How to get a neural network to do what you want?

Loss function engineering

Petar Veličković

Artificial Intelligence Group
Computer Laboratory, University of Cambridge, UK

Introduction

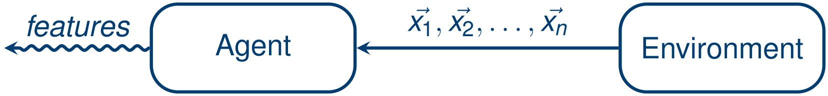
- In this talk, I will explore interesting applications of using neural networks for *what they weren't originally designed to do*.
- This will cover the essentials of how neural networks normally work, with a journey through representation learning…
- …followed by an overview of several interesting applications: adversarial examples, neural style transfer and DeepDream…
- …concluding with an example of how I applied similar techniques within my research (*time permitting*).
- Let's get started!

The three “flavours” of machine learning

- Unsupervised learning
- Supervised learning
  (the only kind of learning neural networks can directly do!)
- Reinforcement learning

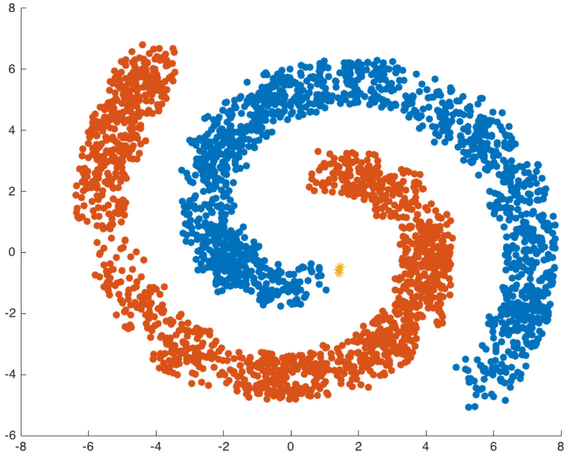
Unsupervised learning

- The environment gives you *unlabelled data*—and asks you to assign useful features/structure to it.



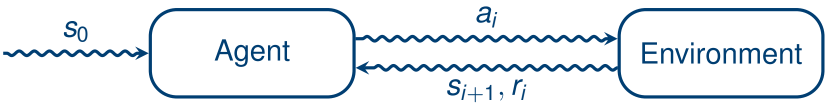
- Example: study data from patients suffering from a disease, in order to discover different (previously unknown) types of it.

Example: Clustering



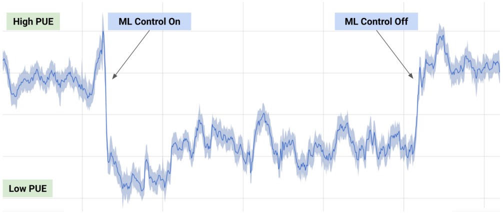
Reinforcement learning

- You are allowed to perform *actions* within the environment, triggering a state change and a reward signal—your objective is to maximise future rewards.



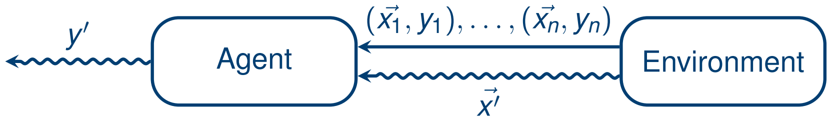
- Example: playing a video game—*states* correspond to RAM/framebuffer contents, *actions* are available key presses (including NOP), *rewards* are changes in score.

Example: Atari



Supervised learning

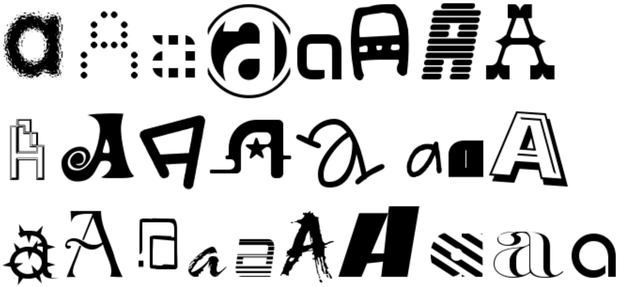
- The environment gives you *labelled data* ($\sim$ known *input/output pairs*)—and asks you to learn the underlying function.



- Example: determining whether a person will be likely to return their loan, given their credit history (and a set of previous data on issued loans to other customers).

Motivation: notMNIST

- Which characters do you see? (*How* did you conclude this?)



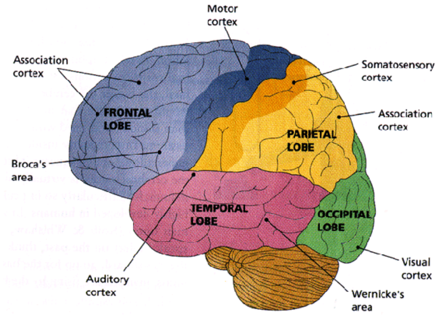
- Imagine someone asked you to write a program that recognises characters from arbitrary glyphs…

Intelligent systems

- Although the previous task was likely *simple* to you, you (probably) couldn't turn your thought process into a concise sequence of instructions for a program!
- Unlike a *"dumb"* program (that just blindly executes preprogrammed instructions), you've been exposed to a lot of `A` characters during your lifetimes, and eventually *"learnt"* the complex features making something an `A`!
- Desire to design such systems (capable of *generalising* from past experiences) is the essence of *machine learning*!
  - How many such systems do we know from nature?

Specialisation in the brain

- We know that *different parts* of the brain perform *different tasks*:

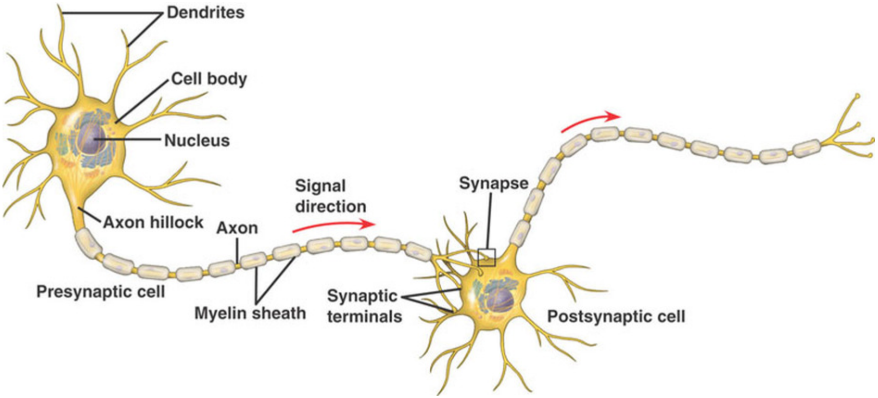


- There is increasing evidence that the brain:
  - Learns from *exposure to data*;
  - Is *not* preprogrammed!

