

Chemical Biophysics Symposium: introduction to neural networks

Erik Spence

SciNet HPC Consortium

28 April 2023

Today's slides and code

Today's slides and code can be found here. Go to the "Chemical Biophysics Symposium" page, under Lectures, "slides".

<https://scinet.courses/1289>

Who am I?

My name is Erik Spence.

- I am an Applications Analyst at SciNet (<https://www.scinethpc.ca>).
- SciNet is a High-Performance-Computing (HPC) consortium, one of six in Canada, run by the University of Toronto.
- These consortia run massively parallel computers, with tens of thousands of cores, to perform computations that couldn't be done otherwise.
- My job at SciNet is to help users get their codes to run on these machines.
- We also educate users on how to write fast, efficient code.

Today I'm going to give an introduction to programming and using neural networks.

Neural networks are commonplace

Neural networks are particularly good at detecting patterns, and for certain problems perform better than any other known class of algorithm. Neural networks are used for

- Image recognition, object detection (pneumonia, cancer).
- Medical diagnosis.
- Natural language processing (voice recognition).
- Novelty detection (detection of outliers).
- Next-word predictions.
- Text sentiment analysis.
- System control (self-driving cars).

Neural networks are finding their way into everything.

Neural networks, motivation

Consider the problem of hand-written digit recognition:

9 2 8 1 2 3

How would you go about writing a program which can tell you what digits are displayed?

- All the algorithms you might use to describe what a given number "looks like" are extremely difficult to implement in code. Where do you even start?
- And yet humans can easily tell what these digits are.
- Neural networks are based on a "biologically inspired" approach to solving such classification problems.
- This is one of the classic problems which have been solved using neural networks.

Neural networks, the approach

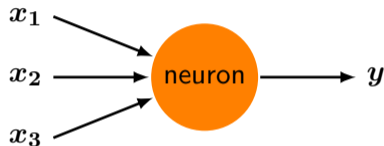
Rather than focus on the details of what individual numbers look like, we will instead ignore those details altogether. We will use a completely different approach:

- Break the data set of numbers into two or three groups: training, testing, and optionally validation.
- As with other supervised machine-learning algorithms, feed the training data to the neural network and train it to recognize one number from another.
- Rather than focus on details of the numbers, let the neural network figure out the details for itself.

This is the goal of this workshop.

Neurons

Neural networks are built upon "neurons". This is just a fancy way of saying a "function that takes multiple inputs and returns a single output".



The function which the neuron implements is up to the programmer, but it must contain free parameters so that the network can be trained. These functions usually take the form

$$f(x_1, x_2, x_3) = f\left(\sum_{i=1}^3 w_i x_i + b\right) = f(\mathbf{w} \cdot \mathbf{x} + b)$$

Where \mathbf{w} are the 'weights' and b is the 'bias'. These are the trainable parameters.

Neurons, continued

What function should we use for f ? One which is usually used, at least when initially teaching about neural networks, is the "sigmoid function" (also called the "logistic function").

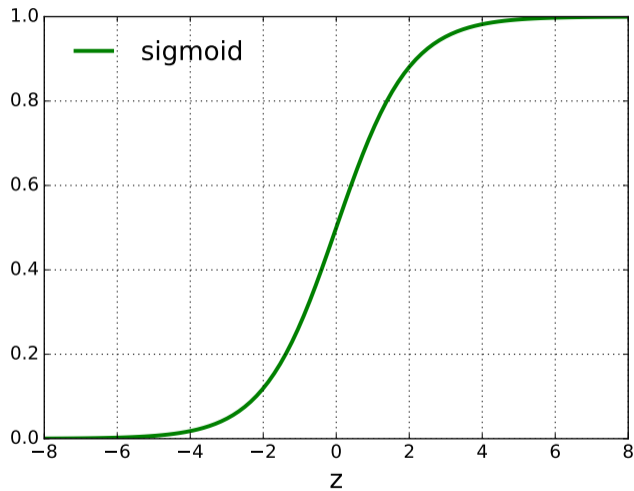
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

And so our neuron function becomes

$$f(x_1, x_2, x_3) = f(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}}$$

Where again \mathbf{w} are the 'weights' and b is the 'bias'.

