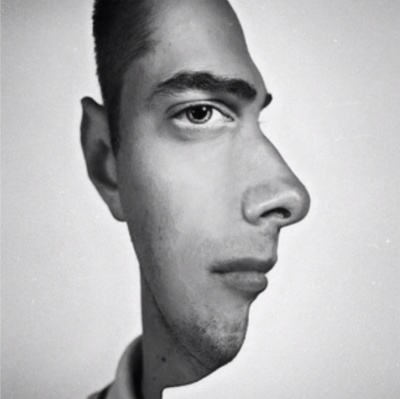
# **The Most Intuitive and Easiest Guide for Convolutional Neural Network**

[**https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480**](https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480)

****

Is it a man with the right face? Or is it a man who is looking at you directly? If your attention is at the nose or the right contour, you’ll probably say the former. If you see the ear or the left contour, you’ll say the latter.

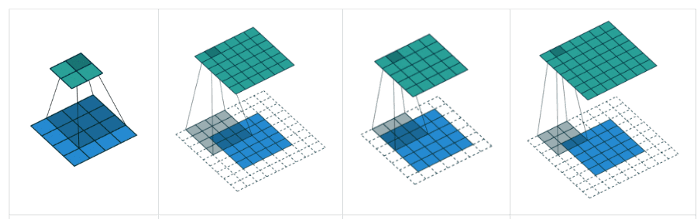
In the terminology of convolutional neural networks, we call the patterns as ‘kernel’, ‘filter’ or ‘feature detector.’ In my opinion, the last one is the most intuitive. So what CNN does is detecting the wanted features from the image data using corresponding filters and extracting the significant features for prediction.

With this computation, we detect a particular feature from the input image and get the result having information about that feature. This is called ‘feature map.’ If we see this with the real image example, the outcome is shown below.

You know our image data consists of three basic colors: Red, Green and Blue. Say our image has 7x7 pixel data with RGB. This means the data has a 7x7x3 volume. If we are to detect the certain features with 4 filters, the convolution computation occurs for each filter.

If we use a bigger size of filters, the height and the width will be bigger. And the number of filters will determine the depth of the outcome.

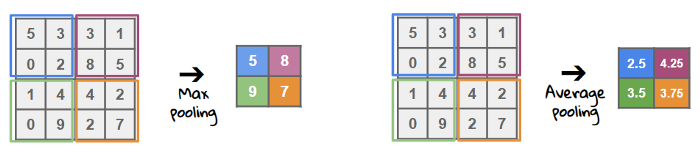
The pixels at the corner are less counted than those in the middle. This means that the pixels don’t get the same amount of weights. Additionally, If we just keep applying the convolution, we might lose the data too fast. Padding is the trick we can use here to fix this problem. As its name, padding means giving additional pixels at the boundary of the data.

****

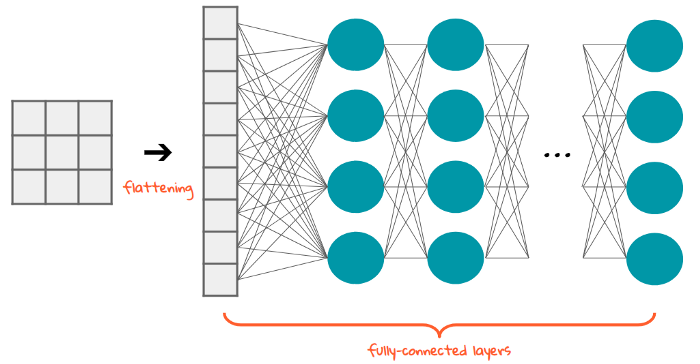
The first example on the picture above is showing what we have done in the previous section. The input image has 4x4 pixels and the filter has 3x3. There is no padding, which is called ‘valid.’ The result becomes 2x2 pixels data (4–3+1 = 2). We can see that the output data is downsized.

Let’s see the third example this time. There is one layer padding with the blank pixels. The input image has 5x5 pixels and the filter has 3x3. So the result gets 5x5 pixels (5 + 1\*2–3 + 1= 5), which is the same size as the input image. We call this ‘same.’ We can even make the outcome bigger than the input data, but the two cases are used the most.

does a filter always have to move one pixel at a time? Of course not. We can also make it move two steps or three steps at a time both in the horizontal and vertical ways. This is called ‘stride.’



But isn’t this losing valuable data? Why are we reducing the size? It could be seen like losing information at the first glimpse, but it’s rather getting more ‘meaningful’ data than losing. By removing some noise in the data and extracting only the significant one, we can reduce overfitting and speed up the computation.



Flattening is converting the data into a 1-dimensional array for inputting it to the next layer. We flatten the output of the convolutional layers to create a single long feature vector. And it is connected to the final classification model, which is called a fully-connected layer. In other words, we put all the pixel data in one line and make connections with the final layer. And once again. What is the final layer for? The classification of ‘the cats and dogs.’

sigmoid (for a binary case)

As the layers go deeper and deeper, the features that the model deals with become more complex. For example, at the early stage of ConvNet, it looks up for oriented line patterns and then finds some simple figures. At the deep stage, it can catch the specific forms of objects and finally able to detect the object of an input image.

**Machine Learning Mastery**

**CNN**

Convolutional Neural Networks expect and preserve the spatial relationship between pixels by learning internal feature representations using small squares of input data. Feature are learned and used across the whole image, allowing for the objects in the images to be shifted or translated in the scene and still detectable by the network.

**1. Convolutional Layers**

Convolutional layers are comprised of filters and feature maps.

**Filters**

The filters are the “neurons” of the layer. The have input weights and output a value. The input size is a fixed square called a patch or a receptive field.

If the convolutional layer is an input layer, then the input patch will be pixel values. If the deeper in the network architecture, then the convolutional layer will take input from a feature map from the previous layer.

**Feature Maps**

The feature map is the output of one filter applied to the previous layer.

**Zero Padding**

The distance that filter is moved across the the input from the previous layer each activation is referred to as the stride.

If the size of the previous layer is not cleanly divisible by the size of the filters receptive field and the size of the stride then it is possible for the receptive field to attempt to read off the edge of the input feature map. In this case, techniques like zero padding can be used to invent mock inputs for the receptive field to read.

**2. Pooling Layers**

The pooling layers down-sample the previous layers feature map.

Pooling layers follow a sequence of one or more convolutional layers and are intended to consolidate the features learned and expressed in the previous layers feature map. As such, pooling may be consider a technique to compress or generalize feature representations and generally reduce the overfitting of the training data by the model.

They too have a receptive field, often much smaller than the convolutional layer. Also, the stride or number of inputs that the receptive field is moved for each activation is often equal to the size of the receptive field to avoid any overlap.

Pooling layers are often very simple, taking the average or the maximum of the input value in order to create its own feature map.

**3. Fully Connected Layers**

Fully connected layers are the normal flat feed-forward neural network layer.

These layers may have a non-linear activation function or a softmax activation in order to output probabilities of class predictions.

Fully connected layers are used at the end of the network after feature extraction and consolidation has been performed by the convolutional and pooling layers. They are used to create final non-linear combinations of features and for making predictions by the network

**1. Image Input Data**

Image input data is expressed as a 3-dimensional matrix of width \* height \* channels. If we were using color images in our example, we would have 3 channels for the red, green and blue pixel values, e.g. 32x32x3.

**2. Convolutional Layer**

Convolutional layers also make use of a nonlinear transfer function as part of activation and the rectifier activation function is the popular default to use.

**3. Pool Layer**

We define a pooling layer with a receptive field with a width of 2 inputs and a height of 2 inputs. We also use a stride of 2 to ensure that there is no overlap.

This results in feature maps that are one half the size of the input feature maps. From 10 different 28×28 feature maps as input to 10 different 14×14 feature maps as output.

We will use a max() operation for each receptive field so that the activation is the maximum input value.

**4. Fully Connected Layer**

Finally, we can flatten out the square feature maps into a traditional flat fully connected layer.

We can define the fully connected layer with 200 hidden neurons, each with 10x14x14 input connections, or 1960 + 1 weights per neuron. That is a total of 392,200 connections and weights to learn in this layer.

We can use a sigmoid or softmax transfer function to output probabilities of class values directly.

Excellent results achieve a prediction error of less than 1%. State-of-the-art prediction error of approximately 0.2% can be achieved with large Convolutional Neural Networks. From MNIST data (modified national institute of science and technology)

The capacity of a deep learning neural network model controls the scope of the types of mapping functions that it is able to le

arn.

**Creating a good model**

A model with too little capacity cannot learn the training dataset meaning it will underfit, whereas a model with too much capacity may memorize the training dataset, meaning it will overfit or may get stuck or lost during the optimization process.

The capacity of a neural network model is defined by configuring the number of nodes and the number of layers.

The number of nodes in a layer is referred to as the width.

Developing wide networks with one layer and many nodes was relatively straightforward. In theory, a network with enough nodes in the single hidden layer can learn to approximate any mapping function, although in practice, we don’t know how many nodes are sufficient or how to train such a model.

The number of layers in a model is referred to as its depth.

Increasing the depth increases the capacity of the model. Training deep models, e.g. those with many hidden layers, can be computationally more efficient than training a single layer network with a vast number of nodes.

We can define a two-dimensional CNN with 32 filter maps, each with a size of 3 by 3, as follows:

layer = Conv2D(32, (3,3))

**Binary Classification Loss Functions**

Binary classification are those predictive modeling problems where examples are assigned one of two labels.

The problem is often framed as predicting a value of 0 or 1 for the first or second class and is often implemented as predicting the probability of the example belonging to class value 1.

sensible default learning rate of 0.01

**Loss function selection**

**Binary Cross-Entropy Loss**

Cross-entropy is the default loss function to use for binary classification problems.

It is intended for use with binary classification where the target values are in the set {0, 1}.

Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if you have a good reason.

Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for predicting class 1. The score is minimized and a perfect cross-entropy value is 0.

Cross-entropy can be specified as the loss function in Keras by specifying ‘binary\_crossentropy‘ when compiling the model.

model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=['accuracy'])

1

model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=['accuracy'])

The function requires that the output layer is configured with a single node and a ‘sigmoid‘ activation in order to predict the probability for class 1.

model.add(Dense(1, activation='sigmoid'))

1

model.add(Dense(1, activation='sigmoid'))

**Hinge Loss**

An alternative to cross-entropy for binary classification problems is the hinge loss function, primarily developed for use with Support Vector Machine (SVM) models.

It is intended for use with binary classification where the target values are in the set {-1, 1}.

The hinge loss function encourages examples to have the correct sign, assigning more error when there is a difference in the sign between the actual and predicted class values.

Reports of performance with the hinge loss are mixed, sometimes resulting in better performance than cross-entropy on binary classification problems.

Firstly, the target variable must be modified to have values in the set {-1, 1}.

# change y from {0,1} to {-1,1}

y[where(y == 0)] = -1

The hinge loss function can then be specified as the ‘hinge‘ in the compile function.

model.compile(loss='hinge', optimizer=opt, metrics=['accuracy'])

1

model.compile(loss='hinge', optimizer=opt, metrics=['accuracy'])

Finally, the output layer of the network must be configured to have a single node with a hyperbolic tangent activation function capable of outputting a single value in the range [-1, 1].

model.add(Dense(1, activation='tanh'))

**Hyperparameter learning rate**

Deep learning neural networks are trained using the stochastic gradient descent optimization algorithm.

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

The learning rate may be the most important hyperparameter when configuring your neural network. Therefore it is vital to know how to investigate the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior.

The amount that the weights are updated during training is referred to as the step size or the “learning rate.”