Brain & data

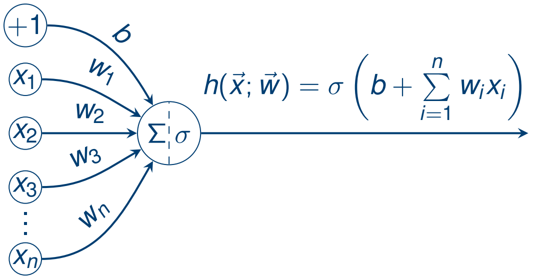
- The majority of what we know about the brain comes from studying *brain damage*:
  - Rerouting visual inputs into the auditory region of baby ferrets makes this region capable of dealing with visual input!
  - As far as we know (for now), the modified region works equally good as the visual cortex of healthy ferrets!
- If there are no major biological differences in learning to process different kinds of input…
- $\implies$ the brain likely uses a *general learning algorithm*, capable of adapting to a wide spectrum of inputs.
- We'd very much like to capture this algorithm!

A *real* neuron!



An *artificial* neuron!

Within this context sometimes also called a *perceptron* (...)

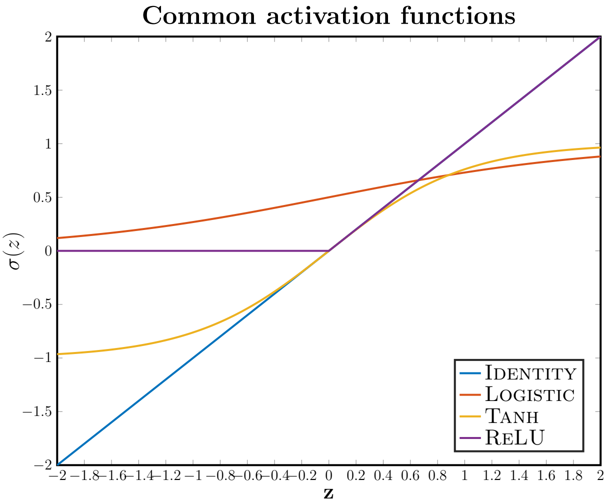


$$h(\vec{x}; \vec{w}) = \sigma\left(b + \sum_{i=1}^{n} w_i x_i\right)$$

Popular choices for the activation function σ:

- *Identity*: $\sigma(x) = x$;
- *Rectified linear unit (ReLU)*: $\sigma(x) = \max(0, x)$;
- *Sigmoid functions*: $\sigma(x) = \frac{1}{1+\exp(-x)}$ (*logistic*); $\sigma(x) = \tanh x$.

Activation functions

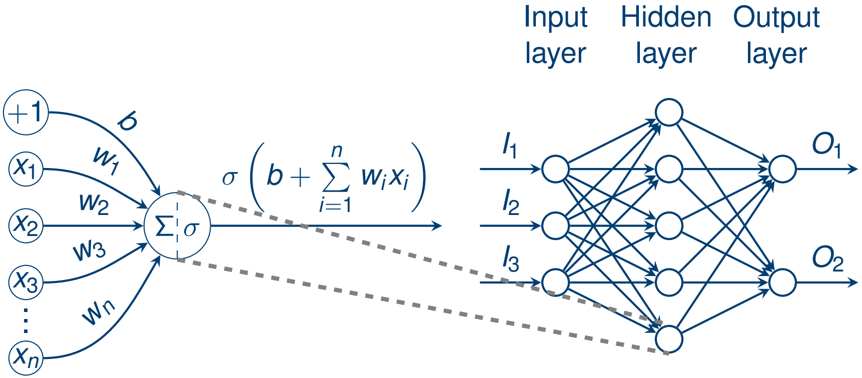


Neural networks and deep learning

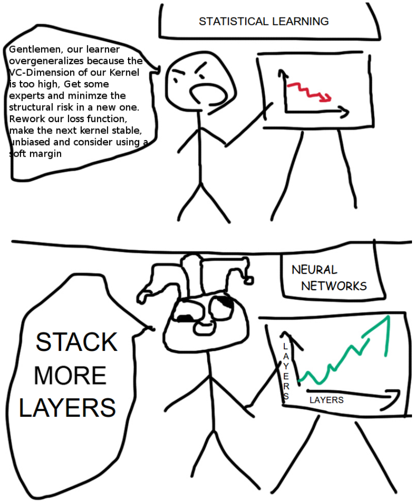
- It is easy to extend a single neuron to a *neural network*—simply connect outputs of neurons to inputs of other neurons.
- We may do this in two ways:
  - **Feedforward**: the computation graph does not have cycles;
  - **Recurrent**: the computation graph has cycles.
- Typically we organise neural networks in a sequence of *layers*, such that a single layer only processes output from the previous layer. Everything with >1 hidden layer is *"deep"*!

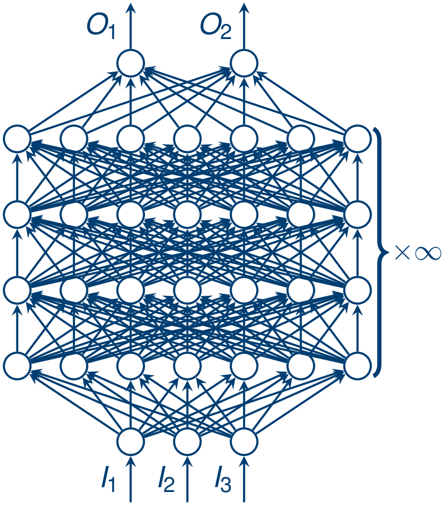
Multilayer perceptrons

The most potent feedforward architecture allows for full connectivity between layers—sometimes also called a *multilayer perceptron*.



Deep neural networks


STATISTICAL LEARNING
Gentlemen, our learner overgeneralizes because the VC-Dimension of our Kernel is too high, Get some experts and minimze the structural risk in a new one. Rework our loss function, make the next kernel stable, unbiased and consider using a soft margin
NEURAL NETWORKS
STACK MORE LAYERS
LAYERS
LAYERS



Quiz: What do we have here?

DeepBlue vs. AlphaGo

- Main idea (roughly) the same: *assume that a grandmaster is only capable of thinking k steps ahead*—then generate a (near-)optimal move when considering $k' > k$ steps ahead.
  - DeepBlue does this exhaustively, AlphaGo sparsely (discarding many "highly unlikely" moves).
- One of the key issues: when *stopping exploration*, how do we determine the *advantage* that player 1 has?

DeepBlue: Gather a team of *chess experts*, and define a function $f : Board \to \mathbb{R}$, to define this advantage.

AlphaGo: Feed the raw state of the board to a deep neural network, and have it *learn* the advantage function *by itself*.

- This highlights an important *paradigm shift* brought about by deep learning...

Feature engineering

- Historically, machine learning problems were tackled by defining a set of *features* to be manually extracted from raw data, and given as inputs for *"shallow"* models.
  - Many scientists built *entire PhDs* focusing on features of interest for just one such problem!
  - Generalisability: very small (often zero)!
- With deep learning, the network *learns the best features by itself*, directly from *raw data*!
  - For the first time connected researchers from fully distinct areas, e.g. *natural language processing* and *computer vision*.
  - $\Longrightarrow$ a person capable of working with deep neural networks may readily apply their knowledge to create state-of-the-art models in virtually any domain (assuming a large dataset)!