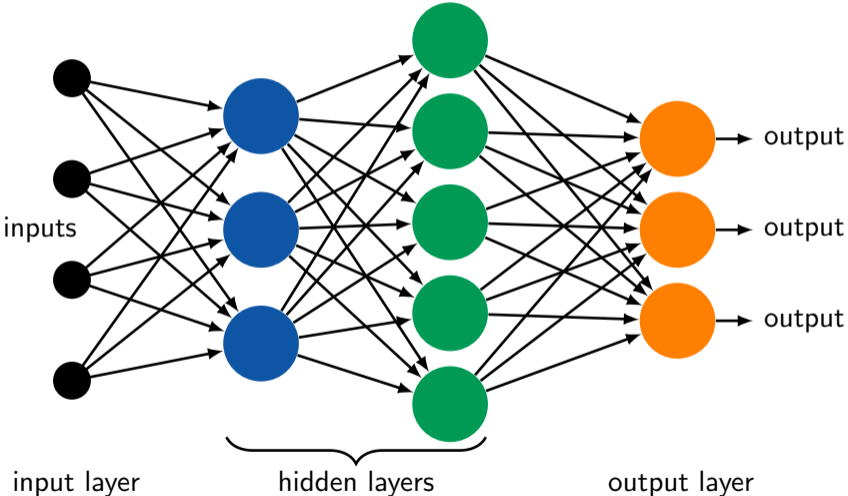
Why the sigmoid function?



Because it ranges from 0 to 1 smoothly.

Neural networks

Suppose we combine many neurons together, into a proper network, consisting of "layers".



Some notes about neural networks

Some details about the graphic on the previous slide:

- The input neurons do not contain any functions. They merely represent the input data being fed into the network.
- Each neuron in the 'hidden' layers and the output layer all contain functions with their own free parameters, \mathbf{w} and \mathbf{b} .
- Each neuron outputs a single value. This output is passed to all of the neurons in the subsequent layer. This type of layer is known as a "fully-connected", or "dense", layer.
- The number of free parameters in the neurons in any given layer depends upon the number of neurons in the previous layer.
- The output from the output layer is aggregated into the desired form.

Seriously?

You might legitimately wonder why on Earth we would think this would lead anywhere.

- As it happens, this topology is similar to some simple biological neural networks.
- Each layer takes the output of the previous layer as its input.
- Each layer makes "decisions" about the information that it receives.
- In this way the later layers are able to make more complex and abstract decisions than the earlier layers.
- A many-layered network can potentially make sophisticated decisions.

However, there are subtleties in training such a network.

Training our neural network

How do we optimize the weights and biases? We need to define some sort of "cost function" (sometimes called "loss" or "objective" function):

$$C = \frac{1}{2} \sum_i (f(\mathbf{x}_i) - \mathbf{y}_i)^2$$

where f is our neural network, and \mathbf{y}_i are the correct answers, based on the data, associated with each \mathbf{x}_i . Here we are using the "quadratic" cost function.

We then use an optimization algorithm to search for the values of \mathbf{w} and \mathbf{b} which generate the minimum of C , given the data \mathbf{x} and \mathbf{y} . We will use the Gradient Descent algorithm to find this minimum.

Gradient descent

Suppose the function we want to minimize has only one parameter.

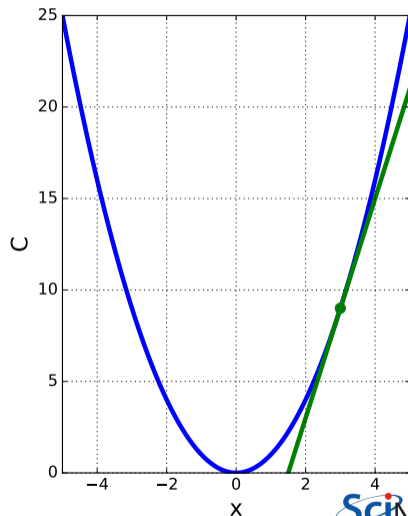
$$C = w^2$$

Suppose we've guessed that the minimum of C is w_j , and we wish to improve the guess. Gradient descent says to move according to the formula:

$$w_{j+1} = w_j - \eta \frac{\partial C}{\partial w_j}$$

where η is called the step size. We then repeat until some stopping criterion is satisfied.

