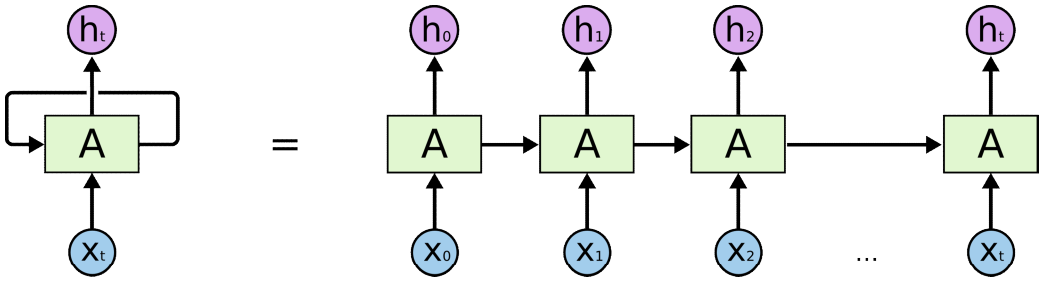
Deep Learning with Python

DAY 2

RNN AND GAN

Recurrent Neural Networks

- Recurrent neural networks have connections that have loops, adding feedback and memory to the networks over time.
- This memory allows this type of network to learn and generalize across sequences of inputs rather than individual patterns.
- Consider the following taxonomy of sequence problems that require a mapping of an input to an output (taken from Andrej Karpathy*)
  - **One-to-Many**: sequence output, for image captioning.
  - **Many-to-One**: sequence input, for sentiment classification
  - **Many-to-Many**: sequence in and out, for machine translation.
  - **Synchronized Many-to-Many**: synced sequences in and out, for video classification.

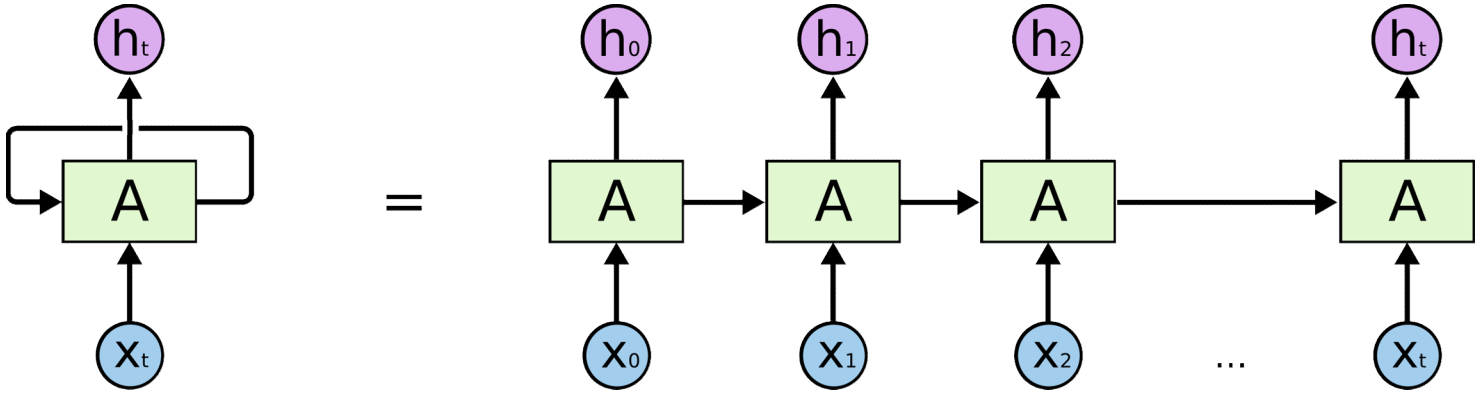


https://colah.github.io/posts/2015-08-Understanding-LSTMs/

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Recurrent Neural Networks

- Given a standard feedforward Multilayer Perceptron network, a recurrent neural network can be thought of as the addition of loops to the architecture.