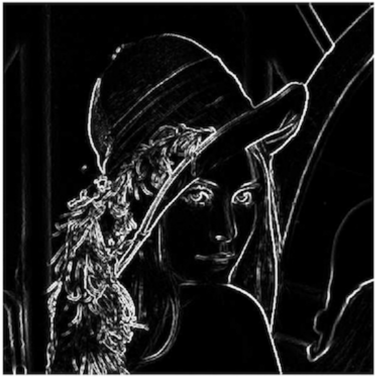
Representation learning

- As inputs propagate through the layers, the network captures more complex *representations* of them.
- It will be extremely valuable for us to be able to reason about these representations!
- Typically, models that deal with *images* will tend to have the best visualisations (and will be the key topic of this talk).
- Therefore, I will provide a brief introduction to these models (convolutional neural networks). Then we can look into the kinds of representations they capture...

Working with images

- Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - Treating each pixel as an independent input…
  - …results in $h \times w \times d$ new parameters per neuron in the first hidden layer…
  - …quickly deteriorating as images become larger—requiring exponentially more data to properly fit those parameters!
- **Key idea:** downsample the image until it is small enough to be tackled by such a network!
  - Would ideally want to extract some useful features first…
- $\Longrightarrow$ exploit spatial structure!

The *convolution* operator



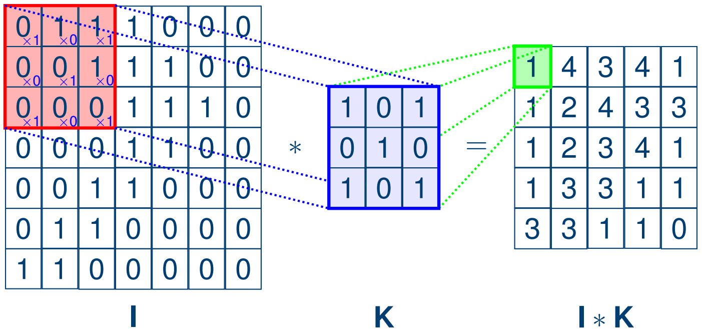
Enter the *convolution* operator

- Define a small (e.g. 3×3) matrix (the kernel, $\mathbf{K}$).
- Overlay it in all possible ways over the input image, $\mathbf{I}$.
- Record sums of elementwise products in a new image.

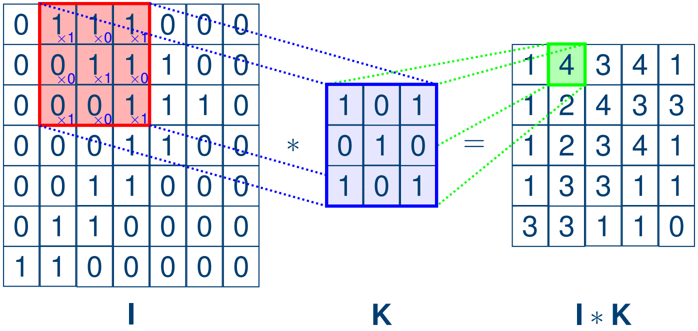
$$(\mathbf{I} * \mathbf{K})_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} \mathbf{K}_{ij} \cdot \mathbf{I}_{x+i-1,y+j-1}$$

- This operator exploits structure—neighbouring pixels influence one another stronger than ones on opposite corners!
- Start with *random kernels*—and let the network find the optimal ones on its own!

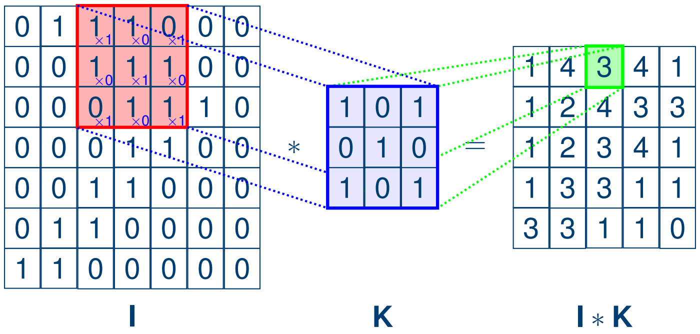
Convolution example



Convolution example



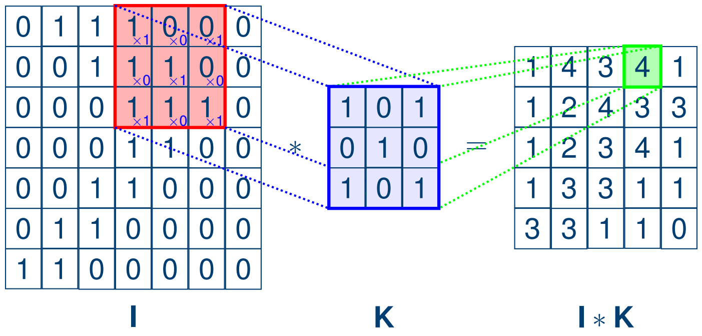
Convolution example



$$\begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 1 & 1_{\times 1} & 1_{\times 0} & 0_{\times 1} & 0 & 0 \\ \hline 0 & 0 & 1_{\times 0} & 1_{\times 1} & 1_{\times 0} & 0 & 0 \\ \hline 0 & 0 & 0_{\times 1} & 1_{\times 0} & 1_{\times 1} & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline 1 & 0 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|}\hline 1 & 4 & \boxed{3} & 4 & 1 \\ \hline 1 & 2 & 4 & 3 & 3 \\ \hline 1 & 2 & 3 & 4 & 1 \\ \hline 1 & 3 & 3 & 1 & 1 \\ \hline 3 & 3 & 1 & 1 & 0 \\ \hline \end{array}$$

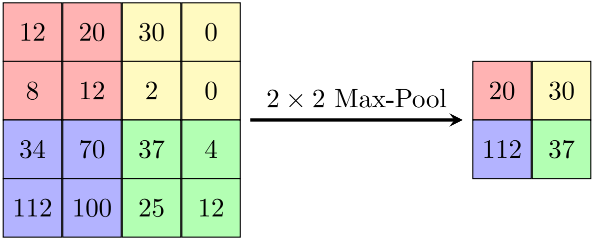
$$\mathbf{I} \qquad \mathbf{K} \qquad \mathbf{I} * \mathbf{K}$$

Convolution example



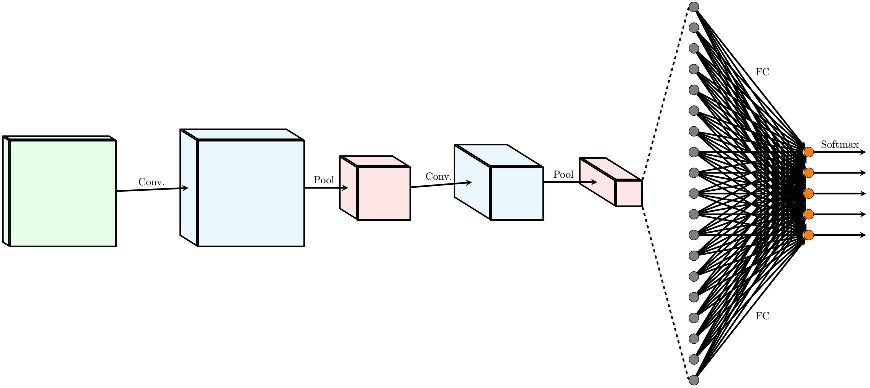
Downsampling ($\sim$ *max-pooling*)

Convolutions *light up* when they detect a particular feature in a region of the image. Therefore, when downsampling, it is a good idea to preserve maximally activated parts. This is the inspiration behind the *max-pooling* operation.



