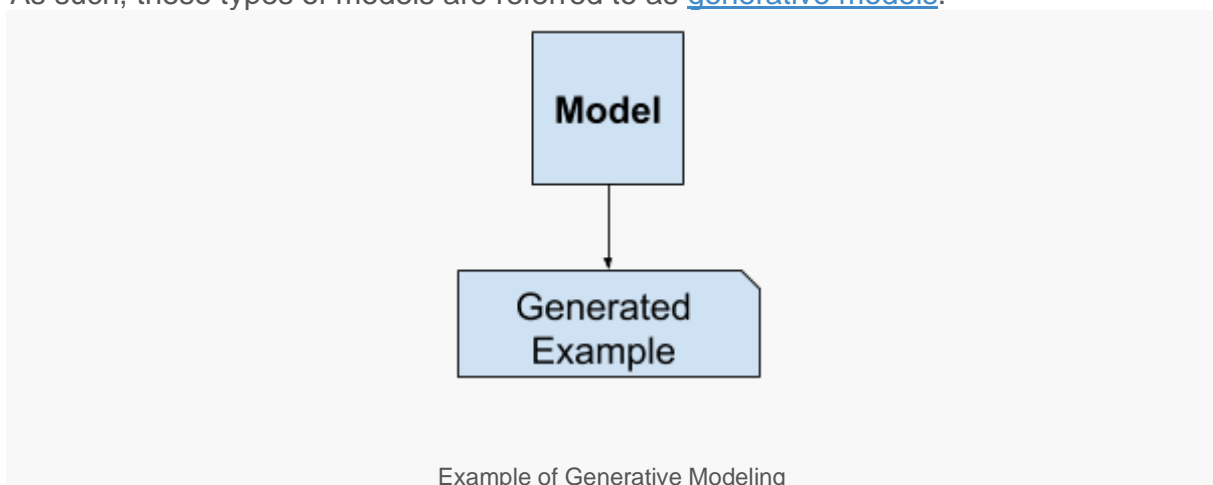
— Page 44, [Pattern Recognition and Machine Learning](#), 2006.

This is because a model must discriminate examples of input variables across classes; it must choose or make a decision as to what class a given example belongs.



Alternately, unsupervised models that summarize the distribution of input variables may be able to be used to create or generate new examples in the input distribution.

As such, these types of models are referred to as [generative models](#).



For example, a single variable may have a known data distribution, such as a [Gaussian distribution](#), or bell shape. A generative model may be able to sufficiently summarize this

data distribution, and then be used to generate new variables that plausibly fit into the distribution of the input variable.

Approaches that explicitly or implicitly model the distribution of inputs as well as outputs are known as generative models, because by sampling from them it is possible to generate synthetic data points in the input space.

— Page 43, [Pattern Recognition and Machine Learning](#), 2006.

In fact, a really good generative model may be able to generate new examples that are not just plausible, but indistinguishable from real examples from the problem domain.

Examples of Generative Models

[Naive Bayes](#) is an example of a generative model that is more often used as a discriminative model.

For example, Naive Bayes works by summarizing the probability distribution of each input variable and the output class. When a prediction is made, the probability for each possible outcome is calculated for each variable, the independent probabilities are combined, and the most likely outcome is predicted. Used in reverse, the probability distributions for each variable can be sampled to generate new plausible (independent) feature values.

Other examples of generative models include Latent Dirichlet Allocation, or LDA, and the Gaussian Mixture Model, or GMM.

Deep learning methods can be used as generative models. Two popular examples include the Restricted Boltzmann Machine, or RBM, and the Deep Belief Network, or DBN.

Two modern examples of deep learning generative modeling algorithms include the Variational Autoencoder, or VAE, and the Generative Adversarial Network, or GAN.

What Are Generative Adversarial Networks?

Generative Adversarial Networks, or GANs, are a deep-learning-based generative model.

More generally, GANs are a model architecture for training a generative model, and it is most common to use deep learning models in this architecture.

The GAN architecture was first described in the 2014 paper by [Ian Goodfellow](#), et al. titled "[Generative Adversarial Networks](#)."

A standardized approach called Deep Convolutional Generative Adversarial Networks, or DCGAN, that led to more stable models was later formalized by [Alec Radford](#), et al. in the 2015 paper titled “[Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks](#)”.

Most GANs today are at least loosely based on the DCGAN architecture ...

— [NIPS 2016 Tutorial: Generative Adversarial Networks](#), 2016.

The GAN model architecture involves two sub-models: a *generator model* for generating new examples and a *discriminator model* for classifying whether generated examples are real, from the domain, or fake, generated by the generator model.

- **Generator.** Model that is used to generate new plausible examples from the problem domain.
- **Discriminator.** Model that is used to classify examples as real (*from the domain*) or fake (*generated*).

Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator.

— Page 699, [Deep Learning](#), 2016.

The Generator Model

The generator model takes a fixed-length random vector as input and generates a sample in the domain.

The vector is drawn from randomly from a Gaussian distribution, and the vector is used to seed the generative process. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution.

This vector space is referred to as a latent space, or a vector space comprised of [latent variables](#). Latent variables, or hidden variables, are those variables that are important for a domain but are not directly observable.

A latent variable is a random variable that we cannot observe directly.

— Page 67, [Deep Learning](#), 2016.

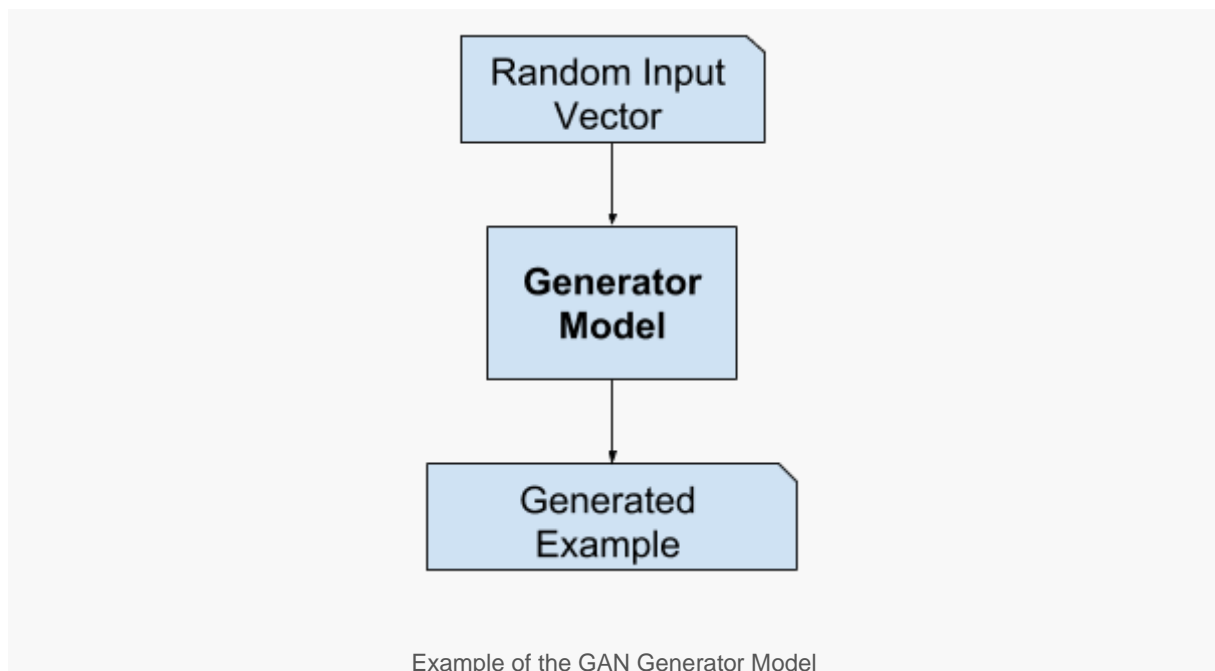
We often refer to latent variables, or a latent space, as a projection or compression of a data distribution. That is, a latent space provides a compression or high-level concepts of the observed raw data such as the input data distribution. In the case of GANs, the generator model applies meaning to points in a chosen latent space, such that new

points drawn from the latent space can be provided to the generator model as input and used to generate new and different output examples.

Machine-learning models can learn the statistical latent space of images, music, and stories, and they can then sample from this space, creating new artworks with characteristics similar to those the model has seen in its training data.

— Page 270, [Deep Learning with Python](#), 2017.

After training, the generator model is kept and used to generate new samples.



The Discriminator Model

The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated).

The real example comes from the training dataset. The generated examples are output by the generator model.

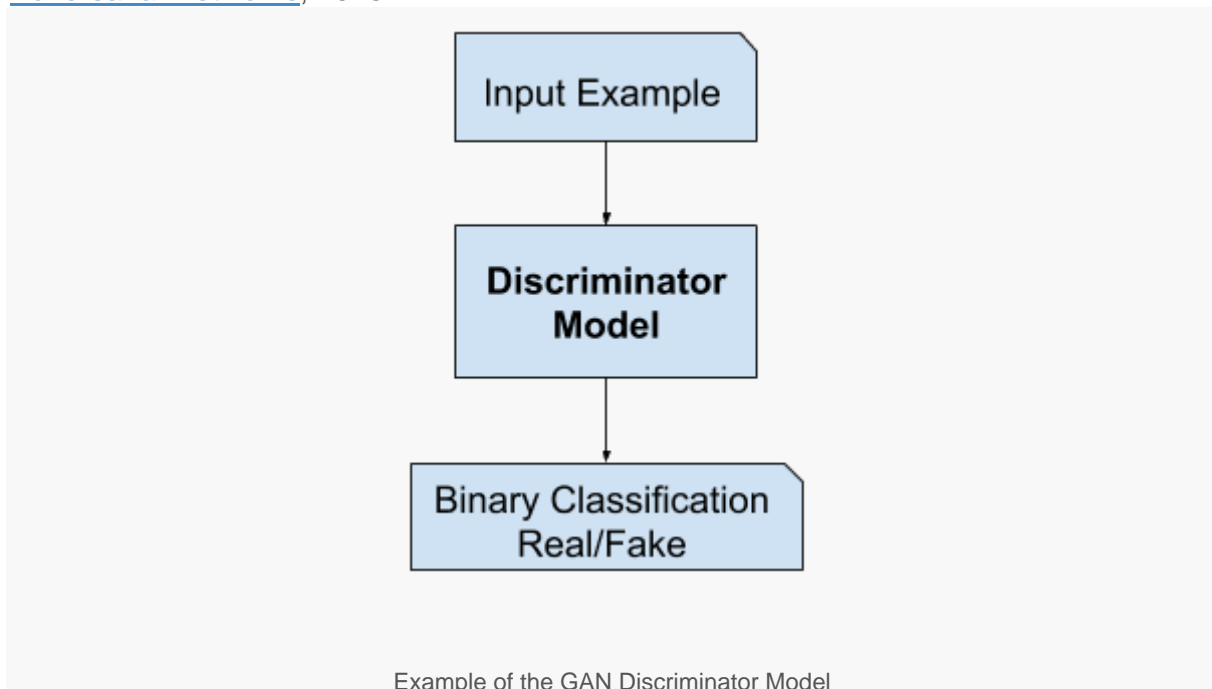
The discriminator is a normal (and well understood) classification model.

After the training process, the discriminator model is discarded as we are interested in the generator.

Sometimes, the generator can be repurposed as it has learned to effectively extract features from examples in the problem domain. Some or all of the feature extraction layers can be used in transfer learning applications using the same or similar input data.

We propose that one way to build good image representations is by training Generative Adversarial Networks (GANs), and later reusing parts of the generator and discriminator networks as feature extractors for supervised tasks

— [Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks](#), 2015.



GANs as a Two Player Game

Generative modeling is an unsupervised learning problem, as we discussed in the previous section, although a clever property of the GAN architecture is that the training of the generative model is framed as a supervised learning problem.

The two models, the generator and discriminator, are trained together. The generator generates a batch of samples, and these, along with real examples from the domain, are provided to the discriminator and classified as real or fake.

The discriminator is then updated to get better at discriminating real and fake samples in the next round, and importantly, the generator is updated based on how well, or not, the generated samples fooled the discriminator.

We can think of the generator as being like a counterfeiter, trying to make fake money, and the discriminator as being like police, trying to allow legitimate money and catch counterfeit money. To succeed in this game, the counterfeiter must learn to make money that is indistinguishable from genuine money, and the generator network must learn to create samples that are drawn from the same distribution as the training data.

— [NIPS 2016 Tutorial: Generative Adversarial Networks](#), 2016.

In this way, the two models are competing against each other, they are adversarial in the game theory sense, and are playing a [zero-sum game](#).

