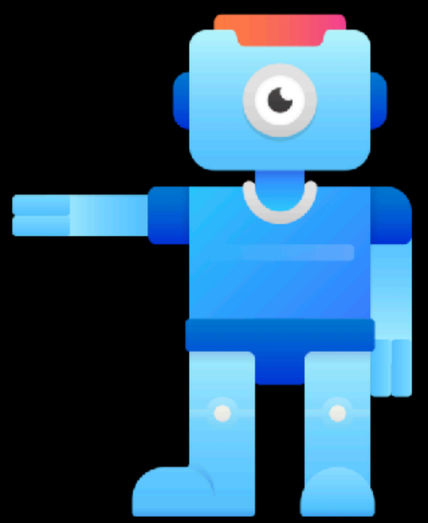


MODERN COMPUTER VISION

BY RAJEEV RATAN

Top-1 and Top-5 Accuracy (aka Rank-1, Rank-5)

How we benchmark large multi-class classification models

Rank-N

- Rank-N Accuracy is a way to measure a classifier's accuracy with a bit more leeway.
- Image our classifier returns this output:

Class Name	Probability
Shetland sheepdog	0.44
collie	0.31
chow	0.1
wire-haired fox terrie	0.09
lion	0.06

- It'll return **'Shetland sheepdog'** as the predicted class (due to highest probability). However, that would be incorrect.
- The correct class **'collie'** is actually the second most probable, which means the classifier is still doing quite well, but is not reflected if we look only at the top predicted class.

Rank-N or Top-N Accuracy

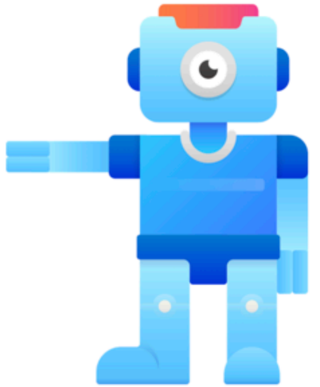
- Rank-N Accuracy considers the top N classes with the highest probabilities.

Class Name	Probability
Shetland sheepdog	0.44
collie	0.31
chow	0.1
wire-haired fox terrie	0.09
lion	0.06

- For example, Rank-5, will consider any of the top 5 most likely classes for the predicted label.

Assessing Classifier Performance

RANK	MODEL	TOP 1 ACCURACY ↑	TOP 5 ACCURACY	NUMBER OF PARAMS	EXTRA TRAINING DATA	PAPER
1	**Meta Pseudo Labels** (EfficientNet-L2)	90.2%	98.8%	480M	✓	Meta Pseudo Labels
2	**Meta Pseudo Labels** (EfficientNet-B6-Wide)	90%	98.7%	390M	✓	Meta Pseudo Labels
3	**NFNet-F4+**	89.2%		527M	✓	High-Performance Large-Scale Image Recognition Without Normalization
4	**ALIGN** (EfficientNet-L2)	88.64%	98.67%	480M	✓	Scaling Up Visual and Vision-Language Representation Learning With Noisy Text Supervision
5	**EfficientNet-L2-475** (SAM)	88.61%		480M	✓	Sharpness-Aware Minimization for Efficiently Improving Generalization

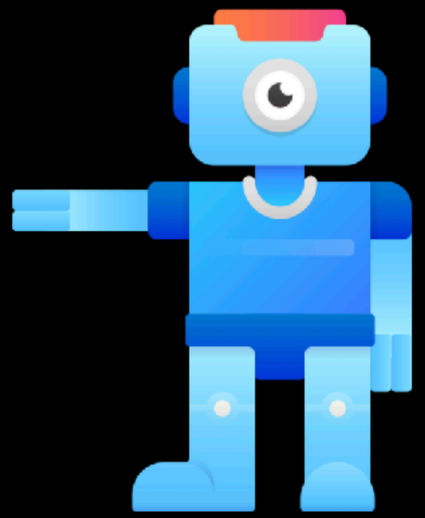


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Callbacks - Early Stopping, Check Pointing, LR Schedule and more



MODERN COMPUTER VISION

BY RAJEEV RATAN

Callbacks - Early Stopping, Check Pointing, LR Schedule and more

Using CallBack methods in Keras and PyTorch

Callbacks

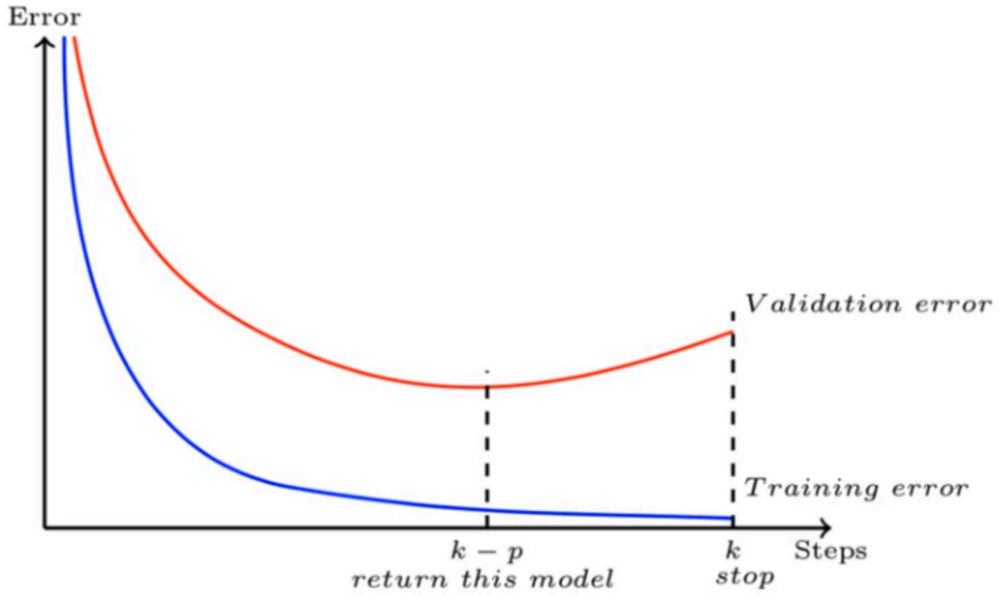
- **Callbacks** are used to perform different actions at different stages during training.
- **Why is this useful?**
  - What if we wanted to save each model after each epoch?
  - What if we set our model to train for 100 epochs but after 30, it started to overfit? Wouldn’t we want some way stop training?
  - What if we wanted to log our information somewhere so that it can analysed later on?

Callbacks are the solution

- Callbacks can be used to perform:
  - Early Stopping
  - Model Checkpointing
  - Learning Rate Scheduler
  - Logging
  - Remote Monitoring
  - Custom Functions

Early Stopping - A Solution for Overfitting

- During our training process our validation loss may **stagnate** or **stop decreasing** and sometimes actually start to increase (overfitting)
- We can use Callbacks to implement **Early Stopping**



Model Checkpointing

- During Training we can periodically save the weights after each epoch.
- This allows us to resume training in the event of a crash
- The checkpoint file contains the model's weights or the model itself
- This can usually be configured in many ways

```
tf.keras.callbacks.ModelCheckpoint(
    filepath,
    monitor="val_loss",
    verbose=0,
    save_best_only=False,
    save_weights_only=False,
    mode="auto",
    save_freq="epoch",
    options=None,
    **kwargs
)
```

Learning Rate Scheduler

- We can avoid having our loss oscillate around the global minimum by attempting to reduce the Learn Rate by a specified amount.
- If no improvement is seen in our monitored metric (val_loss typically), we wait a certain number of epochs (patience) then this callback reduces the learning rate by a factor.

```
from keras.callbacks import ReduceLROnPlateau

reduce_lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.2, patience = 3, verbose = 1, min_delta = 0.0001)
```

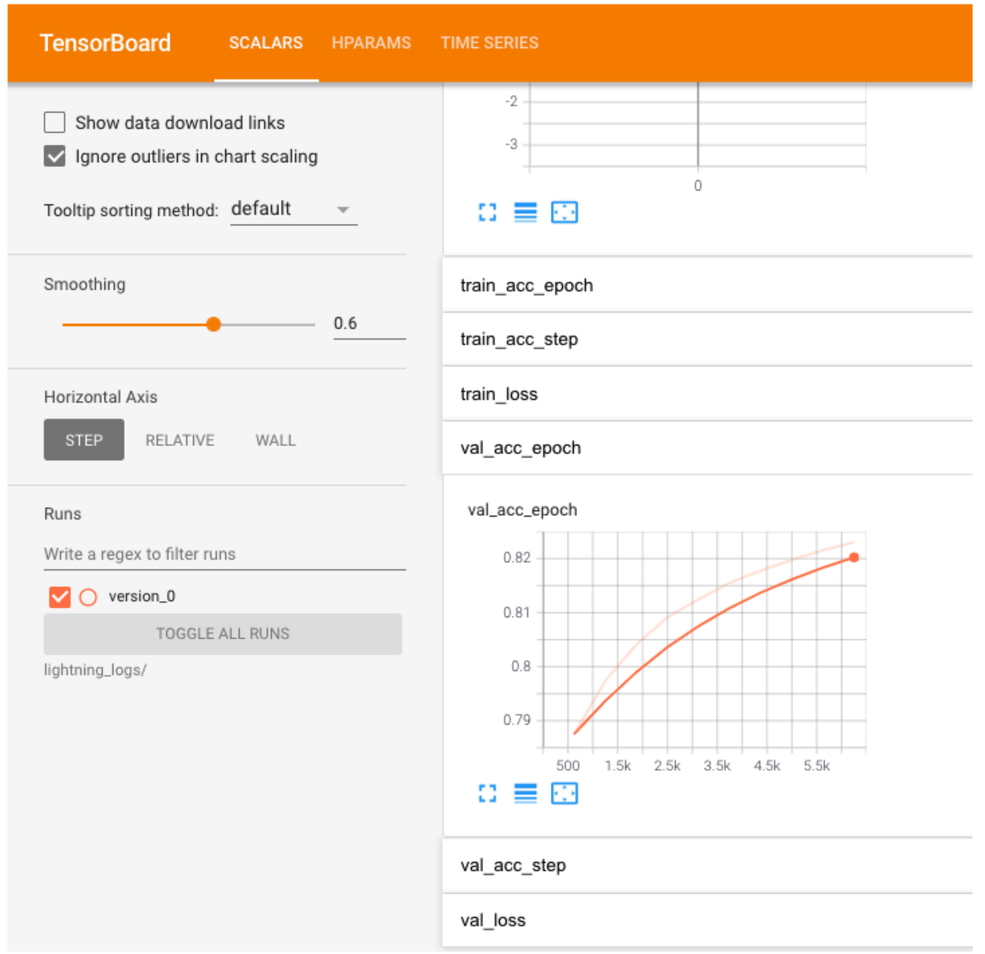
Logging

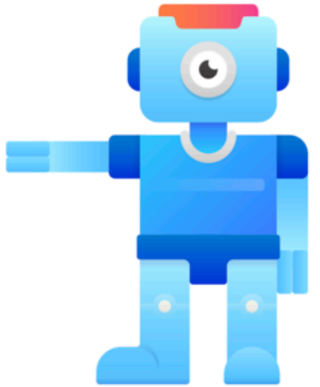
- We can automatically log our model's training stats (training and validation loss/accuracy) and view them later using TensorBoard or others.

```
my_callbacks = [
    tf.keras.callbacks.EarlyStopping(patience=2),
    tf.keras.callbacks.ModelCheckpoint(filepath='model.{epoch:02d}-{val_loss:.2f}.h5')
    tf.keras.callbacks.TensorBoard(log_dir='./logs'),
]
model.fit(dataset, epochs=10, callbacks=my_callbacks)
```

```
tf.keras.callbacks.TensorBoard(
    log_dir="logs",
    histogram_freq=0,
    write_graph=True,
    write_images=False,
    update_freq="epoch",
    profile_batch=2,
    embeddings_freq=0,
    embeddings_metadata=None,
    **kwargs
)
```

```
# Start tensorboard.
%load_ext tensorboard
%tensorboard --logdir lightning_logs/
```



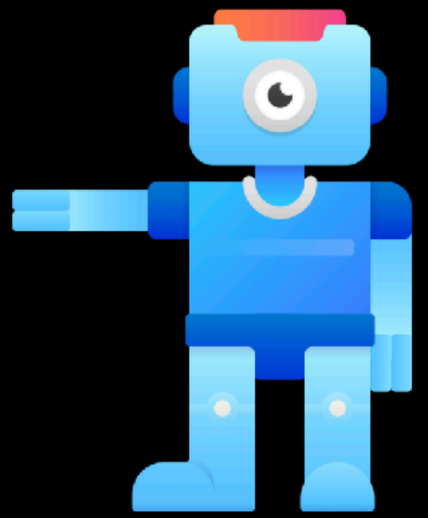


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Callbacks - Early Stopping, Check Pointing, LR Schedule and more



MODERN COMPUTER VISION

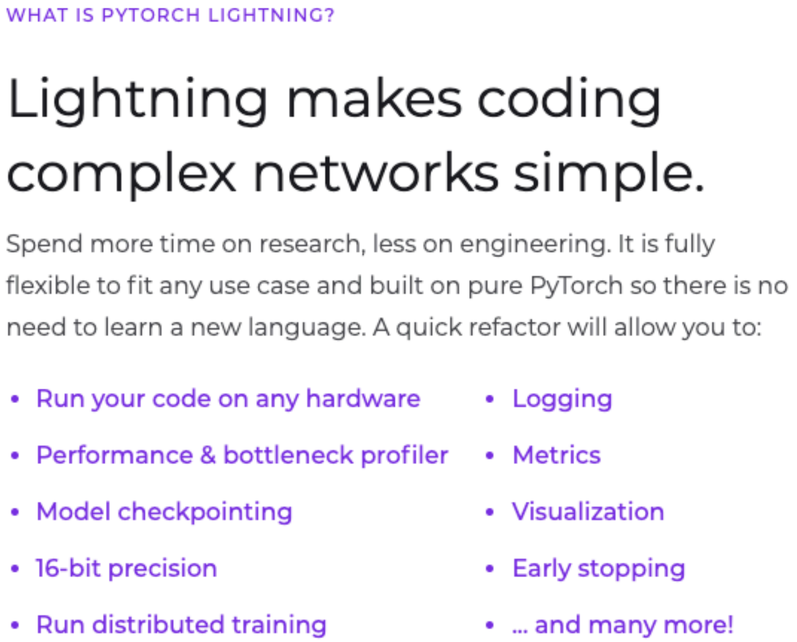
BY RAJEEV RATAN

PyTorch Lightning

A feature rich easy to use PyTorch High Level Interface Library

PyTorch Lightning

- **PyTorch Lightning** is an open-source Python library that provides a high-level interface for PyTorch.