Stacking convolutions and poolings

Rough rule of thumb: increase the *depth* (number of convolutions) as the *height* and *width* decrease.



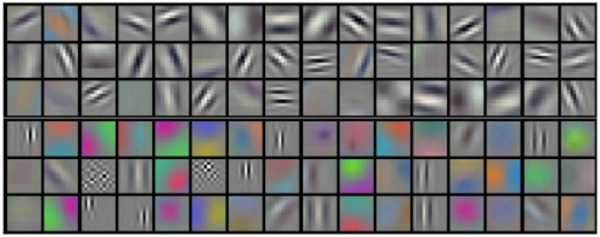
CNN representations

Three ways to examine the CNN's internal representations:

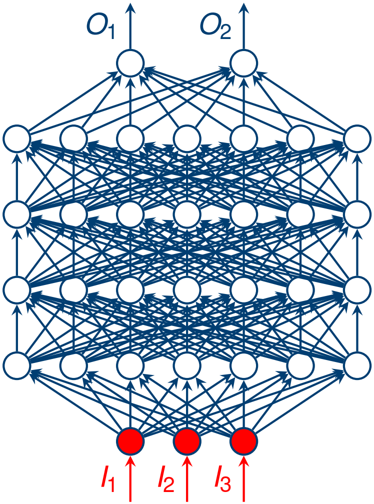
1. Observe the learnt *kernels*;
2. Pass an input through the network, observe the *activations*;
3. Coming later in this talk…

Observing kernels

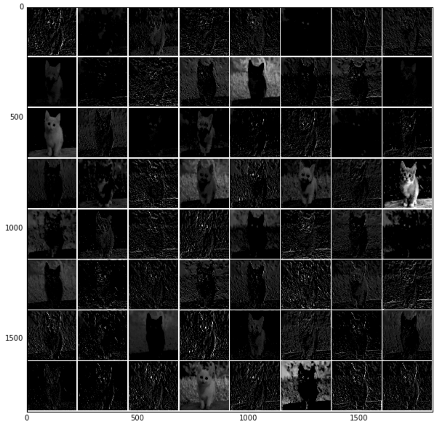
- Typically, as the kernels are small, gaining useful information from them becomes difficult already *past the first layer*.
- However, the first layer of kernels reveals something *magical*... In almost all cases, these kernels will learn to become *edge detectors*!

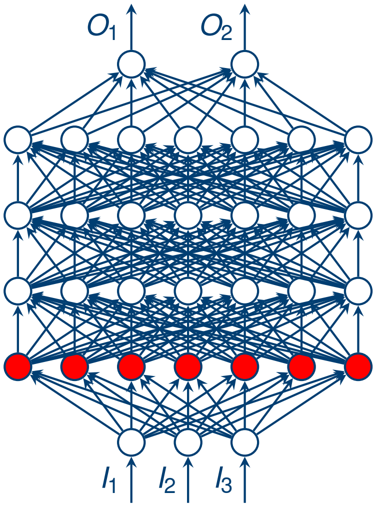


Passing data through the network: *Input*

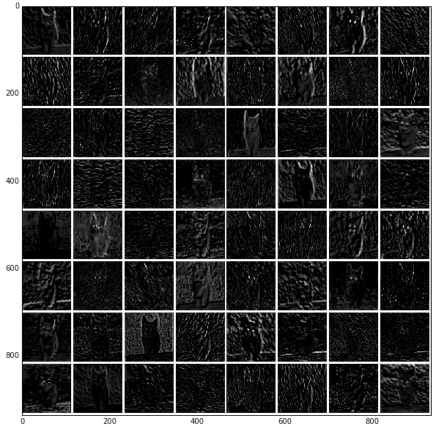


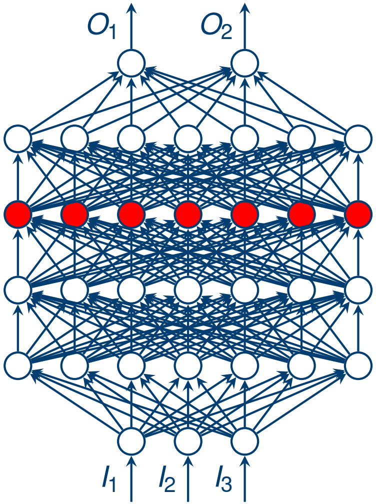
Passing data through the network: *Shallow layer*



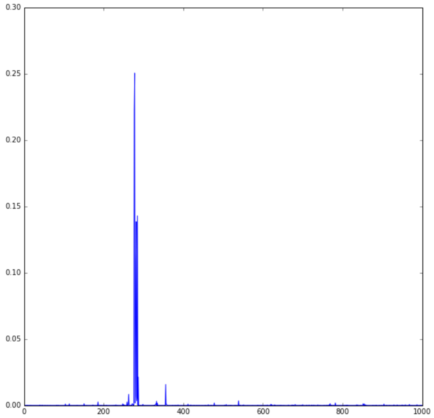


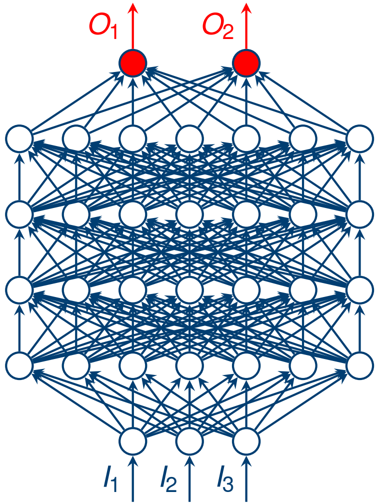
Passing data through the network: *Deep layer*





Passing data through the network: *Output*

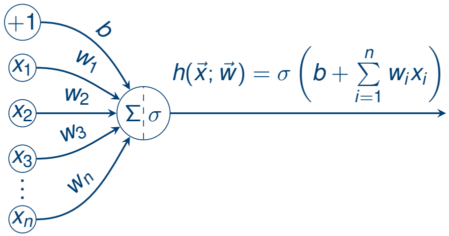




Towards a learning algorithm

- I will now prove an important property of the kinds of neural networks we built so far: **differentiability**.
- This will lead us to the primary *algorithm* which will be used for all the subsequent applications.
- We'll argue by reviewing the basic operations the network is doing, and then making an argument by induction (as every network can be constructed in finitely many such steps).

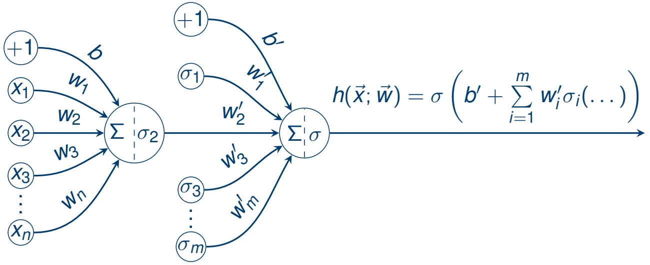
A single neuron is differentiable



$+1$
x_1
x_2
x_3
x_n
b
w_1
w_2
w_3
w_n
Σ
σ
$h(\vec{x}; \vec{w}) = \sigma\left(b + \sum_{i=1}^{n} w_i x_i\right)$

- The input fed to σ is a linear combination, and therefore trivially differentiable in $\vec{x}$, $\vec{w}$ and b.
- σ itself is differentiable! (Except for ReLU at zero, but it's OK)
- By composition, we may differentiate the neuron's function with respect to any parameter.