Recurrent Neural Networks

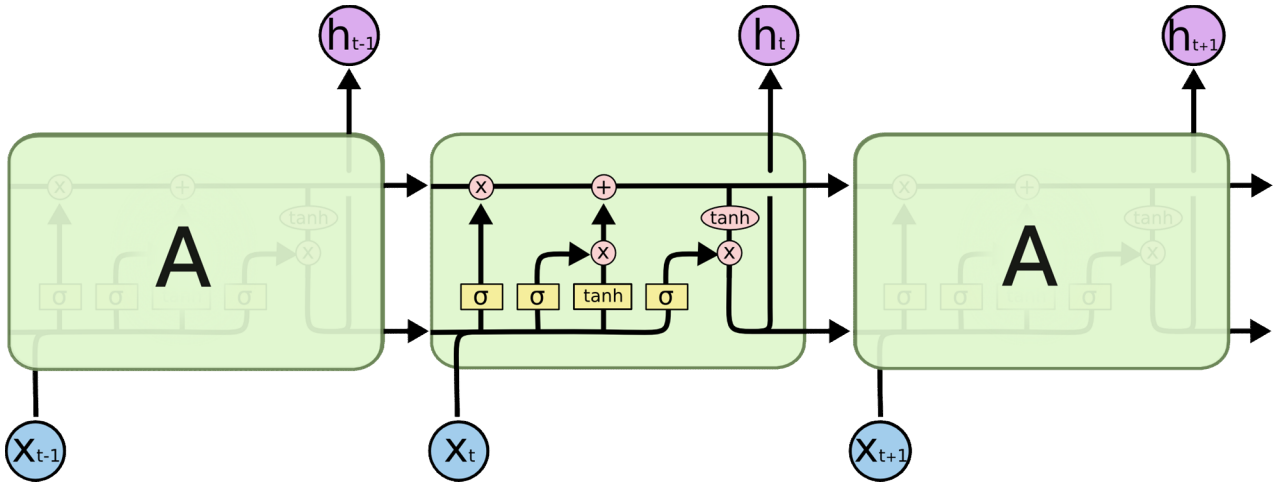
- For the techniques to be effective on real problems, two major issues needed to be resolved for the network to be useful.

    1. **How to train the network with Backpropagation.**
    Backpropagation breaks down in a recurrent neural network, because of the recurrent or loop connections. This was addressed with a modification of the Backpropagation technique called **Backpropagation Through Time or BPTT.**

    2. How to stop gradients vanishing or exploding during training.
    When Backpropagation is used in very deep neural networks and in unrolled recurrent neural networks, the gradients that are calculated in order to update the weights can become unstable.

Long Short-term Memory Network

The Long Short-Term Memory or LSTM network is a recurrent neural network that is trained using Backpropagation Through Time and overcomes the vanishing gradient problem.



Long Short-term Memory Network

- There are three types of gates within a memory unit:
  - **Forget Gate**: conditionally decides what information to discard from the unit.
  - **Input Gate**: conditionally decides which values from the input to update the memory state.
  - **Output Gate**: conditionally decides what to output based on input and the memory of the unit.

- Each unit is like a mini state machine where the gates of the units have weights that are learned during the training procedure.

Time Series Prediction with MLP

- How to phrase time series prediction as a regression problem and develop a neural network model for it

- How to frame time series prediction with a time lag and develop a neural network model for it.

- First method – Multilayer perceptron Regression
- Second method – Multilayer perceptron Using the Window Method

Activity 1

Time Series Prediction with LSTM

- How to develop LSTM networks for a time series prediction problem framed as regression.
- How to develop LSTM networks for a time series prediction problem using a window for both features and time steps.
- How to develop and make predictions using LSTM networks that maintain state (memory) across very long sequences.

- LSTMs are sensitive to the scale of the input data, specifically when the sigmoid. (default) or tanh activation functions are used.
- LSTM Network expects the input data (X) to be provided with a specific array structure in the form of. [samples, time steps, features]

Activity 2

LSTM with Memory Between Batches

- The LSTM network has memory which is capable of remembering across long sequences.
- Normally, the state within the network is reset after each training batch when fitting the model, as well as each call to model.predict() or model.evaluate()
- We can gain finer control over when the internal state of the LSTM network is cleared in Keras by making the LSTM layer stateful.
- This means that it can build state over the entire training sequence and even maintain that state if needed to make predictions.