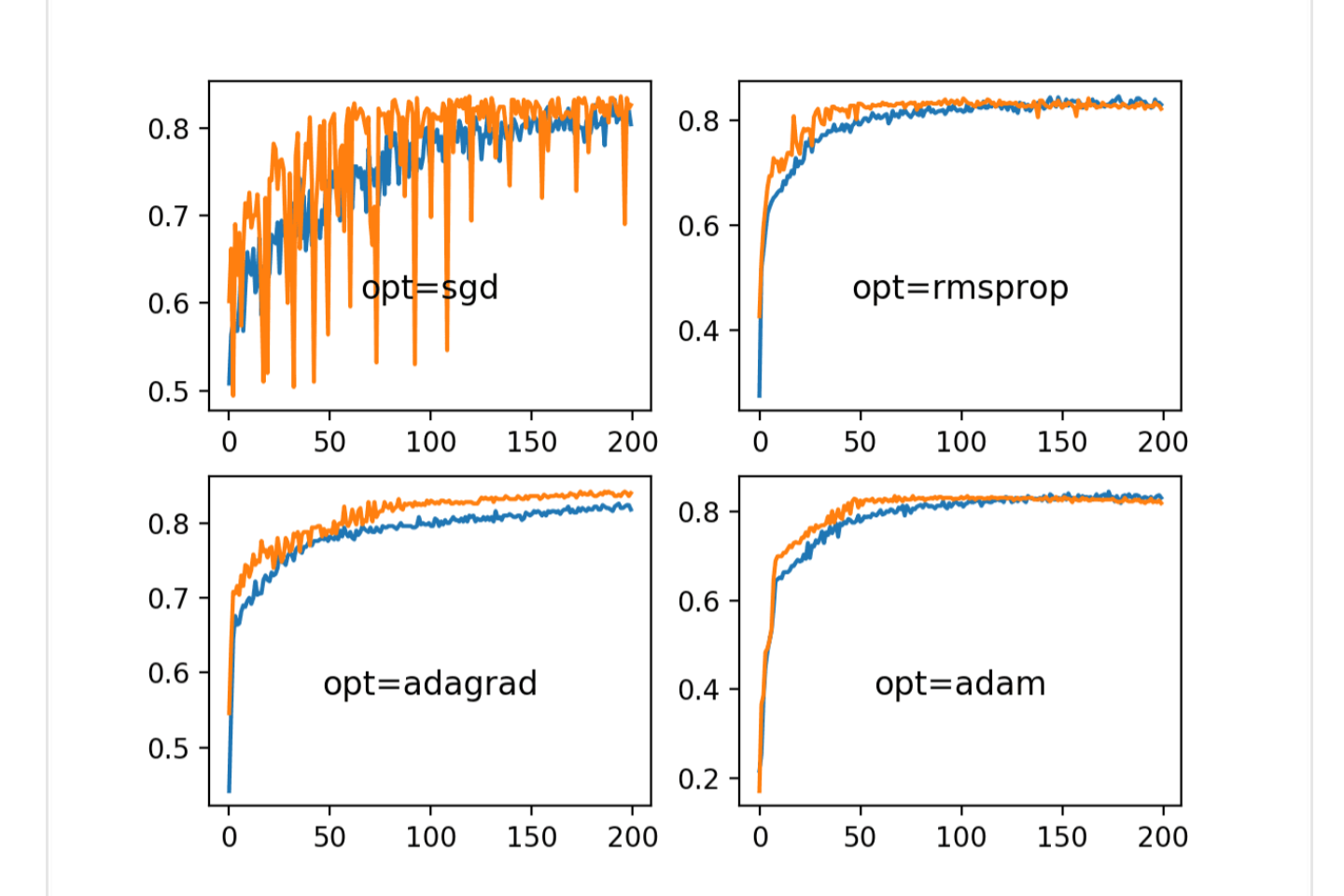
Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

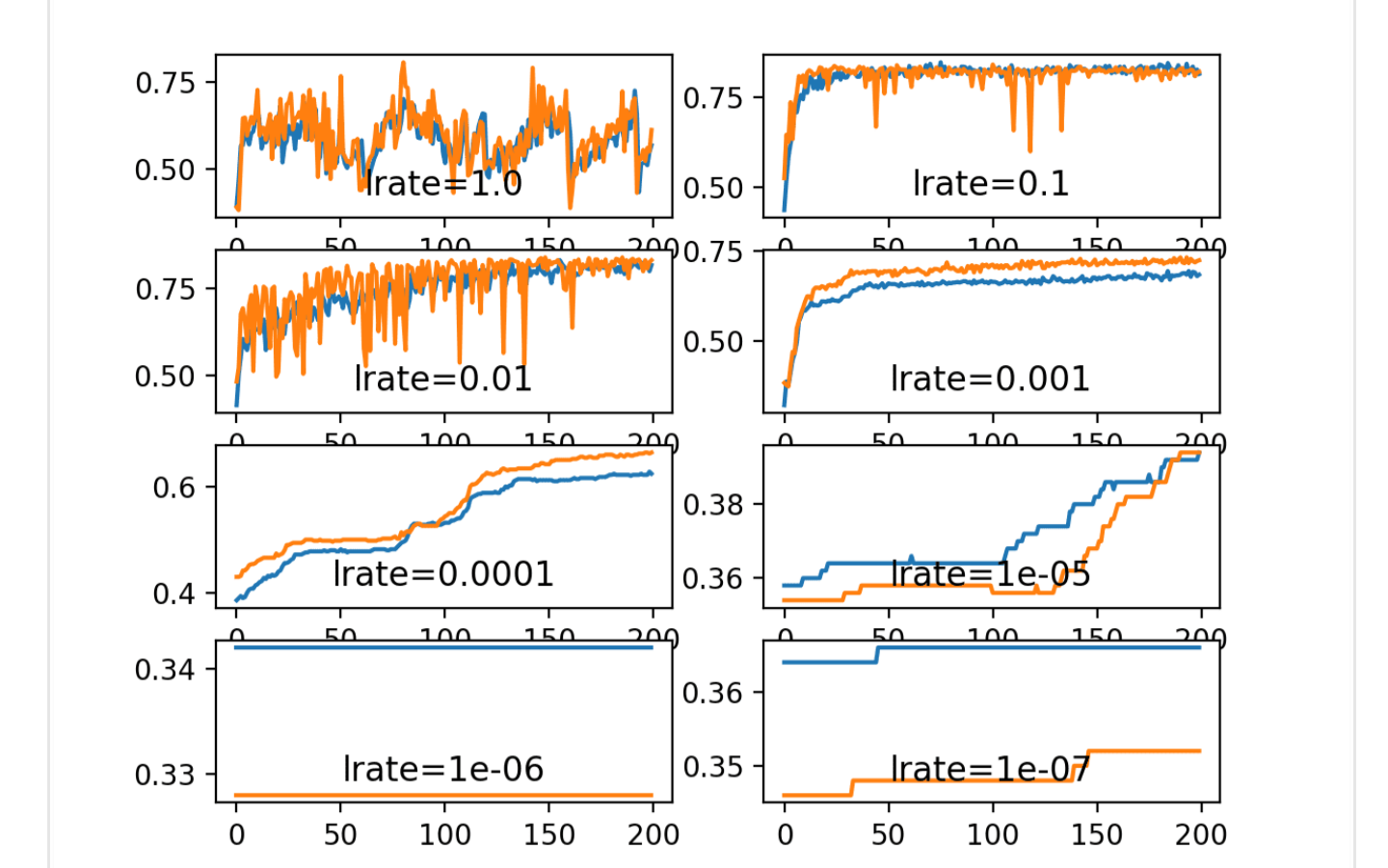
Three commonly used adaptive learning rate methods include:

RMSProp Optimizer

Adagrad Optimizer

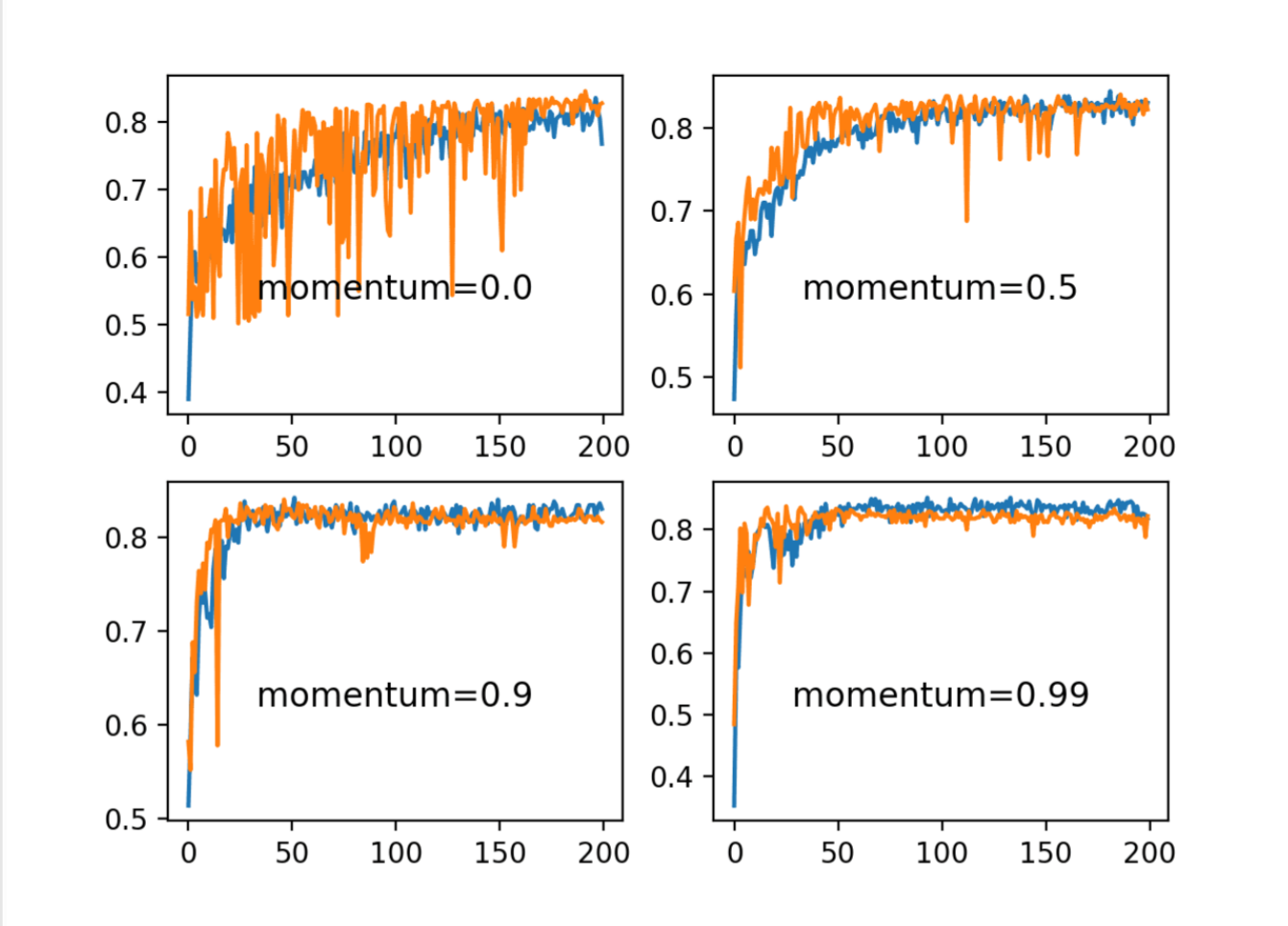
Adam Optimizer (Adaptive Moment Estimation)