Connecting neurons is differentiable



- We compute a linear combination of differentiable functions, then feed it into a differentiable function—by composition, the full output is still differentiable with respect to any parameter.
- This works *regardless of how we connect* (e.g. fully connected, convolutional, . . .)

Max-pooling is differentiable... with a twist

- We can think of max-pooling as a piecewise step function:

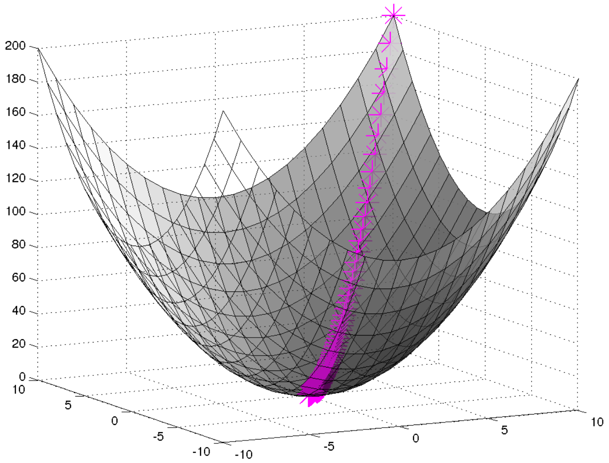
$$pool(\vec{x}) = \begin{cases} x_1 & \forall i \neq 1 . x_1 > x_i \\ x_2 & \forall i \neq 2 . x_2 > x_i \\ \dots & \\ x_n & \forall i \neq n . x_n > x_i \end{cases}$$

- Therefore, for any input $\vec{x}$ we feed, we can differentiate the pooling operation as well (except perhaps when there is no unique maximum, but we can introduce tiebreaking...).

Gradient descent

- We have successfully shown that the neural network's function $h(\vec{x}; \vec{w})$ is differentiable both with respect to its inputs $\vec{x}$ and weights/biases $\vec{w}$.
- If we now define a differentiable **loss function** $\mathcal{L}(\vec{x}; \vec{w})$, we can then train the network to minimise it, using gradient descent.
- At each step, follow the direction of the negative gradient of the loss function.
- Once we have decided on a neural network architecture, the loss function is what determines what the network is going to learn in training!

Gradient descent



(Simple) loss function engineering

- A very simple loss is the *squared error*: for a given training example $(\vec{x}_i, y_i)$, minimise the squared error of the network's prediction with respect to y_i:

$$\mathcal{L}(\vec{w}) = (y_i - h(\vec{x}_i; \vec{w}))^2$$

- A simple example of engineering a more targeted loss function from there is adding *weight decay*. Often, large weights are undesirable (numerically), so penalise them:

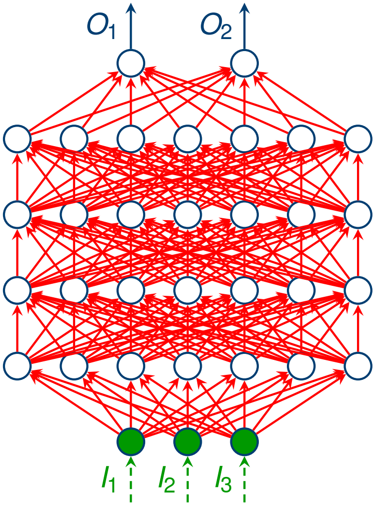
$$\mathcal{L}(\vec{w}) = (y_i - h(\vec{x}_i; \vec{w}))^2 + \lambda||\vec{w}||^2$$

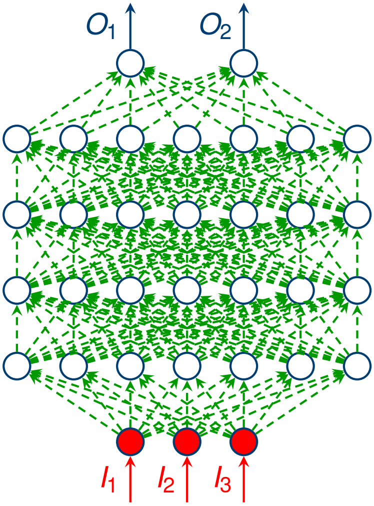
- Here, λ represents the tradeoff between fitting the training data and minimising the weights (*careful!*).

Change the game

- These losses are only parametrised in the weights (inputs are *fixed* to whatever's in the training set), and therefore are useful for learning better weights.
- For the purposes of our applications, we are going to *change the game*: assume the weights are fixed, and then define loss functions on the inputs!
- To be able to get nice results out of this, we need nice *pre-trained models*...

Change the game

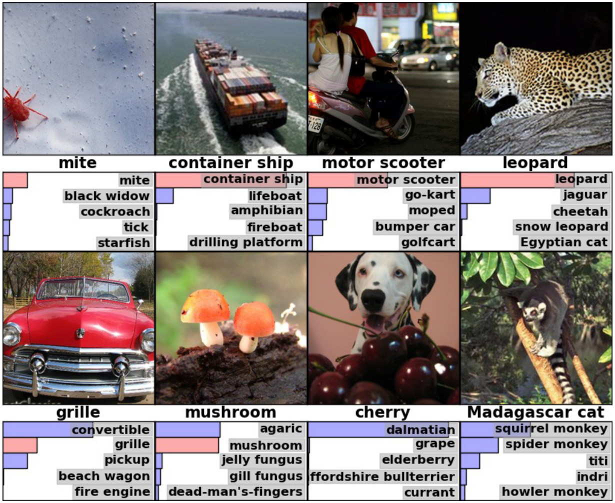




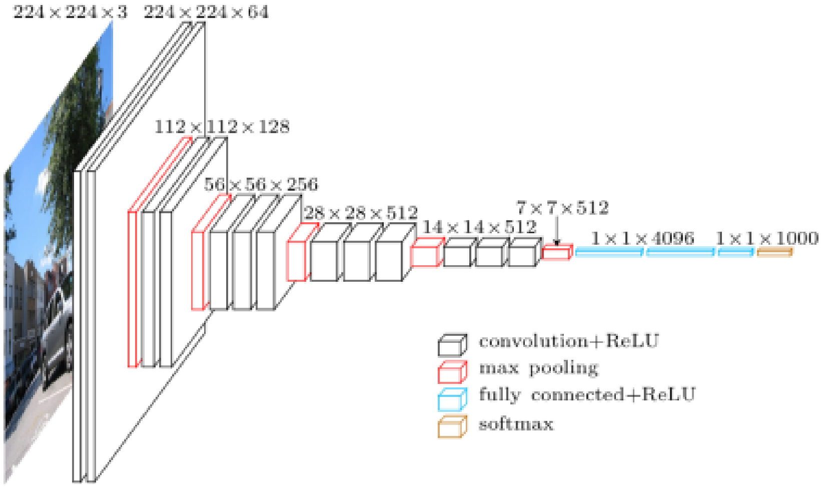
ImageNet

- A 1000-class image classification problem, with classes that are both very *diverse* (animals, transportation, people…) and very *specific* (100 breeds of dogs!).
- A state-of-the-art predictor needs to be very good at extracting features from virtually any image!
- Early success story of deep learning (2012); human performance ($\sim$ 94%) surpassed by a 150-layer neural network in 2015.
- Pre-trained models are readily available in deep learning libraries (such as Keras, which I will be using for all the demos).