Stacked LSTMs with Memory Between Batches

- LSTM networks can be stacked in Keras in the same way that other layer types can be stacked.
- One addition to the configuration that is required is that an LSTM layer prior to each subsequent LSTM layer must return the sequence. This can be done by setting the return sequences parameter on the layer to True. We can extend the stateful LSTM in the previous section to have two layers, as follows:

```
01 model.add(LSTM(4, batch_input_shape=(batch_size, look_back,
02 1), stateful=True, return_sequences=True))
03 model.add(LSTM(4, batch_input_shape=(batch_size, look_back,
04 1), stateful=True))
```

Project: Sequence Classification of Movie Reviews

- How to develop an LSTM model for a sequence classification problem.
- How to reduce overfitting in your LSTM models through the use of dropout.
- How to combine LSTM models with Convolutional Neural Networks that excel at learning spatial relationships.

- Simple LSTM for Sequence Classification
- LSTM For Sequence Classification With Dropout
  - Recurrent Neural networks like LSTM generally have the problem of overfitting. Dropout can be applied between layers using the Dropout Keras layer.
- LSTM and CNN for Sequence Classification

Activity 3

Sequence Classification of Movie Reviews

- LSTM and CNN For Sequence Classification
  - The IMDB review data does have a one-dimensional spatial structure in the sequence of words in reviews and the CNN may be able to pick out invariant features for good and bad sentiment.
  - We can easily add a one-dimensional CNN and max pooling layers after the Embedding layer which then feed the consolidated features to the LSTM.
  - We can use a smallish set of 32 features with a small filter length of 3. The pooling layer can use the standard length of 2 to halve the feature map size.

```
01 model = Sequential()
02 model.add(Embedding(top_words, embedding_vecor_length,
   input_length=max_review_length))
03 model.add(Conv1D(filters=32, kernel_size=3, padding='same',
   activation='relu'))
04 model.add(MaxPooling1D(pool_size=2))
05 model.add(LSTM(100))
06 model.add(Dense(1, activation='sigmoid'))
```

Quiz

http://bit.ly/2Y8MTi2

Generative Adversarial Networks

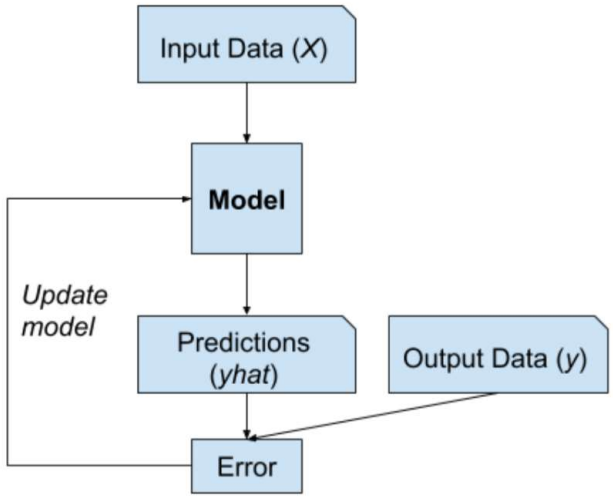
- GANs have been defined as the most interesting idea in the last 10 years in Machine Learning. – Yann LeCun

What are GANs

- Generative modelling 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 an adversarial zero-sum game until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

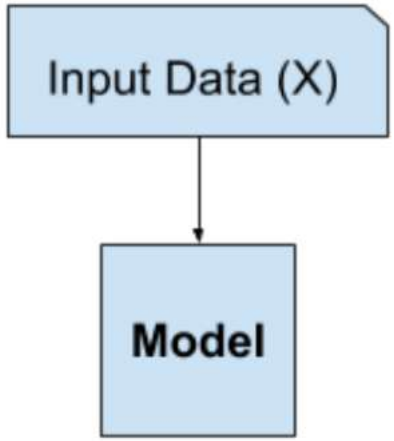
Supervised vs. Unsupervised Learning

- 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.
- This correction of the model is generally referred to as a supervised form of learning, or supervised learning. It is supervised because there is a real expected outcome to which a prediction is compared.



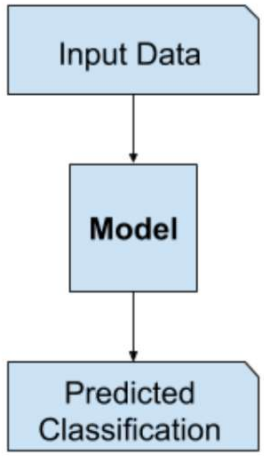
Supervised vs. Unsupervised Learning

- 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.