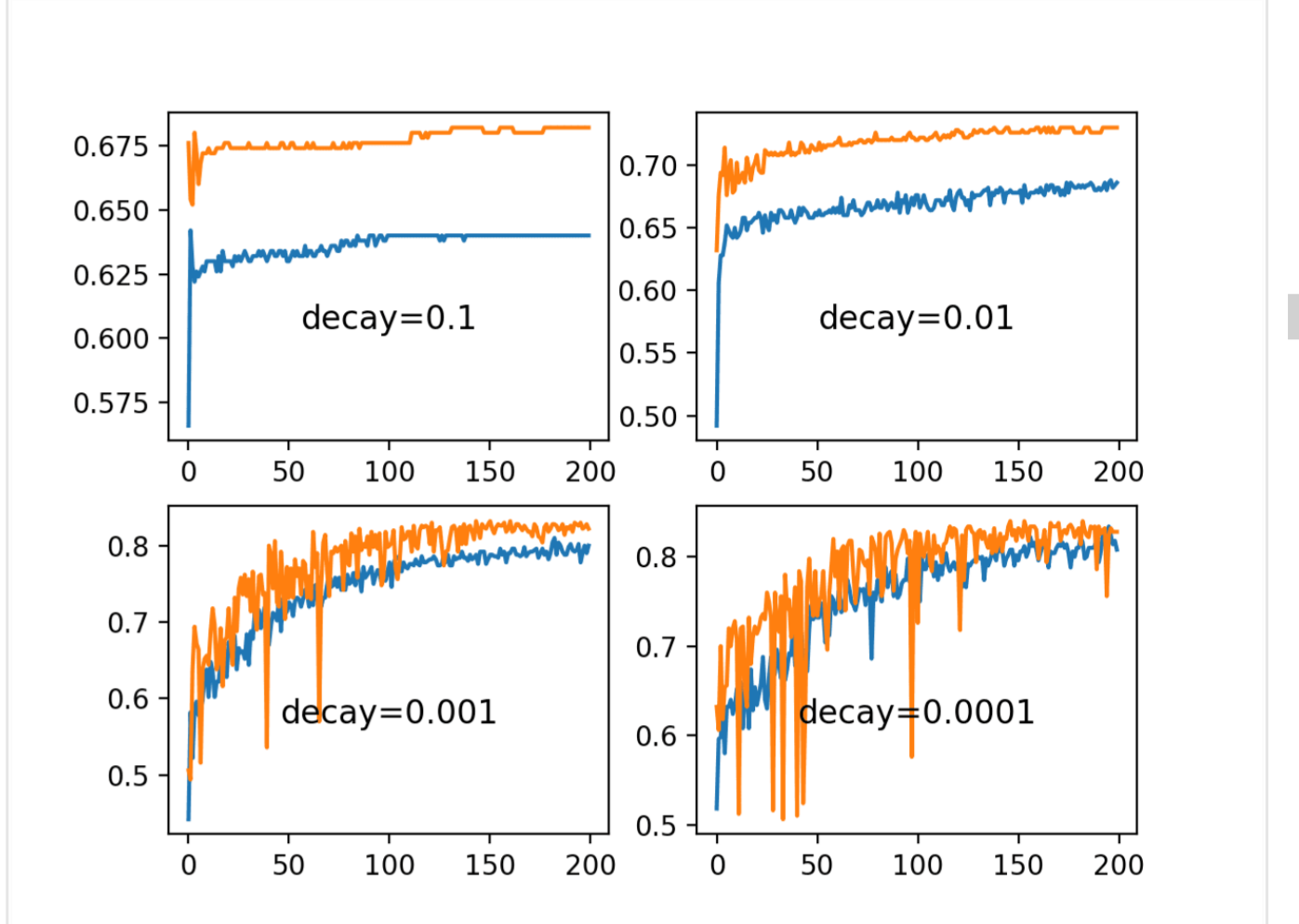
**Momentum Dynamics**

Momentum can smooth the progression of the learning algorithm that, in turn, can accelerate the training process.



**Decay**

Fixing the learning rate at 0.01 and not using momentum, we would expect that a very small learning rate decay would be preferred, as a large learning rate decay would rapidly result in a learning rate that is too small for the model to learn effectively.



**Vanishing gradient and relu**

The vanishing gradients problem is one example of unstable behavior that you may encounter when training a deep neural network.

It describes the situation where a deep multilayer feed-forward network or a recurrent neural network is unable to propagate useful gradient information from the output end of the model back to the layers near the input end of the model.

The result is the general inability of models with many layers to learn on a given dataset, or for models with many layers to prematurely converge to a poor solution.

Many fixes and workarounds have been proposed and investigated, such as alternate weight initialization schemes, unsupervised pre-training, layer-wise training, and variations on gradient descent. Perhaps the most common change is the use of the rectified linear activation function that has become the new default, instead of the hyperbolic tangent activation function that was the default through the late 1990s and 2000s.

It is desirable to train neural networks with many layers, as the addition of more layers increases the capacity of the network, making it capable of learning a large training dataset and efficiently representing more complex mapping functions from inputs to outputs.

A problem with training networks with many layers (e.g. 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.

The term vanishing gradient refers to the fact that in a feedforward network (FFN) the backpropagated error signal typically decreases (or increases) exponentially as a function of the distance from the final layer.

The vanishing gradients problem may be manifest in a Multilayer Perceptron by a slow rate of improvement of a model during training and perhaps premature convergence, e.g. continued training does not result in any further improvement.

The rectified linear activation function has supplanted the hyperbolic tangent activation function as the new preferred default when developing Multilayer Perceptron networks, as well as other network types like CNNs.

This is because the activation function looks and acts like a linear function, making it easier to train and less likely to saturate, but is, in fact, a nonlinear function, forcing negative inputs to the value 0. It is claimed as one possible approach to addressing the vanishing gradients problem when training deeper models.

**Dropout**

Dropout regularization is a computationally cheap way to regularize a deep neural network.

Dropout works by probabilistically removing, or “dropping out,” inputs to a layer, which may be input variables in the data sample or activations from a previous layer. It has the effect of simulating a large number of networks with very different network structure and, in turn, making nodes in the network generally more robust to the inputs.

The Dropout layer is added to a model between existing layers and applies to outputs of the prior layer that are fed to the subsequent layer.

Often, dropout is only used after the pooling layers, but this is just a rough heuristic.

**Early stopping**

A problem with training neural networks is in the choice of the number of training epochs to use.

Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model. Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

Early stopping requires that a validation dataset is evaluated during training.

This can be achieved by specifying the validation dataset to the fit() function when training your model.

There are two ways of doing this.

The first involves you manually splitting your training data into a train and validation dataset and specifying the validation dataset to the fit() function via the validation\_data argument. For example:

model.fit(train\_X, train\_y, validation\_data=(val\_x, val\_y))

model.fit(train\_X, train\_y, validation\_data=(val\_x, val\_y))

Alternately, the fit() function can automatically split your training dataset into train and validation sets based on a percentage split specified via the validation\_split argument.

The validation\_split is a value between 0 and 1 and defines the percentage amount of the training dataset to use for the validation dataset. For example:

model.fit(train\_X, train\_y, validation\_split=0.3)

model.fit(train\_X, train\_y, validation\_split=0.3)

In both cases, the model is not trained on the validation dataset. Instead, the model is evaluated on the validation dataset at the end of each training epoch.

Keras supports the early stopping of training via a callback called EarlyStopping.

This callback allows you to specify the performance measure to monitor, the trigger, and once triggered, it will stop the training process.

The EarlyStopping callback is configured when instantiated via arguments.

The “monitor” allows you to specify the performance measure to monitor in order to end training. Recall from the previous section that the calculation of measures on the validation dataset will have the ‘val\_‘ prefix, such as ‘val\_loss‘ for the loss on the validation dataset.

es = EarlyStopping(monitor='val\_loss')

Based on the choice of performance measure, the “mode” argument will need to be specified as whether the objective of the chosen metric is to increase (maximize or ‘max‘) or to decrease (minimize or ‘min‘).

For example, we would seek a minimum for validation loss and a minimum for validation mean squared error, whereas we would seek a maximum for validation accuracy.

es = EarlyStopping(monitor='val\_loss', mode='min')

1

By default, mode is set to ‘auto‘ and knows that you want to minimize loss or maximize accuracy.

Often, the first sign of no further improvement may not be the best time to stop training. This is because the model may coast into a plateau of no improvement or even get slightly worse before getting much better.

We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the “patience” argument.

es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1, patience=50)

The exact amount of patience will vary between models and problems. Reviewing plots of your performance measure can be very useful to get an idea of how noisy the optimization process for your model on your data may be.

Finally, it may be desirable to only stop training if performance stays above or below a given threshold or baseline. For example, if you have familiarity with the training of the model (e.g. learning curves) and know that once a validation loss of a given value is achieved that there is no point in continuing training. This can be specified by setting the “baseline” argument.

This might be more useful when fine tuning a model, after the initial wild fluctuations in the performance measure seen in the early stages of training a new model are past.

es = EarlyStopping(monitor='val\_loss', mode='min', baseline=0.4)

“Efficient BackProp” written by Yann LeCun, Leon Bottou, (both at Facebook AI), Genevieve Orr, and Klaus-Robert Muller (also co-editors of the book). 1998

"Backpropagation is a very popular neural network learning algorithm because it is conceptually simple, computationally efficient, and because it often works. However, getting it to work well, and sometimes to work at all, can seem more of an art than a science.

Designing and training a network using backprop requires making many seemingly arbitrary choices such as the number and types of nodes, layers, learning rates, training and test sets, and so forth. These choices can be critical, yet there is no foolproof recipe for deciding them because they are largely problem and data dependent"

**Shuffling the training set:**

a very simple trick that crudely implements this idea is to simply choose successive examples that are from different classes since training examples belonging to the same class will most likely contain similar information.

This trick can also be implemented by showing and re-showing examples to the model it gets the most wrong or makes the most error on when making a prediction. This approach can be effective, but can also lead to disaster if the examples that are over-represented during training are outliers.

**GAN**

This is an example of unsupervised learning

A GAN is a generative model that is trained using two neural network models. One model is called the “generator” or “generative network” model that learns to generate new plausible samples. The other model is called the “discriminator” or “discriminative network” and learns to differentiate generated examples from real examples.

The two models are set up in a contest or a game (in a game theory sense) where the generator model seeks to fool the discriminator model, and the discriminator is provided with both examples of real and generated samples.

After training, the generative model can then be used to create new plausible samples on demand.

Generating new plausible samples was the application described in the original paper by Ian Goodfellow, et al. in the 2014 paper “Generative Adversarial Networks” where GANs were used to generate new plausible examples for the MNIST handwritten digit dataset, the CIFAR-10 small object photograph dataset, and the Toronto Face Database

Examples from this paper were used in a 2018 report titled “The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation” to demonstrate the rapid progress of GANs from 2014 to 2017 (found via this tweet by Ian Goodfellow).

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.

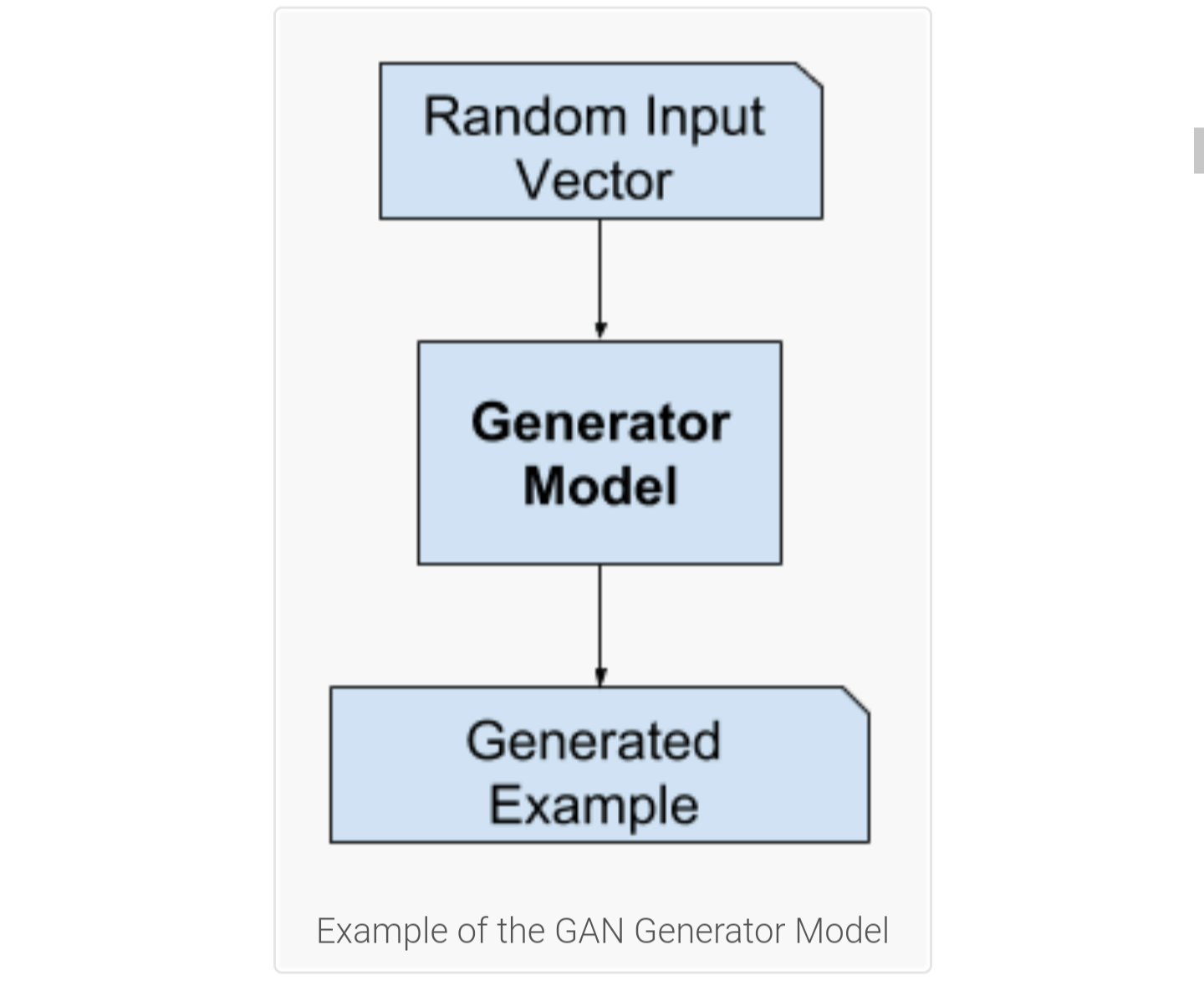
GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating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

A 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“.

