Machine Learning, JavaScript, and Art

A Neural Algorithm of Artistic Style

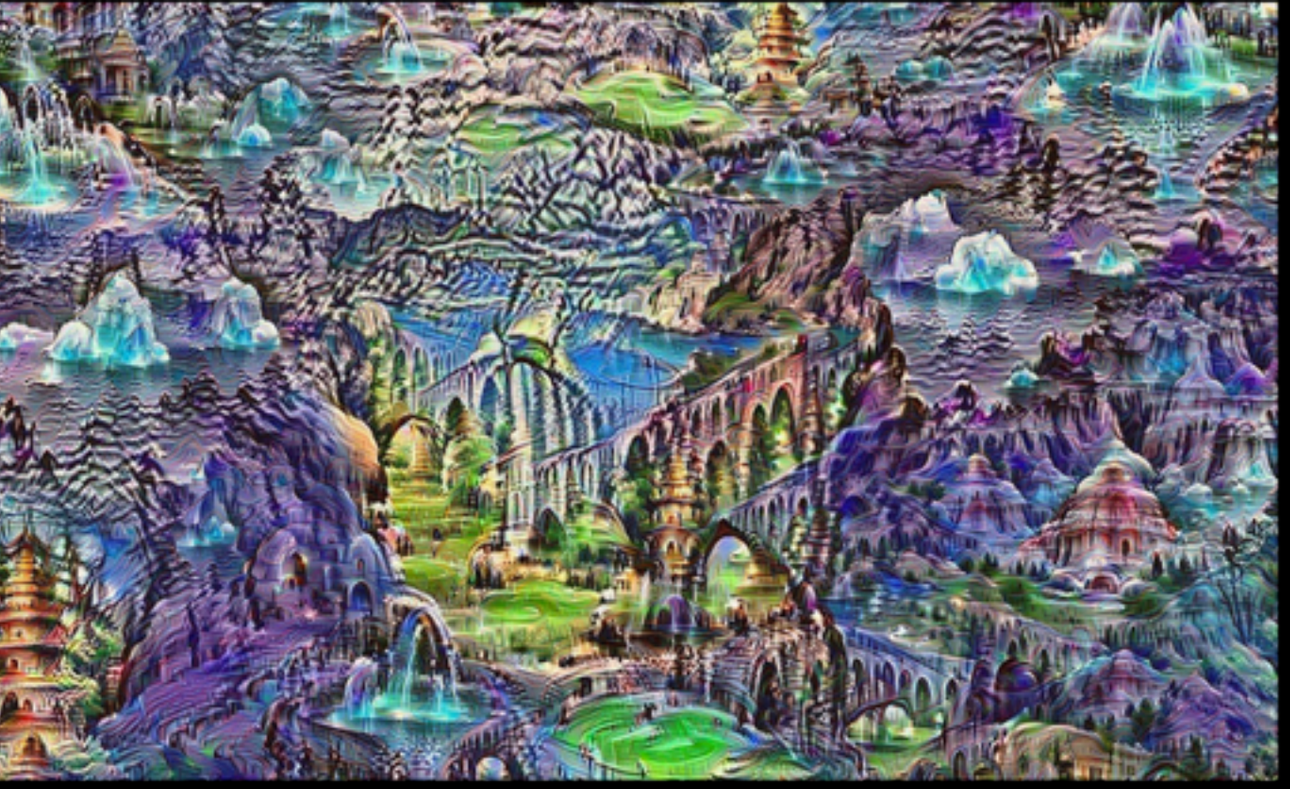
Leon A. Gatys,[1,2,3*] Alexander S. Ecker,[1,2,4,5] Matthias Bethge[1,2,4]