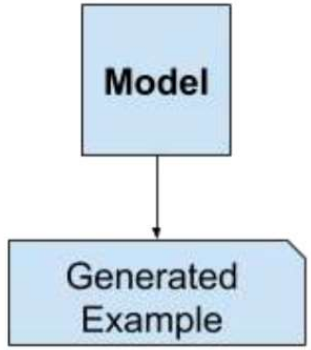
Discriminative vs Generative Modelling

- In supervised learning, we may be interested in developing a model to predict a class label given an example of input variables. This predictive modelling task is called classification. Classification is also traditionally referred to as **discriminative** modelling.
- 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.



Discriminative vs Generative Modelling

- 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 examples that plausibly fit into the distribution of the input variable***.
- 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.



Discriminative vs Generative Modelling

- Naive Bayes is an example of a generative model that is more often used as a discriminative model.
  - 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
  - Latent Dirichlet Allocation, or LDA,
  - the Gaussian Mixture Model, or GMM
- Deep Learning methods
  - Variational Autoencoder, or VAE,
  - Generative Adversarial Network, or GAN

What are GANs?

- 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.
- The initial models worked but where unstable and difficult to train.
- 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*.

What are GANs?

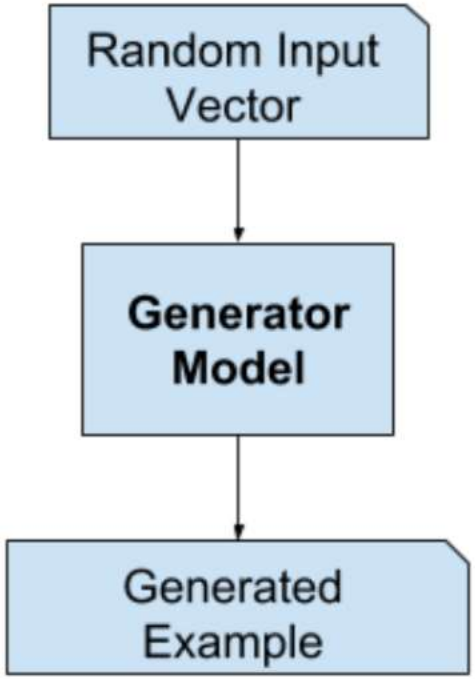
- The GAN model architecture involves two sub-models: a generator model for generating new examples and a discriminator model for classifying whether generated examples are real (from the domain) or fake (generated by the generator model).
  - **Generator**. Model that is used to generate new plausible 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.

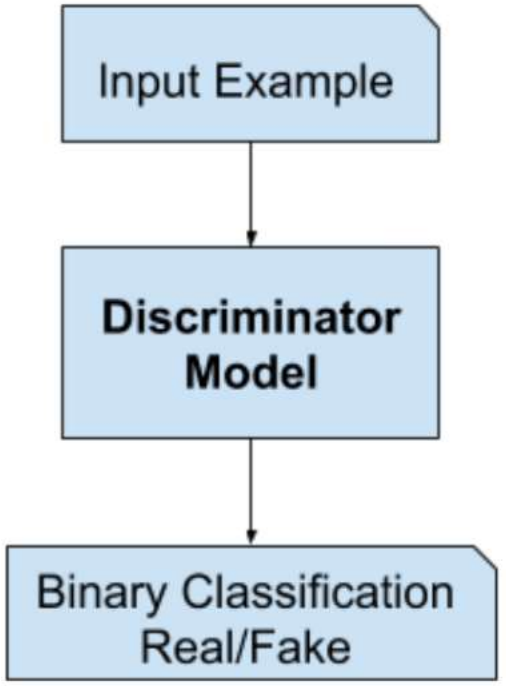
The Generator Model

- The generator model takes a fixed-length random vector as input and generates a sample in the domain, such as an image.
- This **vector space is referred to as a latent space**, or a vector space comprised of latent variables.
- 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.
- After training, the generator model is kept and used to generate new samples.



The Discriminator Model

- The discriminator model takes an example from the problem 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 classification model.
- After the training process, the discriminator model is discarded as we are interested in the generator.