If we have multiple parameters, we step them all.




Training a neural network

How do we apply Gradient Descent to a neural network?

- Suppose that we decide to try to use gradient descent to train the network from five slides ago (slide 10).
- Each of the neurons has its own set of free parameters, \mathbf{w} and \mathbf{b} . There are lots of free parameters!
- To update the parameters we need to calculate every $\frac{\partial C}{\partial w_i}$ and $\frac{\partial C}{\partial b}$ for every neuron!
- But how do we calculate those derivatives, especially for the parameters associated with the neurons that are several layers away from the output?

Actually, as it happens, this is a solved problem. The algorithm is called Backpropagation, but we won't cover it today.

Handwritten digits

A row of six handwritten digits: 9, 2, 8, 1, 2, 3. The digits are written in a simple, slightly irregular black ink on a white background.

One of the classic data sets on which to test neural-network techniques is the MNIST data set.

- A database of handwritten digits, compiled by NIST.
- Contains 60000 training, and 10000 test examples.
- The training digits were written by 250 different people; the test data by 250 different people.
- The digits have been size-normalized and centred.
- Each image is grey scale, 28 x 28 pixels.

We can create a neural network to classify these digits.

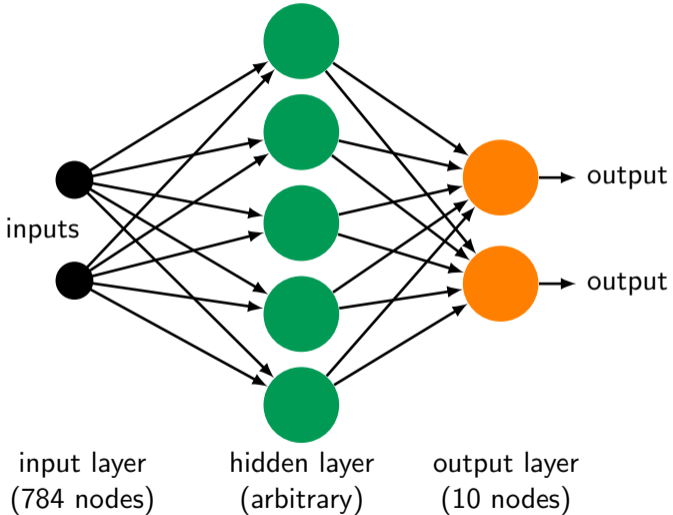
Our network

How would we design a network to analyze this data?

- Each image is $28 \times 28 = 784$ pixels. Let the input layer consist of 784 input nodes. Each node will consist of the grey value for that pixel.
- The output will consist of a one-hot-encoding of the networks analysis of the input data. This means that, if the input image depicts a '7', the output vector should be $[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]$.
- Thus, let there be 10 output nodes, one for each possible digit.
- To start, let's just use a single hidden layer.

Fortunately, packages exist which make coding such a network quite easy.

Our neural network



Neural network frameworks

Now that we have a plan for our network, how are we going to code it? The standard way is to use a neural network 'framework'. Why would you do that?

- Coding your own networks from scratch can be a bit of work.
- Neural network (NN) frameworks have been specifically designed to solve NN problems.
- Python, of course, is not a high-performance language.
- The NN frameworks which have been developed are compiled before being used, thus being much faster than interpreted Python.
- The NN frameworks are also designed to use GPUs, which make things significantly faster than just using CPUs.
- Standard NN frameworks include TensorFlow, Torch, MXNet, Caffe and many many others.

We will use Keras on a TensorFlow backend.

Keras

We will use Keras on top of TensorFlow.

- Keras is a NN framework, but it's only the top-most level.
- More accurately, it's an API standard for creating neural networks.
- Designed for fast development of networks.
- The original version ran on top of a 'back end', which by default is now TensorFlow, as Keras is being absorbed into TensorFlow.
- Historically it ran on top of many other backends also: Theano, CNTK, MXNet, TypeScript, JavaScript, PlaidML, Scala, CoreML, and others.
- Because it's a proper framework, all of the NN goodies you need are already built into it.
- Because the recommended way is to use Keras through TensorFlow, that is the way we will be using it.