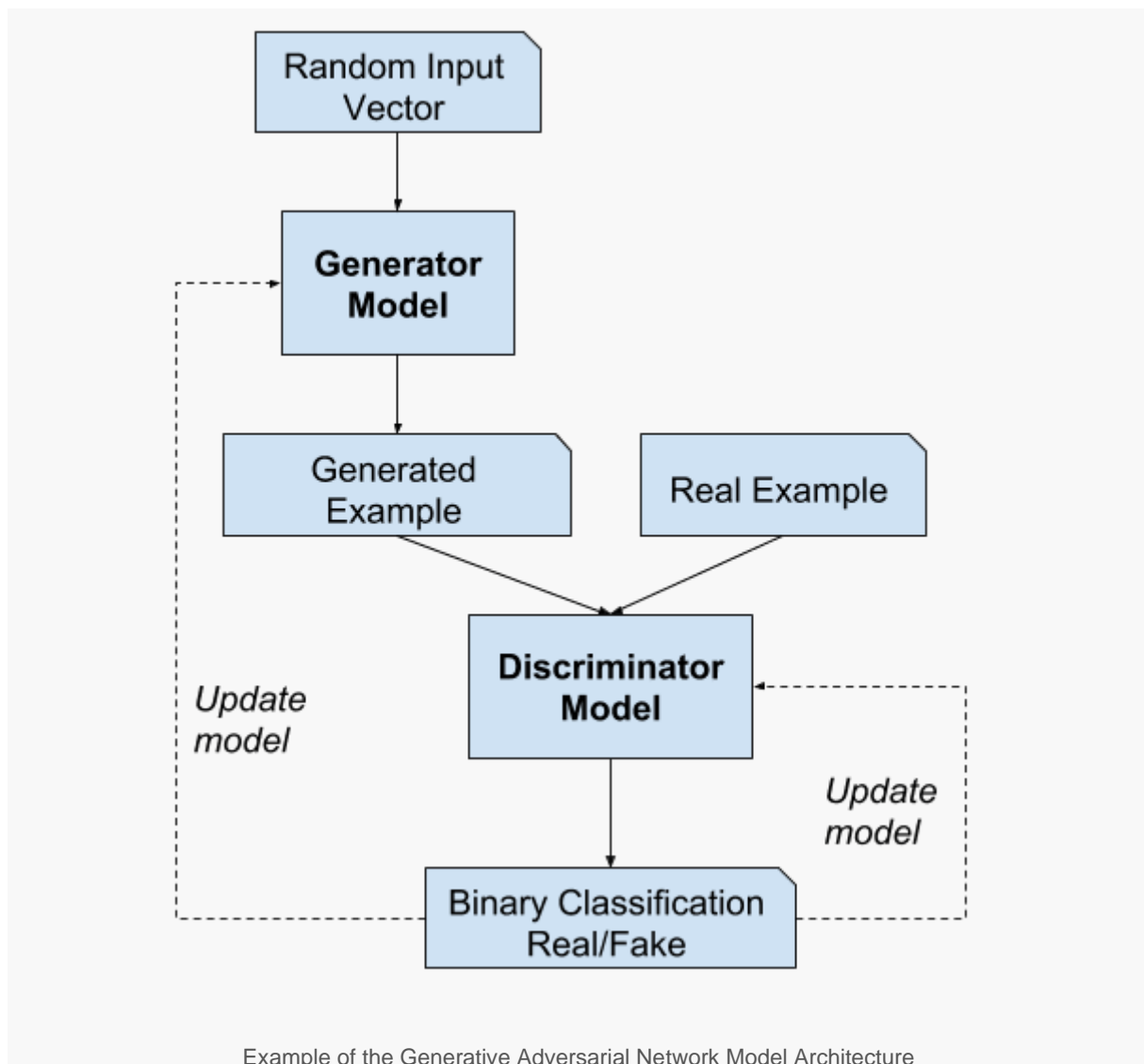
Because the GAN framework can naturally be analyzed with the tools of game theory, we call GANs “adversarial”.

— [NIPS 2016 Tutorial: Generative Adversarial Networks](#), 2016.

In this case, zero-sum means that when the discriminator successfully identifies real and fake samples, it is rewarded or no change is needed to the model parameters, whereas the generator is penalized with large updates to model parameters.

Alternately, when the generator fools the discriminator, it is rewarded, or no change is needed to the model parameters, but the discriminator is penalized and its model parameters are updated.

At a limit, the generator generates perfect replicas from the input domain every time, and the discriminator cannot tell the difference and predicts “unsure” (e.g. 50% for real and fake) in every case. This is just an example of an idealized case; we do not need to get to this point to arrive at a useful generator model.



Example of the Generative Adversarial Network Model Architecture

[training] drives the discriminator to attempt to learn to correctly classify samples as real or fake. Simultaneously, the generator attempts to fool the classifier into believing its samples are real. At convergence, the generator's samples are indistinguishable from real data, and the discriminator outputs 1/2 everywhere. The discriminator may then be discarded.

— Page 700, [Deep Learning](#), 2016.

GANs and Convolutional Neural Networks

GANs typically work with image data and use Convolutional Neural Networks, or CNNs, as the generator and discriminator models.

The reason for this may be both because the first description of the technique was in the field of computer vision and used CNNs and image data, and because of the remarkable progress that has been seen in recent years using CNNs more generally to achieve

state-of-the-art results on a suite of computer vision tasks such as object detection and face recognition.

Modeling image data means that the latent space, the input to the generator, provides a compressed representation of the set of images or photographs used to train the model. It also means that the generator generates new images or photographs, providing an output that can be easily viewed and assessed by developers or users of the model.

It may be this fact above others, the ability to visually assess the quality of the generated output, that has both led to the focus of computer vision applications with CNNs and on the massive leaps in the capability of GANs as compared to other generative models, deep learning based or otherwise.

Conditional GANs

An important extension to the GAN is in their use for conditionally generating an output.

The generative model can be trained to generate new examples from the input domain, where the input, the random vector from the latent space, is provided with (conditioned by) some additional input.

The additional input could be a class value, such as male or female in the generation of photographs of people, or a digit, in the case of generating images of handwritten digits.

Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information y . y could be any kind of auxiliary information, such as class labels or data from other modalities. We can perform the conditioning by feeding y into the both the discriminator and generator as [an] additional input layer.

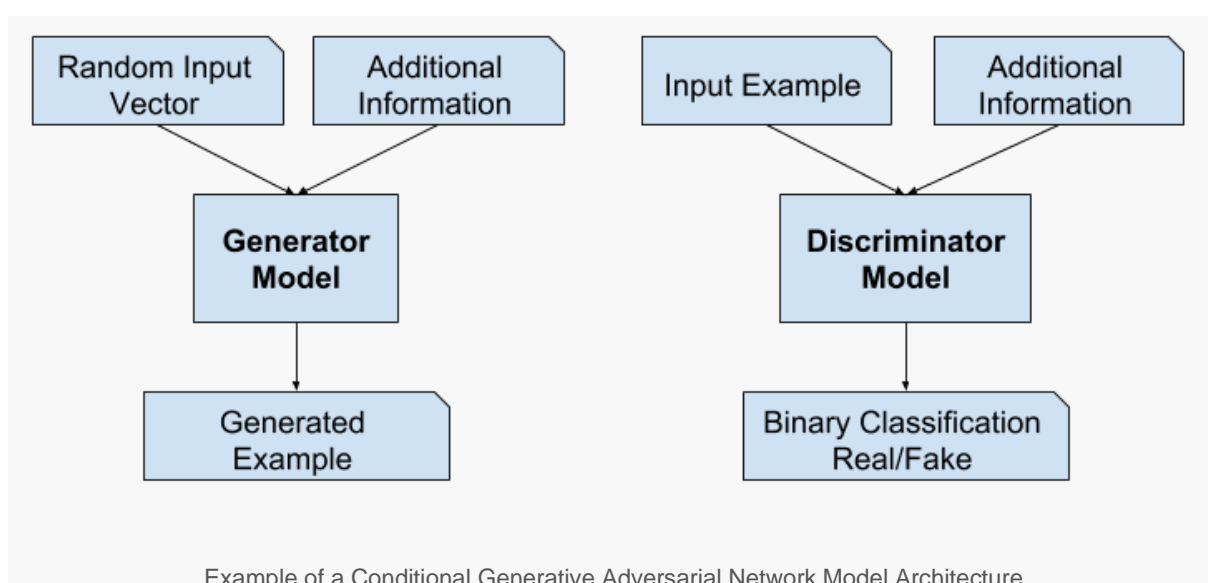
— [Conditional Generative Adversarial Nets](#), 2014.

The discriminator is also conditioned, meaning that it is provided both with an input image that is either real or fake and the additional input. In the case of a classification label type conditional input, the discriminator would then expect that the input would be of that class, in turn teaching the generator to generate examples of that class in order to fool the discriminator.

In this way, a conditional GAN can be used to generate examples from a domain of a given type.

Taken one step further, the GAN models can be conditioned on an example from the domain, such as an image. This allows for applications of GANs such as text-to-image translation, or image-to-image translation. This allows for some of the more impressive applications of GANs, such as style transfer, photo colorization, transforming photos from summer to winter or day to night, and so on.

In the case of conditional GANs for image-to-image translation, such as transforming day to night, the discriminator is provided examples of real and generated nighttime photos as well as (conditioned on) real daytime photos as input. The generator is provided with a random vector from the latent space as well as (conditioned on) real daytime photos as input.



Why Generative Adversarial Networks?

One of the many major advancements in the use of deep learning methods in domains such as computer vision is a technique called [data augmentation](#).

Data augmentation results in better performing models, both increasing model skill and providing a regularizing effect, reducing generalization error. It works by creating new, artificial but plausible examples from the input problem domain on which the model is trained.

The techniques are primitive in the case of image data, involving crops, flips, zooms, and other simple transforms of existing images in the training dataset.

Successful generative modeling provides an alternative and potentially more domain-specific approach for data augmentation. In fact, data augmentation is a simplified version of generative modeling, although it is rarely described this way.

In complex domains or domains with a limited amount of data, generative modeling provides a path towards more training for modeling. GANs have seen much success in this use case in domains such as deep reinforcement learning.

In complex domains or domains with a limited amount of data, generative modeling provides a path towards more training for modeling. GANs have seen much success in this use case in domains such as deep reinforcement learning.

Briefly explain GANs? What are their advantages?

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks.

Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake.

A Gentle Introduction to Generative Adversarial Networks (GANs)

by [Jason Brownlee](#) on June 17, 2019 in [Generative Adversarial Networks](#)

Tweet Tweet Share

Last Updated on July 19, 2019

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks.

Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way

that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake.

In this post, you will discover a gentle introduction to Generative Adversarial Networks, or GANs.

After reading this post, you will know:

- Context for GANs, including supervised vs. unsupervised learning and discriminative vs. generative modeling.
- GANs are an architecture for automatically training a generative model by treating the unsupervised problem as supervised and using both a generative and a discriminative model.
- GANs provide a path to sophisticated domain-specific data augmentation and a solution to problems that require a generative solution, such as image-to-image translation.

Kick-start your project with my new book [Generative Adversarial Networks with Python](#), including *step-by-step tutorials* and the *Python source code* files for all examples. Let's get started.



A Gentle Introduction to Generative Adversarial Networks (GANs)

Photo by [Barney Moss](#), some rights reserved.

Overview

This tutorial is divided into three parts; they are:

1. What Are Generative Models?
2. What Are Generative Adversarial Networks?
3. Why Generative Adversarial Networks?

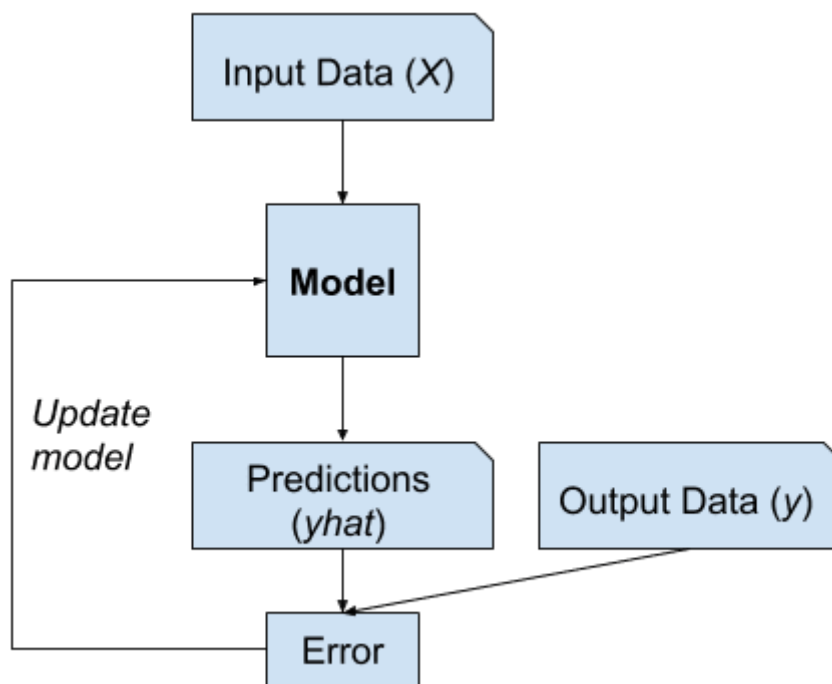
What Are Generative Models?

In this section, we will review the idea of generative models, stepping over the supervised vs. unsupervised learning paradigms and discriminative vs. generative modeling.

Supervised vs. Unsupervised Learning

A typical machine learning problem involves using a model to make a prediction, e.g. [predictive modeling](#).

This requires a training dataset that is used to train a model, comprised of multiple examples, called samples, each with input variables (X) and output class labels (y). A model is trained by showing examples of inputs, having it predict outputs, and correcting the model to make the outputs more like the expected outputs.



What Are Generative Adversarial Networks?

Generative Adversarial Networks, or GANs, are a deep-learning-based generative model.

More generally, GANs are a model architecture for training a generative model, and it is most common to use deep learning models in this architecture.

The GAN architecture was first described in the 2014 paper by [Ian Goodfellow](#), et al. titled “[Generative Adversarial Networks](#).”

A standardized approach called Deep Convolutional Generative Adversarial Networks, or DCGAN, that led to more stable models was later formalized by [Alec Radford](#), et al. in the 2015 paper titled “[Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks](#)”.

Most GANs today are at least loosely based on the DCGAN architecture ...

— [NIPS 2016 Tutorial: Generative Adversarial Networks](#), 2016.

The GAN model architecture involves two sub-models: a *generator model* for generating new examples and a *discriminator model* for classifying whether generated examples are real, from the domain, or fake, generated by the generator model.

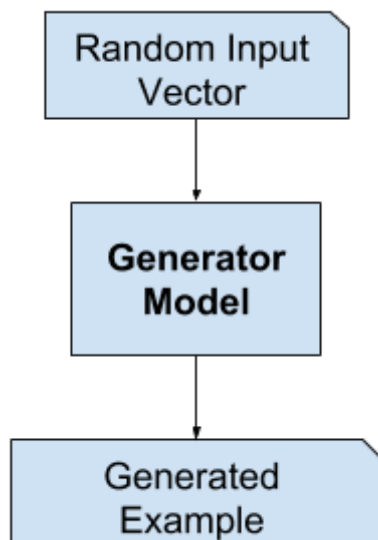
- **Generator.** Model that is used to generate new plausible examples from the problem domain.
- **Discriminator.** Model that is used to classify examples as real (*from the domain*) or fake (*generated*).