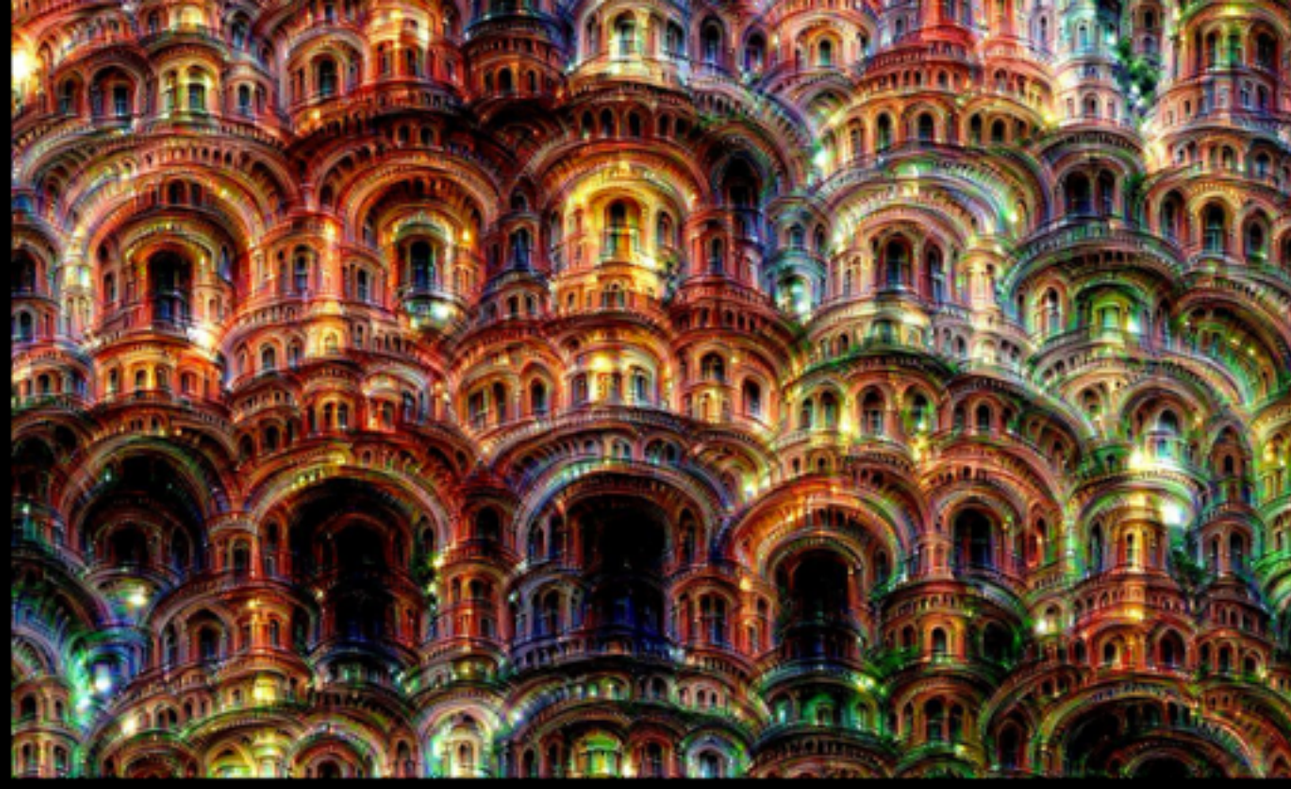
[1]Werner Reichardt Centre for Integrative Neuroscience
and Institute of Theoretical Physics, University of Tübingen, Germany
[2]Bernstein Center for Computational Neuroscience, Tübingen, Germany
[3]Graduate School for Neural Information Processing, Tübingen, Germany
[4]Max Planck Institute for Biological Cybernetics, Tübingen, Germany
[5]Department of Neuroscience, Baylor College of Medicine, Houston, TX, USA
*To whom correspondence should be addressed; E-mail: leon.gatys@bethgelab.org