Getting the data

Let us implement our neural network using Keras. First let us get the MNIST data.

- The data can be automatically downloaded by Keras.
- The data comes pre-split into training and testing data sets.
- We one-hot-encode the target data before returning.

```
# get_mnist.py
from tensorflow.keras.datasets import mnist
import tensorflow.keras.utils as ku

def get_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    y_train = ku.to_categorical(y_train, 10)
    y_test = ku.to_categorical(y_test, 10)

    return x_train, x_test, y_train, y_test
```

```
In [1]: import get_mnist
-----
In [2]:
-----
In [2]: x_train, x_test, y_train, y_test = get_mnist.get_data()
-----
In [3]: x_train.shape
Out[3]: (60000, 28, 28)
```

Reshaping the data

The data needs to be in a specific format:

- As mentioned, we want the data to have dimension 784, rather than 28×28 .
- If the input data is 2D, it must have a 'depth' added, to make it 3D, even if the depth is 1.
- If the input data is 1D no depth is needed.

```
In [4]:
```

```
x_train = x_train.reshape(60000, 784)
```

```
x_test = x_test.reshape(10000, 784)
```

```
In [6]:
```

```
x_test.shape
```

```
Out[6]: (10000, 784)
```

```
In [7]:
```

Our network using Keras

We implement our neural network using Keras, with a single hidden layer.

- A "Sequential" model means the layers are stacked on one another in a linear fashion.
- A "Dense" ("fully-connected") layer is the layer we've already discussed.
- Use "input_dim" in the first layer to indicate the shape of the incoming data.
- The "activation" is the output function of the neuron.

```
# model1.py
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl

def build_model(numnodes):

    model = km.Sequential()

    model.add(kl.Dense(numnodes,
                        input_dim = 784,
                        activation = 'sigmoid'))

    model.add(kl.Dense(10,
                        activation = 'sigmoid'))

    return model
```

Our network using Keras, continued

We implement our neural network using Keras, with 30 neurons in the hidden layer.

We can use the summary function to look at the details of the network.

```
In [7]:  
-----  
In [7]: import model1  
-----  
In [8]:  
-----  
In [8]: model = model1.build_model(30)  
-----  
In [9]:  
-----  
In [9]: model.summary()  
  
-----  
Layer (type) Output Shape Param #  
-----  
dense_1 (Dense) (None, 30) 23550  
-----  
dense_2 (Dense) (None, 10) 310  
-----  
Total params: 23,860  
Trainable params: 23,860  
Non-trainable params: 0  
-----  
In [10]:
```


Our network using Keras, continued more

Now that the network is constructed, it must be compiled.

- The loss function must be specified.
- The optimizer indicates what minimization algorithm to use. Here we use Stochastic Gradient Descent (SGD), which is a variation on regular Gradient Descent.
- The 'metrics' flag indicates what to print out during the training of the network.
- The 'fit' command is used to execute the training.
- The number of epochs, and the batch size, are parameters which apply to Stochastic Gradient Descent.

Our network using Keras, continued

```
In [10]:
```

```
In [10]: model.compile(loss = 'mean_squared_error', optimizer = 'sgd', metrics = ['accuracy'])
```

```
In [11]:
```

```
In [11]: fit = model.fit(x_train, y_train, epochs = 200, batch_size = 128)
```

```
Epoch 1/200
```

```
469/469 [=====] - 0s - loss: 0.1368 - acc: 0.1933
```

```
Epoch 2/200
```

```
469/469 [=====] - 0s - loss: 0.0923 - acc: 0.3126
```

```
⋮
```

```
Epoch 199/200
```

```
469/469 [=====] - 0s - loss: 0.0225 - acc: 0.8947
```

```
Epoch 200/200
```