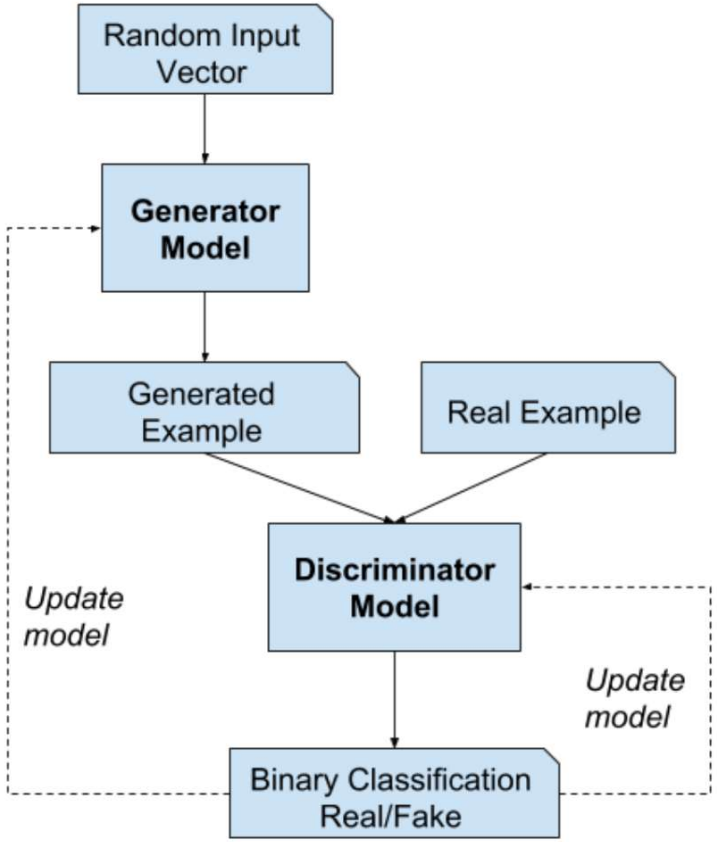


GANs as a Two Player Game

- Generative modelling is an unsupervised learning problem, 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.***
- 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.

GANs as a Two Player Game

- 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.



GANs and CNNs

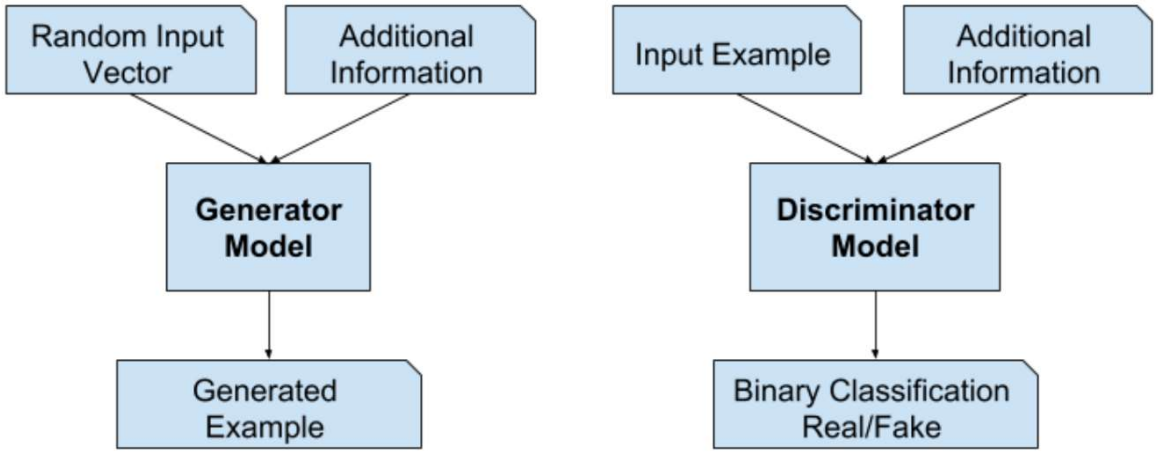
- Modelling image data means that the latent space, the input to the generator, provides a compressed representation of the set of images or photographs used to train the model.
- It also means that the generator generates new images, providing an output that can be easily viewed and assessed by developers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

Conditional GANs

- 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.



Why GANs?

- Successful generative modelling provides an alternative and potentially more domain-specific approach for data augmentation.
- The most compelling application of GANs is in conditional GANs for tasks that require the generation of new examples. Here, Goodfellow indicates three main examples:
  - **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.

How to Upsample with CNN?

- Two common types of layers that can be used in the generator model are
  - a **upsample** layer that simply doubles the dimensions of the input and
  - the **transpose** convolutional layer that performs an inverse convolution operation.