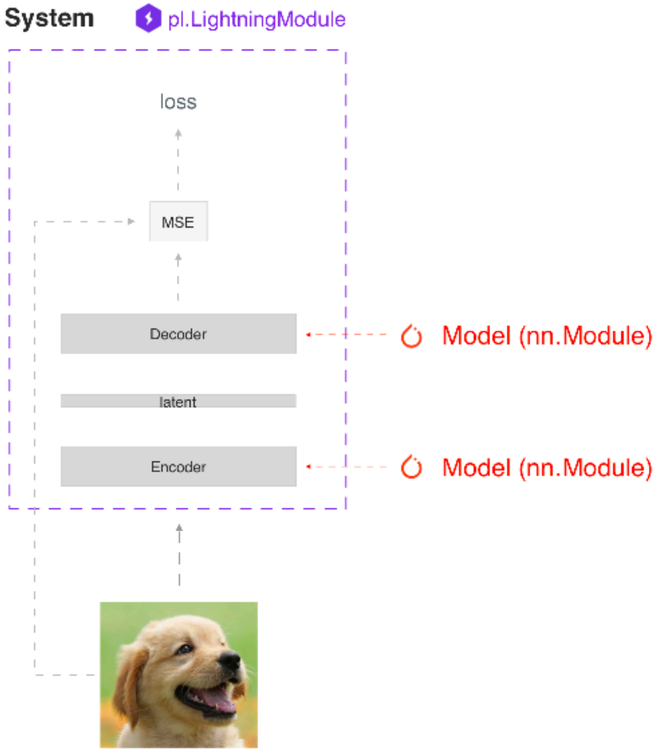
https://www.pytorchlightning.ai/

Why Use Lightning?

- It offers a nice easy to use, modular approach to coding PyTorch Models.
- It was intended for researchers to focus more on science and research and spend less time worrying about how they'd deploy or scale their models during training.
- A model created using Lightning can be trained on multiple-GPUs, TPUs, with Floating Point 16 precision etc.!
- It decouples research code from engineering
- Integrates easily with logging tools such as TensorBoard, MLFlow, Nepture.ai, Comet.ai and Wandb.
- Provides auto-batch selection, learning rate selection and callbacks.

The Lightning Philosophy

- Lightning proposes we encase all our model's building blocks into a **Lightning Module**.
- It requires basically us reorganising our existing PyTorch Code



The Lightning Module

- The **Lightning Module** organises our PyTorch code into 5 sections:
  - Computations (init).
  - Train loop (training_step)
  - Validation loop (validation_step)
  - Test loop (test_step)
  - Optimizers (configure_optimizers)

Minimal Example

Here are the only required methods.

```
>>> import pytorch_lightning as pl
>>> class LitModel(pl.LightningModule):
...
...     def __init__(self):
...         super().__init__()
...         self.l1 = nn.Linear(28 * 28, 10)
...
...     def forward(self, x):
...         return torch.relu(self.l1(x.view(x.size(0), -1)))
...
...     def training_step(self, batch, batch_idx):
...         x, y = batch
...         y_hat = self(x)
...         loss = F.cross_entropy(y_hat, y)
...         return loss
...
...     def configure_optimizers(self):
...         return torch.optim.Adam(self.parameters(), lr=0.02)
```

The Lightning Module - Computations

PyTorch

```
[ ]  import torch
     from torch import nn

     class MNISTClassifier(nn.Module):

       def __init__(self):
         super(MNISTClassifier, self).__init__()

         # mnist images are (1, 28, 28) (channels, width, height)
         self.layer_1 = torch.nn.Linear(28 * 28, 128)
         self.layer_2 = torch.nn.Linear(128, 256)
         self.layer_3 = torch.nn.Linear(256, 10)

       def forward(self, x):
         batch_size, channels, width, height = x.size()

         # (b, 1, 28, 28) -> (b, 1*28*28)
         x = x.view(batch_size, -1)

         # layer 1
         x = self.layer_1(x)
         x = torch.relu(x)

         # layer 2
         x = self.layer_2(x)
         x = torch.relu(x)

         # layer 3
         x = self.layer_3(x)

         # probability distribution over labels
         x = torch.log_softmax(x, dim=1)

         return x
```

PyTorch Lightning

```
[ ]  import torch
     from torch import nn
     import pytorch_lightning as pl

     class LightningMNISTClassifier(pl.LightningModule):

       def __init__(self):
         super(LightningMNISTClassifier, self).__init__()

         # mnist images are (1, 28, 28) (channels, width, height)
         self.layer_1 = torch.nn.Linear(28 * 28, 128)
         self.layer_2 = torch.nn.Linear(128, 256)
         self.layer_3 = torch.nn.Linear(256, 10)

       def forward(self, x):
         batch_size, channels, width, height = x.siz()

         # (b, 1, 28, 28) -> (b, 1*28*28)
         x = x.view(batch_size, -1)

         # layer 1
         x = self.layer_1(x)
         x = torch.relu(x)

         # layer 2
         x = self.layer_2(x)
         x = torch.relu(x)

         # layer 3
         x = self.layer_3(x)

         # probability distribution over labels
         x = torch.log_softmax(x, dim=1)

         return x
```

https://towardsdatascience.com/from-pytorch-to-pytorch-lightning-a-gentle-introduction-b371b7caaf09

The Lightning Module - Trainers

PyTorch

```
# ----------------------
# TRAINING LOOP
# ----------------------
num_epochs = 1
for epoch in range(num_epochs):

   # TRAINING LOOP
   for train_batch in mnist_train:
      x, y = train_batch

      logits = pytorch_model(x)
      loss = cross_entropy_loss(logits, y)
      print('train_loss: ', loss.item())

      loss.backward()
      optimizer.step()
      optimizer.zero_grad()


  # VALIDATION LOOP
  with torch.no_grad():
      val_loss = []
      for val_batch in mnist_val:

          x, y = val_batch
          logits = pytorch_model(x)
          val_loss = cross_entropy_loss(logits, y).item()
          val_loss.append(val_loss)

      val_loss = torch.mean(torch.tensor(val_loss))

      print('val_loss:', val_loss.item())
```

PyTorch Lightning

```
class LightningMNISTClassifier(pl.LightningModule):


  def training_step(self, train_batch, batch_idx):
      x, y = train_batch
      logits = self.forward(x)
      loss = self.cross_entropy_loss(logits, y)
      self.log('train_loss', loss)
      return loss

  def validation_step(self, val_batch, batch_idx):
      x, y = val_batch
      logits = self.forward(x)
      loss = self.cross_entropy_loss(logits, y)
      self.log('val_loss', loss)


(automatically reduced across epochs)
```

https://towardsdatascience.com/from-pytorch-to-pytorch-lightning-a-gentle-introduction-b371b7caaf09

The Lightning Module - Optimiser & Loss

PyTorch

```
pytorch_model = MNISTClassifier()
optimizer = torch.optim.Adam(pytorch_model.parameters(), lr=1e-3)
```

PyTorch Lightning

```
class LightningMNISTClassifier(pl.LightningModule):

    def configure_optimizers(self):
      optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
      return optimizer
```

PyTorch

```
from torch.nn import functional as F

def cross_entropy_loss(logits, labels):
  return F.nll_loss(logits, labels)
```

PyTorch Lightning

```
from torch.nn import functional as F

class LightningMNISTClassifier(pl.LightningModule):

  def cross_entropy_loss(self, logits, labels):
    return F.nll_loss(logits, labels)
```

https://towardsdatascience.com/from-pytorch-to-pytorch-lightning-a-gentle-introduction-b371b7caaf09

The Lightning Module - Trainer

PyTorch

```
# ----------------
# TRAINING LOOP
# ----------------
num_epochs = 1
for epoch in range(num_epochs):

  # TRAINING LOOP
  for train_batch in mnist_train:
    x, y = train_batch

    logits = pytorch_model(x)
    loss = cross_entropy_loss(logits, y)
    print('train loss: ', loss.item())

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()

  # VALIDATION LOOP
  with torch.no_grad():
    val_loss = []
    for val_batch in mnist_val:
      x, y = val_batch
      logits = pytorch_model(x)
      val_loss.append(cross_entropy_loss(logits, y).item())

    val_loss = torch.mean(torch.tensor(val_loss))
    print('val_loss: ', val_loss.item())
```

PyTorch Lightning

```
# train loop + val loop + test loop
trainer = pl.Trainer()
trainer.fit(LightningMNISTClassifier())
```

https://towardsdatascience.com/from-pytorch-to-pytorch-lightning-a-gentle-introduction-b371b7caaf09

The Lightning Module - Data Modules

PyTorch

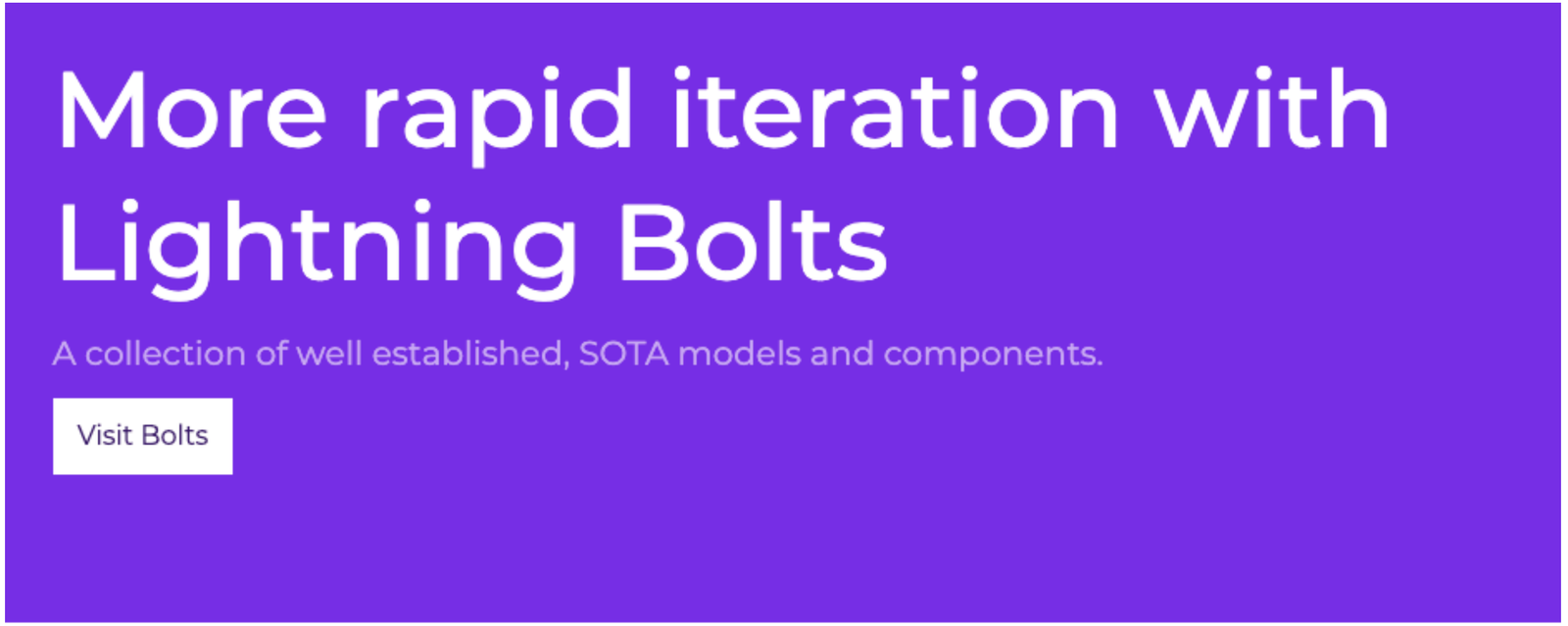
```
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import MNIST
import os
from torchvision import datasets, transforms


# ----------------
# TRANSFORMS
# ----------------
# prepare transforms standard to MNIST
transform=transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((0.1307,), (0.3081,))])

# ----------------
# TRAINING, VAL DATA
# ----------------
mnist_train = MNIST(os.getcwd(), train=True, download=True)

# train (55,000 images), val split (5,000 images)
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])

# ----------------
# TEST DATA
# ----------------
mnist_test = MNIST(os.getcwd(), train=False, download=True)

# ----------------
# DATALOADERS
# ----------------
# The dataloaders handle shuffling, batching, etc...
mnist_train = DataLoader(mnist_train, batch_size=64)
mnist_val = DataLoader(mnist_val, batch_size=64)
mnist_test = DataLoader(mnist_test, batch_size=64)
```

PyTorch Lightning

```
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import MNIST
import os
from torchvision import datasets, transforms

class MNISTDataModule(pl.LightningDataModule):

  def prepare_data(self):
    # prepare transforms standard to MNIST
    MNIST(os.getcwd(), train=True, download=True)
    MNIST(os.getcwd(), train=False, download=True)

  def train_dataloader(self):
    transform=transforms.Compose([transforms.ToTensor(),
                                  transforms.Normalize((0.1307,), (0.3081,))])
    mnist_train = MNIST(os.getcwd(), train=True, download=False,
                        transform=transform)
    self.mnist_train, self.mnist_val = random_split(mnist_train, [55000, 5000])

    mnist_train = DataLoader(mnist_train, batch_size=64)
    return mnist_train

  def val_dataloader(self):
    mnist_val = DataLoader(self.mnist_val, batch_size=64)
    return mnist_val

  def test_dataloader(self):
    transform=transforms.Compose([transforms.ToTensor(),
                                  transforms.Normalize((0.1307,), (0.3081,))])
    mnist_test = MNIST(os.getcwd(), train=False, download=False,
                       transform=transform)
    mnist_test = DataLoader(mnist_test, batch_size=64)
    return mnist_test
```

https://towardsdatascience.com/from-pytorch-to-pytorch-lightning-a-gentle-introduction-b371b7caaf09

Lightning Bolts



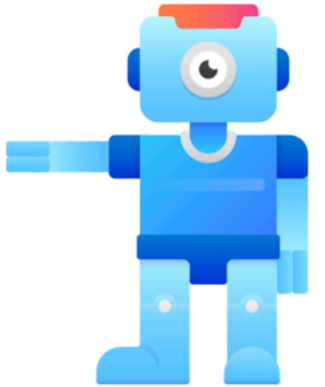
EVEN MORE RAPID ITERATION WITH LIGHTNING

Lightning Bolts

PyTorch Lightning Bolts is a community-built deep learning research and production toolbox, featuring a collection of well established and SOTA models and components, pre-trained weights, callbacks, loss functions, data sets, and data modules.

PyTorch Lightning Exercise

- Convert our Cats vs Dogs PyTorch Code into a Lightning Module
- Auto Batch size finder
- Auto Learning Rate Selector
- Implement Callbacks
  - Early Stopping
  - Model Checkpointing
  - Logging

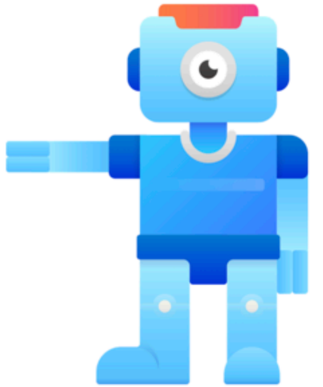


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

PyTorch Lightning Exercise

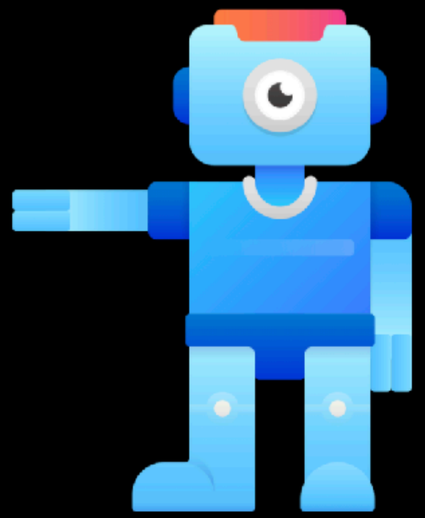


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Transfer Learning



MODERN COMPUTER VISION

BY RAJEEV RATAN

Transfer Learning

Why you don’t always need to train from scratch

Transfer Learning

- In practice, very few people train an entire Convolutional Network from scratch (i.e. random initialization of weights), because it is relatively rare to have a dataset of sufficient size.