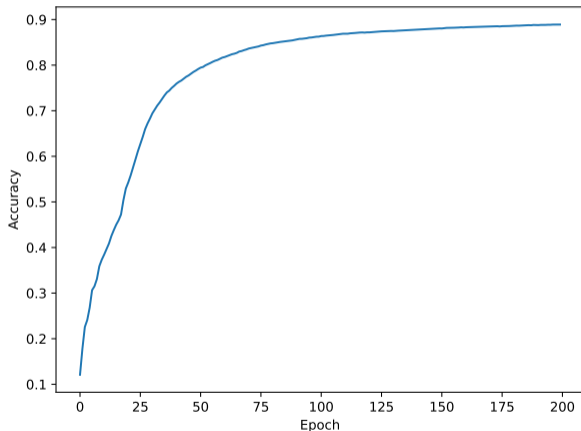
```
469/469 [=====] - 0s - loss: 0.0225 - acc: 0.8947
```

```
In [12]:
```

Plotting the training

The `fit.history` dictionary contains useful information about the training. Sometimes plotting can give you some insight into the quality of the training, and whether or not it's finished.

```
In [12]:  
-----  
In [12]: import matplotlib.pyplot as plt  
-----  
In [13]: plt.plot(fit.history['accuracy'])  
-----  
In [14]: plt.xlabel('Epoch')  
-----  
In [15]: plt.ylabel('Accuracy')  
-----  
In [16]:
```



Our network using Keras, continued even more

Now check against the test data.
88.5%!

This isn't great. We can do
better.

```
In [16]:  
-----  
In [16]: score = model.evaluate(x_test, y_test)  
313/313 [======>.....] - ETA: 0s  
-----  
In [17]:  
-----  
In [17]: score  
[0.024616853955388068, 0.8851999999999999]  
-----  
In [18]:
```

The next steps

We can do better. What might we do? There are a few simple approaches that might be explored.

- Change the activation function.
- Change the cost function.
- Change the optimization algorithm.
- Change the way things are initialized.
- Add regularization, to try to deal with over-fitting.

The most important technique, however:

- Completely overhaul the network strategy.

This field is huge; we've barely scratched the surface.

Other activation functions: relu

Two commonly-used functions:

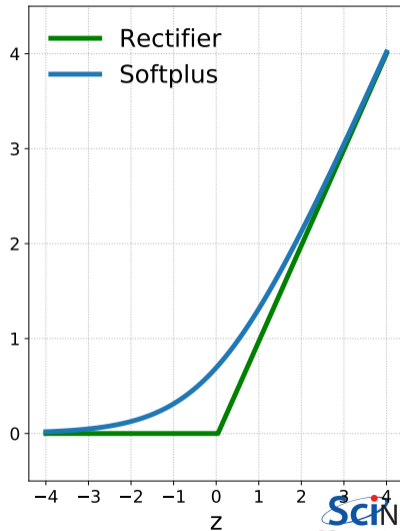
- Rectifier (also called Rectifier Linear Units, or RELUs):

$$f(z) = \max(0, z).$$

- Softplus:

$$f(z) = \ln(1 + e^z).$$

- Good: doesn't suffer from the vanishing-gradient problem.
- Bad: unbounded, could blow up.
- Other variants: leaky RELU, and SELU (scaled exponential).

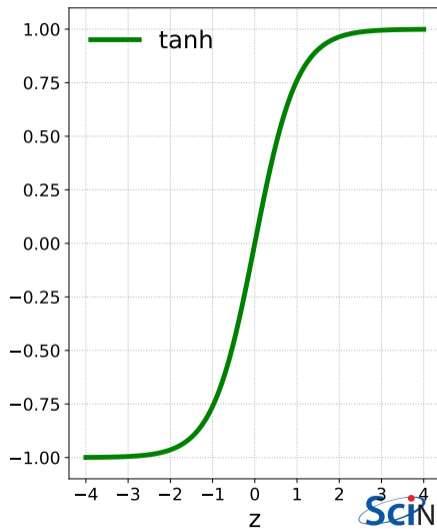


Other activation functions: tanh

Another commonly-used activation function is tanh:

$$f(z) = \tanh(z).$$

- Good: stronger gradients than sigmoid, faster learning rate, doesn't suffer from the vanishing-gradient problem.
- Good: because the function is anti-symmetric about zero. This also results in faster learning, at least for deeper networks.



Other activation functions: softmax

One of the more-commonly used output-layer activation functions is the softmax function:

$$s(z_j) = \frac{e^{z_j}}{\sum_{k=1}^N e^{z_k}},$$

where N is the number of output neurons. The advantage of this function is that it converts the output to a probability.