**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.



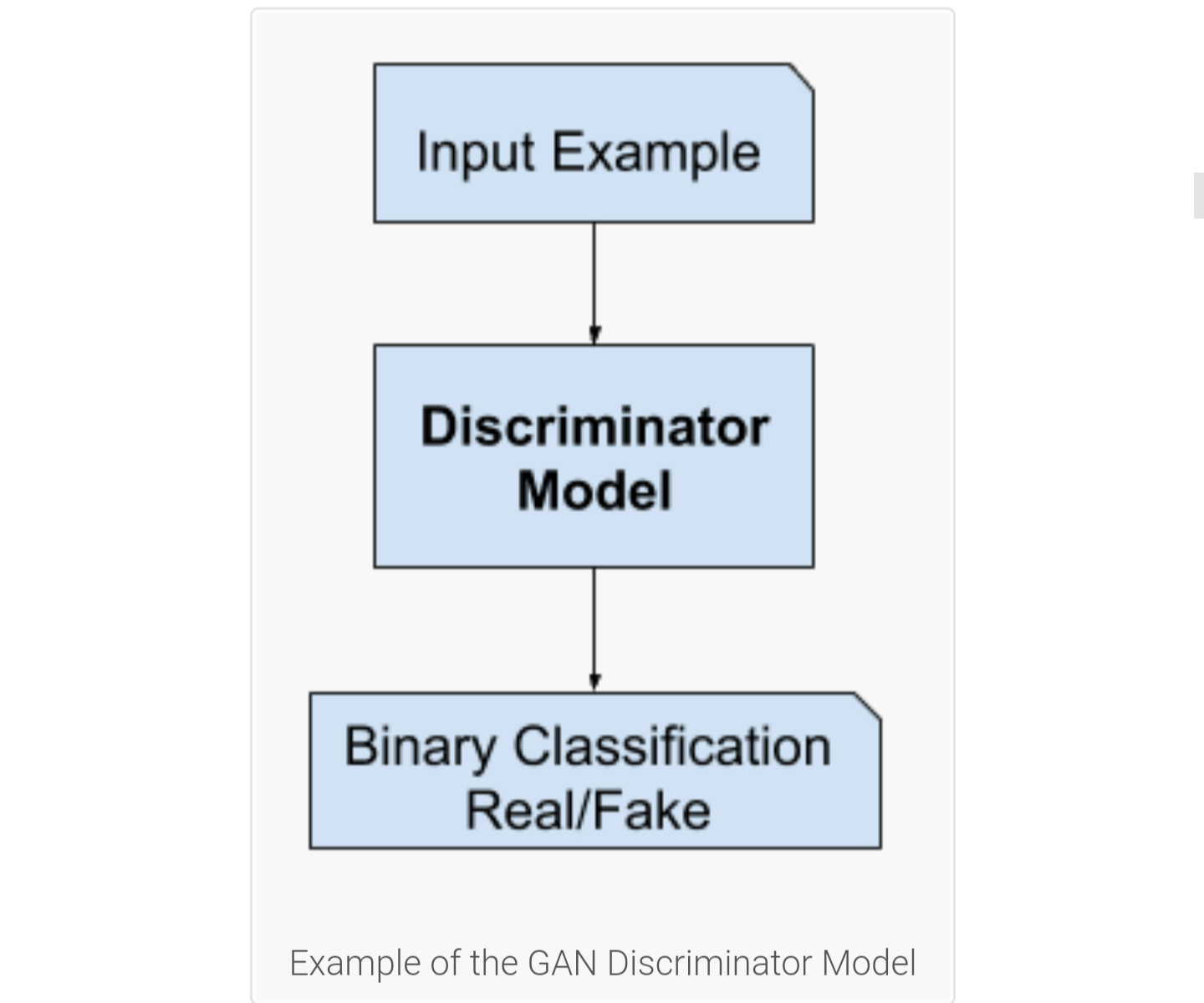
**The Discriminator Model**

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.

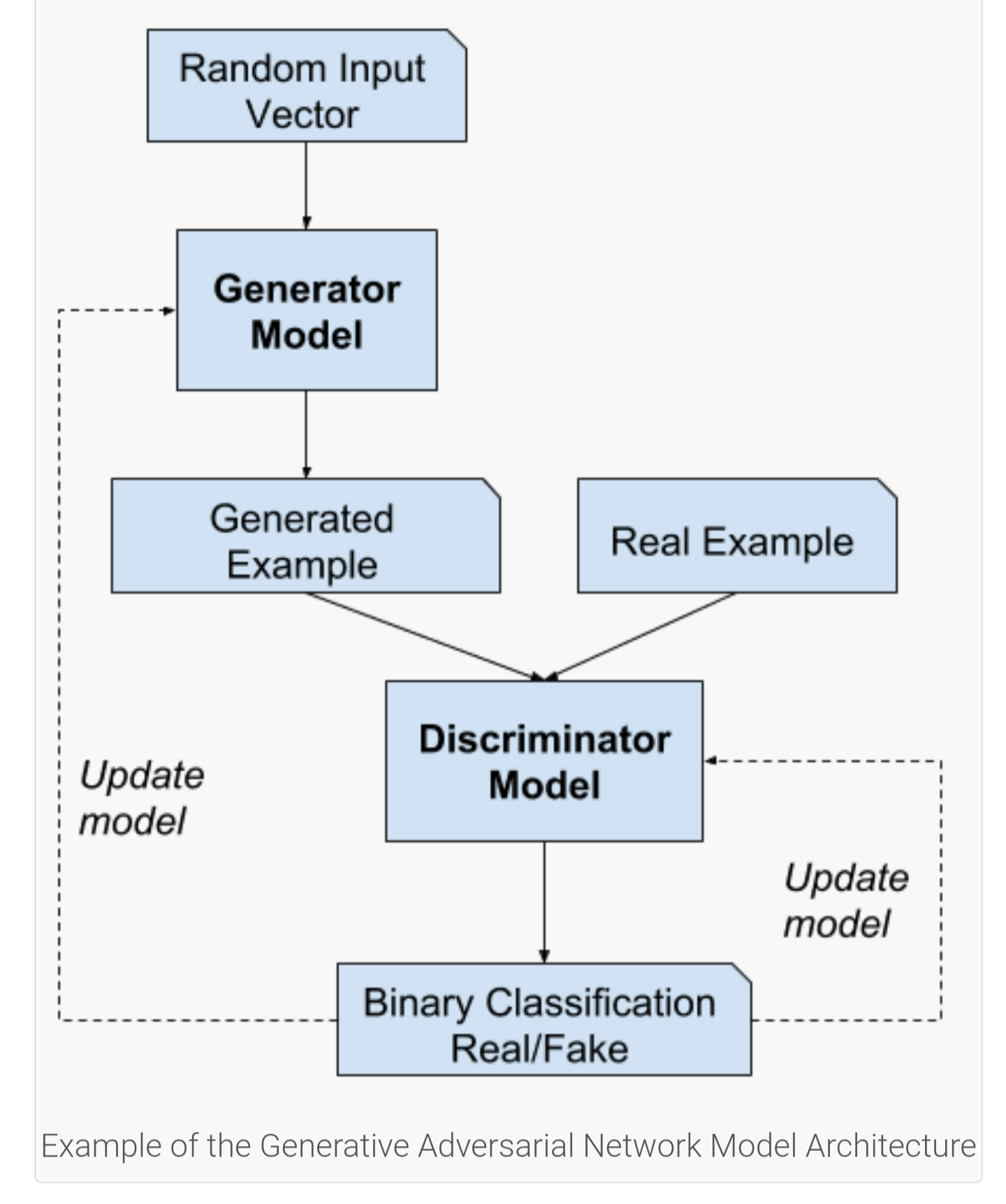


The discriminator is then updated to get better at discriminating real and fake samples in the next round, and importantly, the generator is updated based on how well, or not, the generated samples fooled the discriminator.

"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.

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.



**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.

**Some applications:**

Image Super-Resolution. The ability to generate high-resolution versions of input images.

Creating Art. The ability to great new and artistic images, sketches, painting, and more.

Image-to-Image Translation. The ability to translate photographs across domains, such as day to night, summer to winter, and more

2014 paper:

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake.

— Generative Adversarial Networks, 2014

**Deep Convolutional Generative Adversarial Network (DCGAN)**

The deep convolutional generative adversarial network, or DCGAN for short, is an extension of the GAN architecture for using deep convolutional neural networks for both the generator and discriminator models and configurations for the models and training that result in the stable training of a generator model.

We introduce a class of CNNs called deep convolutional generative adversarial networks (DCGANs), that have certain architectural constraints, and demonstrate that they are a strong candidate for unsupervised learning.

— Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.

The DCGAN is important because it suggested the constraints on the model required to effectively develop high-quality generator models in practice. This architecture, in turn, provided the basis for the rapid development of a large number of GAN extensions and applications.

We propose and evaluate a set of constraints on the architectural topology of Convolutional GANs that make them stable to train in most settings.

— Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.

This architecture involves seven best practices to consider when implementing your GAN model:

Downsample Using Strided Convolutions (e.g. don’t use pooling layers).

Upsample Using Strided Convolutions (e.g. use the transpose convolutional layer).

Use LeakyReLU (e.g. don’t use the standard ReLU).

Use Batch Normalization (e.g. standardize layer outputs after the activation).

Use Gaussian Weight Initialization (e.g. a mean of 0.0 and stdev of 0.02).

Use Adam Stochastic Gradient Descent (e.g. learning rate of 0.0002 and beta1 of 0.5).

Scale Images to the Range [-1,1] (e.g. use tanh in the output of the generator).

Unlike other deep learning neural network models that are trained with a loss function until convergence, a GAN generator model is trained using a second model called a discriminator that learns to classify images as real or generated. Both the generator and discriminator model are trained together to maintain an equilibrium.

As such, there is no objective loss function used to train the GAN generator models and no way to objectively assess the progress of the training and the relative or absolute quality of the model from loss alone.

Instead, a suite of qualitative and quantitative techniques have been developed to assess the performance of a GAN model based on the quality and diversity of the generated synthetic images.

The objective evaluation of GAN generator models remains an open problem.

**Manual GAN Generator Evaluation (by eye)**

Qualitative GAN Generator Evaluation

Five qualitative techniques for evaluating GAN generator models are listed below.

Nearest Neighbors.

Rapid Scene Categorization.

Rating and Preference Judgment.

Evaluating Mode Drop and Mode Collapse.

Investigating and Visualizing the Internals of Networks.

Rapid and preference judgement is where human judges are asked to rank or compare examples of real and generated images from the domain.