Upsample

- 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.
- For example, an input image with the shape 2 × 2 would be output as 4 × 4.

```
        1, 2
Input = 3, 4

         1, 1, 2, 2
Output = 1, 1, 2, 2
         3, 3, 4, 4
         3, 3, 4, 4
```

```
...
# define model
model = Sequential()
model.add(UpSampling2D())
```

* By default, the UpSampling2D will double each input dimension

Upsample

- You may want to use different factors on each dimension, such as double the width and triple the height. This could be achieved by setting the size argument to (2, 3).

```
# example of using different scale factors for each dimension
model.add(UpSampling2D(size=(2, 3)))
```

- The result of applying this operation to a 2 × 2 image would be a 4×6 output image (e.g. 2×2 and 2×3).
- by default, the **UpSampling2D** layer will use a nearest neighbor algorithm to fill in the new rows and columns. This has the effect of simply doubling rows and columns, as described and is specified by the interpolation argument set to '**nearest**'.
- Alternately, a bilinear interpolation method can be used which draws upon multiple surrounding points. This can be specified via setting the interpolation argument to '**bilinear**'.

```
...
# example of using bilinear interpolation when upsampling
model.add(UpSampling2D(interpolation="bilinear"))
```

Transpose

- A simple way to think about it is that it both performs the upsample operation and interprets the coarse input data to fill in the detail while it is upsampling.
- It is like a layer that combines the UpSampling2D and Conv2D layers into one layer.
- In fact, the transpose convolutional layer performs an inverse convolution operation. Specifically, the forward and backward passes of the convolutional layer are reversed
- It is sometimes called a **deconvolution** or **deconvolutional** layer and models that use these layers can be referred to as deconvolutional networks, or deconvnets.

Transpose

- The transpose convolutional layer is much like a normal convolutional layer.
- It requires that you specify the number of filters and the kernel size of each filter. The key to the layer is the stride.
- Typically, the stride of a convolutional layer is (1 × 1), that is a filter is moved along one pixel horizontally for each read from left-to-right, then down pixel for the next row of reads. *A stride of 2 × 2 on a normal convolutional layer has the effect of downsampling the input, much like a pooling layer.* In fact, a 2 × 2 stride can be used instead of a pooling layer in the discriminator model.

Transpose

- The transpose convolutional layer is like an inverse convolutional layer. As such, you would, intuitively think that a 2 × 2 stride would upsample the input instead of downsample, which is exactly what happens.
- Stride or strides refers to the manner of a filter scanning across an input in a traditional convolutional layer.
- Whereas, in a transpose convolutional layer, stride refers to the manner in which outputs in the feature map are laid down.
- *This effect can be implemented with a normal convolutional layer using a fractional input stride (f) , e.g. with a stride of f = 1/2*
- When inverted, the output stride is set to the numerator of this fraction, e.g. f = 2.

Transpose

- Assuming a single filter with a 1 x 1 kernel and model weights that result in no change to the inputs when output (e.g. a model weight of 1.0 and a bias of 0.0)

```
      1, 2
Input = 3, 4
```

```
       1, 2
Output = 3, 4
```

- With an output stride of (2,2), the 1 × 1 convolution requires the insertion of additional rows and columns into the input image so that the reads of the operation can be performed

```
        1, 0, 2, 0
Input = 0, 0, 0, 0
        3, 0, 4, 0
        0, 0, 0, 0
```

```
         1, 0, 2, 0
Output = 0, 0, 0, 0
         3, 0, 4, 0
         0, 0, 0, 0
```

Conv2DTranspose

- Keras provides the transpose convolution capability via the Conv2DTranspose layer.
- It can be added to your model directly; for example:

```
# define model
model = Sequential()
model.add(Conv2DTranspose(...))
```

GAN Training

- The algorithm is summarized in the figure below, taken from the original 2014 paper by Goodfellow, et al. titled Generative Adversarial Networks.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

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

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

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

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

How to train GAN in practice

- We can implement the discriminator directly by configuring the discriminator model to predict a probability of 1 for real images and 0 for fake images and minimizing the cross-entropy loss, specifically the binary cross-entropy loss.
- The generator is trained to maximize the discriminator predicting a high probability of class = 1 for generated (fake) images.
  - This is achieved by updating the generator via the discriminator with the class label of 1 for the generated images.
- The discriminator is not updated in this operation but provides the gradient information required to update the weights of the generator model.
  - For example, if the discriminator predicts a low average probability of being real for the batch of generated images, then this will result in a large error signal propagated backward into the generator given the expected probability for the samples was 1.0 for real. This large error signal, in turn, results in relatively large changes to the generator to hopefully improve its ability at generating fake samples on the next batch.