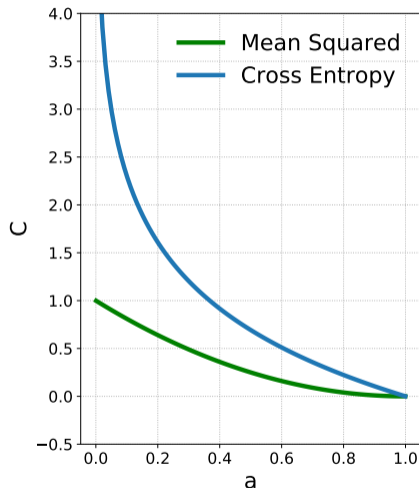
This is the activation function that is always used on the output layer when doing classification.

Other cost functions: cross entropy

The most-commonly used cost function for categorical output data is cross entropy:

$$C = -\frac{1}{n} \sum_i^n [y_i \log(a_i) + (1 - y_i) \log(1 - a_i)]$$

- The above equation is for 2 categories, but it easily generalizes to more.
- Good: the gradient of cross entropy is directly proportional to the error; learning is faster than with mean squared error.
- Because $0 \leq a \leq 1$, this is always used with the softmax activation function as output.
- $y = 1$ in the example on the right.



Our Keras network, revisited

What's our new strategy for our MNIST neural network?

- Change our hidden layer activation function to tanh.
- Change our output layer activation function to softmax.
- Use the cross-entropy cost function.
- Use the Adam minimization algorithm.

Using regular gradient descent would also probably work. Using the rectifier linear unit activation function on the hidden layer is also an option.

```
# model2.py
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl

def build_model(numnodes):

    model = km.Sequential()

    model.add(kl.Dense(numnodes,
                        input_dim = 784,
                        activation = 'tanh'))

    model.add(kl.Dense(10,
                        activation = 'softmax'))

    return model
```

Our Keras network, revisited, continued

```
In [18]: import model2
In [19]:
In [19]: model = model2.build_model(30)
In [20]:
In [20]: model.compile(loss = "categorical_crossentropy", optimizer = "adam",
...:                 metrics = ['accuracy'])
In [21]:
In [21]: fit = model.fit(x_train, y_train, epochs = 100, batch_size = 128, verbose = 2)
Epoch 1/100
469/469 [=====] - 1s - loss: 0.0688 - acc: 0.4576
Epoch 2/100
469/469 [=====] - 1s - loss: 0.3661 - acc: 0.7246
:
Epoch 100/100
469/469 [=====] - 1s - loss: 0.0103 - acc: 0.9338
In [22]:
```

Our Keras network, revisited, continued more

Now check against the test data.

93%! Better!

```
In [22]:  
-----  
In [22]: score = model.evaluate(x_test, y_test)  
-----  
In [23]:  
-----  
In [23]: score  
Out[23]: [0.010993927612225524, 0.9294999999999999]  
-----  
In [24]:
```

The next steps, an aside

There are a lot of things we can tweak to make the network do better on the testing data (number of layers, neurons, activation functions, etc.). How do we know what to do?

- In many ways, implementing a network is an art.
- Certain forms and functions and parameters are known to lead to certain types of behaviour, and thus are used in certain situations.
- Choosing the correct values of parameters can often seem like a matter of trial-and-error.
- And choosing the correct activation functions, number of nodes, can also seem like trial-and-error.
- But there are more-sophisticated ways of finding the optimum parameter choices. We won't delve into this today.

Practice is often needed to know how to approach various types of problems. Consult your colleagues, and the literature.

Other topics

This workshop is short. There is not time to cover every topic. Some topics you should look into further, if you're going to use NNs in your research:

- preprocessing data: remove unnecessary degrees of freedom, scale and centre the data.
- parameter initialization: how the weights and biases are initialized sometimes matters.
- activation functions: there are several activation functions which are used in specific areas of neural networks. Learn which ones are used in your field.
- more cost functions: there are other cost functions which are used in specific applications.
- training failures: the disappearing gradient problem, the exploding gradient problem.