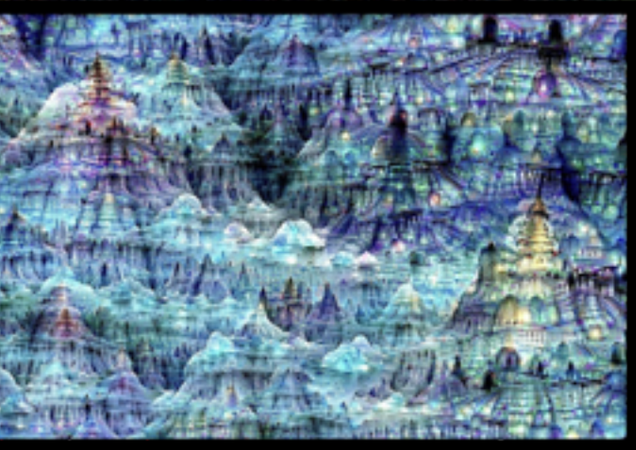
In fine art, especially painting, humans have mastered the skill to create unique visual experiences through composing a complex interplay between the content and style of an image. Thus far the algorithmic basis of this process is unknown and there exists no artificial system with similar capabilities. However, in other key areas of visual perception such as object and face recognition near-human performance was recently demonstrated by a class of biologically inspired vision models called Deep Neural Networks.[1,2] Here we introduce an

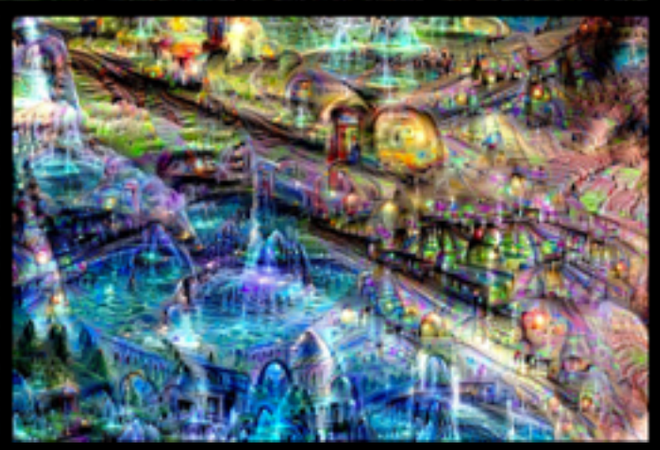

A
B
C
D
E
F

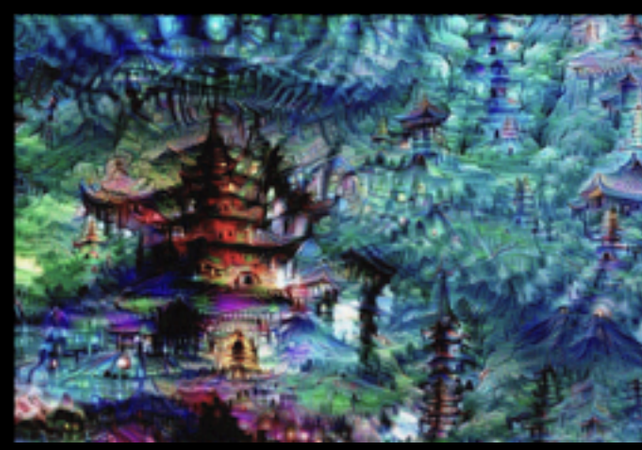
Source: https://dmitryulyanov.github.io/feed-forward-neural-doodle/