- Instead, it is common to **pretrain** a ConvNet on a very large dataset (e.g. **ImageNet**, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an **initialization** or a **fixed feature extractor** for the task of interest. The three major Transfer Learning scenarios look as follows:

https://cs231n.github.io/transfer-learning/

Transfer Learning Rationale

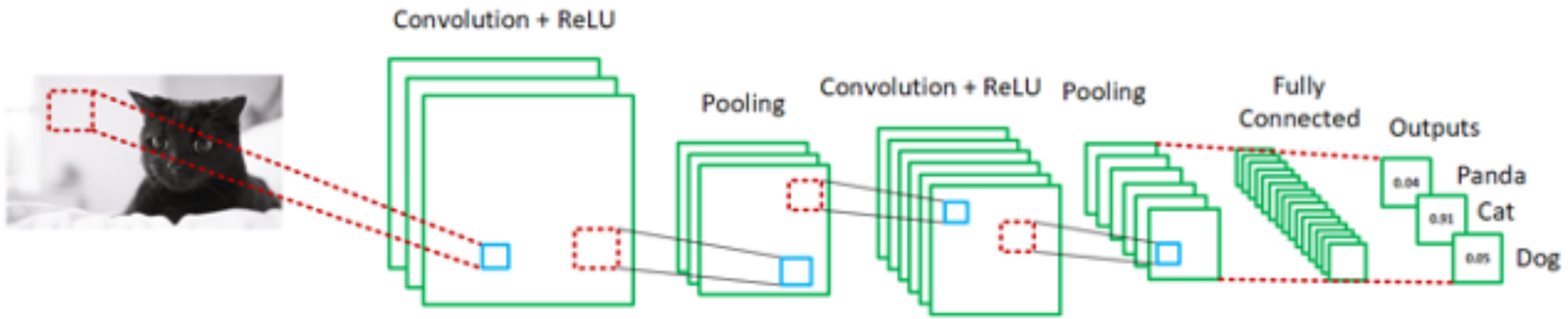
- Training extensive models on vast image datasets like ImageNet can sometimes take weeks.
- A model trained on so much data would have useful embeddings that can be applied to other image domains e.g. edge detectors, pattern and blob detectors
- Example, an ImageNet model trained to detect Tigers, Lions and Horses might be useful to detect other 4-legged mammals
- Transfer Learning is the concept where we utilise trained models in other domains to attain great accuracy as well as faster training times.

Transfer Learning - 2 Major Types

- Feature Extractor
- Fine Tuning

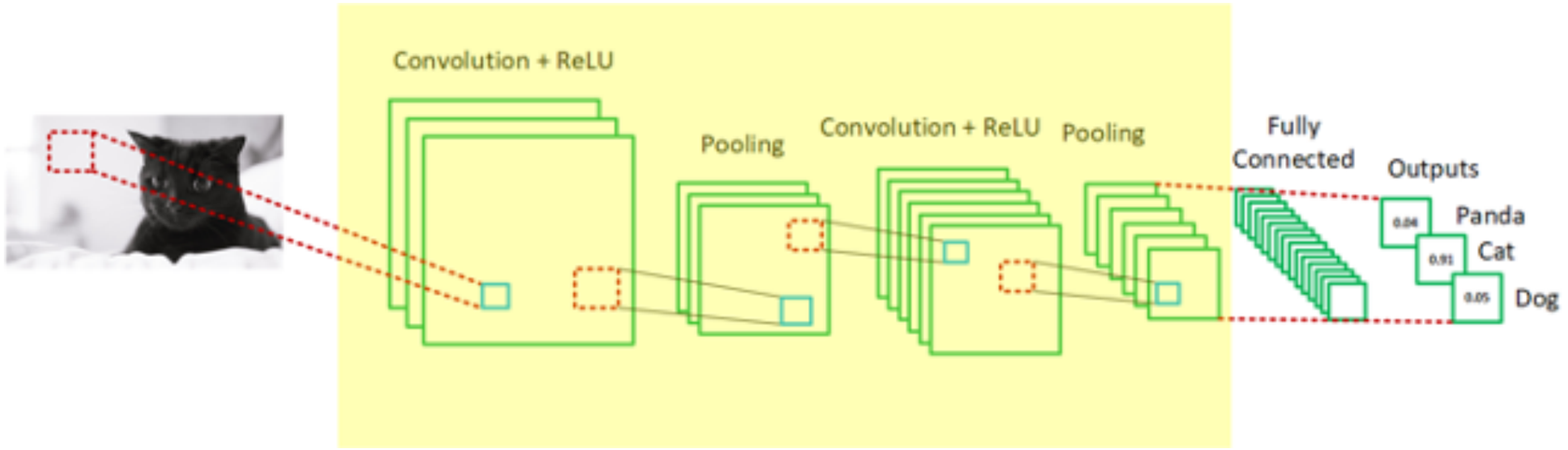
Transfer Learning - Feature Extractor

Take a pre-trained Network



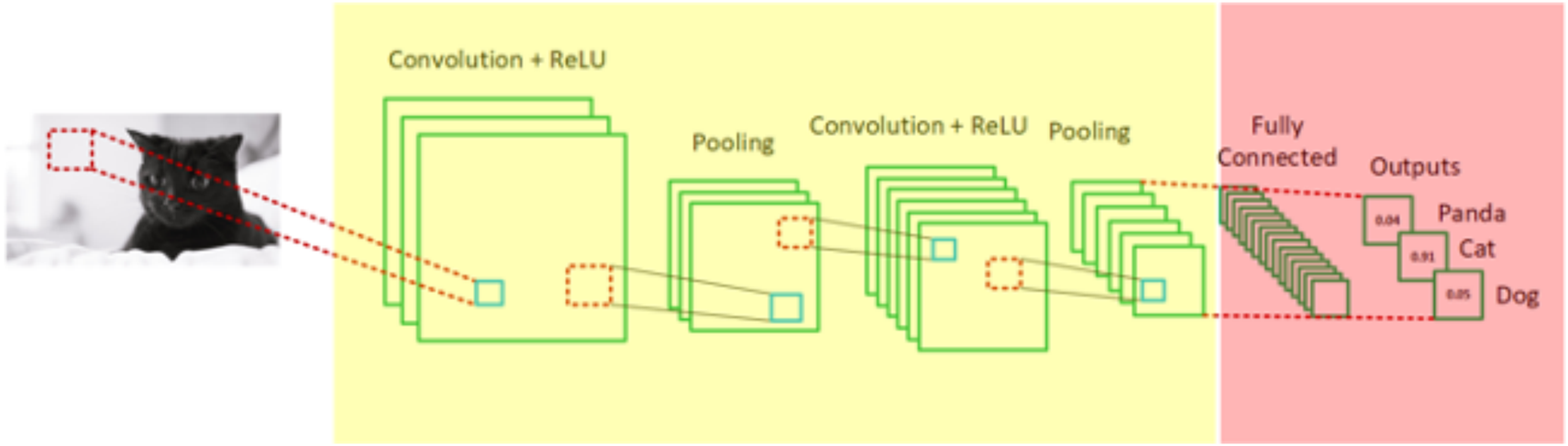
Transfer Learning - Feature Extractor

Freeze the CONV weights/layers - meaning they remain fixed



Transfer Learning - Feature Extractor

Replace the Top Layer with your Top Layer and train it



Transfer Learning - Feature Extractor

The steps involved in Transfer Learning by Feature Extraction

1. **Freeze bottom layers** of a pre-trained network

2. **Replace the top half of the network with your top**, so that it outputs only the number of classes in your dataset

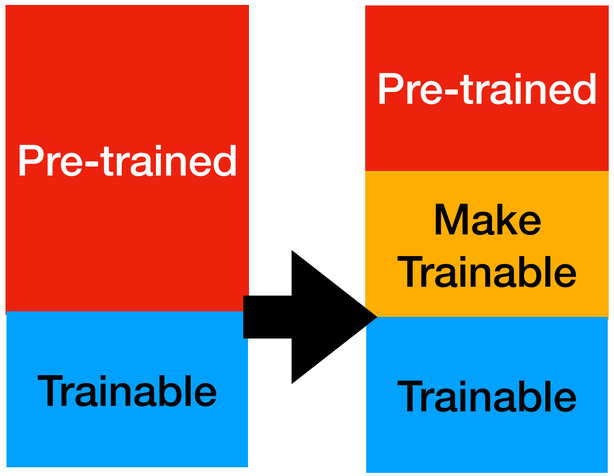
3. Train the model on your new dataset

Transfer Learning - Fine Tuning

In Fine Tuning, we complete all the steps in Feature Extraction but then we:

1. Unfreeze all or parts of the pre-trained model
2. Train the model for a few epochs, this is where we **‘fine tune’** the weights of the pre-trained model.

- The intuition behind this is earlier features maps in a ConvNet learn generic features, while the later layers learn specifics about the image dataset. By fine tuning we change those specifics from the pre-trained model to the specifics of our dataset.



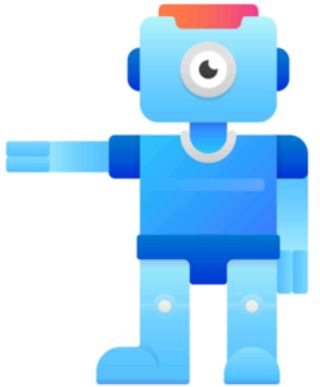
Transfer Learning - When Do We Use?

- Ideal scenario - New dataset is large and similar to the pre-trained original dataset. Models should not overfit.
- Not ideal but still recommended - New data is large but different.
- If data is small (even if similar) Transfer Learning and Fine Tuning can often overfit on the training data. A useful idea at times is to train a linear classier on the CNN outputs.

Transfer Learning Advice

- **Learning Rates** - use very small learning rates for pre-trained models especially when fine tuning. This is because we know the pertained weights are already very good and thus don't want to change time too much

- Due to parameter sharing you can train a pre-trained network on images of different sizes

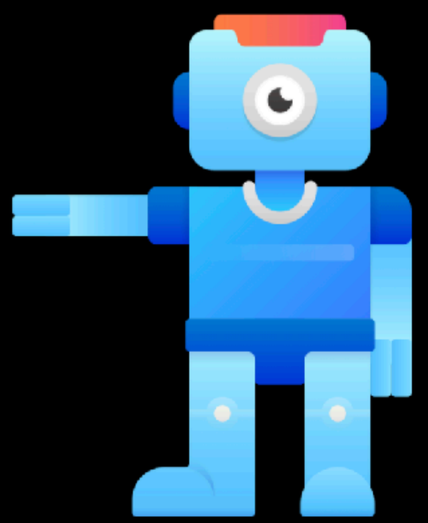


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Transfer Learning and Fine Tuning in Keras and PyTorch



MODERN COMPUTER VISION

BY RAJEEV RATAN

Google DeepDream

How to produce trippy effects in images

What is DeepDream?

- Simply, we take an image and then output ‘dreams’ or hallucinogenic effects on the image
- So what’s the process to do this?