But at this point we have covered enough of the very basics to get you started.

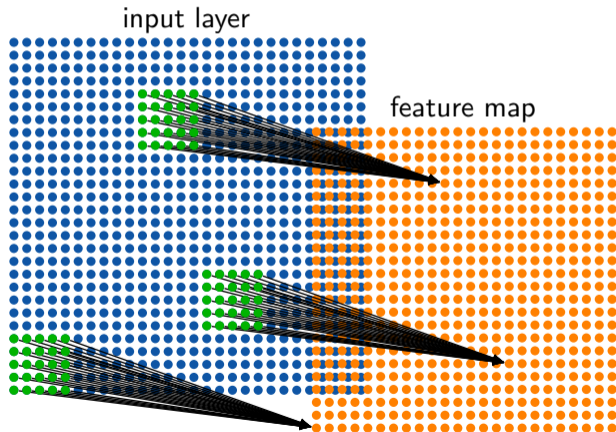
What next?

What we've done so far is pretty good, but it's not going to scale well.

- These are small images, and only black-and-white.
- Imagine we had a more-typical image size (200×200) and 3 colours? Now we're up to 120,000 input parameters.
- We need an approach that is more efficient.
- A good place to start would be an approach that doesn't throw away all of the spatial information.
- The data is (28×28), not (1×784).
- We should redesign our network to account for the spatial information. How do we do that?
- The first step called a Convolutional Layer. This is the bread-and-butter of all neural network image analysis.

Convolutional layers: feature maps

Create a set of neurons that, instead of using all of the data as input, only takes input from a small area of the image. This set of neurons is called a "feature map".



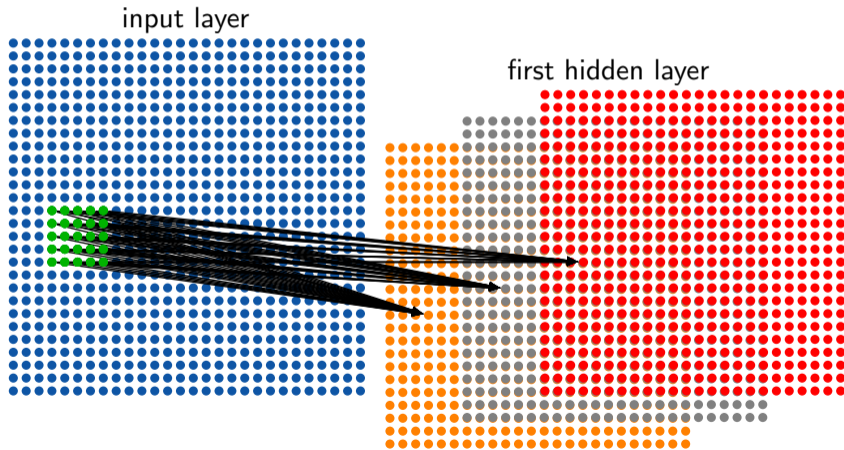
Feature maps

Some notes about feature maps.

- Notice that the feature map is smaller (24×24) than the input layer (28×28).
- The size of the feature map is partially set by the 'stride', meaning the number of pixels we shift to use as the input to the next neuron. In this case I've used a stride of 1.
- The **weights and biases are shared by all the neurons in the feature map**.
- Why? The goal is to train the feature map to recognize a single feature in the input, regardless of its location in the image.
- Consequently, it makes no sense to have a single feature map as the first hidden layer. Rather, multiple feature maps are used as the first layer.
- Feature maps are also called "filters" and "kernels".