The Generator Model

The generator model takes a fixed-length random vector as input and generates a sample in the domain.

The vector is drawn from randomly from a Gaussian distribution, and the vector is used to seed the generative process. After training, points in this multidimensional vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution.

This vector space is referred to as a latent space, or a vector space comprised of [latent variables](#). Latent variables, or hidden variables, are those variables that are important for a domain but are not directly observable.



The Discriminator Model

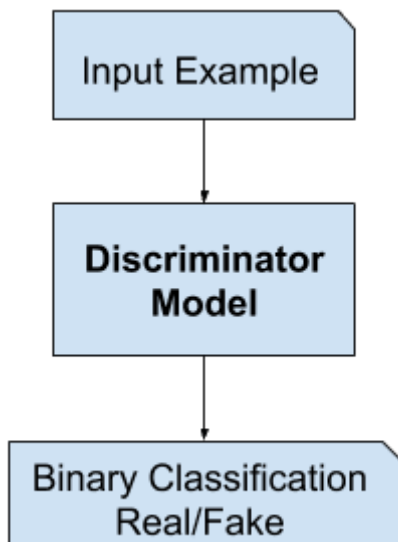
The discriminator model takes an example from the domain as input (real or generated) and predicts a binary class label of real or fake (generated).

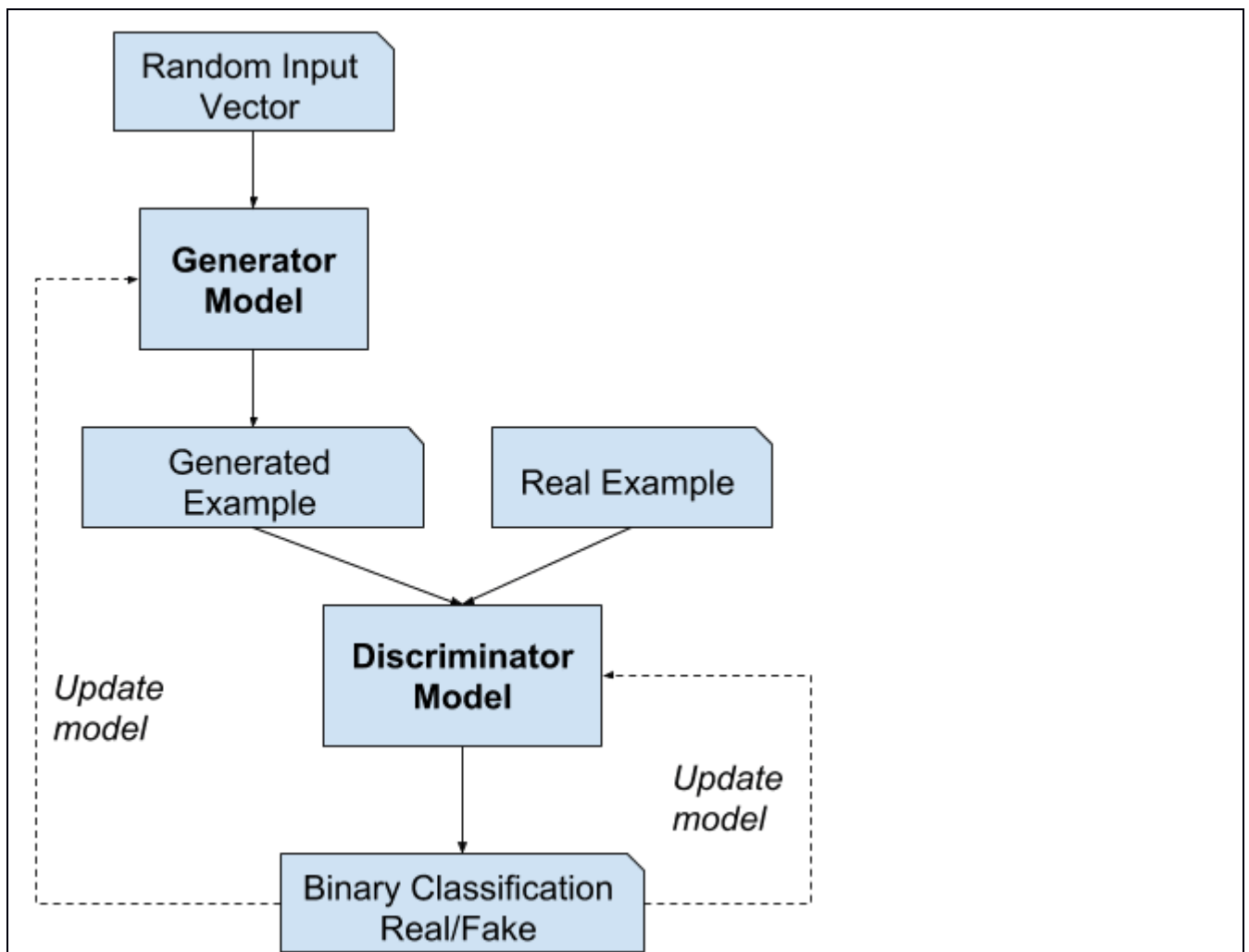
The real example comes from the training dataset. The generated examples are output by the generator model.

The discriminator is a normal (and well understood) classification model.

After the training process, the discriminator model is discarded as we are interested in the generator.

Sometimes, the generator can be repurposed as it has learned to effectively extract features from examples in the problem domain. Some or all of the feature extraction layers can be used in transfer learning applications using the same or similar input data.





GANs and Convolutional Neural Networks

GANs typically work with image data and use Convolutional Neural Networks, or CNNs, as the generator and discriminator models.

The reason for this may be both because the first description of the technique was in the field of computer vision and used CNNs and image data, and because of the remarkable progress that has been seen in recent years using CNNs more generally to achieve state-of-the-art results on a suite of computer vision tasks such as object detection and face recognition.

Modeling image data means that the latent space, the input to the generator, provides a compressed representation of the set of images or photographs used to train the model. It also means that the generator generates new images or photographs, providing an output that can be easily viewed and assessed by developers or users of the model.

It may be this fact above others, the ability to visually assess the quality of the generated output, that has both led to the focus of computer vision applications with CNNs and on the massive leaps in the capability of GANs as compared to other generative models, deep learning based or otherwise.

GANs and Convolutional Neural Networks

GANs typically work with image data and use Convolutional Neural Networks, or CNNs, as the generator and discriminator models.

The reason for this may be both because the first description of the technique was in the field of computer vision and used CNNs and image data, and because of the remarkable progress that has been seen in recent years using CNNs more generally to achieve state-of-the-art results on a suite of computer vision tasks such as object detection and face recognition.

Modeling image data means that the latent space, the input to the generator, provides a compressed representation of the set of images or photographs used to train the model. It also means that the generator generates new images or photographs, providing an output that can be easily viewed and assessed by developers or users of the model.

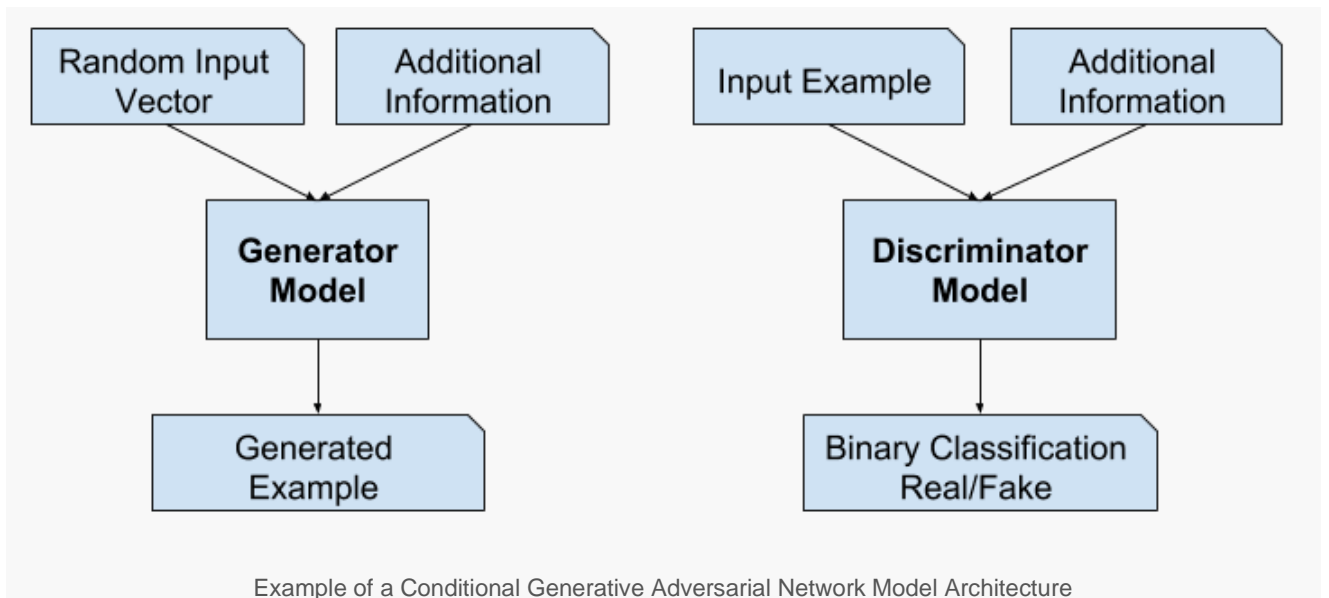
It may be this fact above others, the ability to visually assess the quality of the generated output, that has both led to the focus of computer vision applications with CNNs and on the massive leaps in the capability of GANs as compared to other generative models, deep learning based or otherwise.

The discriminator is also conditioned, meaning that it is provided both with an input image that is either real or fake and the additional input. In the case of a classification label type conditional input, the discriminator would then expect that the input would be of that class, in turn teaching the generator to generate examples of that class in order to fool the discriminator.

In this way, a conditional GAN can be used to generate examples from a domain of a given type.

Taken one step further, the GAN models can be conditioned on an example from the domain, such as an image. This allows for applications of GANs such as text-to-image translation, or image-to-image translation. This allows for some of the more impressive applications of GANs, such as style transfer, photo colorization, transforming photos from summer to winter or day to night, and so on.

In the case of conditional GANs for image-to-image translation, such as transforming day to night, the discriminator is provided examples of real and generated nighttime photos as well as (conditioned on) real daytime photos as input. The generator is provided with a random vector from the latent space as well as (conditioned on) real daytime photos as input.



Apr 2022

1 a) Which metric will be preferred for object detection ? Brief about it. 4

Average Precision (AP) and mean Average Precision (mAP) are the most popular metrics used to evaluate object detection models, such as Faster R_CNN, Mask R-CNN, and YOLO, among others. The same metrics have also been used to evaluate submissions in competitions like COCO and PASCAL VOC challenges.

$$\text{IoU} = \frac{\text{area}(gt \cap pd)}{\text{area}(gt \cup pd)}$$

Diagrammatically, IoU is defined as follows (the area of the intersection divided by the area of union between ground-truth and predicted box).

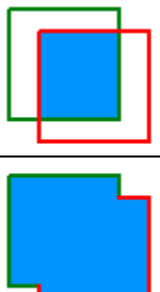
$$IOU = \frac{\text{area of overlap}}{\text{area of union}} = \frac{\text{area of intersection}}{\text{area of union}}$$


Fig 3: IoU (Source: Author)

IoU ranges between 0 and 1, where 0 shows no overlap, and 1 means perfect overlap between gt and pd . IoU metric is useful through thresholding; that is, we need a threshold (α , say) to determine whether detection is correct.

For the IoU threshold at α , True Positive(TP) is a detection for which $\text{IoU}(gt, pd) \geq \alpha$ and False Positive is a detection for which $\text{IoU}(gt, pd) < \alpha$. False Negative is a ground truth missed together with gt for which $\text{IoU}(gt, pd) < \alpha$.

Is that clear? If not, the following Figure should make the definitions clearer.

Considering the IoU threshold, $\alpha = 0.5$, then TP, FP and FNs can be identified as shown in Fig 4 below.

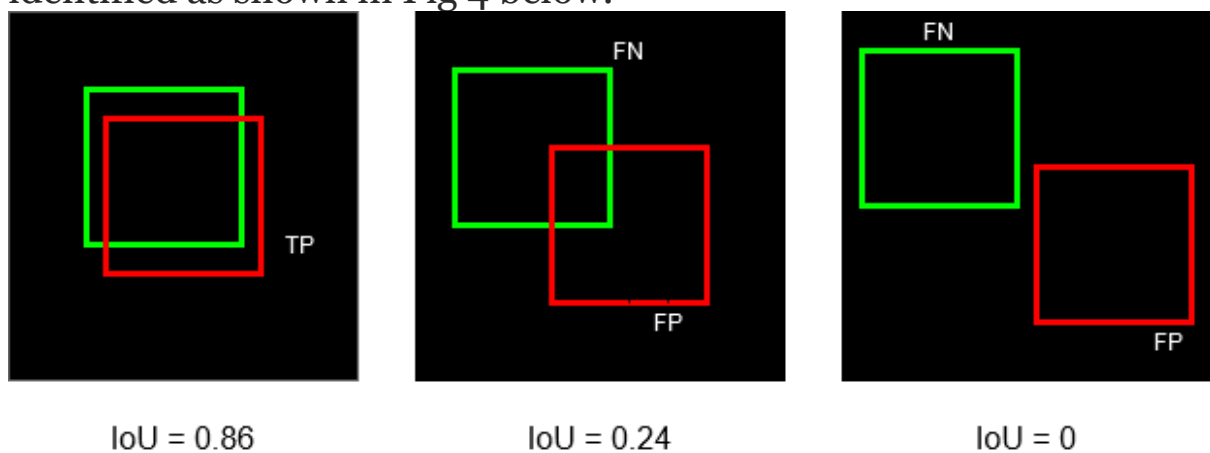


Fig 4: Identification of TP, FP and FN through IoU thresholding.