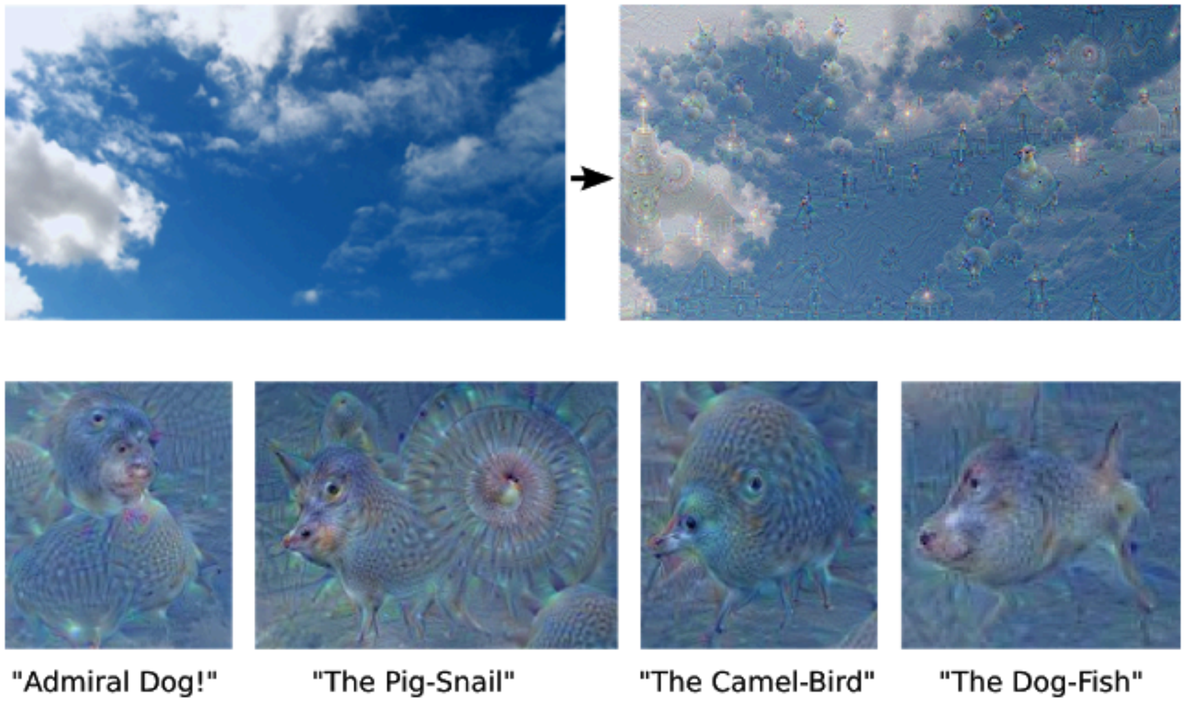


Image Source: https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.htm

Trippy isn't it

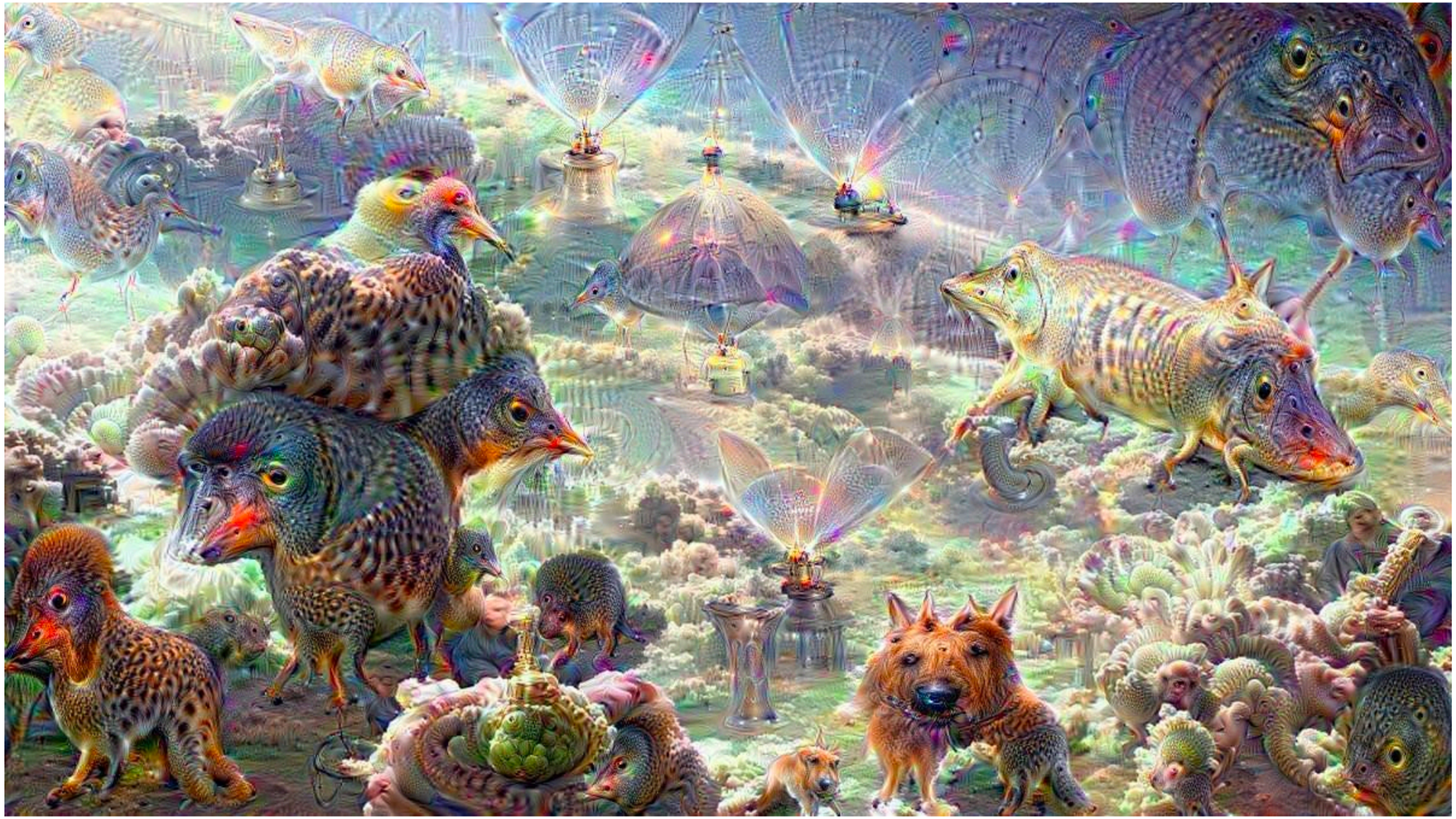


Image Source: https://www.kaggle.com/learn-forum/186491

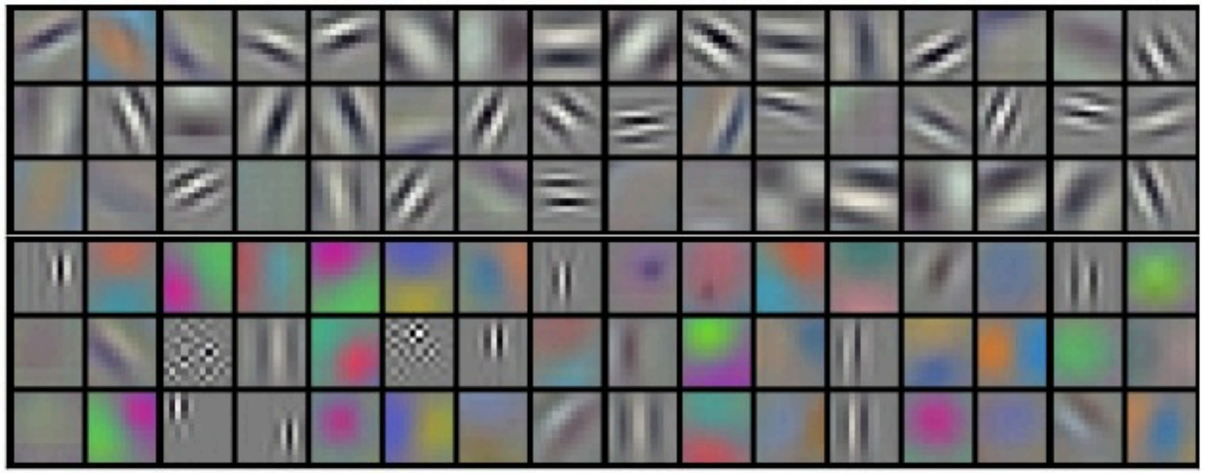
DeepDream Process Overview

- The DeepDream algorithm effectively is asking a pre-trained CNN to take a look at an image, identify patterns you recognise and then amplify it.
- It uses **representations** learned by CNNs to produce these hallucinogenic or ‘trippy’ images.
- It was introduced by C. Olah A. Mordvintsev and M. Tyka of Google Research in 2015 in publication titled “DeepDream: A Code Example for Visualizing Neural Networks”

Image Source: https://deepdreamgenerator.com

The DeepDream Algorithm - High Level

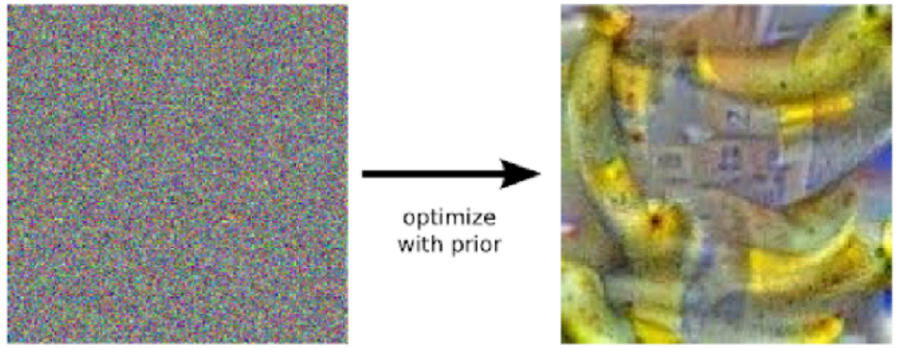
- Remember how we created our filter visualisations?

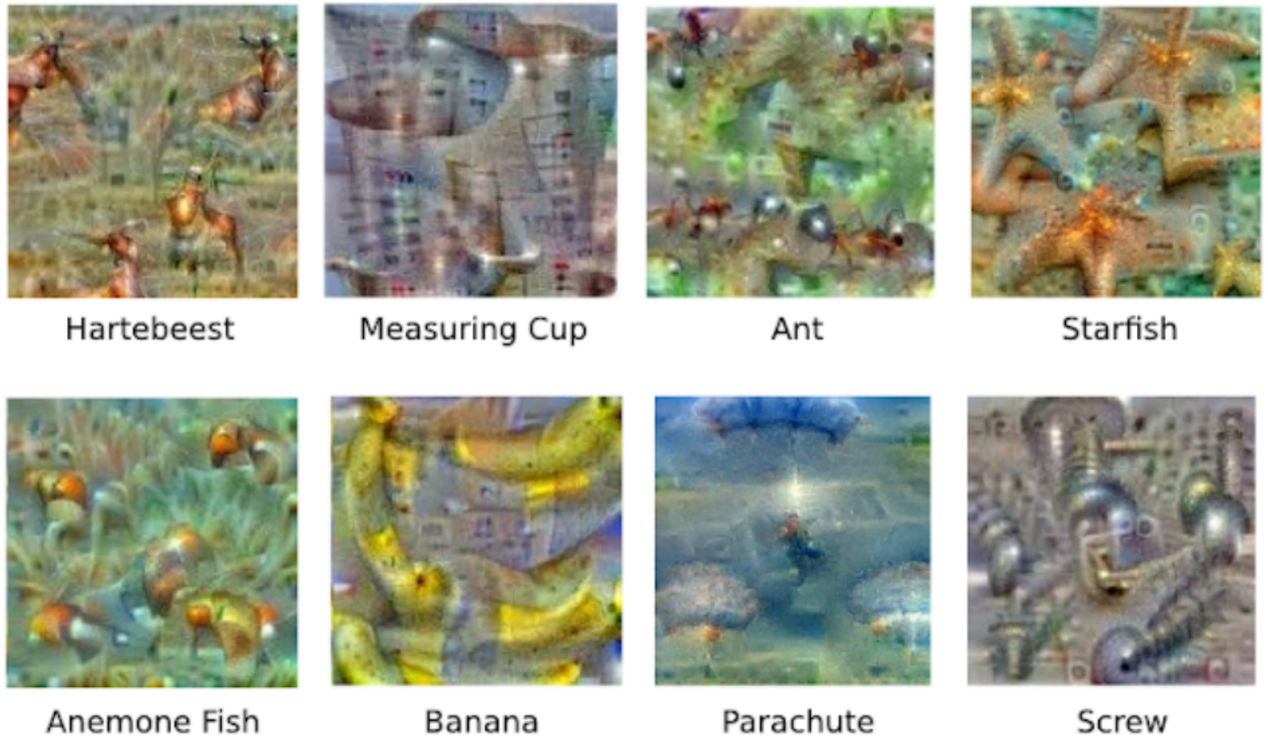


Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

The DeepDream Algorithm - High Level

- And class maximisation visualisations?

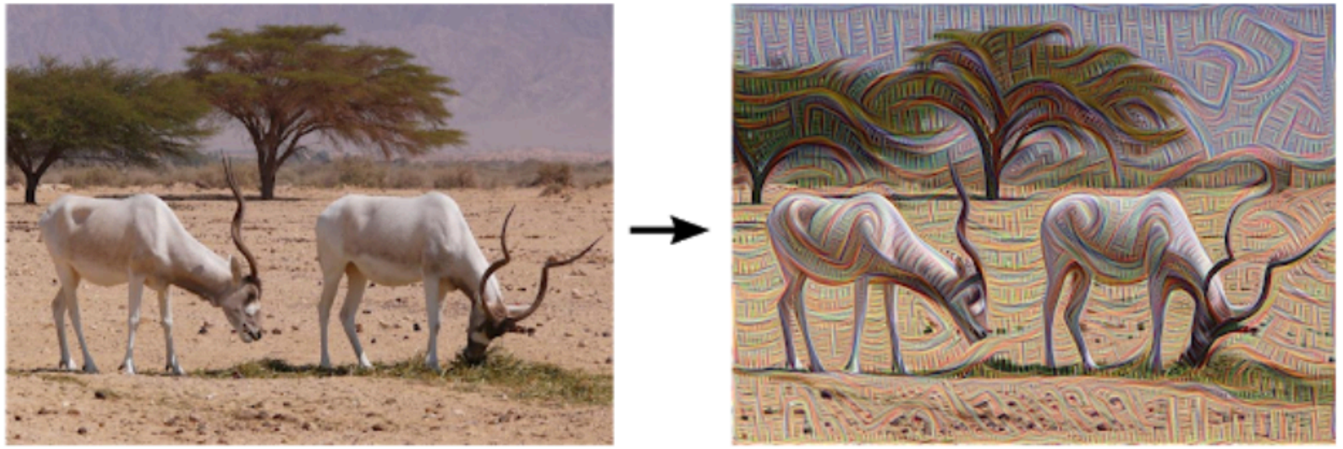




https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

The DeepDream Algorithm - High Level

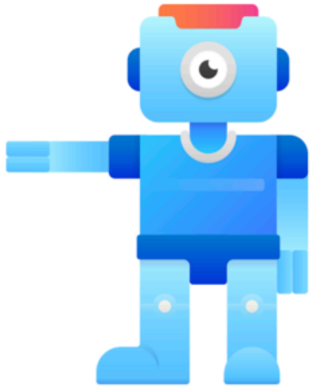
- In Deep Dream, instead of maximising a class output, we let the network decide.
- We feed it an input image and allow the network to analyse the image
- We then choose a layer and ask the network to amplify or enhance whatever it detected.
- Remember each layer deals with different features (high-level, low level) so the complexity of the pattern depends on these features. E.g. low level features look like brushes or strokes.



Left: Original photo by Zachi Evenor. Right: processed by Günther Noack, Software Engineer

The DeepDream Algorithm Low Level

- Get an input image
- Load a pertained model (VGG16, InceptionV3 etc.) to be used as a **feature extractor**
- **Calculate loss** which is the sum of the activations between the chosen layers. We also **normalise** at each layer so contributions from each layer are ‘equal’
- We use **gradient ascent** to do this, unlike gradient descent where we’re trying to find the local minima, gradient ascent tries to **maximise its loss** which allows the network to find less meaningful patters.
- Apply this at different scales (sizes) so that the patterns don’t occur at the same granularity

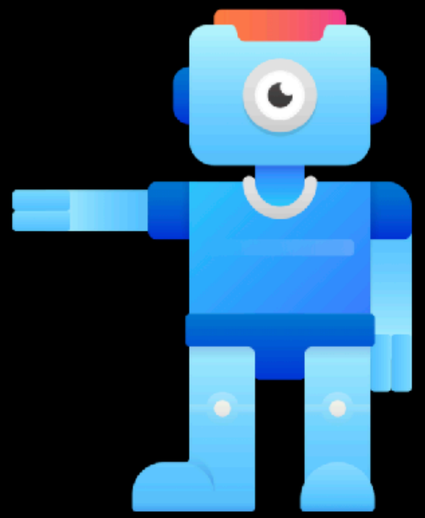


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Experiment with Google Deep Dream in Keras and PyTorch!



MODERN COMPUTER VISION

BY RAJEEV RATAN

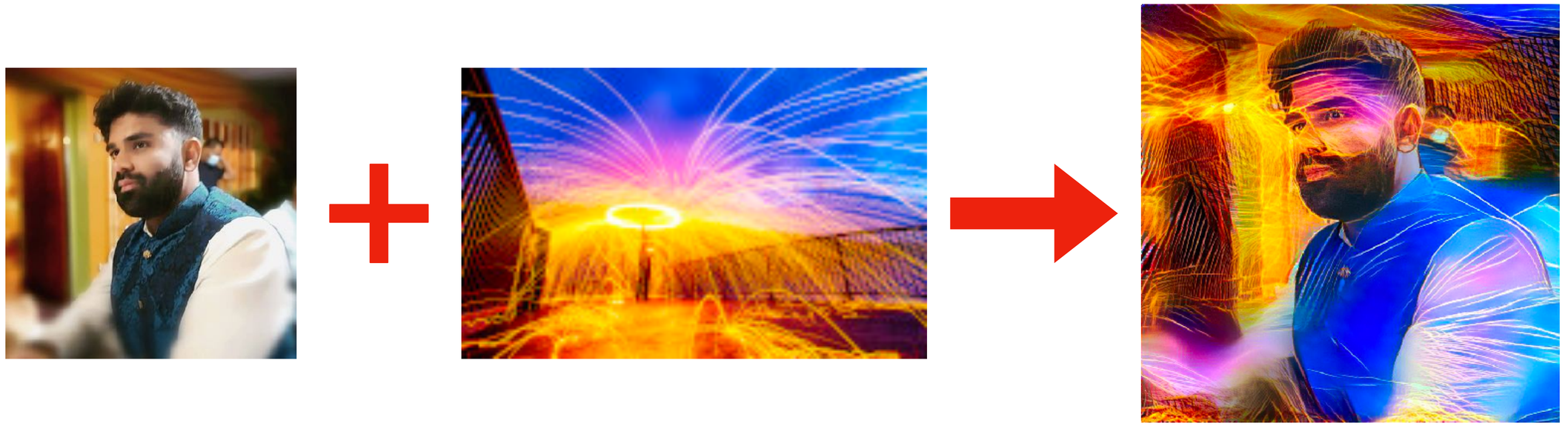
Neural Style Transfer

Copying Artistic Style onto images

Neural Style Transfer - AI Art

- Introduced by Leon Gatys et al. in 2015, in their paper titled "**A Neural Algorithm for Artistic Style**", the **Neural Style Transfer** algorithm went viral resulting in an explosion of further work and mobile apps.