The “Rapid Scene Categorization” method is generally the same, although images are presented to human judges for a very limited amount of time, such as a fraction of a second, and classified as real or fake.

Images are often presented in pairs and the human judge is asked which image they prefer, e.g. which image is more realistic. A score or rating is determined based on the number of times a specific model generated images on such tournaments. Variance in the judging is reduced by averaging the ratings across multiple different human judges.

This is a labor-intensive exercise, although costs can be lowered by using a crowdsourcing platform like Amazon’s Mechanical Turk, and efficiency can be increased by using a web interface.

**Quantitative GAN Generator Evaluation**

Average Log-likelihood

Coverage Metric

Inception Score (IS)

Modified Inception Score (m-IS)

Mode Score

AM Score

Frechet Inception Distance (FID)

Maximum Mean Discrepancy (MMD)

The Wasserstein Critic

Birthday Paradox Test

Classifier Two-sample Tests (C2ST)

Classification Performance

Boundary Distortion

Number of Statistically-Different Bins (NDB)

Image Retrieval Performance

Generative Adversarial Metric (GAM)

Tournament Win Rate and Skill Rating

Normalized Relative Discriminative Score (NRDS)

Adversarial Accuracy and Adversarial Divergence

Geometry Score

Reconstruction Error

Image Quality Measures (SSIM, PSNR and Sharpness Difference)

Low-level Image Statistics

Precision, Recall and F1 Score

The original 2014 GAN paper by Goodfellow, et al. titled “Generative Adversarial Networks” used the “Average Log-likelihood” method, also referred to as kernel estimation or Parzen density estimation, to summarize the quality of the generated images.

This involves the challenging approach of estimating how well the generator captures the probability distribution of images in the domain and has generally been found not to be effective for evaluating GANs.

Two widely adopted metrics for evaluating generated images are the Inception Score and the Frechet Inception Distance.

The inception score was proposed by Tim Salimans, et al. in their 2016 paper titled “Improved Techniques for Training GANs.”

Inception Score (IS) […] is perhaps the most widely adopted score for GAN evaluation.

**— Pros and Cons of GAN Evaluation Measures, 2018.**

Calculating the inception score involves using a pre-trained deep learning neural network model for image classification to classify the generated images. Specifically, the Inception v3 model described by Christian Szegedy, et al. in their 2015 paper titled “Rethinking the Inception Architecture for Computer Vision.” The reliance on the inception model gives the inception score its name.

A large number of generated images are classified using the model. Specifically, the probability of the image belonging to each class is predicted. The probabilities are then summarized in the score to both capture how much each image looks like a known class and how diverse the set of images are across the known classes.

A higher inception score indicates better-quality generated images.

The Frechet Inception Distance, or FID, score was proposed and used by Martin Heusel, et al. in their 2017 paper titled “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.” The score was proposed as an improvement over the existing Inception Score.

FID performs well in terms of discriminability, robustness and computational efficiency. […] It has been shown that FID is consistent with human judgments and is more robust to noise than IS.

— Pros and Cons of GAN Evaluation Measures, 2018.

Like the inception score, the FID score uses the inception v3 model. Specifically, the coding layer of the model (the last pooling layer prior to the output classification of images) is used to capture computer vision specific features of an input image. These activations are calculated for a collection of real and generated images.

The activations for each real and generated image are summarized as a multivariate Gaussian and the distance between these two distributions is then calculated using the Frechet distance, also called the Wasserstein-2 distance.

A lower FID score indicates more realistic images that match the statistical properties of real images.

**Upsampling and downsampling**

A traditional convolutional neural network for image classification, and related tasks, will use pooling layers to downsample input images. For example, an average pooling or max pooling layer will reduce the feature maps from a convolutional by half on each dimension, resulting in an output that is one quarter the area of the input.

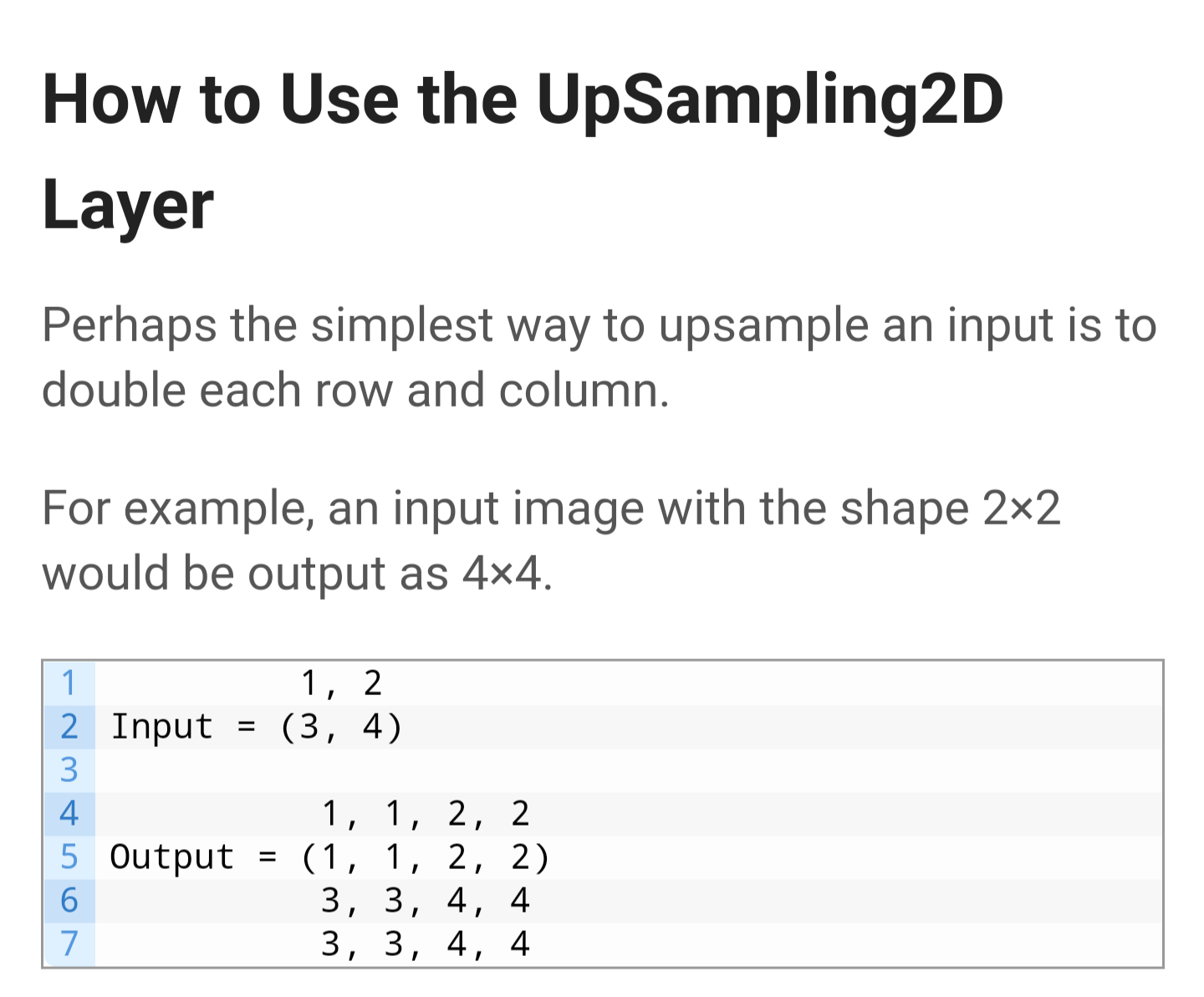
Convolutional layers themselves also perform a form of downsampling by applying each filter across the input images or feature maps; the resulting activations are an output feature map that is smaller because of the border effects. Often padding is used to counter this effect.

The generator model in a GAN requires an inverse operation of a pooling layer in a traditional convolutional layer. It needs a layer to translate from coarse salient features to a more dense and detailed output.

A simple version of an unpooling or opposite pooling layer is called an upsampling layer. It works by repeating the rows and columns of the input.

A more elaborate approach is to perform a backwards convolutional operation, originally referred to as a deconvolution, which is incorrect, but is more commonly referred to as a fractional convolutional layer or a transposed convolutional layer.

Both of these layers can be used on a GAN to perform the required upsampling operation to transform a small input into a large image output



**Reports**

**CNNs**

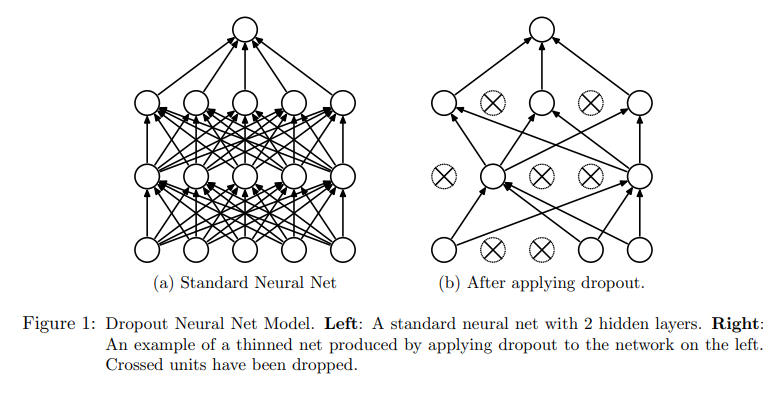
**Dropout: A Simple Way to Prevent Neural Networks from Overfitting**

**Nitish Srivastava nitish@cs.toronto.edu Geoffrey Hinton hinton@cs.toronto.edu Alex Krizhevsky kriz@cs.toronto.edu Ilya Sutskever ilya@cs.toronto.edu Ruslan Salakhutdinov**

[**https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf**](https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)

**2014**

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different “thinned” networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods.



**An Introduction to Convolutional Neural Networks**

**Keiron O’Shea**

**and Ryan Nash**

**2015**

The basic structure of a ANN can be modelled as shown in Figure 1. We would

load the input, usually in the form of a multidimensional vector to the input

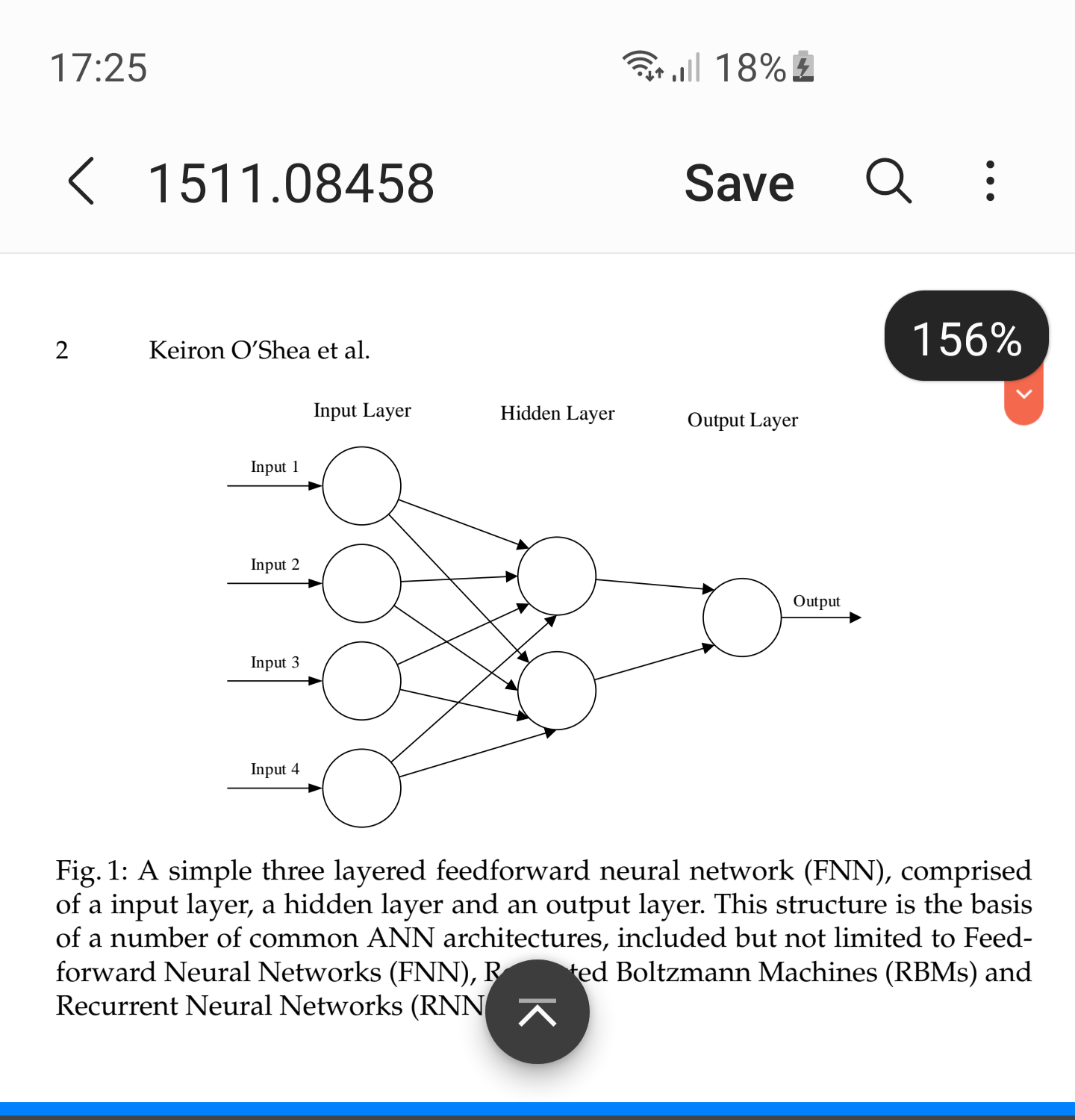
layer of which will distribute it to the hidden layers. The hidden layers will then

make decisions from the previous layer and weigh up how a stochastic change

within itself detriments or improves the final output, and this is referred to as

the process of learning.