ImageNet classification



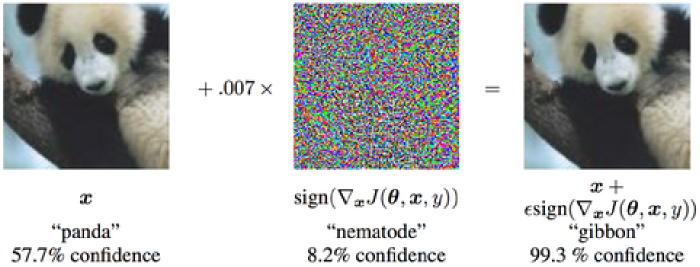
VGG-16



224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax

Adversarial examples

Our first application seeks to demonstrate how easy it is to “dethrone” such models with minimal effort.


$+ .007 \times$
$=$
$\boldsymbol{x}$
“panda”
57.7% confidence
$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
“nematode”
8.2% confidence
$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
“gibbon”
99.3 % confidence

The adversarial loss function

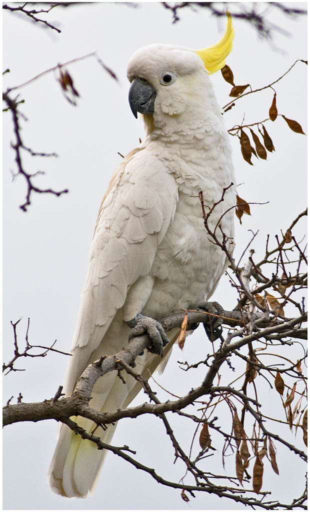
- A classification model outputs a vector of *confidences*: $h(\vec{x}; \vec{w})_i = \mathbb{P}(\vec{x} \text{ in class } i)$.

- Define a loss function to make the input maximise any class confidence you want!

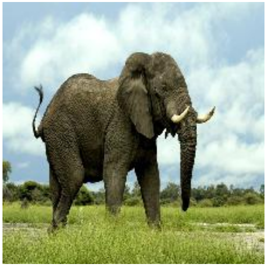
$$\mathcal{L}^i_{adv}(\vec{x}) = -h(\vec{x}; \vec{w})_i$$

- For my demo, I choose the class $i = 89$, i.e. a *sulphur-crested cockatoo*. :)

Totally a sulphur-crested cockatoo

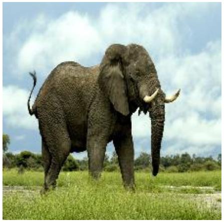


Totally <u>not</u> a sulphur-crested cockatoo

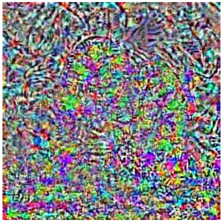


“tusker” (54.96% confidence)

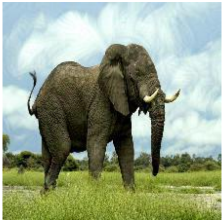
Totally not a sulphur-crested cockatoo (?!)



$+ \quad \varepsilon \times$

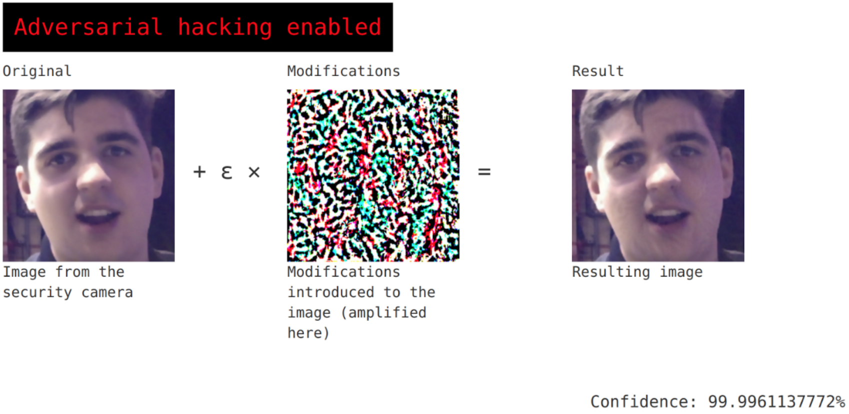


$=$



"sulphur-crested cockatoo" (99.99% confidence!)

Security implications: *facejack*



```
Adversarial hacking enabled

Original                    Modifications              Result

            + ε ×                          =

Image from the              Modifications              Resulting image
security camera             introduced to the
                            image (amplified
                            here)

                                                    Confidence: 99.9961137772%
```

https://github.com/PetarV-/facejack

Don’t even need to mess with the image. . .

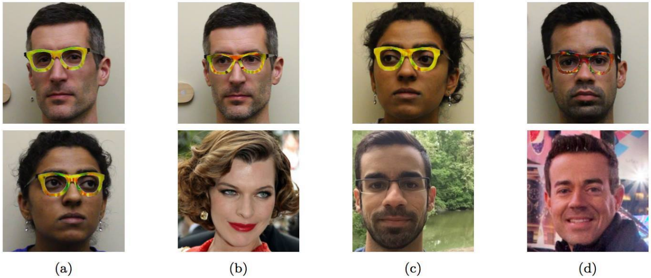


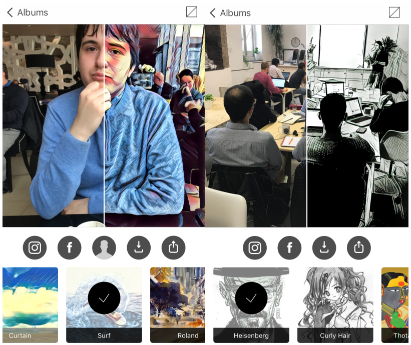
Figure 4: Examples of successful impersonation and dodging attacks. Fig. (a) shows S_A (top) and S_B (bottom) dodging against DNN_B. Fig. (b)–(d) show impersonations. Impersonators carrying out the attack are shown in the top row and corresponding impersonation targets in the bottom row. Fig. (b) shows S_A impersonating Milla Jovovich (by Georges Biard / CC BY-SA / cropped from https://goo.gl/GlsWlC); (c) S_B impersonating S_C; and (d) S_C impersonating Carson Daly (by Anthony Quintano / CC BY / cropped from https://goo.gl/VfnDct).

Some more scary facts

- Don’t even need full gradients—only their *sign* suffices!
- Typically, there’s an entire *space* of adversarial examples around any particular training example.
  - This is because adversarial examples are quite artificial (they “embed a pattern within an example”), and are hence unlikely to be found in a training set.
- The examples often exhibit *transferability*: *what’s adversarial for one model is likely to be adversarial for another*.
- Re-training models on adversarial examples is now a well-known technique for making them more *robust*.