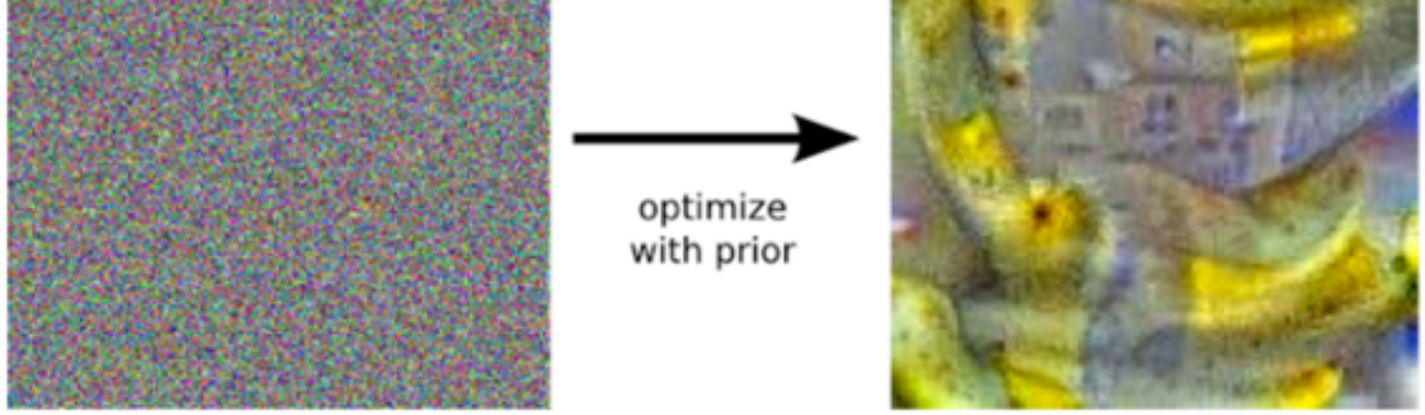








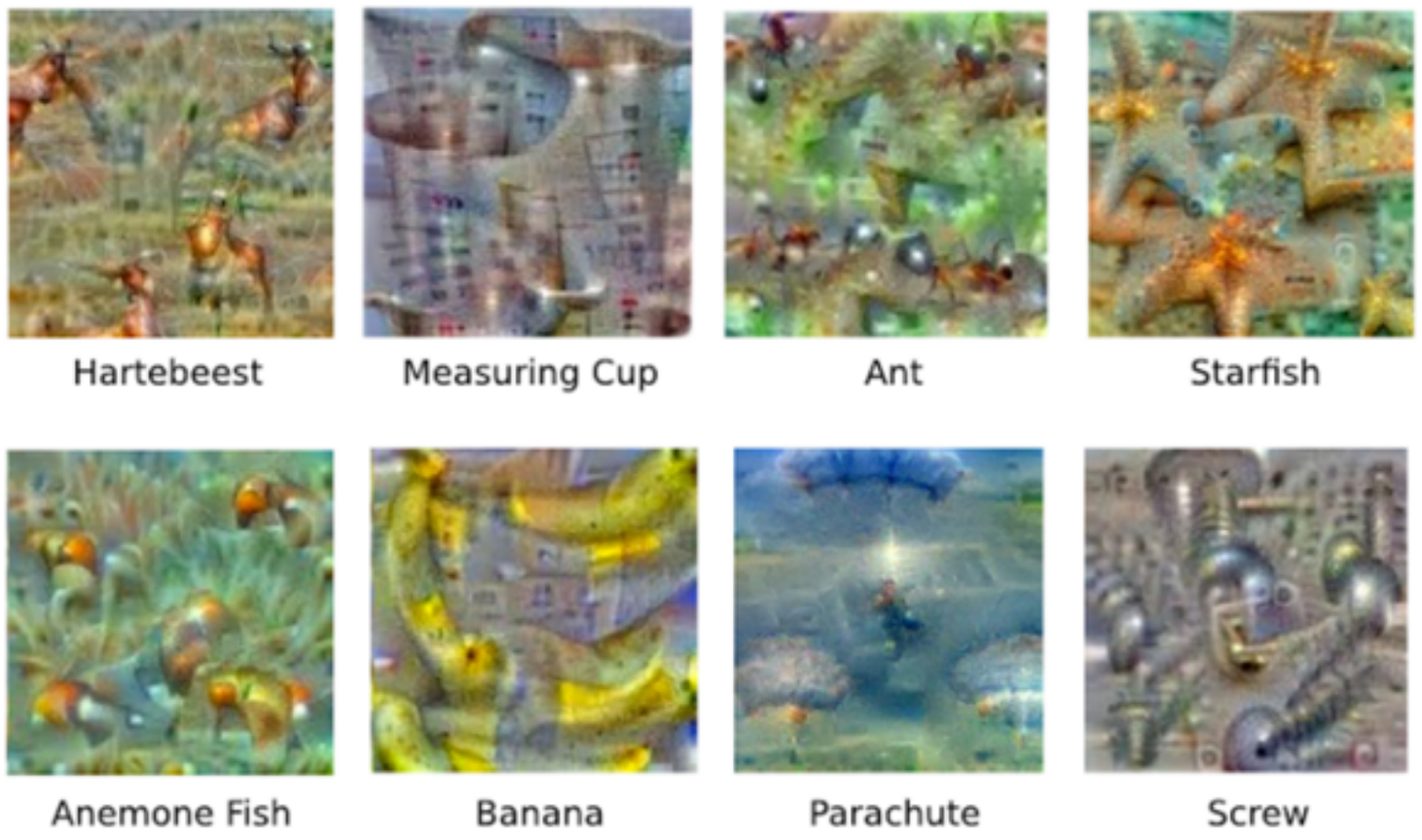
Source: https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html