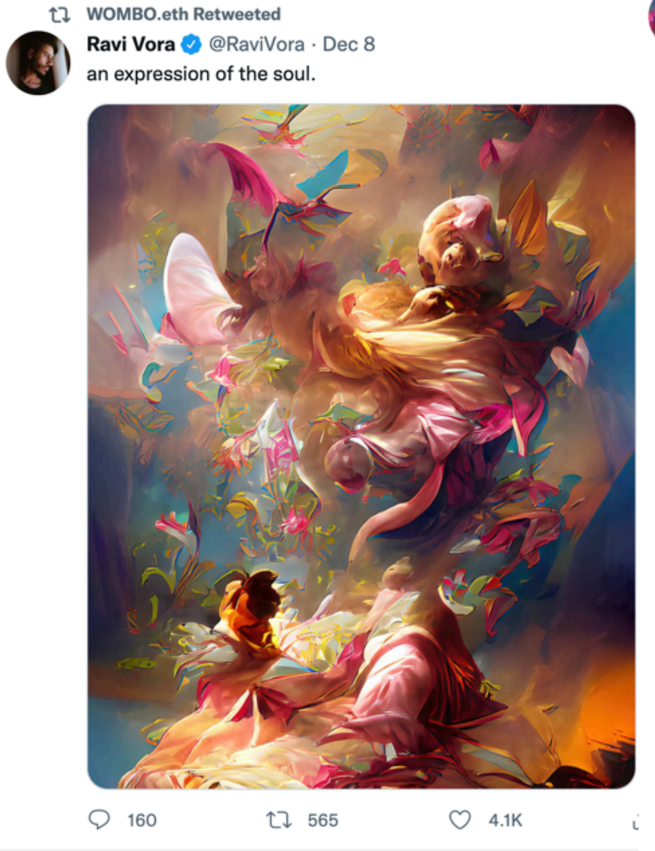
- Neural Style Transfer enables the **artistic style** of an image to be applied to another image! It copies the colour patterns, combinations and brush strokes of the original source image and applies it to your input image.



Source: DeepDreamGenerator.com

Amazing Apps likes Deep Dream Generator and Womba

Top: DeepDreamGenerator.com
Right: Womba.ai



How does it Work?

- NST uses two images, the **content** image and the **style** image
- We take the **style** of one image and **transfer** it onto the **content** of another image

Content image

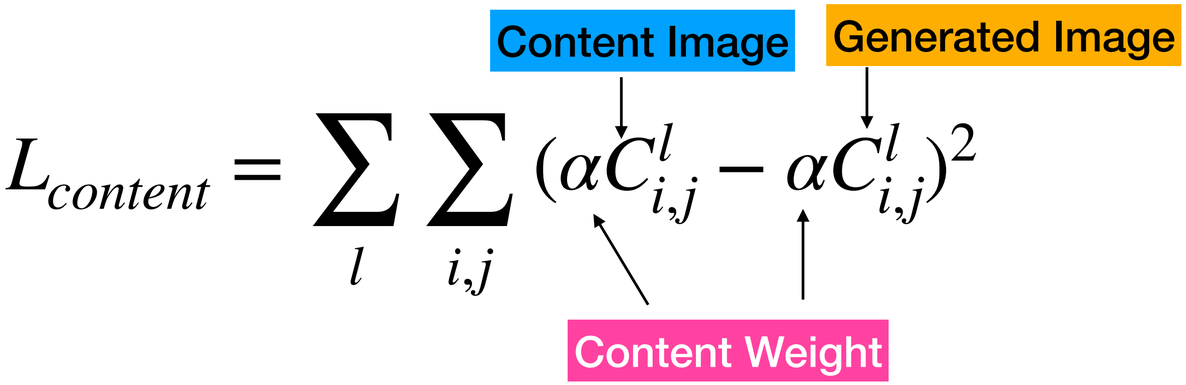
Style image

Source: DeepDreamGenerator.com

Neural Style Transfer using Neural Networks

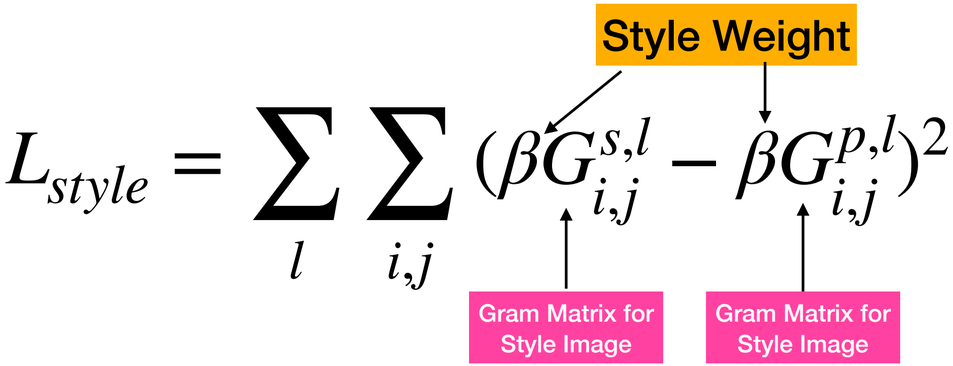
- We take a pre-trained network (VGG19 on ImageNet)
- Define and combine three loss functions which we will minimise
  - Content Loss
  - Style Loss
  - Total-variation Loss

Content Loss



- The content loss function measures how **similar the generated image is to the content image**
- It uses Euclidian distance (L2-Norm) to measure the difference between features of the content image and generated image
- We build this loss function by using a pre-trained Network like VGG16
- We then select a higher-level layer that serves as our Content Loss
- We then compute the activation of this layer for both the content and style image
- Then (as you can see above) we take the L2-Norm between these activations

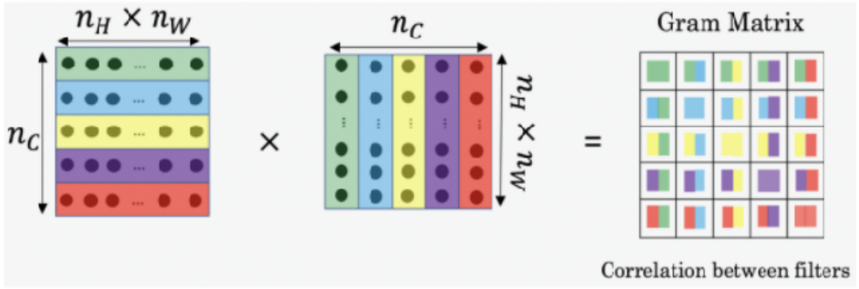
Style Loss



- Measures how **different** the **generated image** is (in style features) from your **style image**.
- Style loss uses multiple layers (or multi-scale representation), this allows it to capture
  - Low level features/layers such as edges
  - Mid level features such as blobs and corners
  - High level features such as more complex patterns
- The **style** representation is give by the **Gram Matrix**

Gram Matrix

- For the style loss, unlike the content loss, we don't just find the difference in activations.
- We find the correlation between these activations across different channels of the same layer.



Source: deeplearning.ai

- Each filter activates upon 'seeing' a feature e.g. a cat's nose.
- Now if another filter in that layer activates upon detecting a cat's eye this would mean they both activate together when detecting a cat. This means they're correlated.

Gram Matrix

- In order to capture the style but not the global arrangement of the content image, we must rely on the correlations between our feature values
- We take the the L2 norm between the gram matrix between layer activations
- We minimise loss between the style of the output image with our original style image, thereby forcing the style of the output image to correlate with the style of the style image.

Total Variation Loss

- The variation loss was included as it reduces noisy and pixelated looking outputs.
- It allows us to maintain smoothness and spatial continuity.
- $L_{total}(i,j,k) = \alpha L_{content}(i,k) + \beta L_{style}(j,k)$

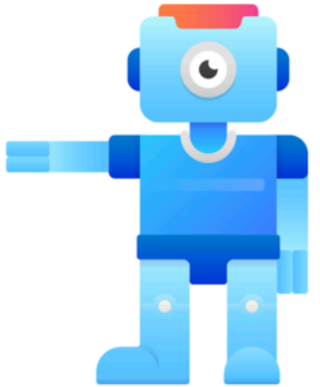
Content Weight

Style Weight

- It operates only on the output image with its goal being to enhance the output image.

Combining Loss Functions

- The final loss function is the **sum of all previous loss functions** with each component weighted
- This weighting allows us to **tweak** how we want our resultant image i.e. do we want more style or more of the content?
- To **implement** we start iterating (for a preset number) using gradient descent (sometimes using L-BFGS) to minimise our loss functions.

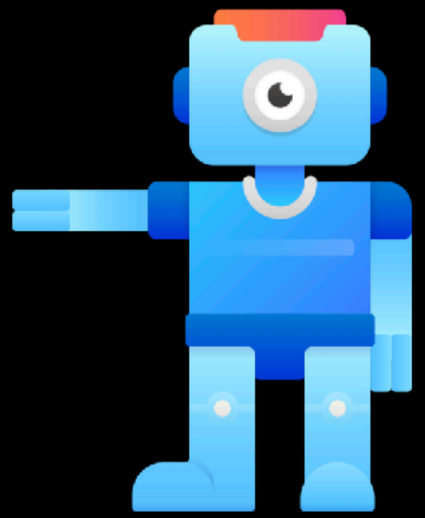


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Implement Neural Style Transfer in Keras and PyTorch



MODERN COMPUTER VISION

BY RAJEEV RATAN

Autoencoders

How Neural Networks can perform Representational Learning

Autoencoders

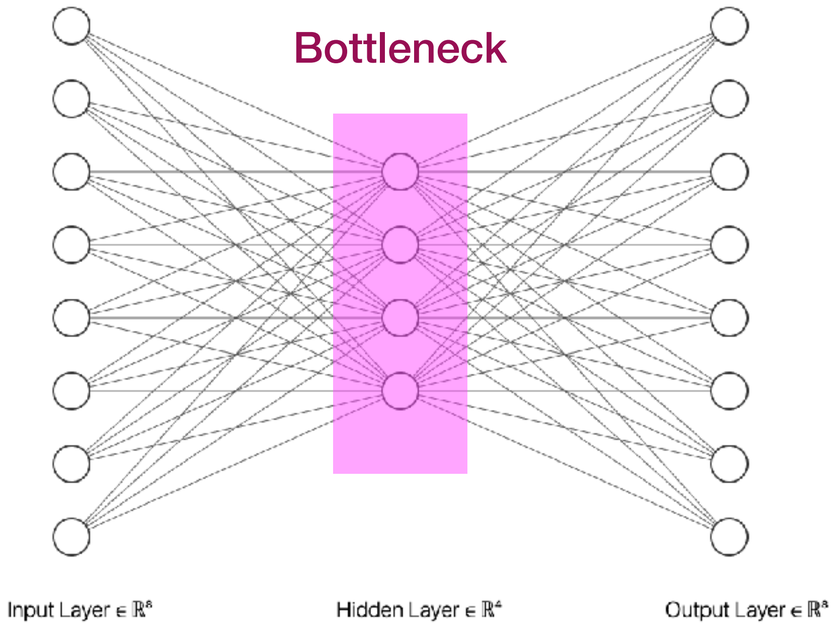
- An **Unsupervised** Learning technique that is used for **representational learning**
- Recall that our **CNN filters act as feature detectors**:
  - high level such as patterns
  - low level such as edges or blobs
- What if we could exploit what a CNN learns about a dataset so that it acts as a method of compression?

What do Autoencoders do?

- They learn to **compress** data based on their correlations between input features
- Some applications include:
  - Denoising (images, even audio)
  - Image Inpainting
  - Information Retrieval
  - Anomaly Detection

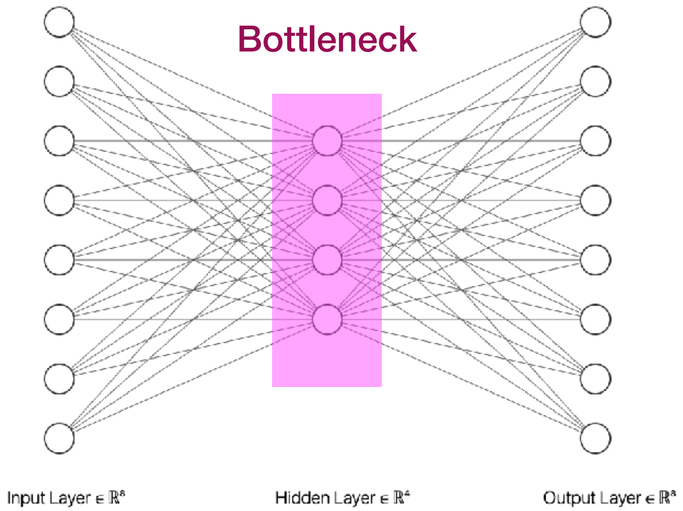
Autoencoders

- Autoencoders are neural networks that use a **bottleneck** architecture which forces a **compressed** knowledge representation of the input data.
- Autoencoders work very well with data that has **correlated** input features (i.e. not independent).