Neural style transfer

The second application seeks to generate images that preserve "content" of a *base* image but capture "style" of a *reference* image.



The content loss function

- Let $\mathcal{A}_i(\vec{x}; \vec{w})$ be the outputs of the *i*-th layer in the network.
- Also, let $\vec{x}_{base}$ be the base image, and $\vec{x}_{ref}$ the reference image.
- We previously saw that deeper layers of neural networks are able to capture content rather well. Therefore it is natural to enforce content similarity by making the outputs of one of these layers be "close" for our base image and generated image:

$$\mathcal{L}_{content}(\vec{x}) = ||\mathcal{A}_i(\vec{x}; \vec{w}) - \mathcal{A}_i(\vec{x}_{base}; \vec{w})||^2$$

The style loss function

- Style is a bit more complicated to capture. We may still make advantage of the outputs, but:
  - Style corresponds less to content; more to *correlation* between parts of the content.
  - Correlations can be reasonably approximated using a *Gram matrix* (an outer product of the features with themselves). For features x and y of the i-th layer:

$$\mathcal{G}_i(\vec{x}; \vec{w})_{xy} = \mathcal{A}_i(\vec{x}; \vec{w})_x \mathcal{A}_i(\vec{x}; \vec{w})_y$$

  - Typically we'll want to match style at multiple scales j.
- Therefore:

$$\mathcal{L}_{style}(\vec{x}) = \sum_j ||\mathcal{G}_j(\vec{x}; \vec{w}) - \mathcal{G}_j(\vec{x}_{ref}; \vec{w})||^2$$

The continuity loss function

- There is another desirable property of the input—we wouldn't want the generated image to be too *"jumpy"* (i.e. changing colour intensities too much with adjacent pixels).

- Enforce this with a *continuity loss*:

$$\mathcal{L}_{continuity}(\vec{x}) = \sum_{i=1}^{n}\sum_{j=2}^{m}(x_{ij} - x_{i,j-1})^2 + \sum_{i=2}^{n}\sum_{j=1}^{m}(x_{ij} - x_{i-1,j})^2$$

Combining the losses

- Similarly as before:

$$\mathcal{L}_{styletrans}(\vec{x}) = \mathcal{L}_{content}(\vec{x}) + \alpha\mathcal{L}_{style}(\vec{x}) + \beta\mathcal{L}_{continuity}(\vec{x})$$

- As usual, correctly choosing α and β is *important* and requires quite some experimentation. Different values give different results, many of which are useless.

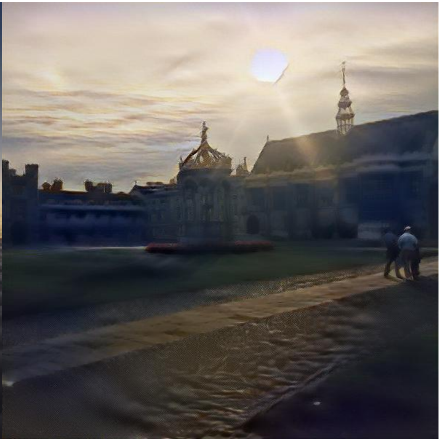
Demo time: Trinity!

Petar Veličković

24 September 2016 ·

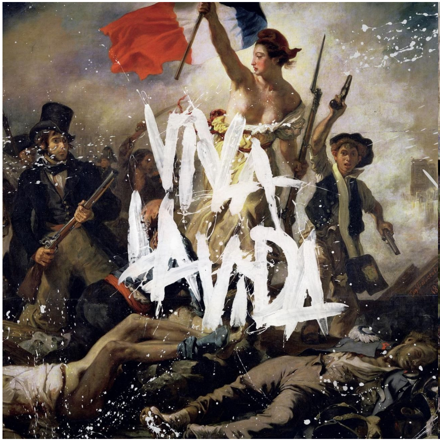
trin looking warm... and cool! how does it know? — at Trinity Great Court.

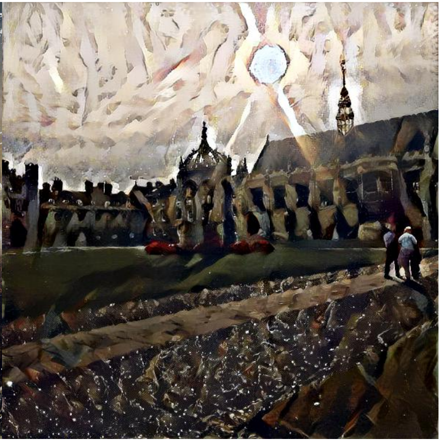
Demo time: Trinity + Pink Floyd!



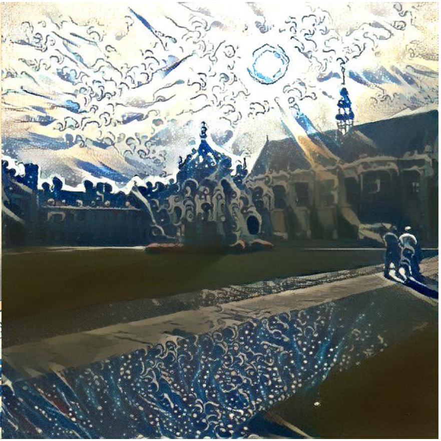
UNIVERSITY OF CAMBRIDGE

Demo time: Trinity + Coldplay!





Demo time: Trinity + Wave!



Further applications

- Aside from its direct use in apps such as Prisma, neural style transfer has made its way into the music and movie industries.

Bringing Impressionism to Life with Neural Style Transfer in *Come Swim*

Bhautik J Joshi*
Research Engineer, Adobe

Kristen Stewart
Director, *Come Swim*

David Shapiro
Producer, Starlight Studios

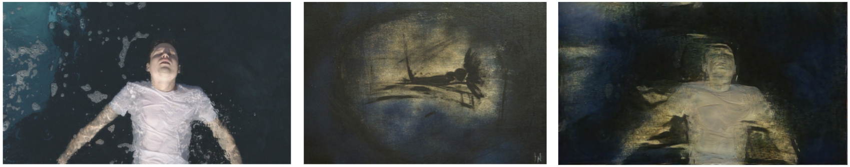
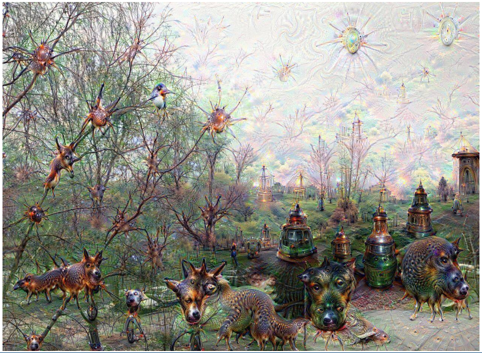


Figure 1: *Usage of Neural Style Transfer in Come Swim; left: content image, middle: style image, right: upsampled result. Images used with permission, (c) 2017 Starlight Studios LLC & Kristen Stewart.*

DeepDream

The third application seeks to provide an alternate way into visualising *feature representations*, by capturing what is it that a neural network will respond highly to *(and it just happened to put images in a bad trip as well)*.