So here's one surprise: neural networks that were trained to discriminate between different kinds of images have quite a bit of the information needed to

generate

images too. Check out some more examples across different classes:



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

Buy a print ▾ HD file ▾

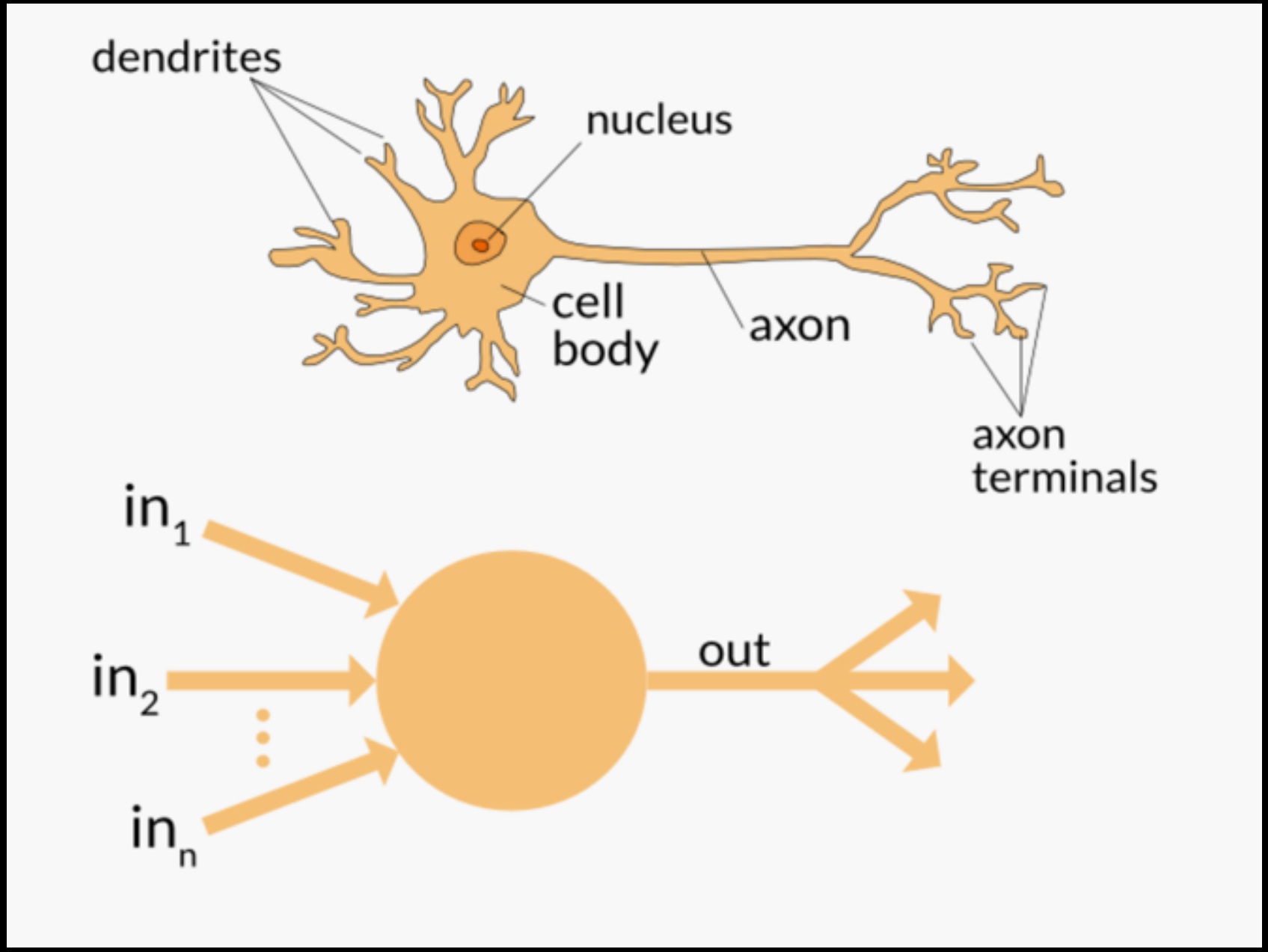
Use as style Download

What is machine learning?

Implementing solution.

Problem solving by analyzing similar problems and their solutions.

AKA Supervised Learning



Source: https://appliedgo.net/perceptron/

synaptic.js

```
Perceptron: function Perceptron() {
  var args = Array.prototype.slice.call(arguments); // convert arguments to Array
  if (args.length < 3)
    throw new Error("not enough layers (minimum 3) !!");

  var inputs = args.shift(); // first argument
  var outputs = args.pop(); // last argument
  var layers = args; // all the arguments in the middle

  var input = new Layer(inputs);
  var hidden = [];
  var output = new Layer(outputs);

  var previous = input;

  // generate hidden layers
  for (var i = 0; i < layers.length; i++) {
    var size = layers[i];
    var layer = new Layer(size);
    hidden.push(layer);
    previous.project(layer);
    previous = layer;
  }
  previous.project(output);

  // set layers of the neural network
  this.set({
    input: input,
    hidden: hidden,
```

```
for (var i = 0; i < layers.length; i++) {
  var size = layers[i];
  var layer = new Layer(size);
  hidden.push(layer);
  previous.project(layer);
  previous = layer;
}
```

```
    var derivative = getVar(this, 'derivative');
    switch (this.squash) {
      case Neuron.squash.LOGISTIC:
        buildSentence(activation, ' = (1 / (1 + Math.exp(-', state, ')))',
          store_activation);
        buildSentence(derivative, ' = ', activation, ' * (1 - ',
          activation, ')', store_activation);
        break;
      case Neuron.squash.TANH:
        var eP = getVar('aux');
        var eN = getVar('aux_2');
        buildSentence(eP, ' = Math.exp(', state, ')', store_activation);
        buildSentence(eN, ' = 1 / ', eP, store_activation);
        buildSentence(activation, ' = (', eP, ' - ', eN, ') / (', eP, ' + ', eN,
', store_activation);
        buildSentence(derivative, ' = 1 - (', activation, ' * ', activation, ')',
tore_activation);
        break;
      case Neuron.squash.IDENTITY:
        buildSentence(activation, ' = ', state, store_activation);
        buildSentence(derivative, ' = 1', store_activation);
        break;
      case Neuron.squash.HLIM:
        buildSentence(activation, ' = +(', state, ' > 0)', store_activation);
        buildSentence(derivative, ' = 1', store_activation);
      case Neuron.squash.RELU:
        buildSentence(activation, ' = ', state, ' > 0 ? ', state, ' : 0',
tore_activation);
        buildSentence(derivative, ' = ', state, ' > 0 ? 1 : 0', store_activation);
        break;
```

```
      case Neuron.squash.TANH:
        var eP = getVar('aux');
        var eN = getVar('aux_2');
        buildSentence(eP, ' = Math.exp(', state, ')', store_activation);
        buildSentence(eN, ' = 1 / ', eP, store_activation);
        buildSentence(activation, ' = (', eP, ' - ', eN, ') / (', eP, ' + ', eN,
', store_activation);
        buildSentence(derivative, ' = 1 - (', activation, ' * ', activation, ')',
ore_activation);
        break;
```