Having multiple hidden layers is called deep learning



The two key learning paradigms in image processing tasks are supervised and

unsupervised learning. Supervised learning is learning through pre-labelled

inputs, which act as targets. For each training example there will be a set of

input values (vectors) and one or more associated designated output values.

The goal of this form of training is to reduce the models overall classification

error, through correct calculation of the output value of training example by

training.

Unsupervised learning differs in that the training set does not include any labels.

Success is usually determined by whether the network is able to reduce or

increase an associated cost function. However, it is important to note that most

image-focused pattern-recognition tasks usually depend on classification using

supervised learning.

Convolutional Neural Networks (CNNs) are analogous to traditional ANNs

in that they are comprised of neurons that self-optimise through learning. Each

neuron will still receive an input and perform a operation (such as a scalar

product followed by a non-linear function) - the basis of countless ANNs. From

the input raw image vectors to the final output of the class score, the entire of

the network will still express a single perceptive score function (the weight).

The last layer will contain loss functions associated with the classes, and all of

the regular tips and tricks developed for traditional ANNs still apply

Cnns are more suited for pattern recognition.

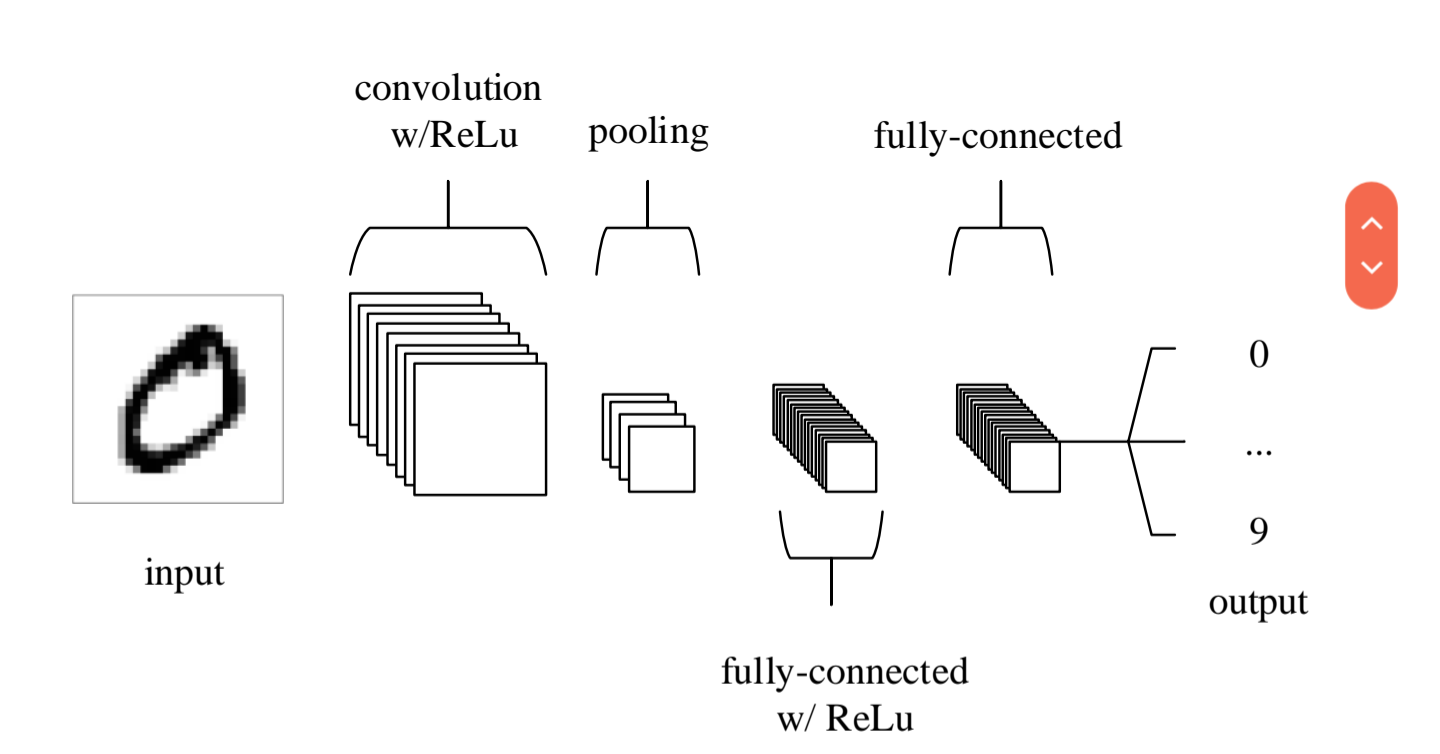
The number of weights in the first layer is determined by the dimensionality of the image. For example the handwritten numbers in the MNIST database are 28×28 and are black and white. Therefore the number of weights is 28+28+1=784. CNNS are more suited to handle more weights.

The problem with arbitrarily increasing the number of hidden layers and neurons per layer is the risk of overfitting (and computing power). An overfit network is unable to pinpoint generalised features in the test set. Overfitting is an issue with unnecessarily complex Networks.

Unlike standard anns, the neurons in cnns connect only to a small region of neurons in the preceeding layer

The input layer has dimensionality image\_width×image\_height×rubber whereas the output layer has dimensionality 1×1×n where n is the number of possible classes.

CNNs are comprised of convolutional layers, pooling layers and fully connected layers which are stacked into a cnn architecture.

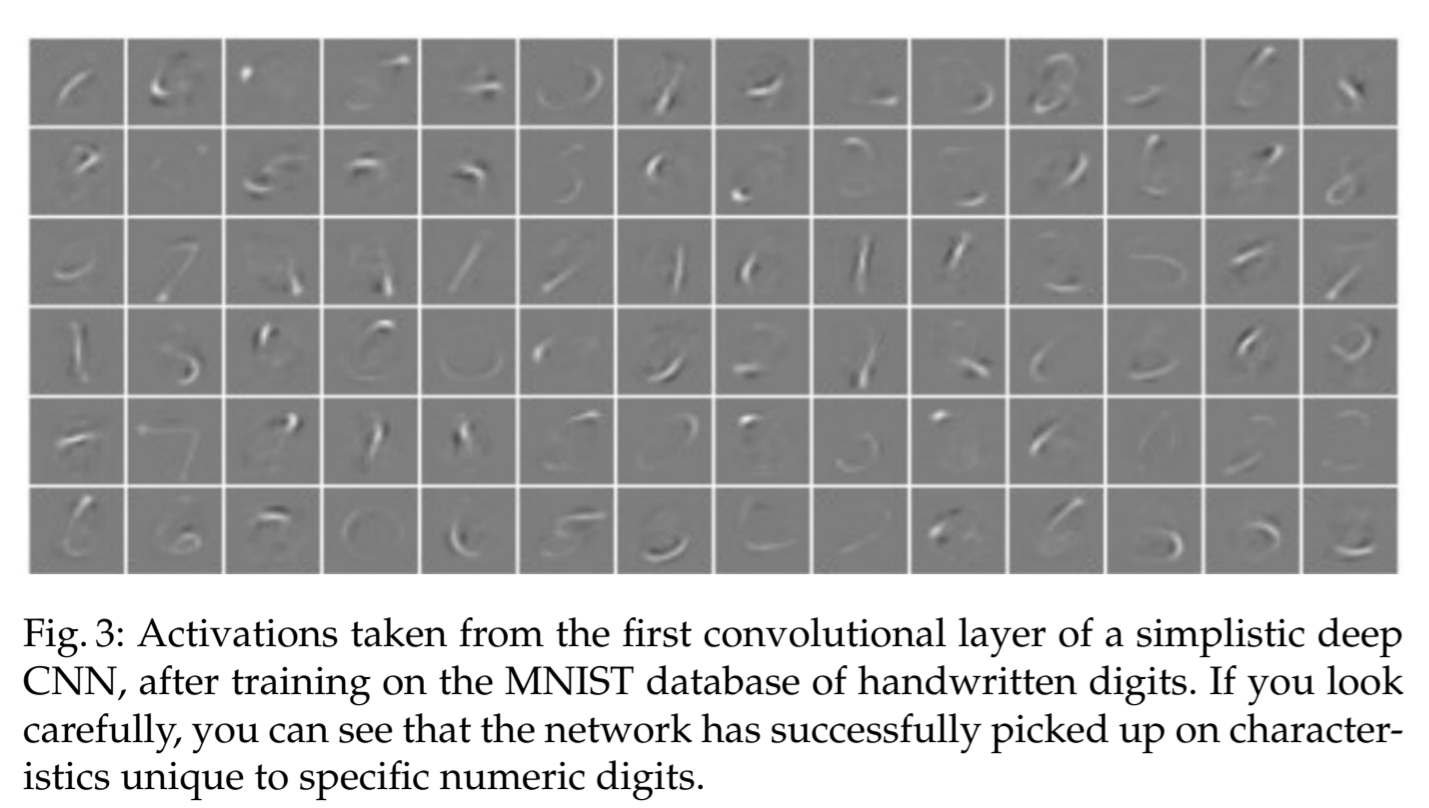


The input layer holds pixel values of the image

The convolutional layer determines the output of neurons connected to a local region of the input. E.g (2×2) region etc. This is done by a scalar product of the weights with the input of the region followed by some non linear scaling, eng relu.

The pooling layer performs downscaling alon the spacial dimensionality of an input, further reducing the number of parameters.

Fully connected layers then attempt to produce a class score from the activation required for classification.