The dream loss function

- Recall that convolutional layer outputs *"light up"* (attain higher values) when the corresponding pattern has been detected.
- Therefore, we should choose some layers, j, in the network, and aim to *maximise* the *intensity* of their output:

$$\mathcal{L}_{dream}(\vec{x}) = -\sum_j ||\mathcal{A}_j(\vec{x}; \vec{w})||^2$$

- The selection of the layers to maximise will depend primarily on whether we want to focus on lower or higher level feature representations (or perhaps the *combination* of them).

The L_2 and continuity losses

- One easy way in which this can fail is that the image simply becomes *too bright*, maximising everything *irrespective* of the features. Include an L_2 loss, which penalises bright pixels:

$$\mathcal{L}_{L_2}(\vec{x}) = ||\vec{x}||^2$$

- Finally, we introduce the continuity loss as before, as continuity is still desirable:

$$\mathcal{L}_{continuity}(\vec{x}) = \sum_{i=1}^{n}\sum_{j=2}^{m}(x_{ij} - x_{i,j-1})^2 + \sum_{i=2}^{n}\sum_{j=1}^{m}(x_{ij} - x_{i-1,j})^2$$

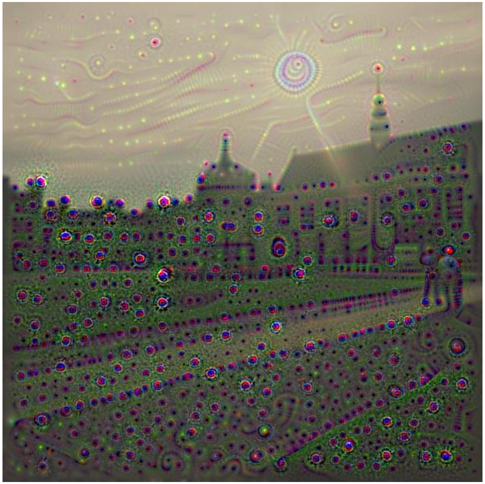
The DeepDream loss

- Similarly as before:

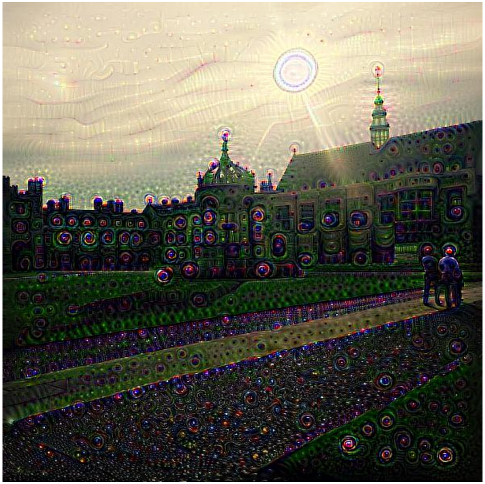
$$\mathcal{L}_{deepdream}(\vec{x}) = \mathcal{L}_{dream}(\vec{x}) + \alpha\mathcal{L}_{L_2}(\vec{x}) + \beta\mathcal{L}_{continuity}(\vec{x})$$

- Same remarks about α and β as usual. In particular, as we change which layers we want to "excite", the appropriate values might change drastically.

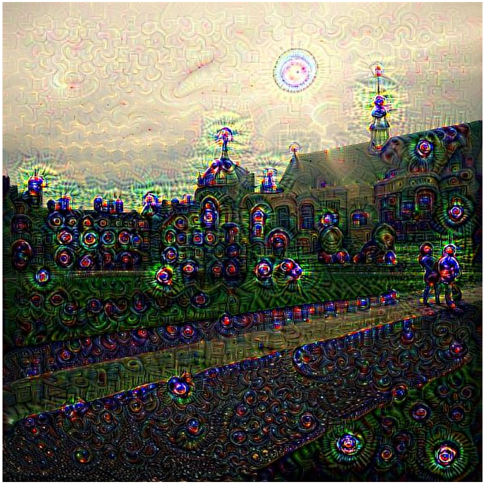
Demo time: Tripity!



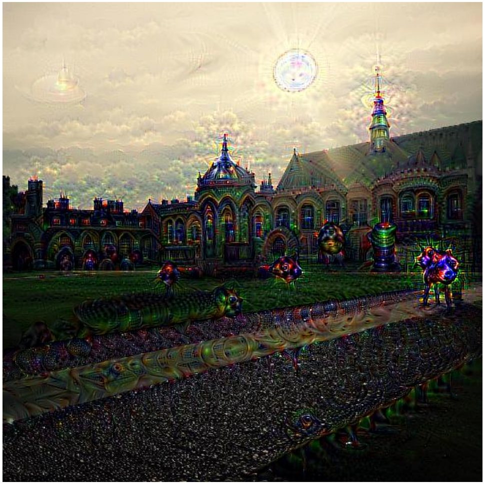
Demo time: Tripity!



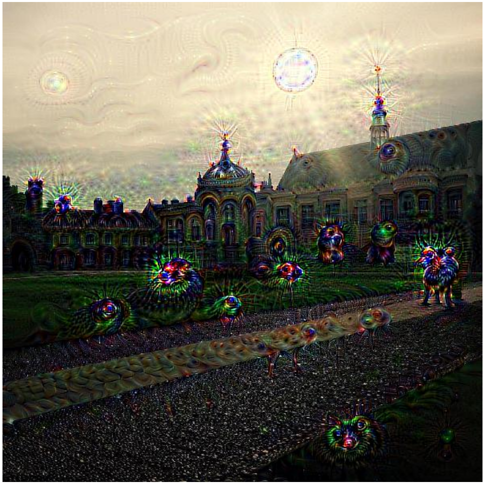
Demo time: Tripity!



Demo time: Tripity!



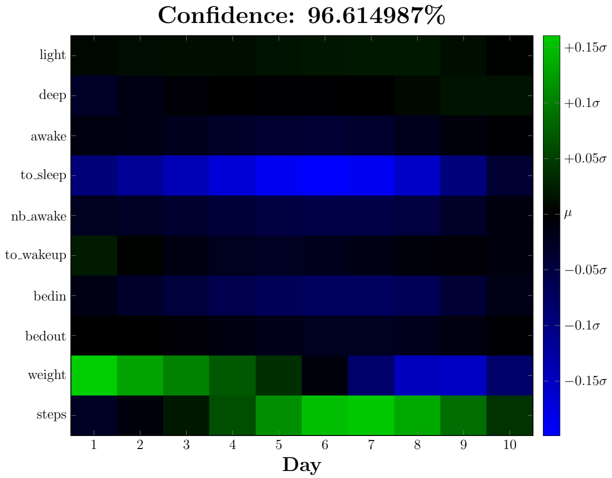
Demo time: Tripity!



Application of DeepDream to time-series data

- In my recent research work, I have produced recurrent neural network-based predictors of weight gain/loss.
- This was done based on very incomplete features (sleep, steps and weight measurements for the past N days), and it's not immediately obvious which sleep features are the most critical for the model's predictions.
- In order to visualise what the network was detecting the most, I applied DeepDream starting from a random ten-day sequence of features, in order to maximise confidence of achieving a -4kg weight objective.

The key to weight loss: *early sleep?*



Thank you!

Questions?

`petar.velickovic@cl.cam.ac.uk`

For all the demo sources, check out:
`https://github.com/PetarV-/deep-lossy-fun`