deeplearn.js

```
export class TanHFunc implements ActivationFunction {
  output<T extends NDArray>(math: NDArrayMath, x: T) {
    return math.scope(() => {
      return math.tanh(x);
    });
  }}
```

The Hyperbolic Tangent Activation Function

Though the logistic sigmoid has a nice biological interpretation, it turns out that the logistic sigmoid can cause a neural network to get "stuck" during training. This is due in part to the fact that if a strongly-negative input is provided to the logistic sigmoid, it outputs values very near zero. Since neural networks use the feed-forward activations to calculate parameter gradients (again, see this previous post for details), this can result in model parameters that are updated less regularly than we would like, and are thus "stuck" in their current state.

An alternative to the logistic sigmoid is the hyperbolic tangent, or tanh function (Figure 1, green curves):

$$\begin{aligned} g_{\tanh}(z) &= \frac{\sinh(z)}{\cosh(z)} \\ &= \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}. \end{aligned}$$

Like the logistic sigmoid, the tanh function is also sigmoidal ("s"-shaped), but instead outputs values that range $(-1, 1)$. Thus strongly negative inputs to the tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs. These properties make the network less likely to get "stuck" during training. Calculating the gradient for the tanh function also uses the quotient rule:

$$\begin{aligned} g'_{\tanh}(z) &= \frac{\partial}{\partial z}\frac{\sinh(z)}{\cosh(z)} \\ &= \frac{\frac{\partial}{\partial z}\sinh(z)\times\cosh(z)-\frac{\partial}{\partial z}\cosh(z)\times\sinh(z)}{\cosh^{2}(z)} \\ &= \frac{\cosh^{2}(z)-\sinh^{2}(z)}{\cosh^{2}(z)} \\ &= 1-\frac{\sinh^{2}(z)}{\cosh^{2}(z)} \\ &= 1-\tanh^{2}(z) \end{aligned}$$

Demo Time!

```
    var imgData = getPixelData(sourceImage);

    var iterate = function(){
      for (var x = 0; x < 300; x+=1)
      {
        for(var y = 0; y < 300; y+=1)
        {
          var dynamicRate =  .01/(1+.0005*iteration);
          perceptron.activate([x/300,y/300]);
          perceptron.propagate(dynamicRate,
pixel(imgData,x,y));
        }
      }
      preview();
    };
```

```
var imgData = getPixelData(sourceImage);

var iterate = function(){
  for (var x = 0; x < 300; x+=1)
  {
    for(var y = 0; y < 300; y+=1)
    {
      var dynamicRate =  .01/(1+.0005*iteration);
      perceptron.activate([x/300,y/300]);
      perceptron.propagate(dynamicRate, pixel(imgData,x,y));
    }
  }
  preview();
};
```