How to train GAN in practice

- This can be implemented in Keras by creating a composite model that combines the generator and discriminator models, allowing the output images from the generator to flow into discriminator directly, and in turn, allow the error signals from the predicted probabilities of the discriminator to flow back through the weights of the generator model.

How to train GAN in practice

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
numpy.random.seed(7)
```

The composite model can then be updated using fake images and real class labels.

```
...
# generate points in the latent space
z = randn(latent_dim * n_batch)
# reshape into a batch of inputs for the network
z = z.reshape(n_batch, latent_dim)
# define target labels for real images
y_real = ones((n_batch, 1))
# update generator model
gan_model.train_on_batch(z, y_real)
```

GAN Hacks to Train Stable Models

- The reason they are difficult to train is that both the generator model and the discriminator model are trained simultaneously in a game. This means that improvements to one model come at the expense of the other model.
- The goal of training two models involves finding a point of equilibrium between the two competing concerns.
- In neural network terms, the technical challenge of training two competing neural network at the same time is that they can fail to converge.
- Instead of converging, GANs may suffer from one of a small number of failure modes. A common failure mode is that instead of finding a point of equilibrium, the generator oscillates between generating specific examples in the domain.

GAN Hacks to Train Stable Models

- Perhaps the most challenging model failure is the case where multiple inputs to the generator result in the generation of the same output. This is referred to as **mode collapse**, and may represent one of the most challenging issues when training GANs.

DCGAN

- Perhaps one of the most important steps forward in the design and training of stable GAN models was the 2015 paper by Alec Radford, et al. titled Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In the paper, they describe the Deep Convolutional GAN, or DCGAN, approach to GAN development that has become the de facto standard.

Stabilization of GAN learning remains an open problem. Fortunately, GAN learning performs well when the model architecture and hyperparameters are carefully selected. Radford et al. (2015) crafted a deep convolutional GAN (DCGAN) that performs very well for image synthesis tasks ...