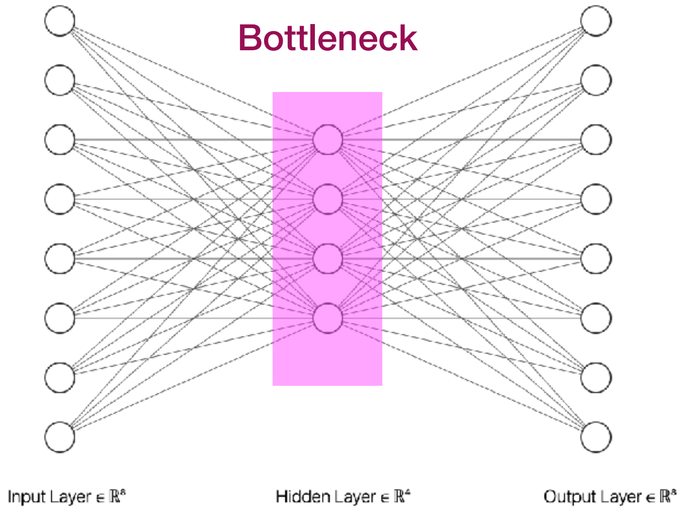
Bottleneck

- The bottleneck constrains the amount of information that is able to traverse the full network.
- This enables the hidden bottleneck layer(s) to learn a compressed representation of the input data



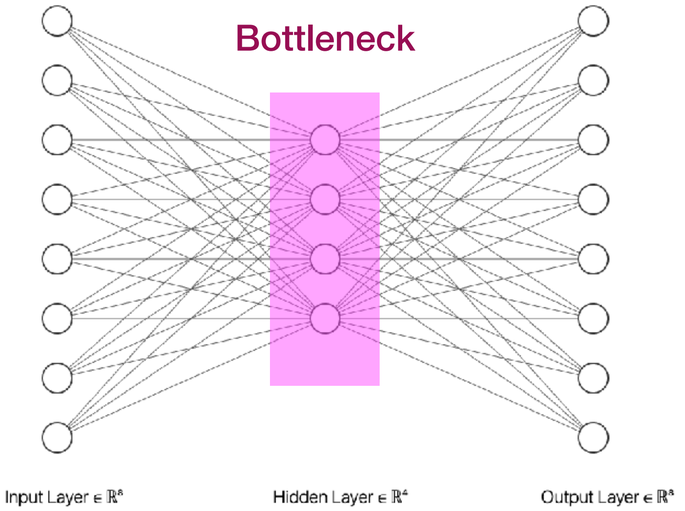
The Ideal Autoencoder

- The ideal Autoencoder is:
  - Sensitive enough to accurately reconstruct the image
  - Insensative enough to inputs so that the model doesn’t overfit the training data



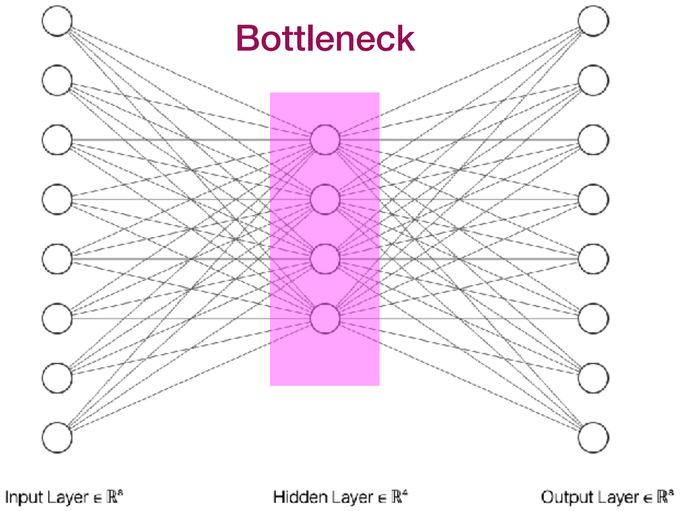
Autoencoder Architecture

- A simple Autoencoder architecture is one where it bottlenecks/constrains the the number of nodes in the hidden layer(s).
- Notice our input and output match in dimensions, that is because we're reconstructing the input
- Our loss function here penalises reconstruction erorr
- This allows the model to learn the most important features needed to reconstruct the data/image



CNN Autoencoder Architecture

- Given that our inputs in Computer Vision applications are images, using Convolution Neural Networks makes total sense
- Using Conv layers provides much better performance