Note: If we raise the IoU threshold above 0.86, the first instance will be FP; if we lower the IoU threshold below 0.24, the second instance becomes TP.

Remark: The decision to mark a detection as TP or FP and ground-truth as FN is completely contingent on the choice of IoU threshold, α . For example, in the above Figure, if we lower the threshold below 0.24, then the detection in the second image becomes TP, and if the

IoU threshold is raised above 0.86, the detection on the first image will be FP.

Precision and Recall

Precision is the degree of exactness of the model in identifying only relevant objects. It is the ratio of TPs over all detections made by the model.

Recall measures the ability of the model to detect all ground truths— proportion of TPs among all ground truths.

$$P = \frac{TP}{TP + FP} = \frac{TP}{\text{all detections}}$$
$$R = \frac{TP}{TP + FN} = \frac{TP}{\text{all ground-truths}}$$

Equation 1: Precision and Recall

A model is said to be good if it has high precision and high recall. A perfect model has zero FNs and zero FPs (precision=1 and recall=1). Often, attaining a perfect model is not feasible.

Precision x Recall Curve (PR Curve)

Heads up: Before going on, look at this Figure and read its caption.



This is a detection made using [Mask R-CNN](#). It shows a bounding box(dotted), segmentation mask, class (fruit) and confidence score(0.999). The object detection model typically outputs the bounding box, confidence score and class. The confidence value is the model's confidence in the detection, which ranges between 0 and 1.

Just like IoU described earlier, confidence scoring also relies on the threshold. Raising the confidence score threshold means that more objects will be missed by the model (more FNs and, therefore, low recall and high precision), whereas a low confidence score will mean that the model gets more FPs (hence low precision and high recall). This means we need to develop some trade-offs for precision and recall.

The precision-recall (PR) curve is a plot of precision and recall at varying confidence values. For a good model, precision and recall stay high even when the confidence score varies.

Average Precision

$AP@α$ is Area Under the Precision-Recall Curve(AUC-PR) evaluated at $α$ IoU threshold. Formally, it is defined as follows.

$$AP@α = \int_0^1 p(r) dr$$

Equation 2: Definition of Average Precision

Notation: $AP@α$ or $APα$ means that AP precision is evaluated at $α$ IoU threshold. If you see metrics like AP_{50} and A_{75} , they mean AP calculated at $IoU=0.5$ and $IoU=0.75$, respectively.

A high Area Under the PR Curve means high recall and high precision. Naturally, the PR curve is a zig-zag-like plot. That means that it is not monotonically decreasing. We can remove this property using interpolation methods. We will discuss two of those interpolation methods below:

1. *11-point interpolation method*
2. *All-point interpolation approach*

11-point interpolation method

An 11-point AP is a plot of interpolated precision scores for model results at 11 equally spaced standard recall levels, namely, 0.0, 0.1, 0.2, . . . 1.0. It is defined as

$$AP@_{\alpha 11} = \frac{1}{11} \sum_{r \in R} p_{interp}(r),$$

Equation 3: 11-point interpolation formula

where, $R = \{0.0, 0.1, 0.2, \dots 1.0\}$ and

$$p_{interp}(r) = \max_{r': r' \geq r} p(r')$$

Equation 4: Supporting equation 3

that is, **interpolated precision at recall value, r** — It is the highest precision for any recall value $r' \geq r$. If this one doesn't make sense, I promise you it will make full sense once we go through an example.

All — point interpolation method

In this case, interpolation is done for all the positions (recall values), that is,

$$AP@_{\alpha} = \sum_i (r_{i+1} - r_i) p_{interp}(r_{i+1}),$$

$$\text{where, } p_{interp}(r_{i+1}) = \max_{r': r' \geq r_{i+1}} p(r')$$

Equation 5: All—point interpolation method

Mean Average Precision (mAP)

Remark (AP and the number of classes): AP is calculated individually for each class. This means that there are as many AP values as the number of classes (loosely). These AP values are averaged to obtain the **mean Average Precision (mAP)** metric.

Definition: The mean Average Precision (mAP) is the average of AP values over all classes.

$$\text{mAP}@_{\alpha} = \frac{1}{n} \sum_{i=1}^n \text{AP}_i \quad \text{for } n \text{ classes.}$$

Remark (AP and IoU): As mentioned earlier, AP is calculated at a given IoU threshold, α . With this reasoning, AP can be calculated over a range of thresholds. [Microsoft COCO](#) calculated the AP of a given category/class at 10 different IoU ranging from 50% to 95% at 5% step-size, usually denoted $\text{AP}@[.50:.5:.95]$. [Mask R-CNN](#) reports the average of $\text{AP}@[.50:.5:.95]$ simply as AP. It says

“We report the standard COCO metrics including AP (averaged over IoU thresholds), AP50, AP75, and APS, APM, APL(AP at different scales)” — Extract from [Mask R-CNN paper](#)

To make all these things clearer, let us go through an example.

Example

Consider the 3 images shown in Figure 5 below. They contain 12 detection (red boxes) and 9 ground truths (green). Each detection has a class marked by a letter and the model confidence. In this example, consider that all the detections are of the same object class, and the IoU threshold is set $\alpha = 50$ per cent. IoU values are shown in Table 1 below.

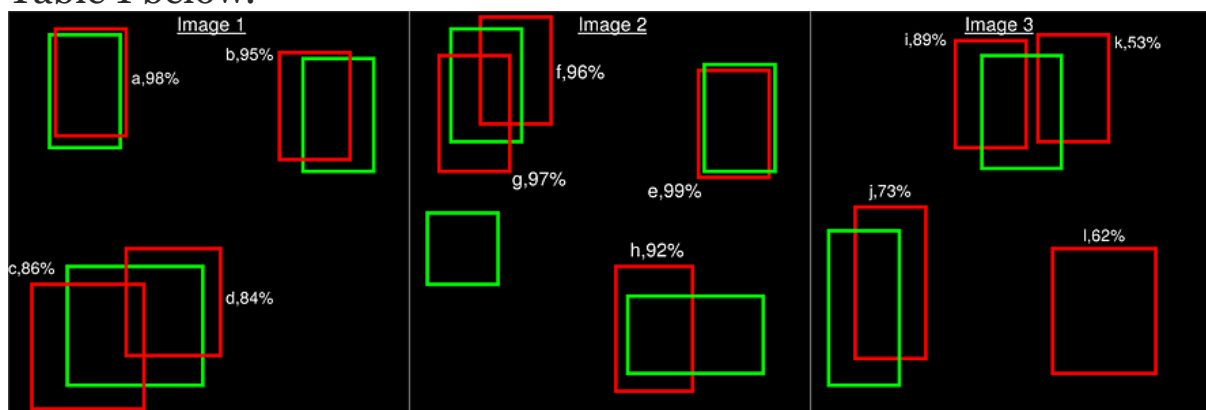


Fig 5: A model detecting objects of the same class. There are 12 detections and 9 ground truths.

Remark (Multiple detections): e.g. **c,d in image 1**, **g,f in image 2** and **i,k in image 3**. For multiple detections like that, a detection with the highest confidence is labelled TP, and all other detections are marked as FPs, provided that the detection has an $\text{IoU} \geq \text{threshold}$ with the truth box. This means that:

- **c and d become FPs** because none of them meets the threshold requirement. *c* and *d* have 47% and 42% IoUs, respectively, against the required 50%.

- **g is a TP, and f is FP.** Both have IoUs greater than 50%, but *g* has a higher confidence of 97% against the confidence of 96% for *f*.
- What about **i** and **k**?

Multiple detections of the same object in an image were considered false detections e.g. 5 detections of a single object counted as 1 correct detection and 4 false detections
— **Source: PASCAL VOC 2012 paper.**

Some detectors can output multiple detections overlapping a single ground truth. For those cases the detection with the highest confidence is considered a TP and the others are considered as FP, as applied by the PASCAL VOC 2012 challenge. -Source: **A Survey on Performance Metrics for Object-Detection Algorithms paper.**

detection	confidence	TP	FP	cumTP	cumFP	all_detections	precision	recall	IoU
e	99	1	0	1	0	1	1	0.11	97
a	98	1	0	2	0	2	1	0.22	92
g	97	1	0	3	0	3	1	0.33	87
f	96	0	1	3	1	4	0.75	0.33	82
b	95	1	0	4	1	5	0.8	0.44	73
h	92	0	1	4	2	6	0.67	0.4	48
i	89	1	0	5	2	7	0.71	0.56	66
c	86	0	1	5	3	8	0.63	0.56	47
d	84	0	1	5	4	9	0.56	0.56	42
j	73	1	0	6	4	10	0.6	0.67	54
l	62	0	1	6	5	11	0.55	0.67	45
k	53	0	1	6	6	12	0.5	0.67	38

Table 1: IoU threshold, $\alpha = 50\%$. cumTP and cumFP are cumulative values for TP and FP columns, respectively. all_detections=cumTP+cumFP.

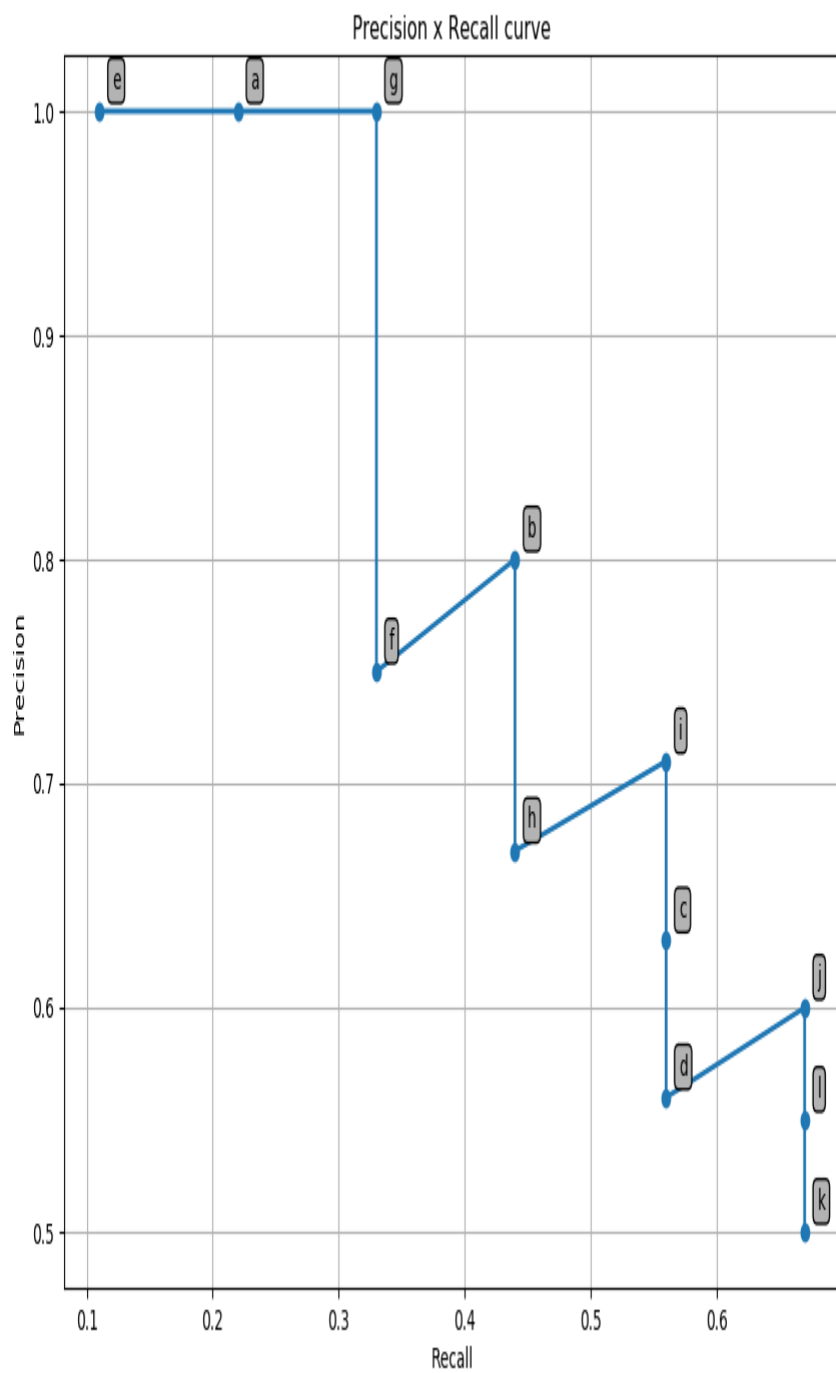
Important: Before filling up *cumTP*, *cumFP*, *all_detection*, *precision*, and *recall*, you need to sort the table values by confidence

in descending order. precision is $\text{cumTP}/\text{all_detections}$ and recall is $\text{cumTP}/\text{number_of_ground_truths}$. We have nine ground truths.