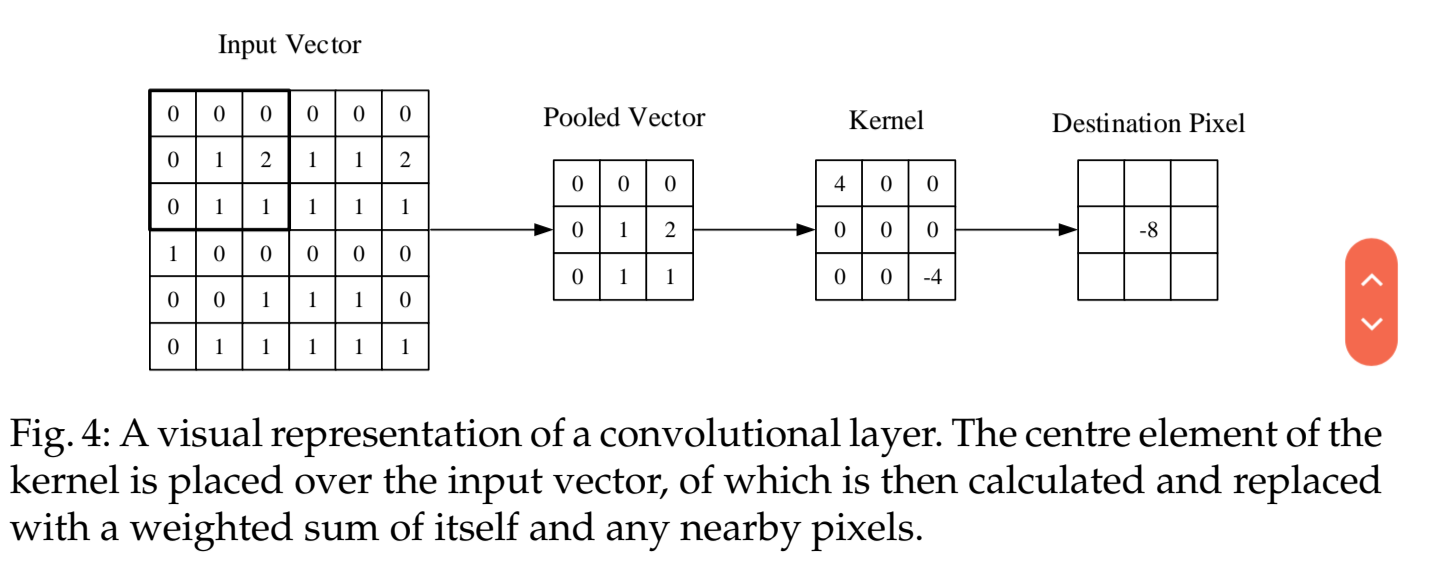
In detail:

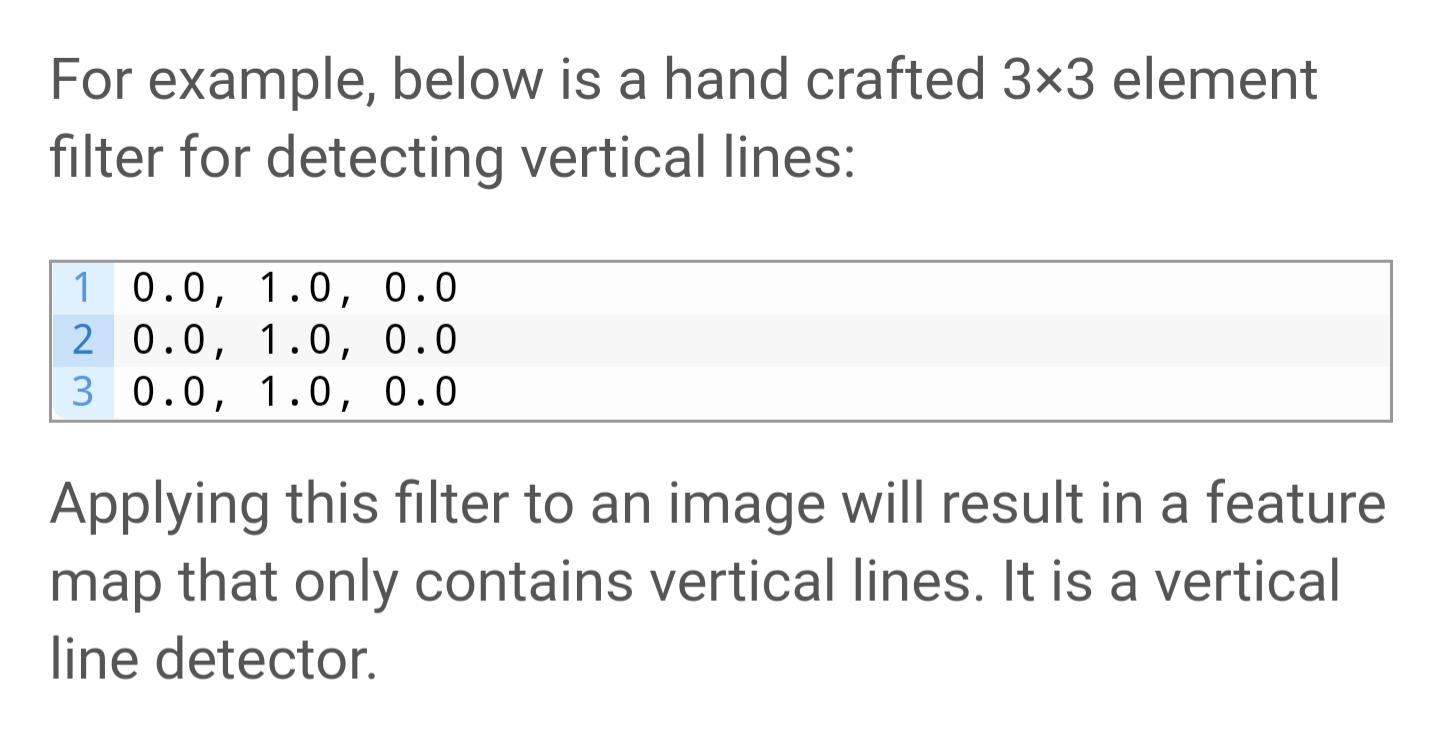
Convolutional layers:

The layers parameters focus around the use of learn able kernels. When data hits a convolutional layer, it produced a 2d activation map, as seen in the fig above.

The input data is an array which is multiplies by an array of weights called a filter or kernel

The scalar product is calculated for each value in that kernel. From this, the network learns kernels which fire when the see a specific feature at a given spatial position of the input. These are know as activations.





Via stochastic gradient decent, the network learns which filters are most successful at identifying a given object, e.g a cat.

Convolutional layers are not only applied to input data, e.g. raw pixel values, but they can also be applied to the output of other layers.

The stacking of convolutional layers allows a hierarchical decomposition of the input.

Consider that the filters that operate directly on the raw pixel values will learn to extract low-level features, such as lines.

The filters that operate on the output of the first line layers may extract features that are combinations of lower-level features, such as features that comprise multiple lines to express shapes.

This process continues until very deep layers are extracting faces, animals, houses, and so on.

This is exactly what we see in practice. The abstraction of features to high and higher orders as the depth of the network is increased.

The benefit of CNNs over ANNs is that cnns only connect to a small number of preceeding neurons. When processing images, that are a large number of inputs, therefore this feature greatly improves computation time. For example, for a 64×64×3 image, if the kernel size were 6×6, each neuron would have 6×6×3 =108 weights whereas an ANN would have 64×64×3=12288 weights!

Convolutional layers also have three hyperparameters, depth, stride and zero-padding.

Depth: Color images have multiple channels, typically one for each color channel, such as red, green, and blue.

From a data perspective, that means that a single image provided as input to the model is, in fact, three images.

A filter must always have the same number of channels as the input, often referred to as “depth“. If an input image has 3 channels (e.g. a depth of 3), then a filter applied to that image must also have 3 channels (e.g. a depth of 3). In this case, a 3×3 filter would in fact be 3x3x3 or [3, 3, 3] for rows, columns, and depth. Regardless of the depth of the input and depth of the filter, the filter is applied to the input using a dot product operation which results in a single value.

We are also able to define the stride in which we set the depth around the spatial

dimensionality of the input in order to place the receptive field. For example if

we were to set a stride as 1, then we would have a heavily overlapped receptive

field producing extremely large activations. Alternatively, setting the stride to a

greater number will reduce the amount of overlapping and produce an output

of lower spatial dimensions.

Zero padding is the process of padding the border of the input.

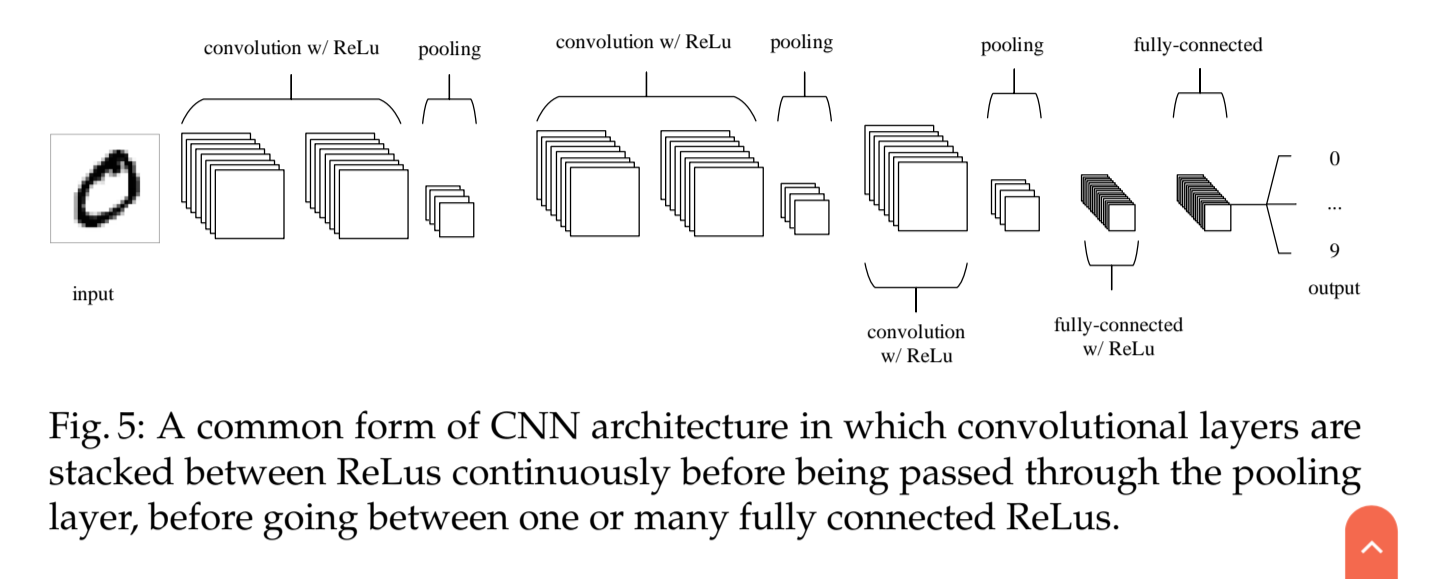
**Pooling layer:**

These aim to gradually reduce the dimensionality of the inputs and subsequently, the complexity of the model. Ultimately the model arrives to a dimensionality of 1×1×n.

Most cnns use a max pooling layer of dimensionality 2×2 which reduces the activation map to 25% of its original size.

Due to the destructive nature of pooling, setting a kernel size above 3 generally will greatly reduce performance of the model.

A common cnn architecture (as well as fig2):



The input layer should be recursively divisible by 2

**A Review of Convolutional Neural Networks**

**Arohan Ajit School of Computer Engineering KIIT Deemed to be University Bhubaneswar, India 1706211@kiit.ac.in Koustav Acharya School of Electrical Engineering KIIT Deemed to be University Bhubaneswar, India 1703207@kiit.ac.in Abhishek Samanta School of Computer Engineering KIIT Deemed to be University Bhubaneswar, India** [**1728169@kiit.ac.in**](mailto:1728169@kiit.ac.in)

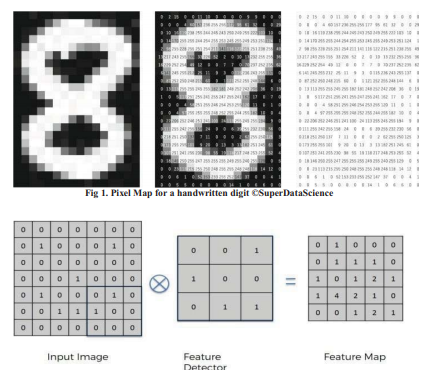
**2020**

[**https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9077735**](https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9077735)

The first step towards developing CNN was taken when a research paper regarding visual cortices of monkeys and birds was published by Hubel and Wiesel. Then in 1980s, convolution process was introduced in field of CNN by Kunihiko Fukushima named neocognitron which was inspired by work of Hubel and Wiesel. However it was Yann Le Cunn who played a major role in bringing CNN to the level it has reached today when he developed a 7 level convolutional network called LeNet-5 using back propagation and adaptive weights for various parameters. All the major architectures present today are different versions LeNet-5.