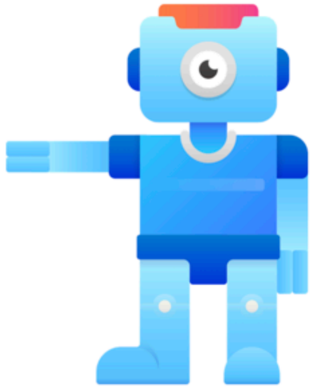
Training an Autoencoder

- The training process is simple, however there are few differences.
- The target data is the same as the training data
- Likewise for validation, as you're testing how well your encoder-decoder model works
- The loss function can be binary cross entropy or even MSE.

```
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

Autoencoder Limitations

- Autoencoders are lossy, meaning the decompressed outputs are degraded compared to the original
- Data-specific meaning that it learns the representation in a specific domain

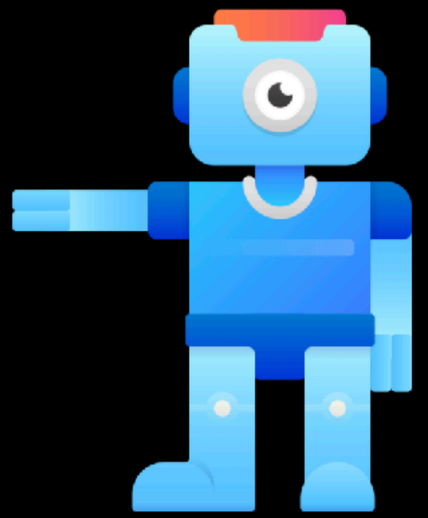


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Autoencoders in Keras and PyTorch

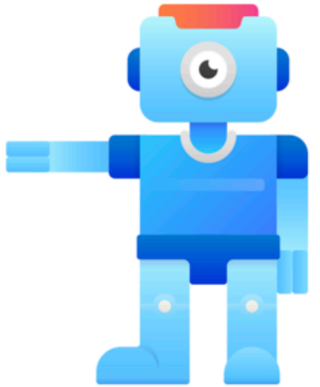


MODERN COMPUTER VISION

BY RAJEEV RATAN

Generative Adversarial Networks (GANs)

An Introduction to Generative Adversarial Networks (GANs)

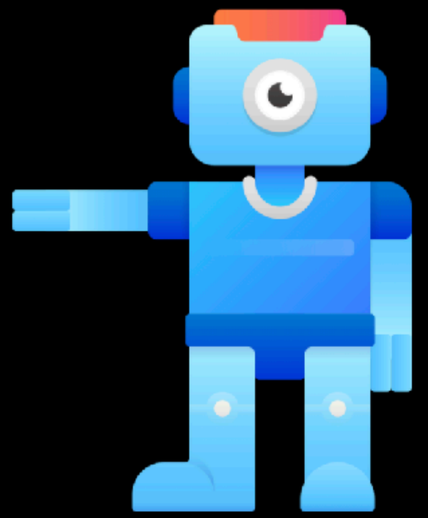


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

How Generative Adversarial Networks (GANs) Work

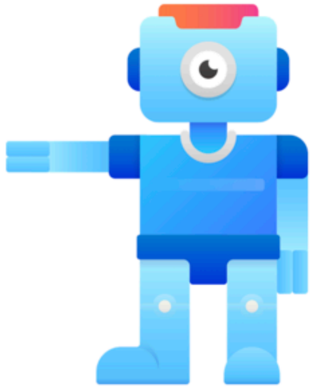


MODERN COMPUTER VISION

BY RAJEEV RATAN

How Do GANs Work?

A high-level overview of the basics of GANs

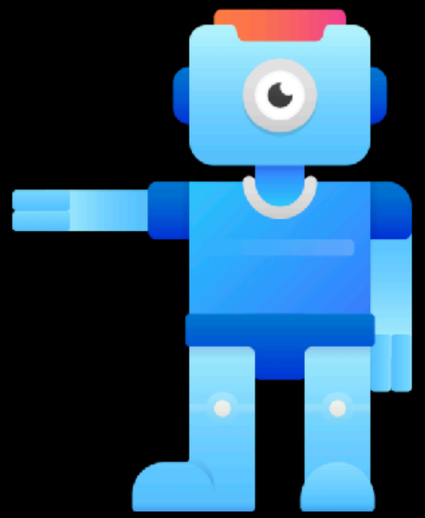


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

A high-level overview of the basics of GANs

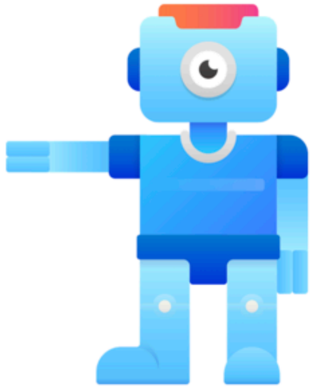


MODERN COMPUTER VISION

BY RAJEEV RATAN

Training Generative Adversarial Networks (GANs)

An Introduction to Generative Adversarial Networks (GANs)

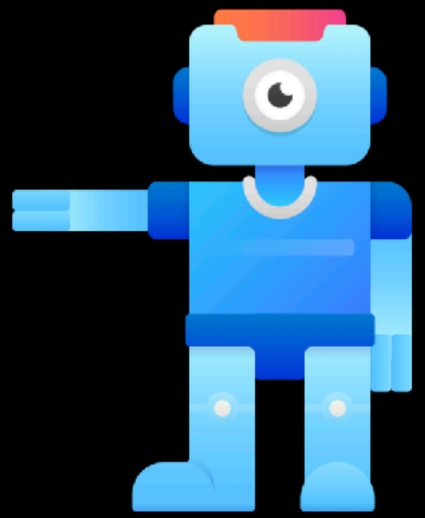


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Challenges in Training GANs

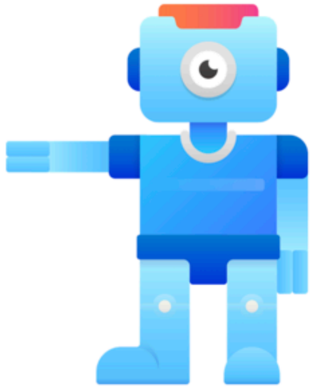


MODERN COMPUTER VISION

BY RAJEEV RATAN

Challenges in Training GANs

Things to watch out for when training GANs

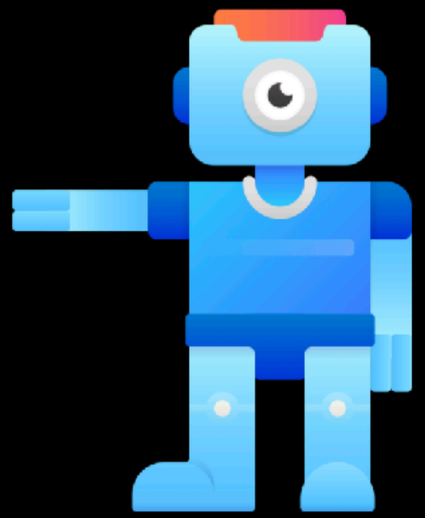


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

GANs in Industry

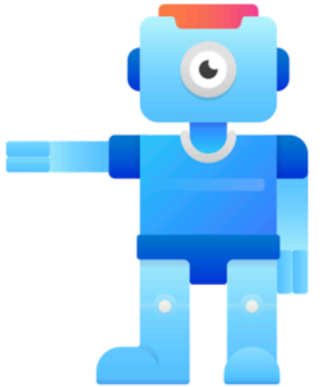


MODERN COMPUTER VISION

BY RAJEEV RATAN

GANs in Industry

Common use cases for GANs in the real world

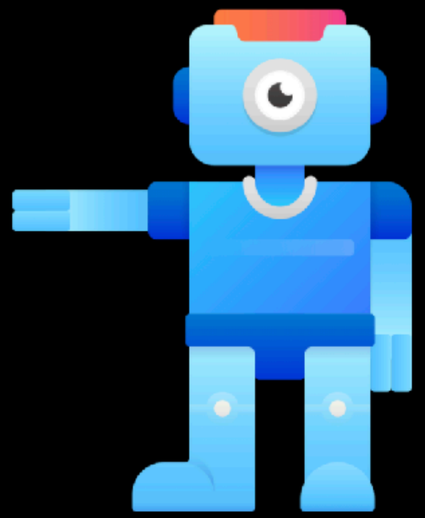


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Siamese Networks



MODERN COMPUTER VISION

BY RAJEEV RATAN

Siamese Networks

Siamese Networks for image similarity

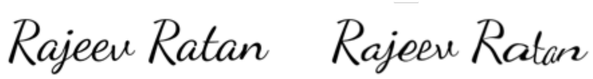
Siamese Networks

- Firstly, what is Siamese?
- The term comes from Siamese twins (conjoined twins) which are twins that unfortunately joined together at birth, sometimes sharing organs.
- Siamese Networks are similarly '**connected**'. They consist of:
  - Two or more identical (in architecture) subnetworks (Neural Networks)
  - The subnetworks share the same parameters & weights
  - Parameter updates are mirrored on both networks (i.e. when one updates, the other one updates as well)

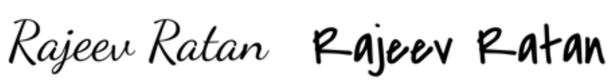
Image Source - https://commons.wikimedia.org/wiki/File:Chang_and_Eng_the_Siamese_twins,_in_a_games_room._Coloured_e_Wellcome_V0007366.jpg

What are Siamese Networks used for?

- Siamese Networks are very useful for **comparing images** or **image similarity tasks**
- Examples of these are Signature Verification, Face Recognition and Finger Print Matching.
- Siamese Networks give a **simple binary output**, Yes if the images match or No if they do not match.

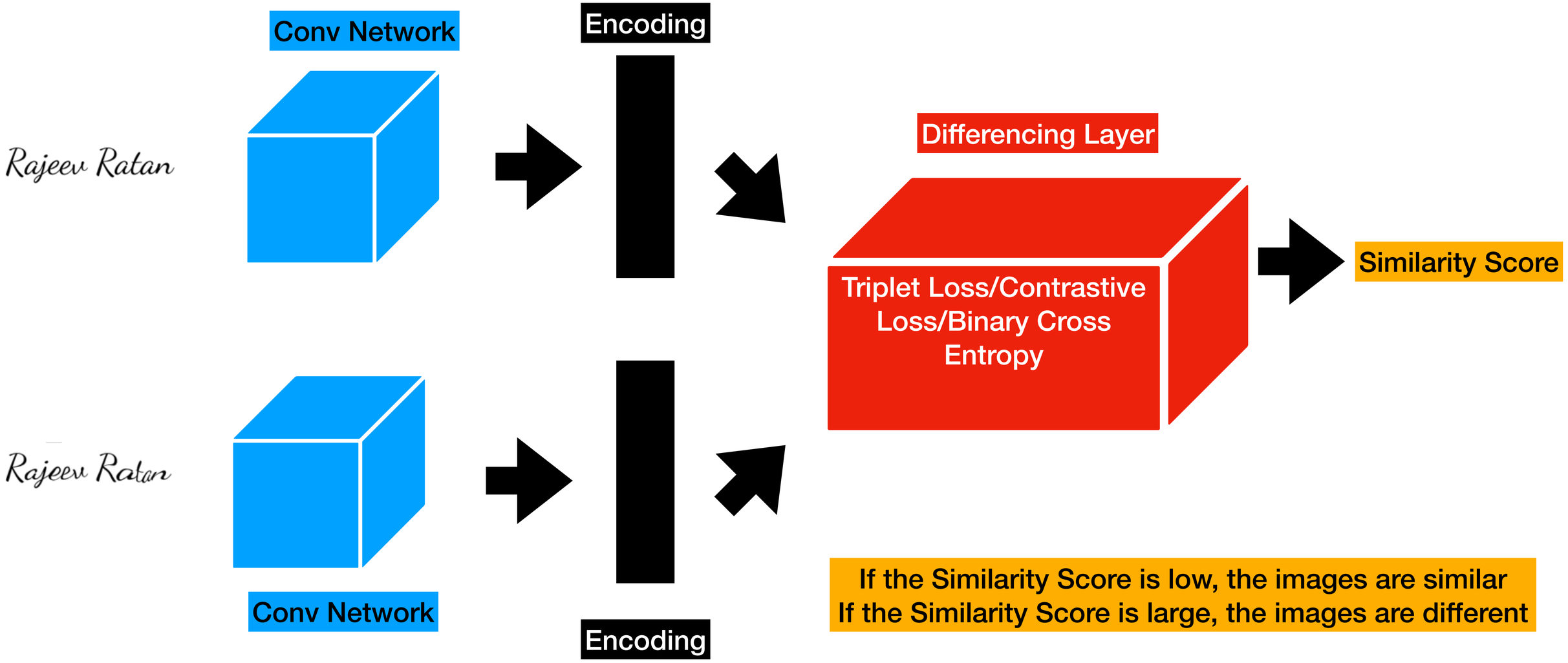


Yes

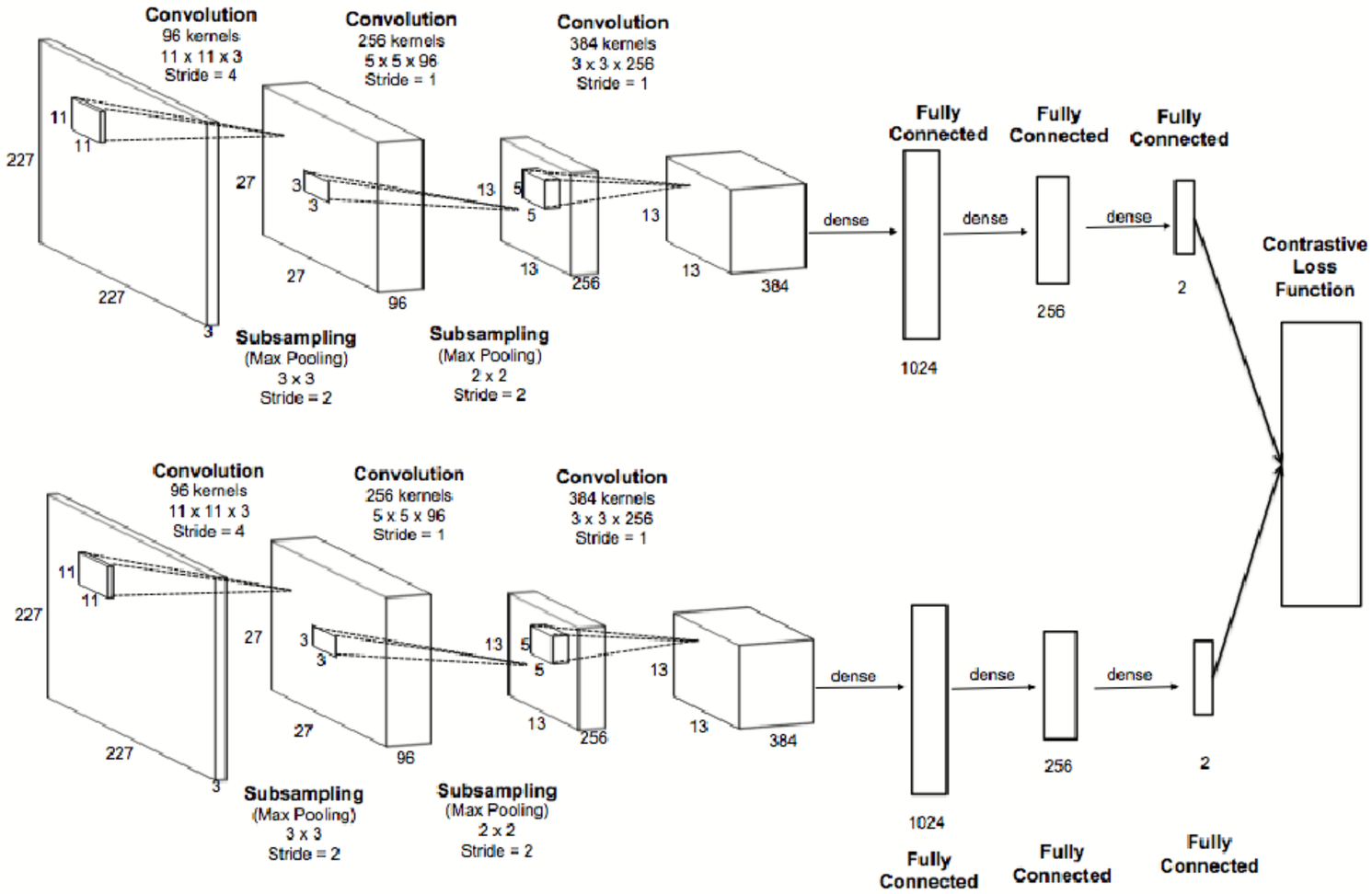


No

High Level Diagram of a Siamese Network



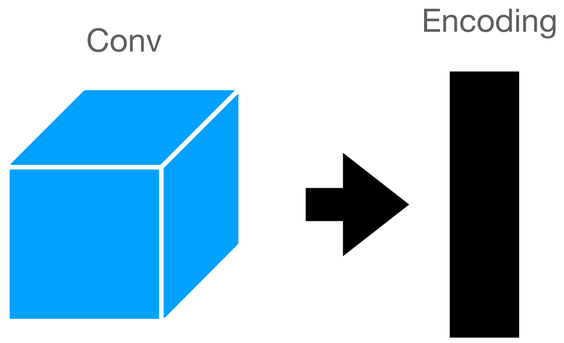
High Level Diagram of a Siamese Network



Example for a siamese network (source: Rao et al. - https://www.semanticscholar.org/paper/A-Deep-Siamese-Neural-Network-Learns-the-Similarity-Rao-Wang/3e16932979250e66cd2cb4d8c9a5411e195273be/figure/0)

Siamese Network Architecture

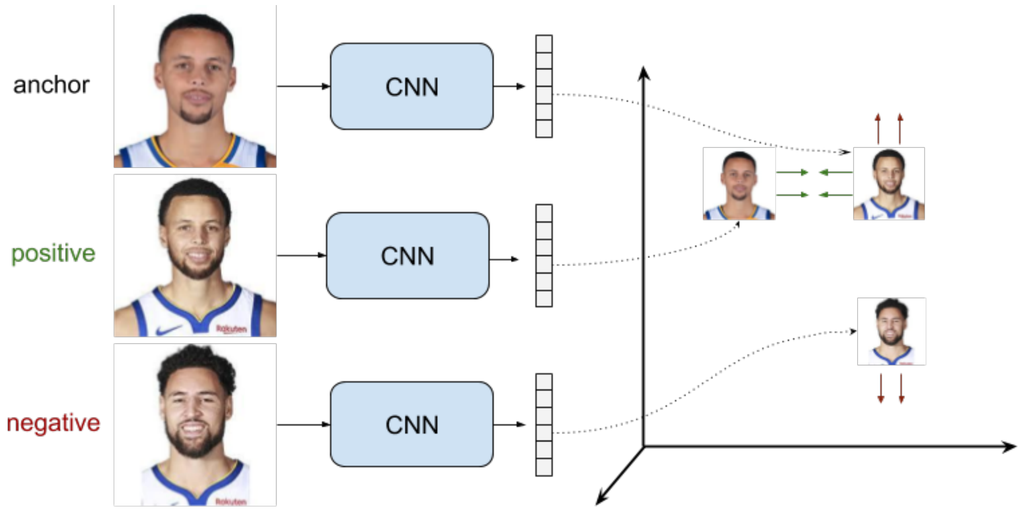
- Notice that the output of Conv Networks in a Siamese Networks are a **flat matrix.**
- This is the **embedding layer or encoding** which can be considered the features extracted from that image.
- The **Differencing Layer** is where we can find the difference using **Euclidian** or **Cosine Difference** between the matrices produced by each network.
- We can then use a **loss function** to assess whether the Siamese Networks made the right decision.



Siamese Networks Loss Functions

Popular Loss functions used when training Siamese Networks are:

- **Triplet Loss** - where a baseline input is compared to a positive input and a negative input. The distance from the baseline input to the positive input is minimized, and the distance from the baseline input to the negative input is maximized
- Contrastive Loss
- Binary Cross-Entropy

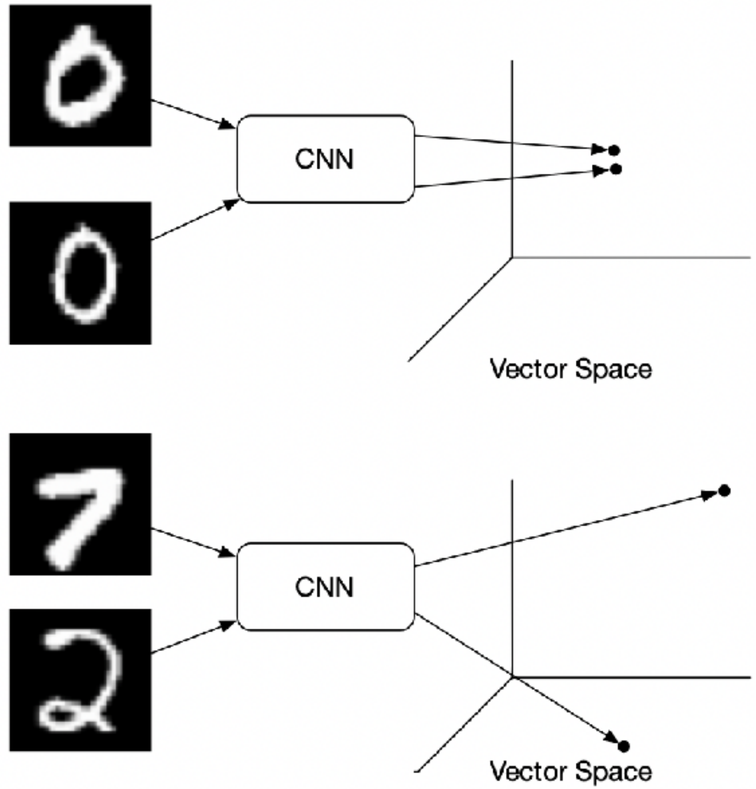


https://commons.wikimedia.org/wiki/File:Triplet_loss.png

Siamese Networks Loss Functions

Popular Loss functions used when training Siamese Networks are:

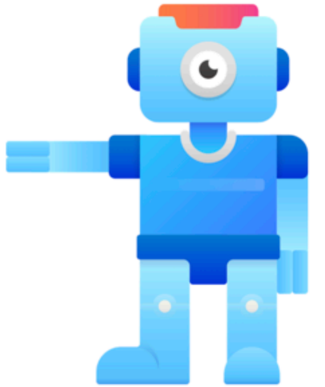
- Triplet Loss
- **Contrastive Loss -** The goal of a siamese networks is to differentiate between pairs of images. Contrastive refers to the fact that these losses are computed contrasting two or more data points representations.
- Binary Cross-Entropy



Siamese Networks Loss Functions

Popular Loss functions used when training Siamese Networks are:

- Triplet Loss
- Contrastive Loss
- **Binary Cross-Entropy** - given the output of our Siamese Model is binary, i.e. similar or dis-similar, using binary cross-entropy loss function is often the obvious default choice.