11-point interpolation

To calculate the approximation of $AP@0.5$ using 11-point interpolation, we need to average the precision values for recall values in R (see Equation 3), that is, for recall values 0.0, 0.1, 0.2, . . . 1.0. as shown in Fig 6 Right below.

$$AP@50\%_{11} = \frac{1}{11}(1 + 1 + 1 + 1 + 0.8 + 0.71 + 0.6 + 0 + 0 + 0 + 0) = 55.55\%$$



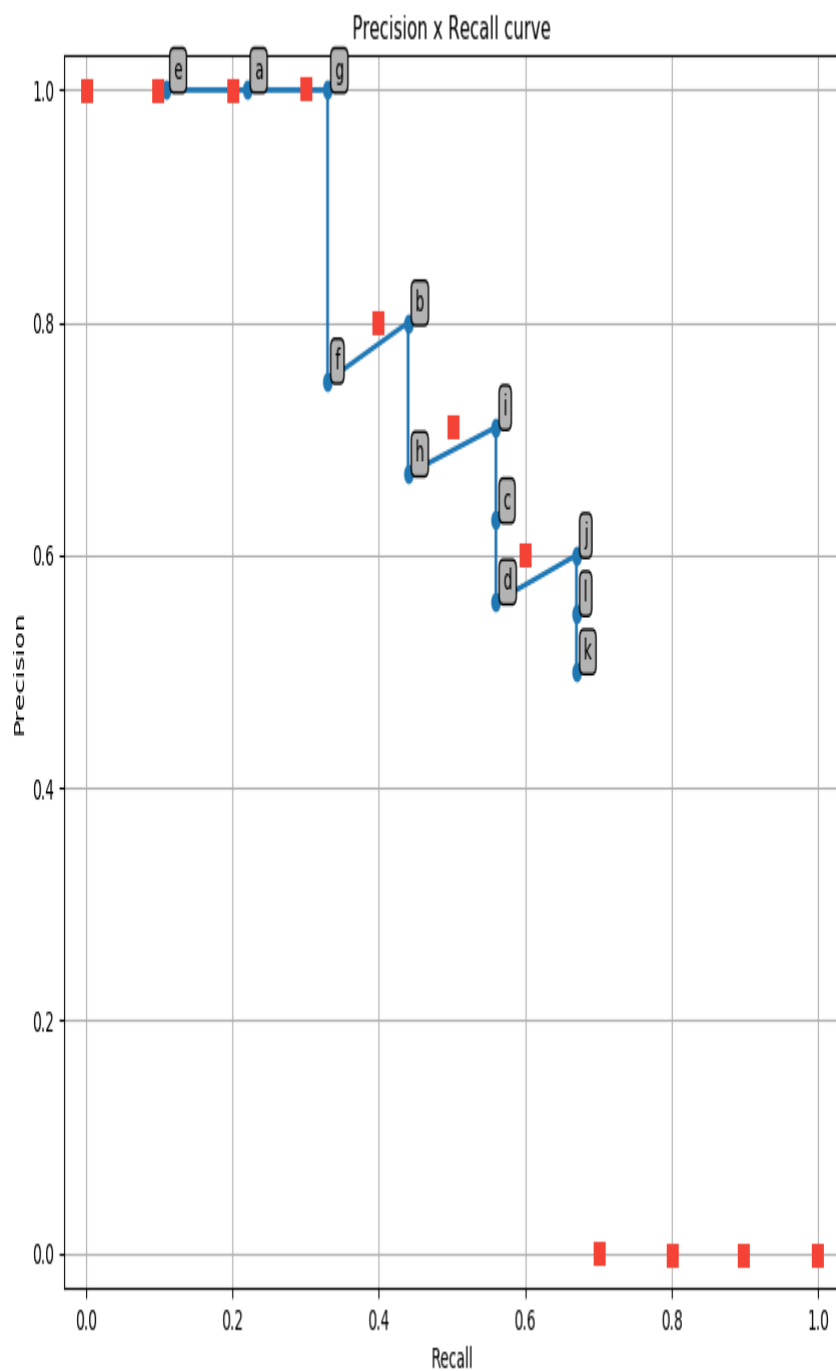
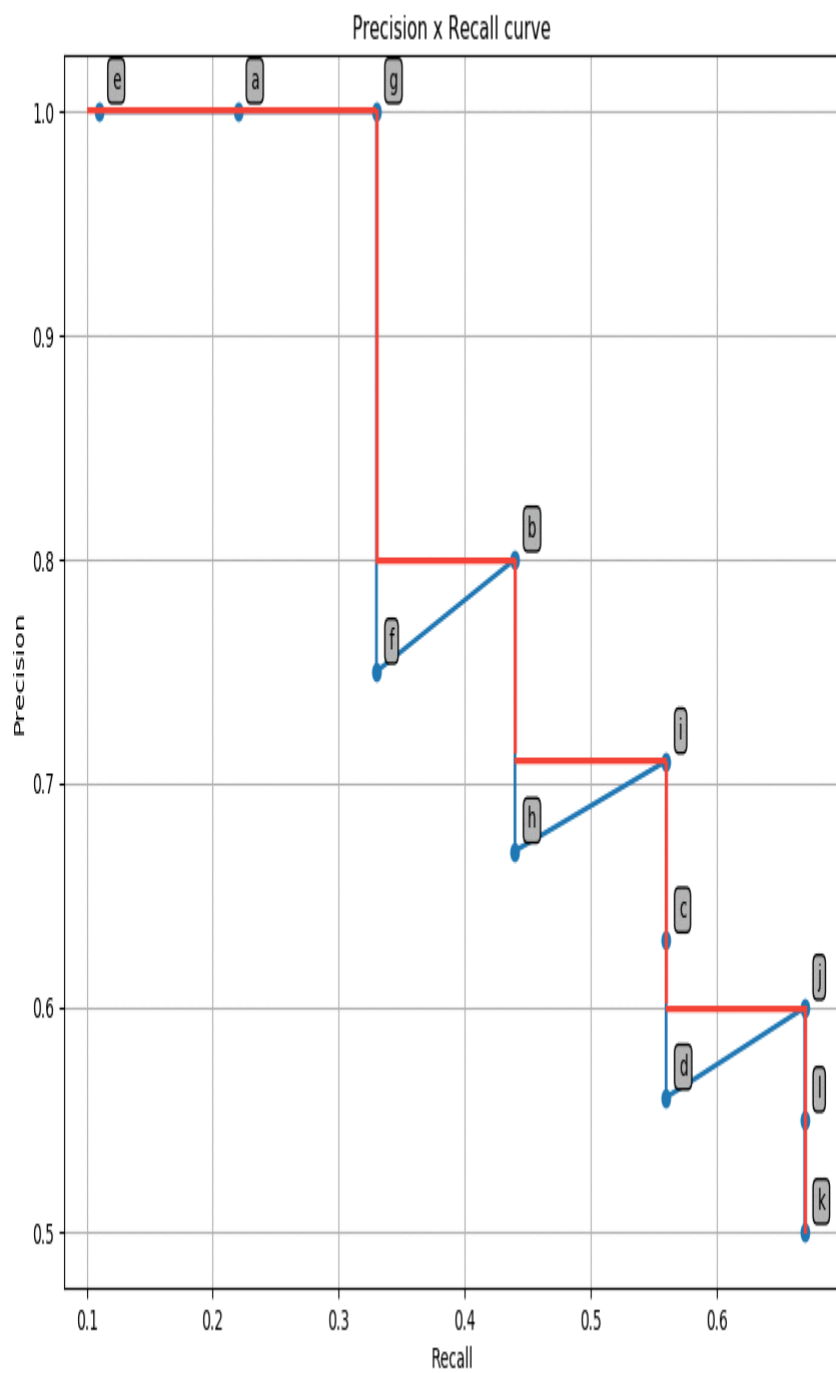


Fig 6: Left: Plot of all precision-recall values. Right: 11 precision values matching the following recall values: 0.0, 0.1, ..., 1.0.

All-point interpolation

From the definition in Equation 5, we can calculate AP@50 using all-point interpolation as follows.

$$\begin{aligned} AP@50 &= 1 * (0.33 - 0) + 0.8 * (0.44 - 0.33) + 0.71 * (0.56 - 0.44) + 0.6 * (0.67 - 0.56) \\ &= 56.92\% \end{aligned}$$



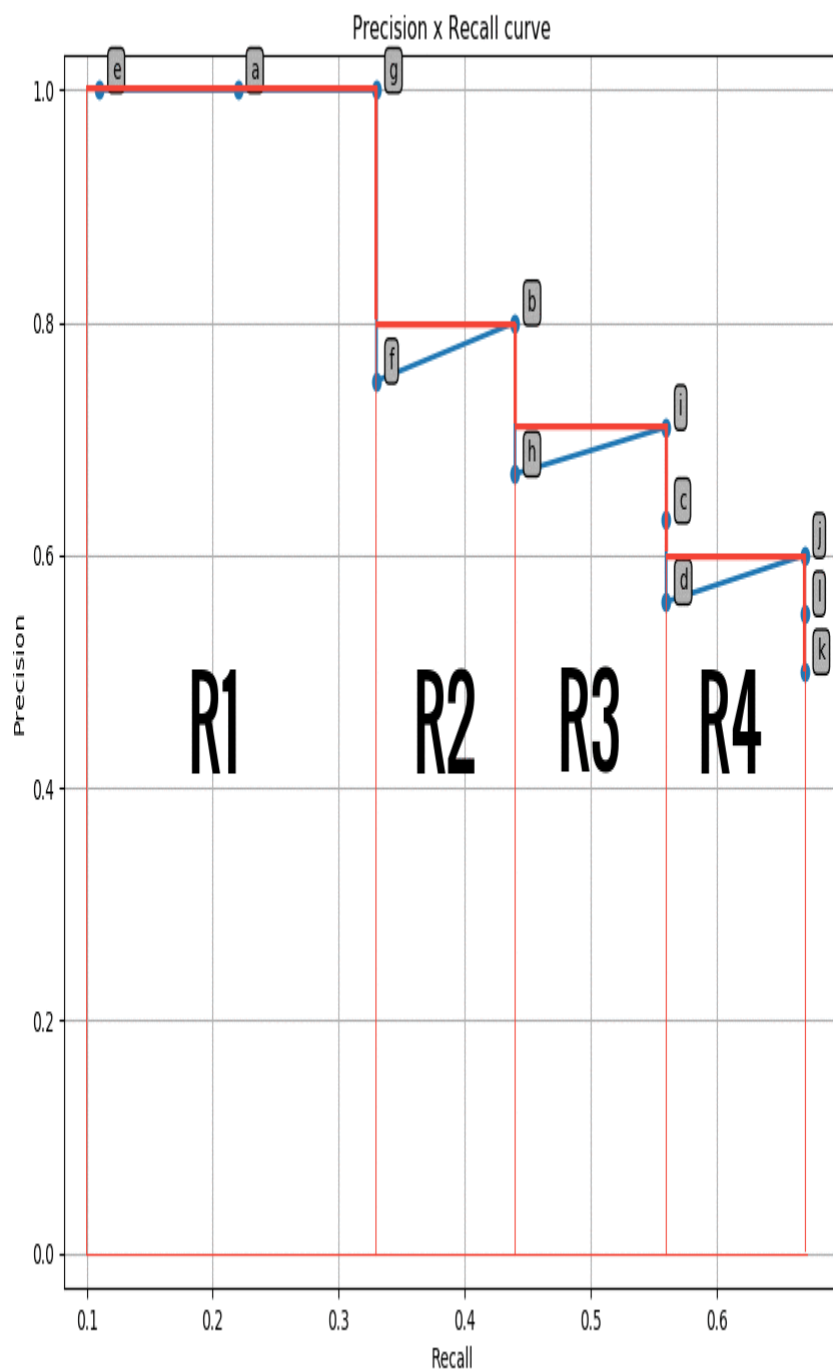


Fig 7: Left: All point interpolation curve(in red) overlaying the original PR curve. Right: Regions for all-point interpolation

Simply put, all-point interpolation involves calculating and summing area values of the four regions (R1, R2, R3 and R4) in Figure 1.3b, that is,

$$AP@50 = \text{Area}(R1) + \text{Area}(R2) + \text{Area}(R3) + \text{Area}(R4),$$

where,

$$\text{Area}(R1) = 1 * (0.33 - 0) = 0.33$$

$$\text{Area}(R2) = 0.8 * (0.44 - 0.33) = 0.088$$

$$\text{Area}(R3) = 0.71 * (0.56 - 0.44) = 0.0852$$

$$\text{Area}(R4) = 0.6 * (0.67 - 0.56) = 0.066$$

and thus $AP@50 = 56.92\%$.

And that is all!

Remark: Recall that we said AP is calculated for each class. Our calculations are for one class. For several classes, we can calculate mAP by simply taking the average of the resulting AP values for the different classes.

b) What is the role of Max-pool layers in CNN? 4

Introducing Max Pooling

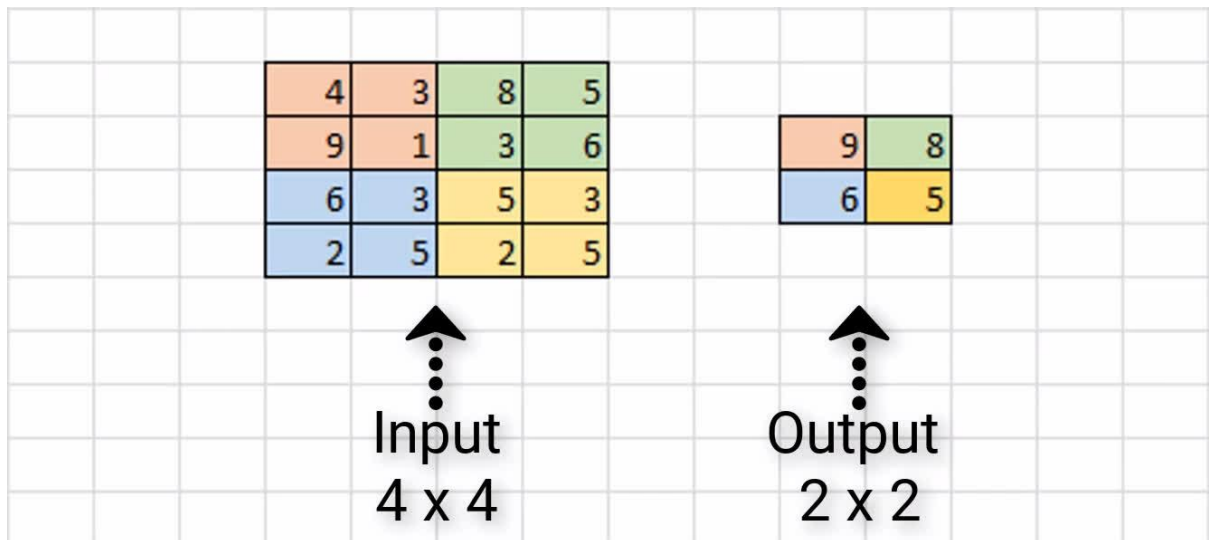
Max pooling is a type of operation that is typically added to CNNs following individual convolutional layers.