Convolutional Layer

Convolution Layer is the most basic but at the same time most important layer in the CNN. It basically convolves or multiplies the pixel matrix generated for the given image or object to produce an activation map for the given image. The main advantage of activation map is that it stores all the distinguishing features of a given image while at the same time reducing the amount of data to be processed. The matrix with which the data is convolved is a feature detector which basically is a set of values with which the machine is compatible. Different versions of image are generated using different values of feature detector. The convoluted model is also trained with backpropagation in order to ascertain minimal error in each layer. According to the lowest error set, depth and padding is set.



Pooling

Pooling is an important step to further reduce the dimensions of the activation map, keeping only the important features while also reducing the spacial invariance. This in turn reduces the number of learnable features for the model. This helps to resolve the problem of overfitting. Pooling allows CNN to incorporate all the different dimensions of an image so that it successfully recognises the given object even if its shape is skewed or is present at a different angle. There are various types of pooling like max pooling, average pooling, stochastic pooling, spatial pyramid pooling.Out of them most popular is max pooling.

Fully Connected Layer

This is the final layer which is feeded to the neural network. Generally matrice is flattened before getting passed on to the neurons. It is hard to follow data after this point due to presence of lot of hidden layer with variable weight for output of each neuron. All the reasoning and computation on data is done here.

HOW CNNS WORK

Input (The Training Data) Input layer is represented in terms of 3 dimensions i.e. width, length and height. It is commonly denoted as width\*height\*depth which is pixels for the image shown in form of a matrix. For example. If input is (64x64x3) then width: 64px, height: 64px, depth: 3px. Depth is mainly used to represent color pictures in form of RGB. Commonly Input layer is even and can be divided by 2 multiple times.

Filter Also called as kernels or feature detectors. Feature detection uses commonly uses a small matrix. For example, in ConvNet, first layer is of dimension 5x5x3.

Convolved Feature Also called Activation Map or Feature Map. A filter of defined is moved through the image taking dot product of each sub matrix producing the Output volume.

Depth The number of filters.

Stride

Sets the amount of distance stride will shift after each convolution. Having a larger stride gives a smaller output volume. For example a filter having a stride 2 will shift 2 columns after each convolution. It is necessary to set stride in such a way that the output volume is an integer. Smaller strides are used for better results.

Zero – Padding Adding zeroes on the border of the input volume so as to maintain the size of input volume and output volume. Not doing so will result in loss of information on border of the image and dimension reduction leading to low performance,

ReLU Layer

ReLU layer is used to apply activation function elementwise which converts all the negative values to zero Therefore converting the threshold to zero. It does not affect volume or hyper parameters.

**GANs**

**Generative Adversarial Nets**

**Ian J. Goodfellow∗ , Jean Pouget-Abadie† , Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair‡ , Aaron Courville, Yoshua Bengio§**

**2014**

**http://datascienceassn.org/sites/default/files/Generative%20Adversarial%20Nets.pdf**

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to 1/2 everywhere.

In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution. The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistiguishable from the genuine articles.

Advantages and disadvantages

D must be synchronized well with G during training (in particular, G must not be trained too much without updating D, in order to avoid “the Helvetica scenario” in which G collapses too many values of z to the same value of x to have enough diversity to model pdata).

The advantages are that Markov chains are never needed, only backprop is used to obtain gradients, no inference is needed during learning, and a wide variety of functions can be incorporated into the model.

**NIPS 2016 Tutorial: Generative Adversarial Networks Ian Goodfellow**

[**https://arxiv.org/pdf/1701.00160.pdf**](https://arxiv.org/pdf/1701.00160.pdf)

Why study generative modeling?

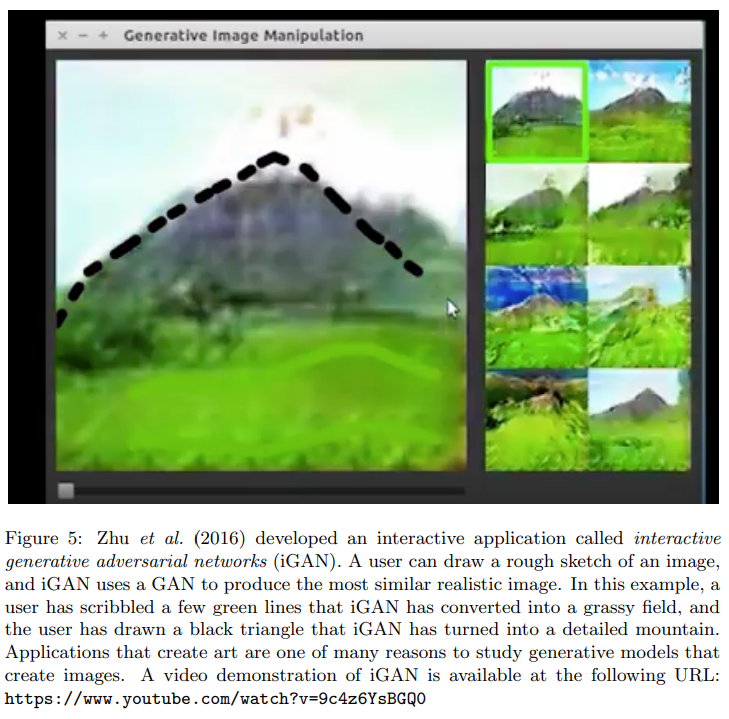
Training and sampling from generative models is an excellent test of our ability to represent and manipulate high-dimensional probability distributions. High-dimensional probability distributions are important objects in a wide variety of applied math and engineering domains.

Generative models can be incorporated into reinforcement learning in several ways. Reinforcement learning algorithms can be divided into two categories; model-based and model-free, with model-based algorithms being those that contain a generative model. Generative models of time-series data can be used to simulate possible futures. Such models could be used for planning and for reinforcement learning in a variety of ways. A generative model used for planning can learn a conditional distribution over future states of the world, given the current state of the world and hypothetical actions an agent might take as input.

Another way that generative models might be used for reinforcement learning is to enable learning in an imaginary environment, where mistaken actions do not cause real damage to the agent. Generative models can also be used to guide exploration by keeping track of how often different states have been visited or different actions have been attempted previously. Generative models, and especially GANs, can also be used for inverse reinforcement learning. Some of these connections to reinforcement learning are described further in section 5.6.

Generative models can be trained with missing data and can provide predictions on inputs that are missing data.

Finally, many tasks intrinsically require realitic generation of samples from some distribution



<https://www.youtube.com/watch?v=9c4z6YsBGQ0>

[other applications include self driving cars and google map creations]

Both players have cost functions that are defined in terms of both players’ parameters. The discriminator wishes to minimize J (D) θ (D) , θ (G) and must do so while controlling only θ (D) . The generator wishes to minimize J (G) θ (D) , θ (G) and must do so while controlling only θ (G) . Because each player’s cost depends on the other player’s parameters, but each player cannot control the other player’s parameters, this scenario is most straightforward to describe as a game rather than as an optimization problem. The solution to an optimization problem is a (local) minimum, a point in parameter space where all neighboring points have greater or equal cost. The solution to a game is a Nash equilibrium



**UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS Alec Radford & Luke Metz indico Research Boston, MA {alec,luke}@indico.io Soumith Chintala Facebook AI Research New York, NY** [**soumith@fb.com**](mailto:soumith@fb.com)

**2016**

<https://arxiv.org/pdf/1511.06434.pdf>

In recent years, supervised learning with convolutional networks (CNNs) has seen huge adoption in computer vision applications. Comparatively, unsupervised learning with CNNs has received less attention. In this work we hope to help bridge the gap between the success of CNNs for supervised learning and unsupervised learning. We introduce a class of CNNs called deep convolutional generative adversarial networks (DCGANs), that have certain architectural constraints, and demonstrate that they are a strong candidate for unsupervised learning. Training on various image datasets, we show convincing evidence that our deep convolutional adversarial pair learns a hierarchy of representations from object parts to scenes in both the generator and discriminator. Additionally, we use the learned features for novel tasks - demonstrating their applicability as general image representations.

One constant criticism of using neural networks has been that they are black-box methods, with little understanding of what the networks do in the form of a simple human-consumable algorithm. In the context of CNNs, Zeiler et. al. (Zeiler & Fergus, 2014) showed that by using deconvolutions and filtering the maximal activations, one can find the approximate purpose of each convolution filter in the network. Similarly, using a gradient descent on the inputs lets us inspect the ideal image that activates certain subsets of filters (Mordvintsev et al.).

Core to our approach is adopting and modifying three recently demonstrated changes to CNN architectures.

replaces deterministic spatial pooling functions (such as maxpooling) with strided convolutions, allowing the network to learn its own spatial downsampling.

We found global average pooling increased model stability but hurt convergence speed.

Third is Batch Normalization (Ioffe & Szegedy, 2015) which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance. This helps deal with training problems that arise due to poor initialization and helps gradient flow in deeper models

The ReLU activation (Nair & Hinton, 2010) is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution. Within the discriminator we found the leaky rectified activation (Maas et al., 2013) (Xu et al., 2015) to work well, especially for higher resolution modeling. This is in contrast to the original GAN paper, which used the maxout activation (Goodfellow et al., 2013).

Architecture guidelines for stable Deep Convolutional GANs

• Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).

• Use batchnorm in both the generator and the discriminator.

• Remove fully connected hidden layers for deeper architectures.

• Use ReLU activation in generator for all layers except for the output, which uses Tanh.

• Use LeakyReLU activation in the discriminator for all layers.



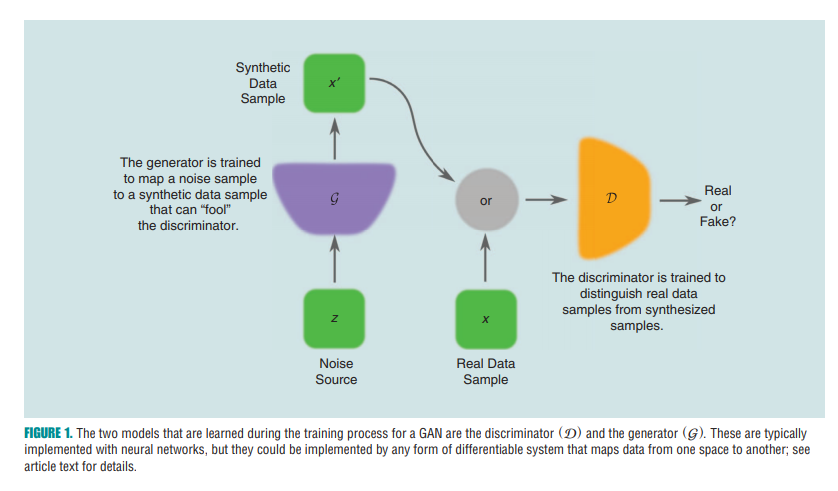
**Generative Adversarial Networks**

**Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath Deep learning for visual understanding: part 2**

[**https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8253599**](https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8253599)

**2018**

Crucially, the generator has no direct access to real images— the only way it learns is through its interaction with the discriminator. The discriminator has access to both the synthetic samples and samples drawn from the stack of real images. The error signal to the discriminator is provided through the simple ground truth of knowing whether the image came from the real stack or from the generator. The same error signal, via the discriminator, can be used to train the generator, leading it toward being able to produce forgeries of better quality



During training, the parameters of one model are updated, while the parameters of the other are fixed. Goodfellow et al. [1] show that, for a fixed generator, there is a unique optimal discriminator, D ( ) x x p p ( ) ( ) x x p ( ) . \* = + data ^ h data g They also show that the generator, G, is optimal when p p g ( ) x x = data ( ), which is equivalent to the optimal discriminator predicting 0.5 for all samples drawn from x. In other words, the generator is optimal when the discriminator, D, is maximally confused and cannot distinguish real samples from ones that are fake.

Ideally, the discriminator is trained until optimal with respect to the current generator; then the generator is again updated. However in practice, the discriminator might not be trained until optimal but rather may only be trained for a small number of iterations, and the generator is updated simultaneously with the discriminator