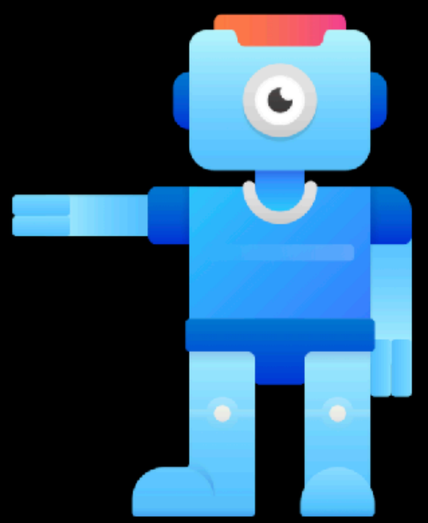


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Training Siamese Networks



MODERN COMPUTER VISION

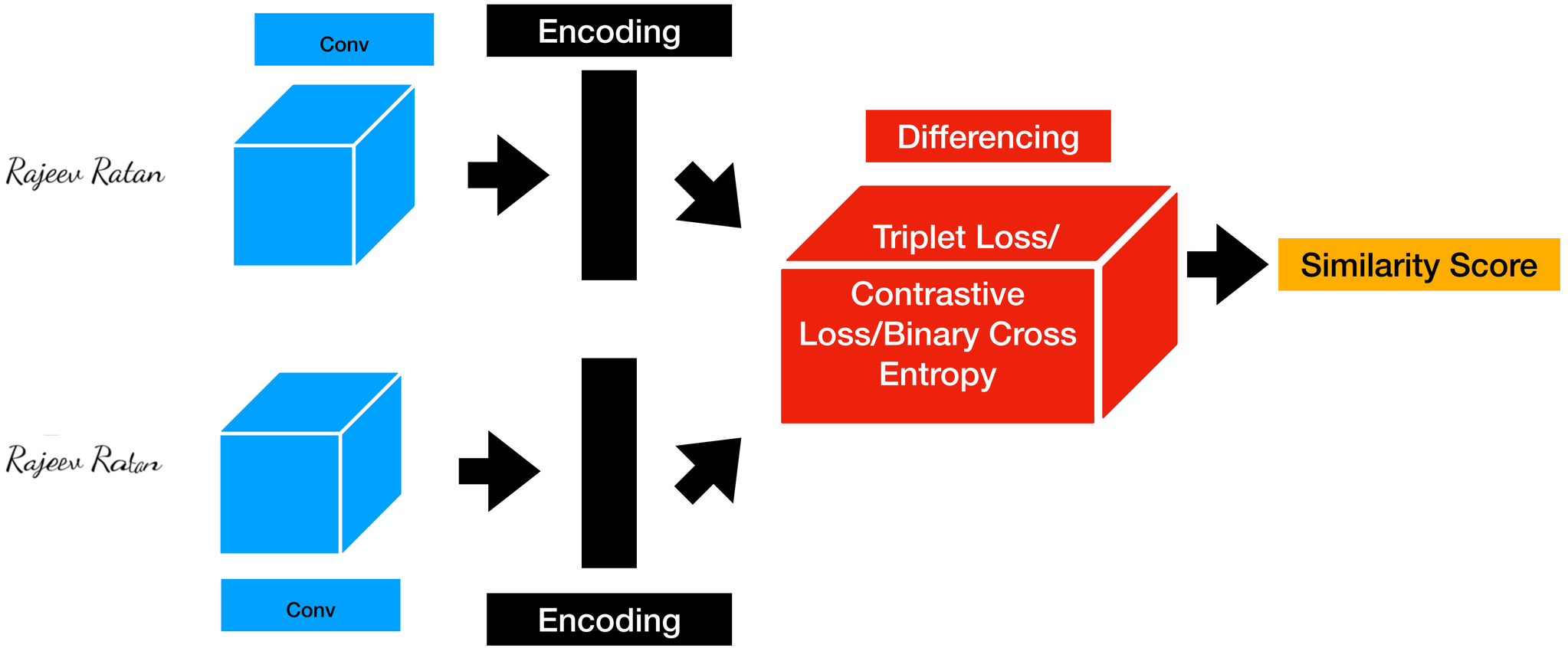
BY RAJEEV RATAN

Training Siamese Networks

How do we go about Training Siamese Networks

Siamese Networks

- Remember the goal of a Siamese Network is to classify if the two inputs are the same or different using the Similarity Score.



Dataset - Image Pairs

- We start the training process by firstly preparing our data by creating **Image Pairs.**
- We create two types of pairs:
  - **Positive** data pair is when both the inputs are the same class
  - **Negative** pair is when the two inputs are difference classes

Positive Pair

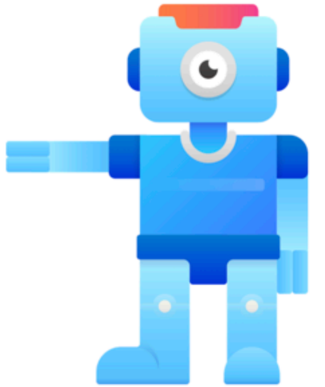
Negative Pair

Building the Network

- We build a CNN that outputs the **feature encoding or embedding** using a fully connected layer.
- **We build the sister** CNN's will have the same architecture, hyperparameters, and weights.
- We then build the **differencing layer to calculate the Euclidian distance** between the output of the two CNN subnetworks encoding.
- The final layer is a **fully-connected layer with a single node** using the sigmoid activation function to output the **Similarity score**.
- Compile the model using one of the loss functions (Contrastive or Triplet Loss work well).

Ready to Train

- Once we have our image pairs dataset and our models built we can begin training.
- Typically we can use RMSprop or any common Gradient Descent Algorithm.
- CNNs are also typically simple as we only need to create relatively simple embeddings.

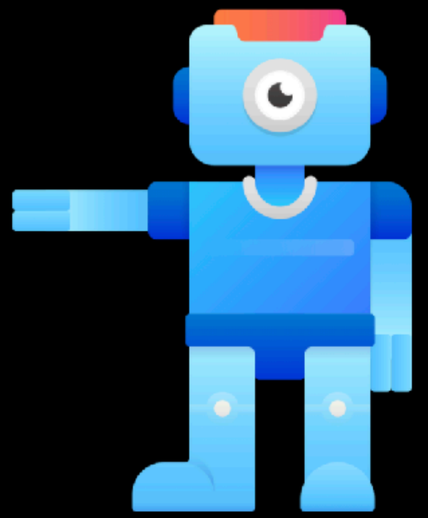


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Siamese Networks in Keras



MODERN COMPUTER VISION

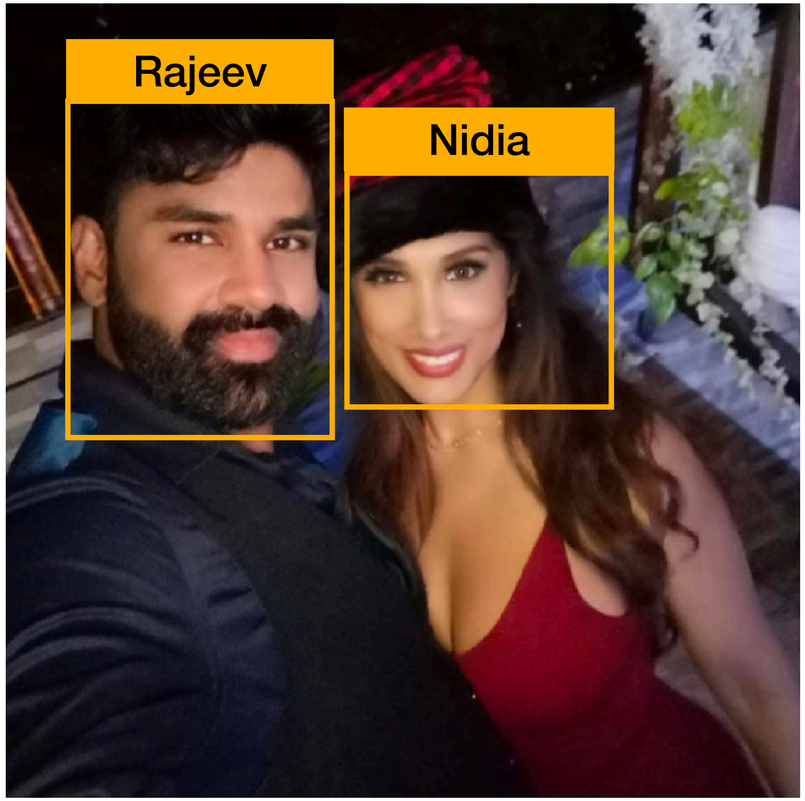
BY RAJEEV RATAN

Facial Recognition

Overview and an intro to VGGFace and FaceNet

What is Facial Recognition?

- The ability to automatically attach an individual's identity to a face
- It's a task that human abilities are absolutely brilliant, and even some animals, dogs, crows, sheep can do it!
- Can machines do it?



Facial Recognition the Early Days

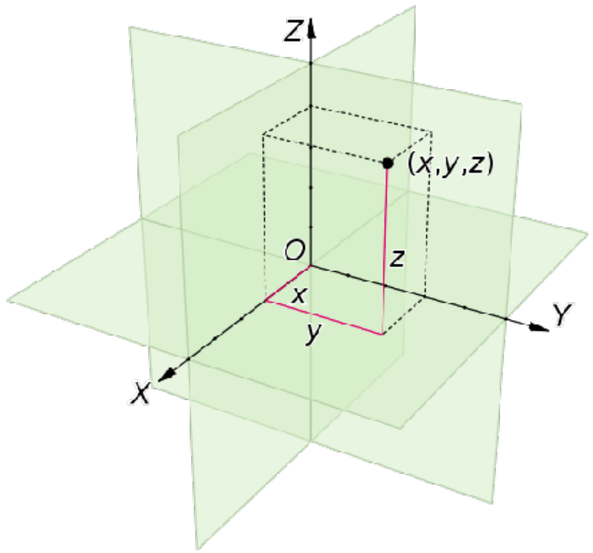
- **OpenCV** has 3 facial recognition libraries, all of which operate similarly.
- They take a dataset of labelled faces, and compute features to represent the images. Their classifiers then utilise these features to classify.
- **Eigenfaces (1987)** - createEigenFaceRecognizer()
- **Fisherfaces (1997)** - createFisherFaceRecognizer()
- **Local Binary Patterns Histograms (1996)**- createLBPHFaceRecognizer()

Facial Recognition using Deep Learning

- Siamese Networks can be used for Facial Recognition.
- Let’s take a look at two popular Deep Learning Facial Recognition Networks:
  - VGGFace
  - FaceNet

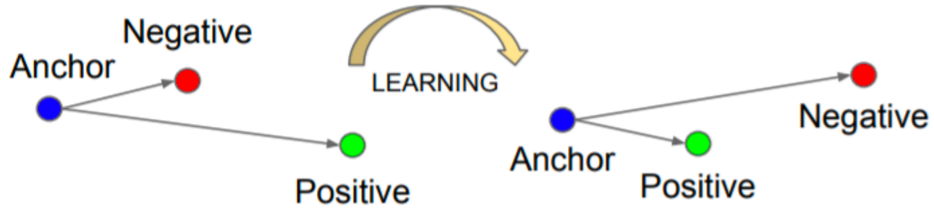
A look at FaceNet

- FaceNet was first introduced by Google in 2015
- It transforms a face into a 128 dimension Euclidian space embedding
- Uses the triplet loss function



3 Dim in euclidian space

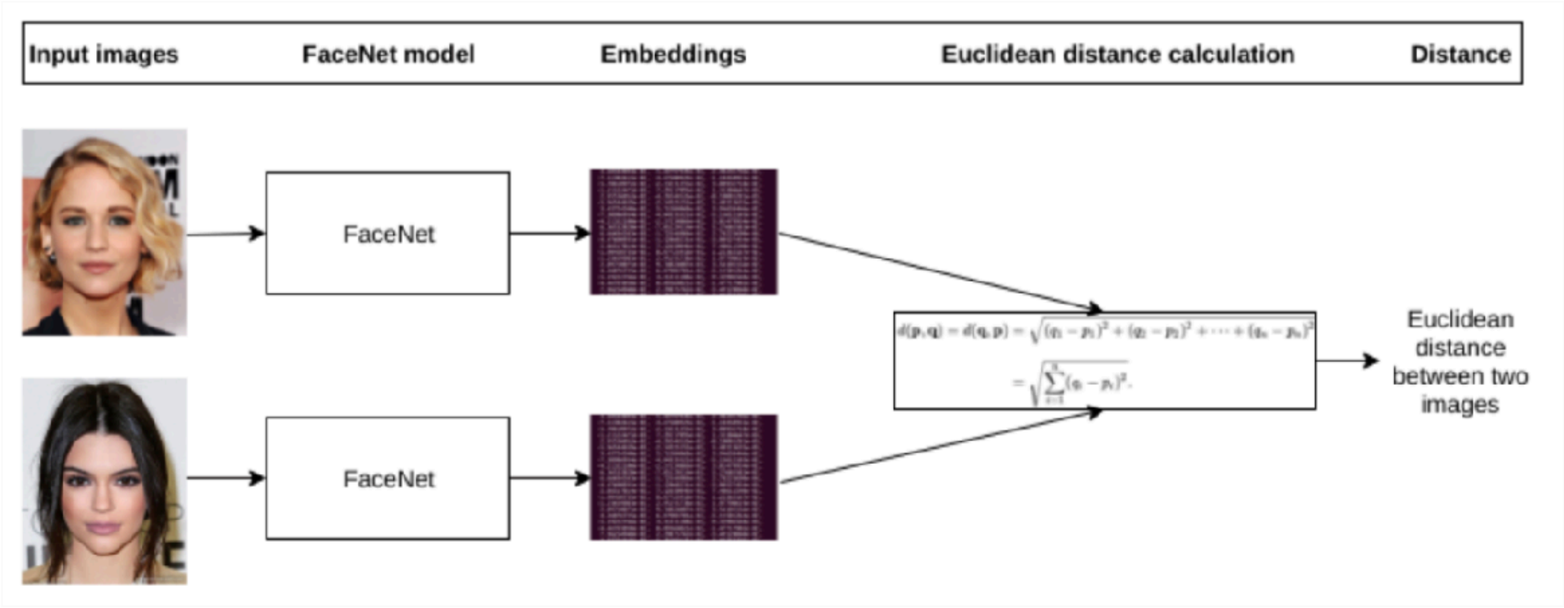
FaceNet Basic Architecture



Triplet Loss Training

https://arxiv.org/pdf/1503.03832.pdf

A look at FaceNet



One shot learning using FaceNet

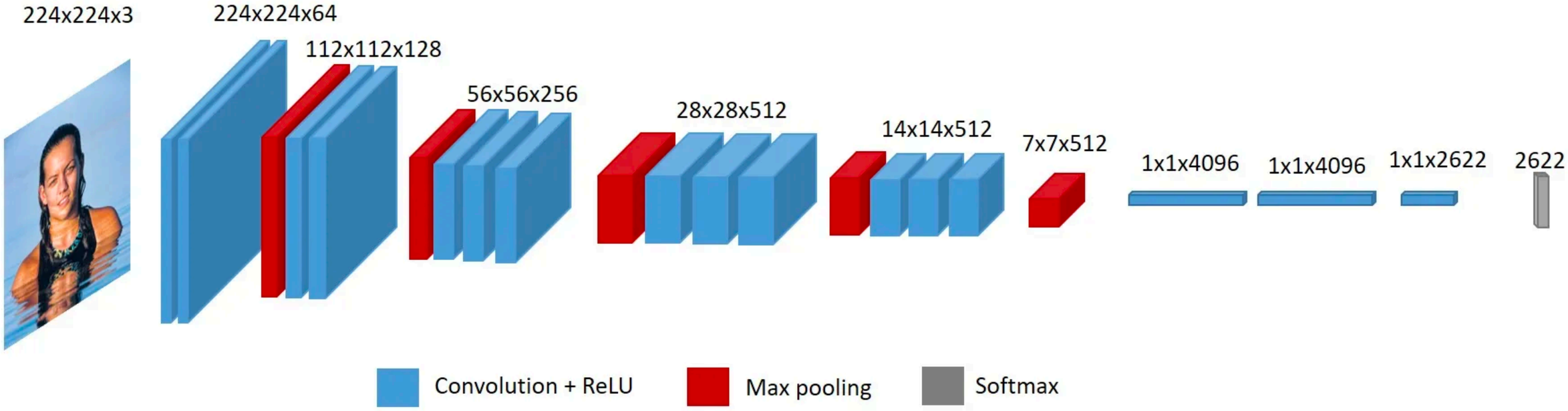
A look at VGGFace

- VGGFace was introduced by Oxford University Researchers Omkar M. Parkhi, Andrea Vedaldi and Andrew Zisserman in their paper titled ‘Deep Face Recognition) in 2015
- They used Triplet Loss and an embedding vector of 2,622 or 1,024 (depending on the configuration) with input image size being 224 x 224

Figure 1: Example images from our dataset for six identities.

https://www.robots.ox.ac.uk/~vgg/publications/2015/Parkhi15/parkhi15.pdf

A look at VGGFace



Dataset	Identities	Images
LFW	5,749	13,233
WDRef [4]	2,995	99,773
CelebFaces [25]	10,177	202,599

Dataset	Identities	Images
Ours	2,622	2.6M
FaceBook [29]	4,030	4.4M
Google [17]	8M	200M

Table 1: **Dataset comparisons:** Our dataset has the largest collection of face images outside industrial datasets by Goole, Facebook, or Baidu, which are not publicly available.

VGGFace Performance

No.	Method	Images	Networks	Acc.
1	Fisher Vector Faces [21]	-	-	93.10
2	DeepFace [29]	4M	3	97.35
3	Fusion [30]	500M	5	98.37
4	DeepID-2,3		200	99.47
5	FaceNet [17]	200M	1	98.87
6	FaceNet [17] + Alignment	200M	1	99.63
7	Ours	2.6M	1	98.95

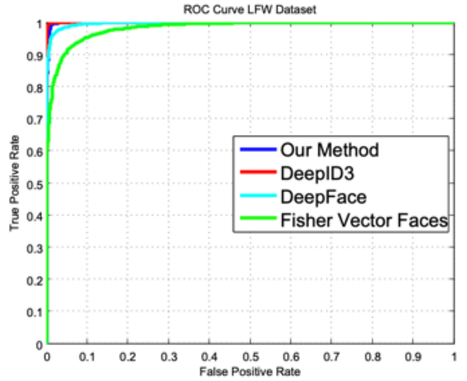
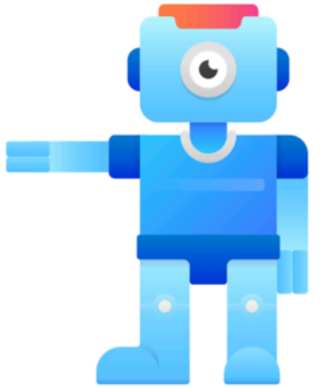


Table 5: **LFW unrestricted setting.** Left: we achieve comparable results to the state of the art whilst requiring less data (than DeepFace and FaceNet) and using a simpler network architecture (than DeepID-2,3). Note, DeepID3 results are for the test set with label errors corrected – which has not been done by any other method. Right: ROC curves.

LFW - Labelled Faces in the Wild

No.	Method	Images	Networks	100%- EER	Acc.
1	Video Fisher Vector Faces [15]	-	-	87.7	83.8
2	DeepFace [29]	4M	1	91.4	91.4
3	DeepID-2,2+,3		200	-	93.2
4	FaceNet [17] + Alignment	200M	1	-	95.1
5	Ours ($K = 100$)	2.6M	1	92.8	91.6
6	Ours ($K = 100$) + Embedding learning	2.6M	1	97.4	97.3

Table 6: **Results on the Youtube Faces Dataset, unrestricted setting**. The value of K indicates the number of faces used to represent each video.



MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Face Similarity, Recognition, VGG Face and FaceNet