GAN Hacks to Train Stable Models

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.

* Summary of Architectural Guidelines for Training Stable Deep Convolutional Generative Adversarial Networks. Taken from: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

7 Best Practices for the DCGAN model architecture

1. Downsample Using Strided Convolutions
2. Upsample Using Strided Convolutions
3. Use Leaky ReLU
4. Use Batch Normalization
5. Use Gaussian Weight Initialization
6. Use Adam Stochastic Gradient Descent
7. Scale Images to the Range [-1,1]

Soumith Chintala's GAN Hacks

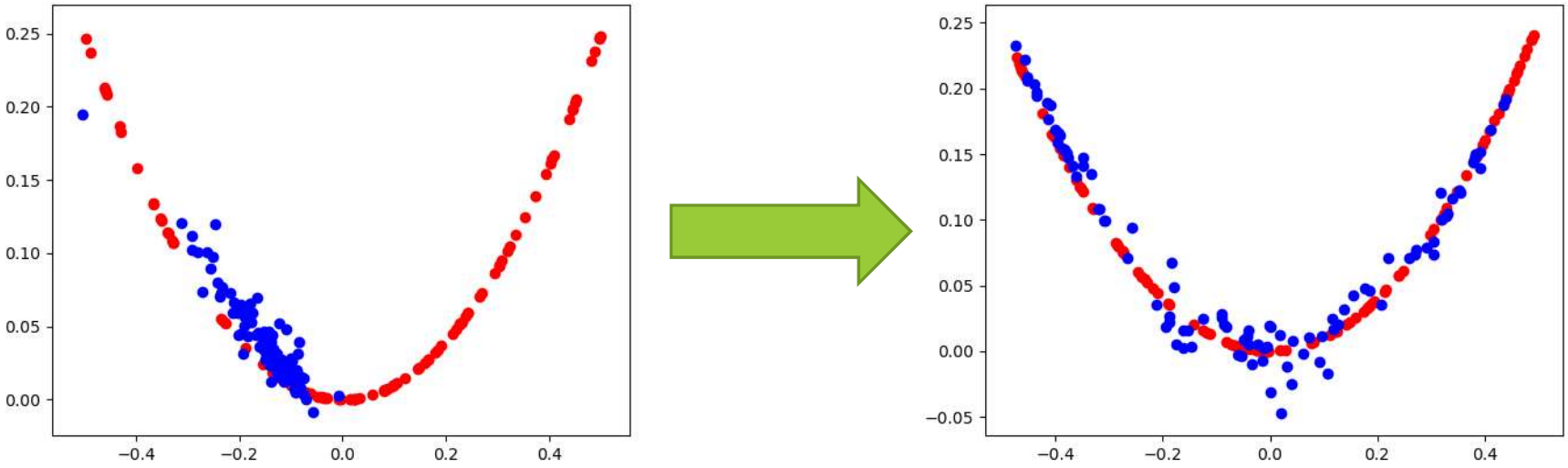
- Soumith Chintala, one of the co-authors of the DCGAN paper, made a presentation at NIPS 2016 titled How to Train a GAN? summarizing many tips and tricks. The video is available on YouTube and is highly recommended.

https://www.youtube.com/watch?v=X1mUN6dD8uE
https://github.com/soumith/ganhacks

-

Develop a 1D GAN

Objective: A generator to output a vector with two elements: one for the input and one for the output of our one-dimensional function.



Activity 4

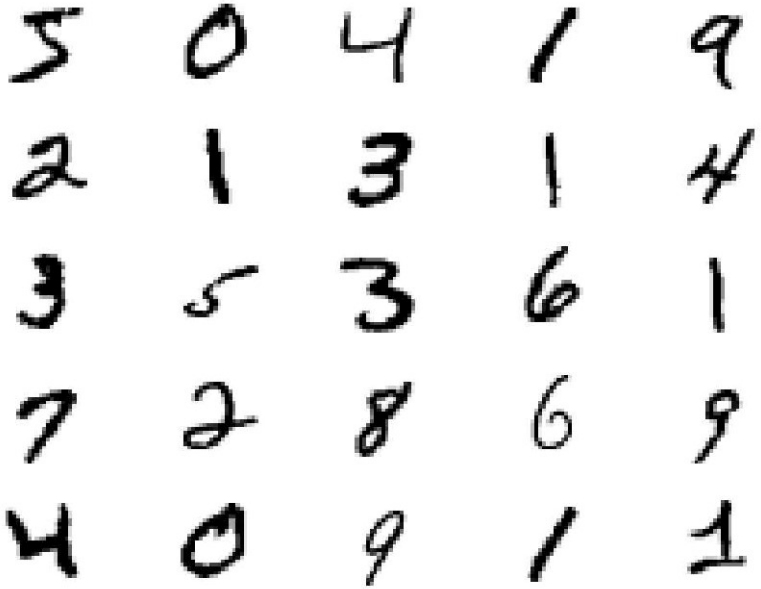
Develop a 1D GAN

1. Select a One-Dimensional Function
2. Define a Discriminator Model
3. Define a Generator Model
4. Training the Generator Model
5. Evaluating the Performance of the GAN
6. Complete Example of Training the GAN

Activity 4

Develop a DCGAN for Grayscale Handwritten Digits

Objective: A generator that learns how to generate new plausible handwritten digits between 0 and 9



Activity 5

Develop a DCGAN for Grayscale Handwritten Digits

1. MNIST Handwritten Digit Dataset
2. How to Define and Train the Discriminator Model
3. How to Define and Use the Generator Model
4. How to Train the Generator Model
5. How to Evaluate GAN Model Performance
6. Complete Example of GAN for MNIST
7. How to Use the Final Generator Model

Activity 5

Application of GAN

- StackGAN: text to Photo-Realistic Image synthesis with Stacked Generative Adversarial Networks (https://github.com/hanzhanggit/StackGAN)

Further Reading

- Generative Adversarial Networks, 2014.
  - https://arxiv.org/abs/1406.2661
- NIPS 2016 Tutorial: Generative Adversarial Networks, 2016.
  - https://arxiv.org/abs/1701.00160
- Wasserstein GAN, 2017.
  - https://arxiv.org/abs/1701.07875

Quiz

http://bit.ly/2Y8MTi2

Thank you