When added to a model, max pooling reduces the dimensionality of images by reducing the number of pixels in the output from the previous convolutional layer.

Let's go ahead and check out a couple of examples to see what exactly max pooling is doing operation-wise, and then we'll come back to discuss why we may want to use max pooling.

*Example Using A Sample From The MNIST Dataset
Scaled Down Example*

Suppose we have the following:



We have some sample input of size 4×4 , and we're assuming that we have a 2×2 filter size with a stride of 2 to do max pooling on this input channel.

Our first 2×2 region is in orange, and we can see the max value of this region is 9, and so we store that over in the output channel.

Next, we slide over by 2 pixels, and we see the max value in the green region is 8. As a result, we store the value over in the output channel.

Since we've reached the edge, we now move back over to the far left, and go down by 2 pixels. Here, the max value in the blue region is 6, and we store that here in our output channel.

Finally, we move to the right by 2, and see the max value of the yellow region is 5. We store this value in our output channel.

This completes the process of max pooling on this sample 4×4 input channel, and the resulting output channel is this 2×2 block. As a result, we can see that our input dimensions were again reduced by a factor of two.

Alright, we know what max pooling is and how it works, so let's discuss why would we want to add this to our network?

Why Use Max Pooling?

There are a couple of reasons why adding max pooling to our network may be helpful.

Reducing Computational Load

Since max pooling is reducing the resolution of the given output of a convolutional layer, the network will be looking at larger areas of the image at a time going forward, which reduces the amount of parameters in the network and consequently reduces computational load.

Reducing Overfitting

Additionally, max pooling may also help to reduce overfitting. The intuition for why max pooling works is that, for a particular image, our network will be looking to extract some particular features.

Maybe, it's trying to identify numbers from the MNIST dataset, and so it's looking for edges, and curves, and circles, and such. From the output of the convolutional layer, we can think of the higher valued pixels as being the ones that are the most activated.

With max pooling, as we're going over each region from the convolutional output, we're able to pick out the most activated pixels and preserve these high values going forward while discarding the lower valued pixels that are not as activated.

Just to mention quickly before going forward, there are other types of pooling that follow the exact same process we've just gone through, except for that it does some other operation on the regions rather than finding the max value.

Average Pooling

For example, average pooling is another type of pooling, and that's where you take the average value from each region rather than the max.

Currently max pooling is used vastly more than average pooling, but I did just want to mention that point.

c) List and brief about the activation functions in neural networks? 4 – **see above – search for activation functions**

d) Briefly explain the concept of Non-Maximum Suppression 4

Non Maximum Suppression

Edit

Non Maximum Suppression is a computer vision method that selects a single entity out of many overlapping entities (for example bounding boxes in object detection). The criteria is usually discarding entities that are below a given probability bound. With remaining entities we repeatedly pick the entity with the highest probability, output that as the prediction, and discard any remaining box where a $\text{IoU} \geq 0.5$ with the box output in the previous step.

What is Non Max Suppression, and why is it used?

Non max suppression is a technique used mainly in object detection that aims at selecting the best bounding box out of a set of overlapping boxes. In the following image, the aim of non max suppression would be to remove the yellow, and blue boxes, so that we are left with only the green box.

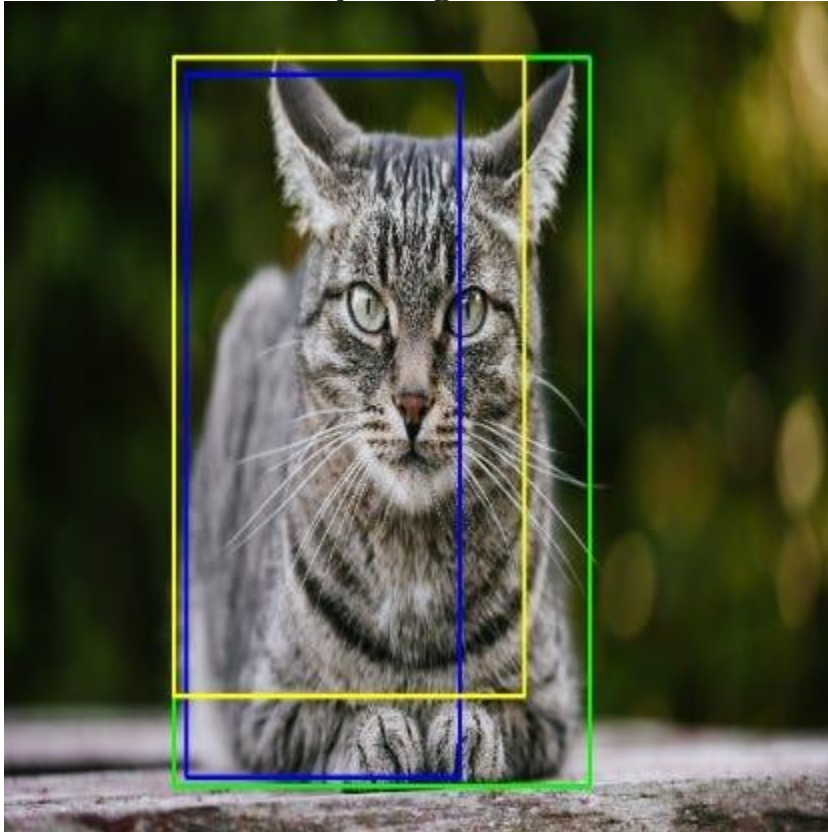


Figure 1: Multiple overlapping boxes for the same object

Procedure for calculating NMS:

To get an overview of what a bounding box is, and what IOU means, I have made two posts on the same. ([Bounding Box](#), and [IOU](#)). The terms and parameters described in the two articles are carried forward in this post. I will first go ahead and describe the procedure of NMS for this particular example, and then explain a more generalized algorithm extending it for different classes.

Explaining the terms used:

- The format for each bounding box that we will use is as follows:
`bbox = [x1, y1, x2, y2, class, confidence]`.
- Let us assume that for this particular image we have a list 3 bounding boxes; i.e `bbox_list = [blue_box, yellow_box, green_box]`.
- `green_box = [x1, y1, x2, y2, "Cat", 0.9]`
`yellow_box = [x5, y5, x6, y6, "Cat", 0.75]`
`blue_box = [x3, y3, x4, y4, "Cat", 0.85]`

Stage 1 (Initial removal of boxes):

- As the first step in NMS, we sort the boxes in descending order of confidences. This gives us:
`bbox_list = [green_box, blue_box, yellow_box]`
- We then define a confidence threshold. Any box that has a confidence below this threshold will be removed. For this example, let us assume a confidence threshold of **0.8**. Using this threshold, we would remove the yellow box as its confidence is < 0.8 . This leaves us with:
`bbox_list = [green_box, blue_box]`

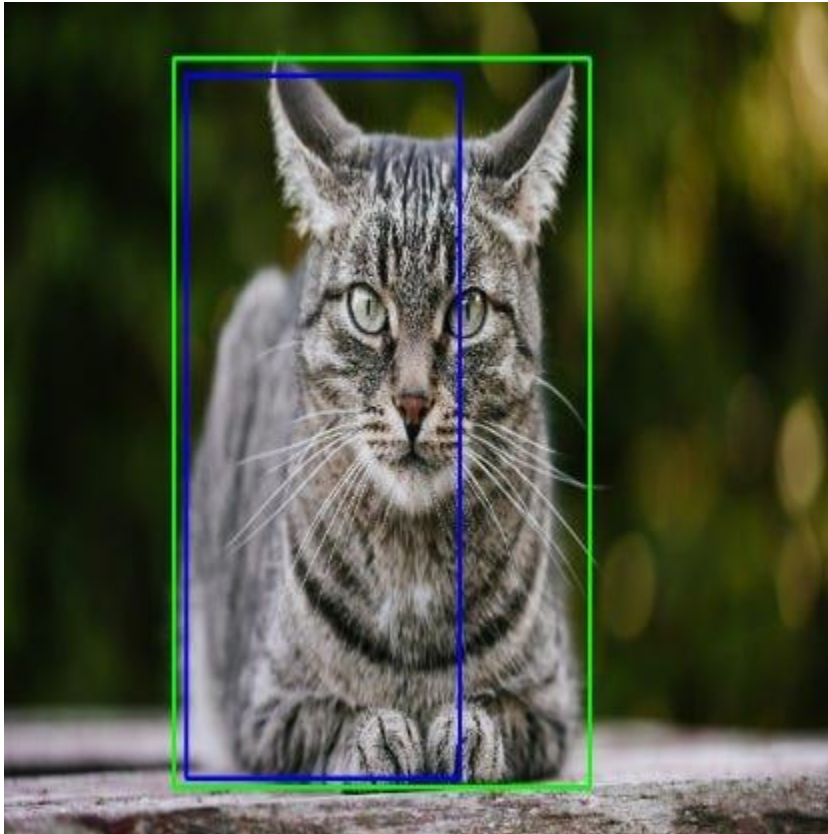


Figure 2: Yellow box removed

Stage 2 (IOU Comparision of boxes):

- Since the boxes are in descending order of confidence, we know that the first box in the list has the highest confidence. We remove this first box from the list and add it to a new list. In our case we would remove the green box, and put it into a new list, say `bbox_list_new`.
- At this stage, we define an additional threshold for IOU. This threshold is used to remove boxes that have a high overlap. The reasoning behind it is as follows: If two boxes have a significant amount of overlap, and they also belong to the same class, it is highly likely that both the boxes are covering the same object (We can verify

this from Figure 2). Since the aim is to have one box per object, we try remove the box that has a lower confidence.

- For our example, let us say that our IOU threshold is **0.5**
- We now start calculating the IOU of the green box with every remaining box in the `bbox_list` that also has the same class. In our case we would calculate the IOU of the green box only with the blue box.
- If the IOU of the green box and the blue box is greater than the threshold we defined of 0.5, we would remove the blue box, since it has a lower confidence, and also has significant overlap.
- This procedure is repeated for every box in the image, to end up with only unique boxes that also have a high confidence.

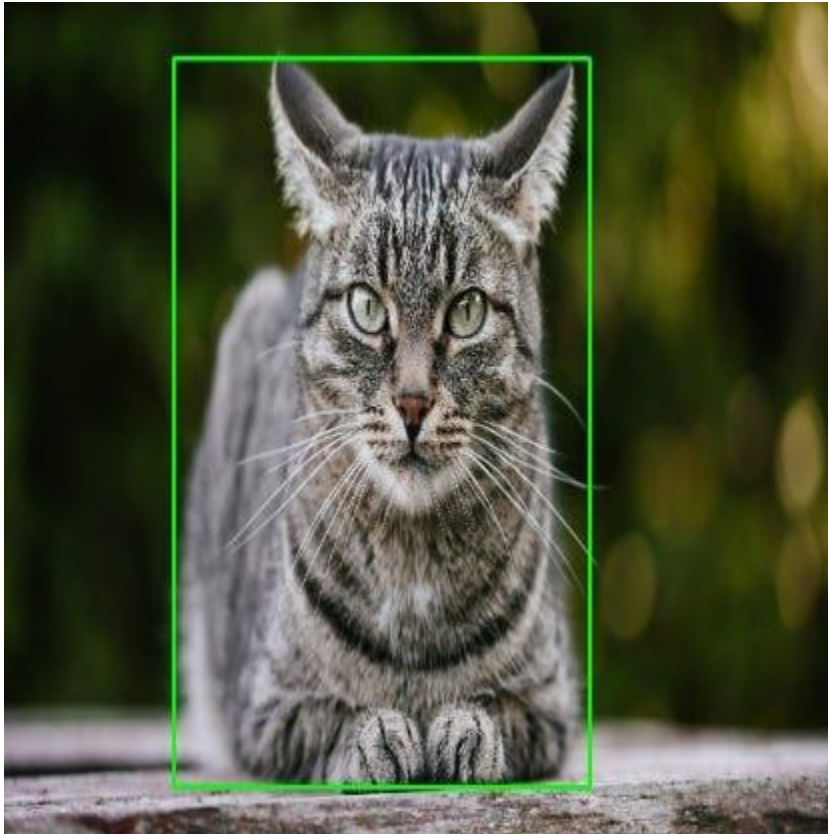


Figure 3: Result of Non Max Suppression

Algorithm:

1. Define a value for Confidence_Threshold, and IOU_Threshold.
2. Sort the bounding boxes in a descending order of confidence.
3. Remove boxes that have a confidence $<$ Confidence_Threshold
4. Loop over all the remaining boxes, starting first with the box that has highest confidence.
5. Calculate the IOU of the current box, with every remaining box that belongs to the same class.

6. If the IOU of the 2 boxes $> \text{IOU_Threshold}$, remove the box with a lower confidence from our list of boxes.
7. Repeat this operation until we have gone through all the boxes in the list.