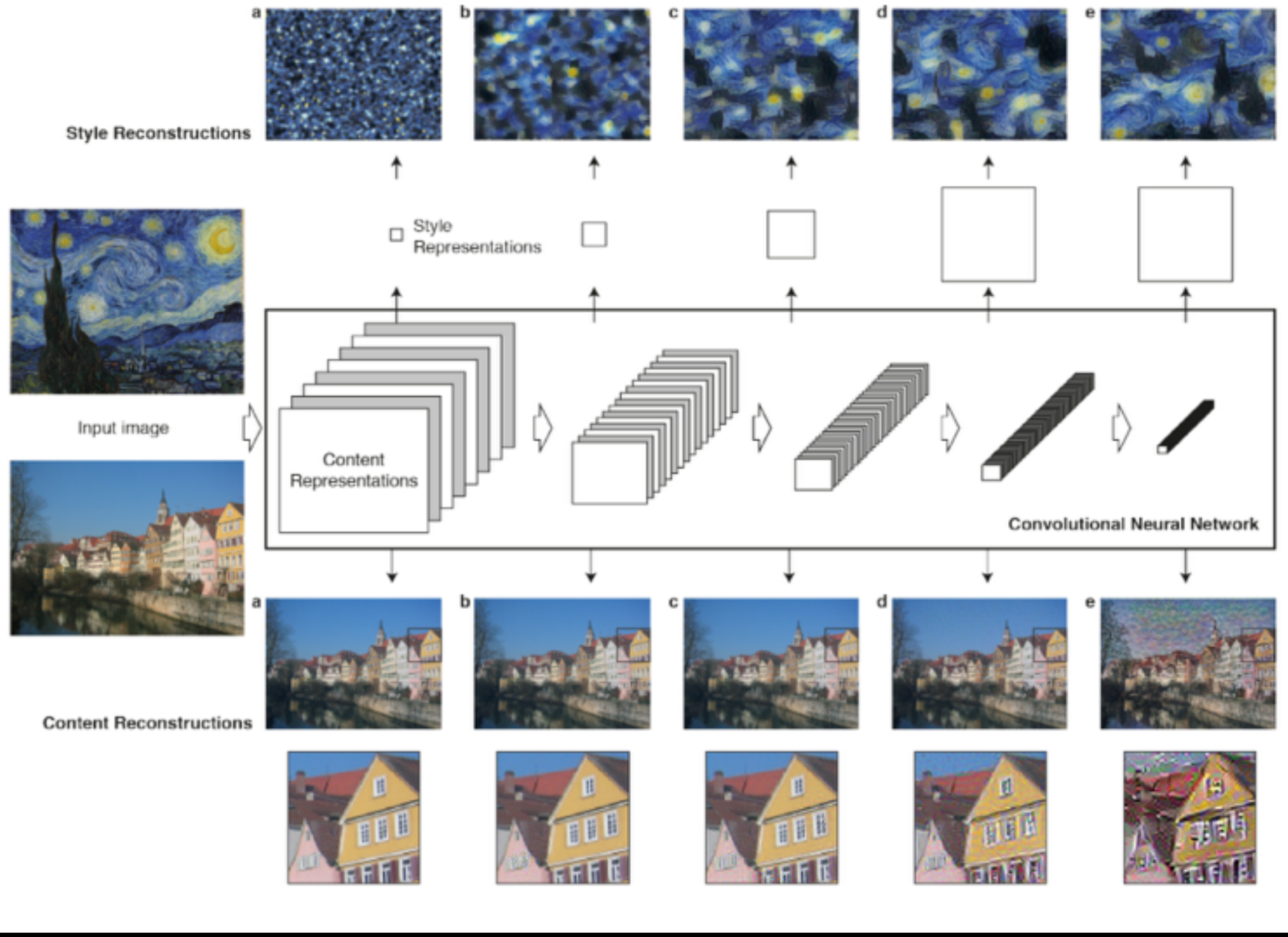

a
b
c
d
e
Style Reconstructions
Style
Representations
Input image
Content
Representations
Convolutional Neural Network
a
b
c
d
e
Content Reconstructions

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Machine learning with JavaScript is doable but kinda slow.

Style Transfer -> Python
DeepDream -> C++
??? -> JavaScript

WebGL maybe????

On translate3d and layer creation hacks

Translate3d is often hailed as something of a silver bullet. In many cases it will drastically improve rendering performance in WebKit and Blink browsers like Chrome, Opera and Safari. Let's take a look at why it can improve rendering performance, and what we need to be careful of.

LA
27

EST
6 M

SHA
TW

TAG
#lay

THE “GO FASTER” HACK!

So you know that magic bullet hack for Chrome (and other WebKit ports) that makes

BUT

Still Doable!

THE END

@am3thyst