Code:

The code below is the basic function to perform Non Max Suppression. The IOU function used in the snippet below is the same function that was used in the previous post(Code can be found: [here](#)). The code below to calculate NMS can be optimized to improve performance.

```
def nms(boxes, conf_threshold=0.7, iou_threshold=0.4):
    bbox_list_thresholder = []
    bbox_list_new = []
    # Stage 1: (Sort boxes, and filter out boxes with low confidence)
    boxes_sorted = sorted(boxes, reverse=True, key = lambda x : x[5])
    for box in boxes_sorted:
        if box[5] > conf_threshold:
            bbox_list_thresholder.append(box)
        else:
            pass
    #Stage 2: (Loop over all boxes, and remove boxes with high IOU)
    while len(bbox_list_thresholder) > 0:
        current_box = bbox_list_thresholder.pop(0)
        bbox_list_new.append(current_box)
        for box in bbox_list_thresholder:
            if current_box[4] == box[4]:
                iou = IOU(current_box[:4], box[:4])
                if iou > iou_threshold:
                    bbox_list_thresholder.remove(box)
    return bbox_list_new
```

Figure 4: Function to perform Non Max Suppression

```
def nms(boxes, conf_threshold=0.7, iou_threshold=0.4):
```


- The function takes as input the list of boxes for the particular image, and a value for confidence threshold, and iou threshold. (I have set default values for them to be 0.7, and 0.4 respectively)

```
bbox_list_thresholded = []  
bbox_list_new = []
```

- We create 2 lists called **bbox_list_thresholded**, and **bbox_list_new**.

bbox_list_thresholded: Contains the new list of boxes after filtering low confidence boxes

bbox_list_new: Contains the final list of boxes after performing NMS

```
boxes_sorted = sorted(boxes, reverse=True, key = lambda x :  
x[5])
```

- We start Stage 1 by sorting the list of boxes in descending order of confidence, and store the new list in the variable **boxes_sorted**. The inbuilt python function called **sorted** iterates through our list of boxes, and sorts it according the keywords we specify. In our case, we specify a keyword **reverse=True** to sort the list in descending order. The second keyword **key** specifies the constraint we want to use for sorting. The lambda function we use provides a mapping that returns the 5th element of each bounding box(confidence). The sorted function while iterating through each box, would look at the lambda function, which would return the 5th element of the

`box(confidence)`, and this would be sorted in a reverse order.

```
for box in boxes_sorted:
    if box[5] > conf_threshold:
        bbox_list_thresholded.append(box)
    else:
        pass
```

- We iterate over all the sorted boxes, and remove the boxes which have a confidence lower than the threshold we set(**conf_threshold=0.7**)

```
while len(bbox_list_thresholded) > 0:
    current_box = bbox_list_thresholded.pop(0)
    bbox_list_new.append(current_box)
```

- In Stage 2, we loop over all the boxes in the list of thresholded boxes(**bbox_list_thresholded**) one by one, till the list is emptied.
We start off by removing(**pop**) the first box from this list(**current_box**), as it has the highest confidence, and append it to our final list (**bbox_list_new**).

```
for box in bbox_list_thresholded:
    if current_box[4] == box[4]:
        iou = IOU(current_box[:4], box[:4])
        if iou > iou_threshold:
            bbox_list_thresholded.remove(box)
```

- We then iterate over all the remaining boxes in the list **bbox_list_thresholded**, and check whether they belong to the same class as the current box. (`box[4]` corresponds to the class)
- In case the two boxes belong to the same class, we calculate the IOU between these boxes (we pass **box[:4]** to the IOU function as it corresponds to

the values of (x1, y1, x2, y2), since our IOU function does not require class and confidence).

- If the $\text{IOU} > \text{iou_threshold}$, we remove the box from the list **bbox_list_thresholded**, as this box is the one with lower confidence.

```
return bbox_list_new
```

- We return the updated list of boxes, after the non max suppression.

Final Points:

- This procedure for non max suppression can be modified according to the application. For example the model YOLOV3 makes use of two sets of confidences as a thresholding measure. Another modification to the algorithm is called soft NMS which I will explain in a further post.
- I have created a **code** that goes through the entire process of NMS threshold for an image with 2 different classes. I will attach the results of it below. The complete code can be found [here](#).

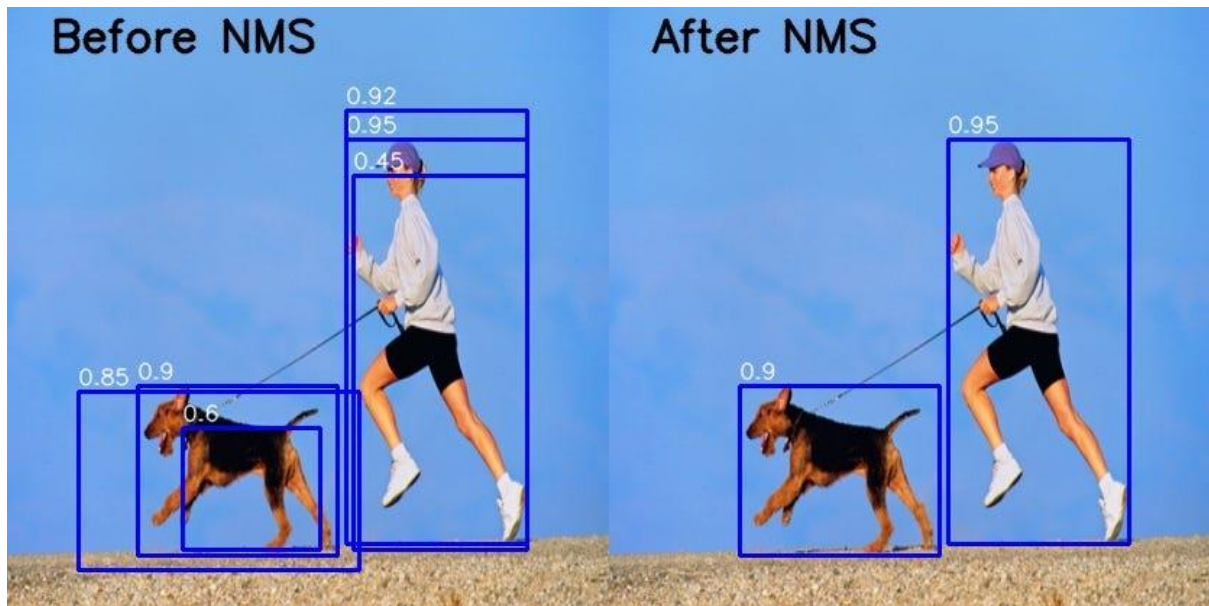


Figure 5: Result of NMS in codes

Non-maximum Suppression

Non Maximum Suppression (NMS) is a computer vision approach for selecting a single entity from a large number of overlapping things. Entities that fall below a certain probability threshold are generally discarded.

Typical One component of the [object detection](#) process is for producing categorization recommendations. Potential regions for the item of interest are referred to as proposals. The majority of methods use a moving window over through the feature map to assign foreground/background scores based on the features calculated in that window. The scores of the neighboring windows are comparable to some extent, and they are regarded as prospective areas. Hundreds of suggestions result as a result of this. We preserve loose limitations in this step since the proposal generating process should have a high recall.

However, it is time-consuming to analyze all of these ideas through the categorization network. It refers to a procedure known as Non-maximum Suppression, which filters suggestions based on a set of criteria.

NMS is used by most object detection algorithms to reduce a huge number of observed rectangles to a handful. In most circumstances, the manner object detection works necessitate the usage of NMS. Windowing is used by most object detectors in a most foundational sense. Hundreds of thousands of windows of various sizes and shapes are created, whether literally on the image or based on a characteristic of the image. These windows are said to contain only one item, and each class is assigned a probability/score by a

classifier. After the detector generates a significant amount of bounding boxes, the best ones must be chosen. The most often used approach for this task is the NMS machine learning algorithm. It is, in essence, a clustering method.

Non-maximum suppression algorithm

A list of Proposal boxes B , matching confidence scores S , and overlap threshold N are all included in the NSM input. Output, on the other hand, is a collection of filtered suggestions D .

- Remove the proposal with the greatest confidence score from B and place it in the proposal list D . (At first, D is empty.)
- Now compare this proposal to all of the others by calculating the IOU of this suggestion with all of the others. Withdraw that suggestion from B if the IOU exceeds the threshold N .
- Remove the suggestion with the highest level of confidence from the other suggestions in B and add it to D .
- Measure the IOU of this proposition with all of the proposals in B again, and remove the boxes with IOUs higher than the threshold.
- This process is continued until B is devoid of any additional offers.

A single threshold value controls the entire filtering process. As a result, choosing the right threshold value is crucial to the model's success. Setting this limit, on the other hand, is difficult.

So how does NMS work?

Assume that 0.3 is the overlap threshold. Even if the confidence is better than other boxes with fewer IOU, if there is indeed a proposal with 0.31 IOU and strong class probabilities, the item will be eliminated. As a result, if two things are placed alongside, one of them will be deleted. Although its confidence is relatively low, a proposition with 0.39 IOU is preserved. Of fact, any threshold-based method has this drawback. So, what are we going to do now?

Soft-NMS is a simple but effective technique to cope with this situation. The concept is simple- rather than fully deleting suggestions with large IOU and high confidence, lower proposal confidence proportionate to IOU value.

So this is simply a single paragraph modification in the code of the NMS algorithm, but it greatly improves precision.

These methods are effective for filtering predictions from a single model; but, what happens if you have forecasts from numerous models? Weighted boxes merging is a new approach for integrating object detection model predictions.

Conclusion

NMS (Non-Maximum Suppression) is a computer vision approach that is employed in several algorithms. It's a group of methods for picking one thing out of a slew of overlapping ones. To get specific results, the selection process can be changed. The most typical criteria are some kind of probability number and some kind of overlap metric (e.g. IOU).

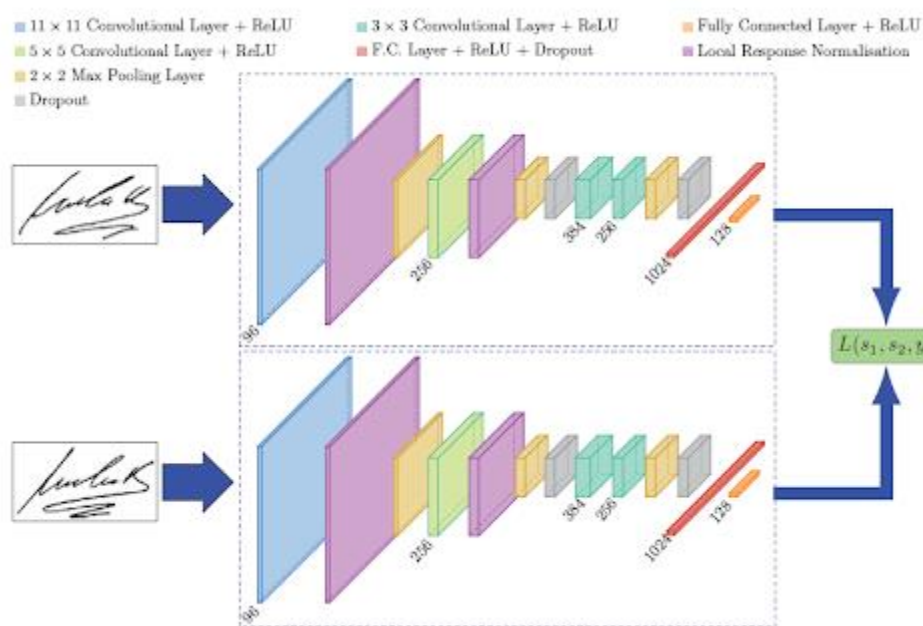
e) Briefly explain Siamese network

WHAT ARE SIAMESE NETWORKS?

A siamese neural network (SNN) is a class of neural network architectures that contain two or more identical sub-networks. "Identical" here means they have the same configuration with the same parameters and weights. Parameter updating is mirrored across both sub-networks and it's used to find similarities between inputs by comparing feature vectors.

Siamese networks use only a few images to get better predictions. The ability to learn from very little data has made siamese networks more popular in recent years. In this article, I'll show you what siamese networks are and how to develop a signature verification system with PyTorch using siamese networks.

What Are Siamese Networks?



Siamese

network used in Signet

A siamese neural network (SNN) is a class of neural network architectures that **contain two or more identical sub-networks**. “Identical” here means they have the same configuration with the same parameters and weights. Parameter updating is mirrored across both sub-networks and it’s used to find similarities between inputs by comparing its feature vectors. These networks are used in many applications.

Traditionally, a neural network learns to predict multiple classes. This poses a problem when we need to add or remove new classes to the data. In this case, we have to update the neural network and retrain it on the whole data set. Also, deep neural networks need a large volume of data on which to train. SNNs, on the other hand, learn a similarity function. Thus, we can train the SNN to see if two images are the same (which I’ll demonstrate below). This process enables us to classify new classes of data without retraining the network.

HOW TO TRAIN A SIAMESE NETWORK

- Initialize the network, loss function and optimizer.
- Pass the first image of the pair through the network.
- Pass the second image of the pair through the network.
- Calculate the loss using the outputs from the first and second images.
- Backpropagate the loss to calculate the gradients of our model.
- Update the weights using an optimizer.
- Save the model.

Pros and Cons of Siamese Networks

SIAMESE NETWORK PROS

More Robust to Class Imbalance

Giving a few images per class is sufficient for siamese networks to recognize those images in the future with the aid of one-shot learning.

Nice to Pair With the Best Classifier

Given that an SNN's learning mechanism is somewhat different from classification models, simply averaging it with a classifier can do much

better than averaging two correlated supervised models (e.g. GBM & RF classifiers).

Learning from Semantic Similarity

SNN focuses on learning embeddings (in the deeper layer) that place the same classes/concepts close together. Hence, we can learn semantic similarity.

SIAMESE NETWORK CONS

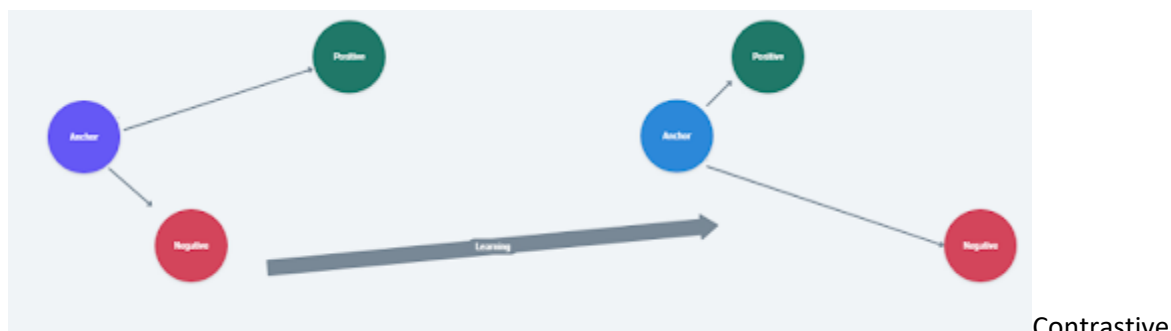
Needs More Training Time Than Normal Networks

Since SNNs involves learning from quadratic pairs (to see all information available) they're slower than the normal classification type of learning (pointwise learning).

Don't Output Probabilities

Since training involves pairwise learning, SNNs won't output the probabilities of the prediction, only distance from each class.

Loss Functions Used in Siamese Networks



loss

Since training SNNs involve pairwise learning, we cannot use cross entropy loss cannot be used. There are two loss functions we typically use to train siamese networks.

TRIPLET LOSS

Triplet loss is a loss function where in we compare a baseline (anchor) input to a positive (truthy) input and a negative (falsy) input. The distance from the baseline (anchor) input to the positive (truthy) input is minimized, and the distance from the baseline (anchor) input to the negative (falsy) input is maximized.

$$\mathcal{L}(A, P, N) = \max\left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0\right)$$

In the above equation, alpha is a margin term used to stretch the distance between similar and dissimilar pairs in the triplet. F_a , F_p , F_n are the feature embeddings for the anchor, positive and negative images.

During the training process, we feed an image triplet (anchor image, negative image, positive image) (anchor image, negative image, positive image) into the model as a single sample. The distance between

the anchor and positive images should be smaller than that between the anchor and negative images.

CONTRASTIVE LOSS

Contrastive loss is an increasingly popular loss function. It's a distance-based loss as opposed to more conventional error-prediction loss. This loss function is used to learn embeddings in which two similar points have a low Euclidean distance and two dissimilar points have a large Euclidean distance.

$$(1 - Y) \frac{1}{2} (D_W)^2 + (Y) \frac{1}{2} \{ \max(0, m - D_W) \}^2$$

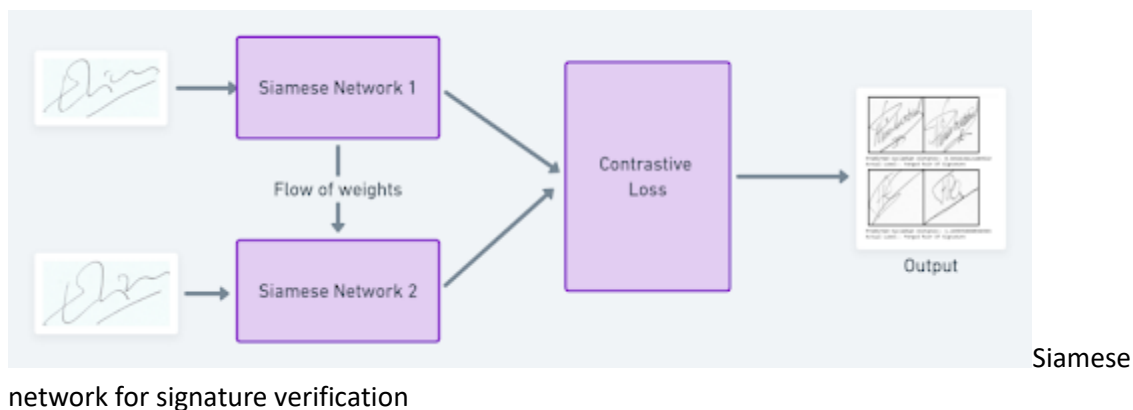
We define D_W (the Euclidean distance) as:

$$\sqrt{\{G_W(X_1) - G_W(X_2)\}^2}$$

G_W is the output of our network for one image.

NEED MORE? WE GOT YOU. [Think You Don't Need Loss Functions in Deep Learning? Think Again.](#)

Signature Verification With Siamese Networks



network for signature verification

As siamese networks are mostly used in verification systems (face recognition, signature verification, etc.), let's implement a signature verification system using siamese neural networks in PyTorch.

DATA SET AND PREPROCESSING THE DATA SET



Signatures in ICDAR data set

We are going to use the ICDAR 2011 data set which consists of genuine and fraudulent Dutch signatures. The data set itself is separated as train and folders. Inside each folder, it consists of files separated as genuine and forgery. The data set also contains the labels as CSV files. You can download the data set [here](#).

To feed this raw data into our neural network, we have to turn all the images into tensors and add the labels from the CSV files to the images. To do this we can use the custom data set class from PyTorch.

Here's what our full code will look like:

```
#preprocessing and loading the data set
class SiameseDataset():
    def __init__(self, training_csv=None, training_dir=None, transform=None):
        # used to prepare the labels and images path
        self.train_df = pd.read_csv(training_csv)
        self.train_df.columns = ["image1", "image2", "label"]
        self.train_dir = training_dir
        self.transform = transform

    def __getitem__(self, index):
        # getting the image path
        image1_path = os.path.join(self.train_dir, self.train_df.iat[index, 0])
        image2_path = os.path.join(self.train_dir, self.train_df.iat[index, 1])

        # Loading the image
        img0 = Image.open(image1_path)
        img1 = Image.open(image2_path)
        img0 = img0.convert("L")
        img1 = img1.convert("L")

        # Apply image transformations
        if self.transform is not None:
            img0 = self.transform(img0)
            img1 = self.transform(img1)

        return img0, img1, torch.from_numpy(np.array([int(self.train_df.iat[index, 2])], dtype=np.float32))

    def __len__(self):
        return len(self.train_df)
```

After preprocessing the data set, we have to load the data set into PyTorch using the DataLoader class. We'll use the transform function to reduce the image size into 105 pixels of height and width for computational purposes.

```
# Load the the dataset from raw image folders
siamese_dataset = SiameseDataset(training_csv, training_dir,
                                  transform=transforms.Compose([
                                      transforms.Resize((105, 105)),
                                      transforms.ToTensor()
                                  ]))
```

NEURAL NETWORK ARCHITECTURE

Now let's create a neural network in PyTorch. We'll use the neural network architecture which will be similar, as described in the SigNet paper.

```

#create a siamese network
class SiameseNetwork(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()
        # Setting up the Sequential of CNN Layers
        self.cnn1 = nn.Sequential(
            nn.Conv2d(1, 96, kernel_size=11, stride=1),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(5, alpha=0.0001, beta=0.75, k=2),
            nn.MaxPool2d(3, stride=2),

            nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(5, alpha=0.0001, beta=0.75, k=2),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout2d(p=0.3),

            nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),

            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout2d(p=0.3),
        )
        # Defining the fully connected layers
        self.fc1 = nn.Sequential(
            nn.Linear(30976, 1024),
            nn.ReLU(inplace=True),
            nn.Dropout2d(p=0.5),

            nn.Linear(1024, 128),
            nn.ReLU(inplace=True),

            nn.Linear(128, 2))

    def forward_once(self, x):
        # Forward pass
        output = self.cnn1(x)
        output = output.view(output.size()[0], -1)
        output = self.fc1(output)
        return output

    def forward(self, input1, input2):
        # forward pass of input 1
        output1 = self.forward_once(input1)
        # forward pass of input 2
        output2 = self.forward_once(input2)
        return output1, output2

```

In the above code, we have created our network as follows:

- The first convolutional layers filter the 105×105 input signature image with 96 kernels of size 11 with a stride of one pixel.

- The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size five.
- The third and fourth convolutional layers are connected to one another without any intervention of pooling or normalization of layers. The third layer has 384 kernels of size three connected to the (normalized, pooled and dropout) output of the second convolutional layer. The fourth convolutional layer has 256 kernels of size three.

This leads to the neural network learning fewer lower-level features for smaller receptive fields and more features for higher-level or more abstract features.

The first fully connected layer has 1024 neurons, whereas the second fully connected layer has 128 neurons. This indicates that the highest learned feature vector from each side of SigNet has a dimension equal to 128.

So where is the other network?

Since the weights are constrained to be identical for both networks, we use one model and feed it two images in succession. After that, we calculate the loss value using both the images and then backpropagate. This saves a lot of memory and also increases computational efficiency.

Andrew Ng on Siamese Networks

LOSS FUNCTION

For this task, we will use contrastive loss because it learns embeddings in which two similar points have a low Euclidean distance and two dissimilar

points have a large Euclidean distance. In PyTorch the implementation of contrastive loss will be as follows:

```
class ContrastiveLoss(torch.nn.Module):
    """
    Contrastive loss function.
    Based on:
    """

    def __init__(self, margin=1.0):
        super(ContrastiveLoss, self).__init__()
        self.margin = margin

    def forward(self, x0, x1, y):
        # euclidian distance
        diff = x0 - x1
        dist_sq = torch.sum(torch.pow(diff, 2), 1)
        dist = torch.sqrt(dist_sq)

        mdist = self.margin - dist
        dist = torch.clamp(mdist, min=0.0)
        loss = y * dist_sq + (1 - y) * torch.pow(dist, 2)
        loss = torch.sum(loss) / 2.0 / x0.size()[0]
        return loss
```

TRAINING THE SIAMESE NETWORK

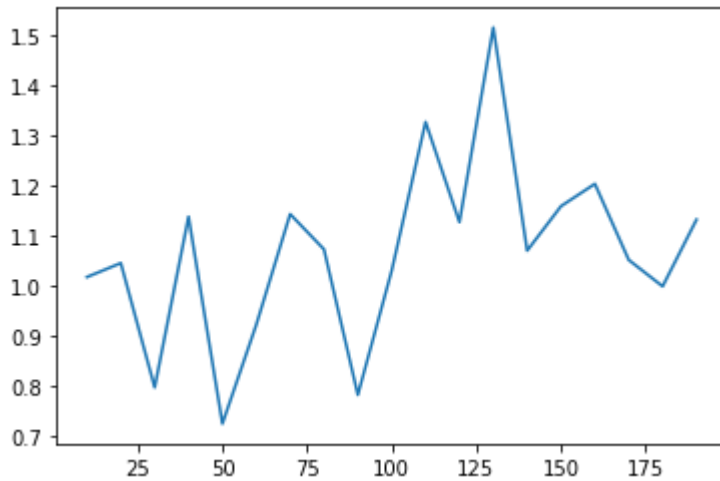
The training process of a siamese network is as follows:

- Initialize the network, loss function and optimizer (we will be using Adam for this project).
- Pass the first image of the pair through the network.
- Pass the second image of the pair through the network.
- Calculate the loss using the outputs from the first and second images.
- Backpropagate the loss to calculate the gradients of our model.

- Update the weights using an optimizer.
- Save the model.

```
# Declare Siamese Network
net = SiameseNetwork().cuda()
# Declalre Loss Function
criterion = ContrastiveLoss()
# Declare Optimizer
optimizer = th.optim.Adam(net.parameters(), lr=1e-3,
weight_decay=0.0005)
#train the model
def train():
    loss=[]
    counter=[]
    iteration_number = 0
    for epoch in range(1,config.epochs):
        for i, data in enumerate(train_dataloader,0):
            img0, img1 , label = data
            img0, img1 , label = img0.cuda(), img1.cuda() ,
            label.cuda()
            optimizer.zero_grad()
            output1,output2 = net(img0,img1)
            loss_contrastive = criterion(output1,output2,label)
            loss_contrastive.backward()
            optimizer.step()
            print("Epoch {} \n Current loss
            {} \n".format(epoch,loss_contrastive.item()))
            iteration_number += 10
            counter.append(iteration_number)
            loss.append(loss_contrastive.item())
        show_plot(counter, loss)
    return net
#set the device to cuda
device = torch.device('cuda' if th.cuda.is available() else 'cpu')
model = train()
torch.save(model.state_dict(), "model.pt")
print("Model Saved Successfully")
```

The model was trained for 20 epochs on Google Colab for an hour; the graph of the loss over time is shown below.



Graph of loss over time

TESTING THE MODEL

Now let's test our signature verification system on the test data set:

- Load the test data set using the DataLoader class from PyTorch.
- Pass the image pairs and the labels.
- Find the Euclidean distance between the images.
- Print the output based on the Euclidean distance.

```
# Load the test data set
test_dataset = SiameseDataset(training_csv=testing_csv, training_dir=testing_dir,
                               transform=transforms.Compose([t
ransforms.Resize((105,105)),
ransforms.ToTensor()
]))

test_dataloader = DataLoader(test_dataset, num_workers=6, batch_size=1, shuffle=True)
#test the network
count=0
```

```

for i, data in enumerate(test_dataloader,0):
    x0, x1, label = data
    concat = torch.cat((x0,x1),0)
    output1,output2 = model(x0.to(device),x1.to(device))

    eucledian distance = F.pairwise_distance(output1, output2)

    if label==torch.FloatTensor([[0]]):
        label="Original Pair Of Signature"
    else:
        label="Forged Pair Of Signature"

    imshow(torchvision.utils.make_grid(concat))
    print("Predicted Eucledian Distance:-",eucledian_distance.item())
    print("Actual Label:-",label)
    count=count+1
    if count ==10:
        break

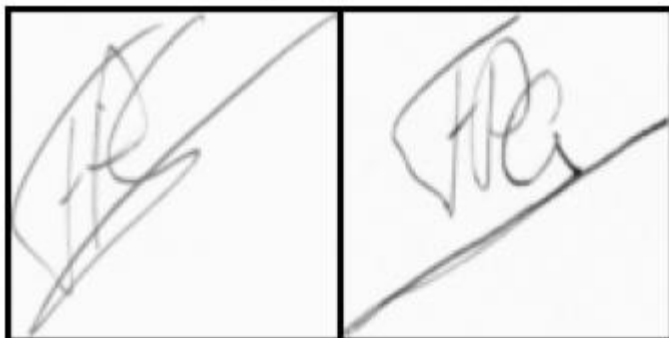
```

The predictions are as follows:



Predicted Eucledian Distance:- 0.5056638121604919

Actual Label:- Forged Pair Of Signature



Predicted Eucledian Distance:- 1.1049458980560303

Actual Label:- Forged Pair Of Signature