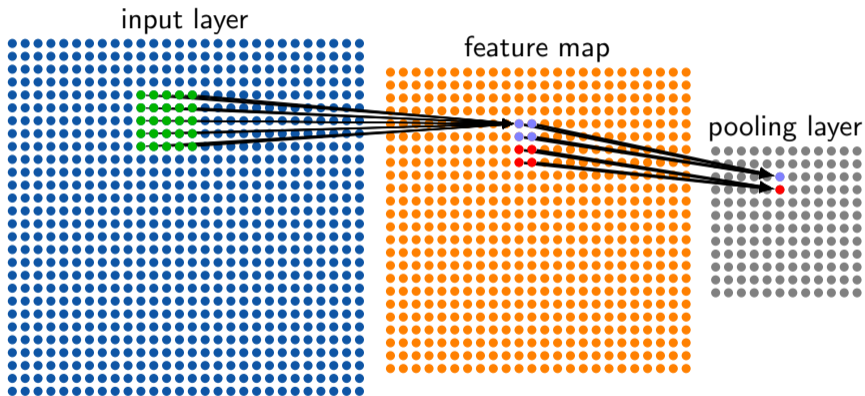
Convolutional layers, continued more

The first hidden layer, a "convolutional layer", consists of multiple feature maps. The same inputs are fed to the neurons in different feature maps.



Pooling layers

Each feature map is often followed by a "pooling layer".



In this case, 2×2 feature map neurons are mapped to a single pooling layer neuron.

Pooling layers, continued

Some notes about pooling layers.

- The purpose of a pooling layer is to reduce the size of the data, and thus the number of free parameters in the network.
- The reduction in data also helps with over-fitting.
- Rather than use one of the activation functions we've already discussed, pooling layers use other functions.
- These functions do not have free parameters (weights and biases) in them which need to be fit. They are merely functions which operate on the input.
- The most common function used is 'max', simply taking the maximum input value.
- Other functions are sometimes used, average pooling, L2-norm pooling.

Re-prepping the data

First let us gather the data again, so that the data is once again 28 x 28 pixels.

We also confirm that the target data is one-hot-encoded.

```
In [24]:
```

```
In [24]: x_train, x_test, y_train, y_test = get_mnist.get_data()
```

```
In [25]:
```

```
In [25]: x_train.shape
```

```
Out[25]: (60000, 28, 28)
```

```
In [26]:
```

```
In [26]: y_train.shape
```

```
Out[26]: (60000, 10)
```

```
In [27]:
```

2D data formatting

Generally, 2D images are actually 3D, to deal with colours. We need to add this third dimension to the data, since the Convolutional layers are expecting it.

Where the third dimension (the "channels") shows up in the dimensionality is given by the "image_data_format" function, which is part of keras.backend.

Having the channels at the end seems to have become standard.

```
In [27]:
```

```
In [27]: import tensorflow.keras.backend as K
```

```
In [28]:
```

```
In [28]: K.image_data_format()
```

```
Out[28]: 'channels_last'
```

```
In [29]:
```

```
In [29]: x_train.shape
```

```
Out[29]: (60000, 28, 28)
```

```
In [30]:
```

```
In [30]: x_train = x_train.reshape(60000, 28, 28, 1)
```

```
In [31]: x_test = x_test.reshape(10000, 28, 28, 1)
```

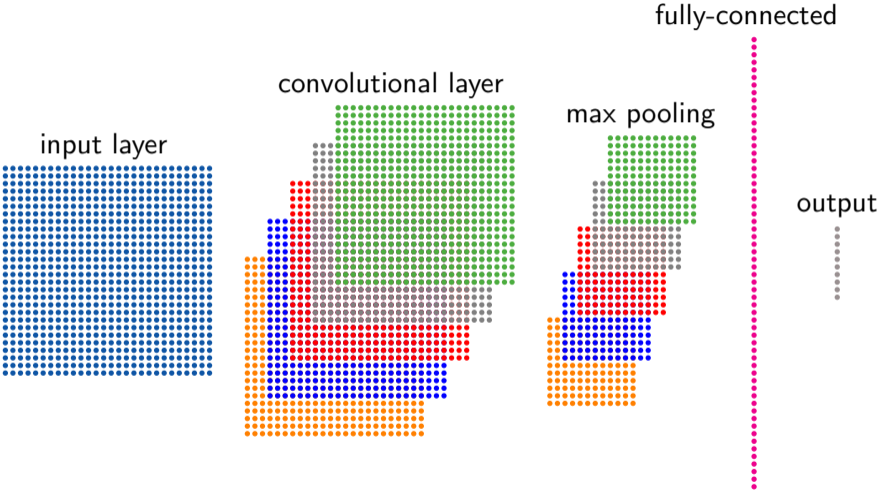
```
In [32]:
```

```
In [32]: x_train.shape
```

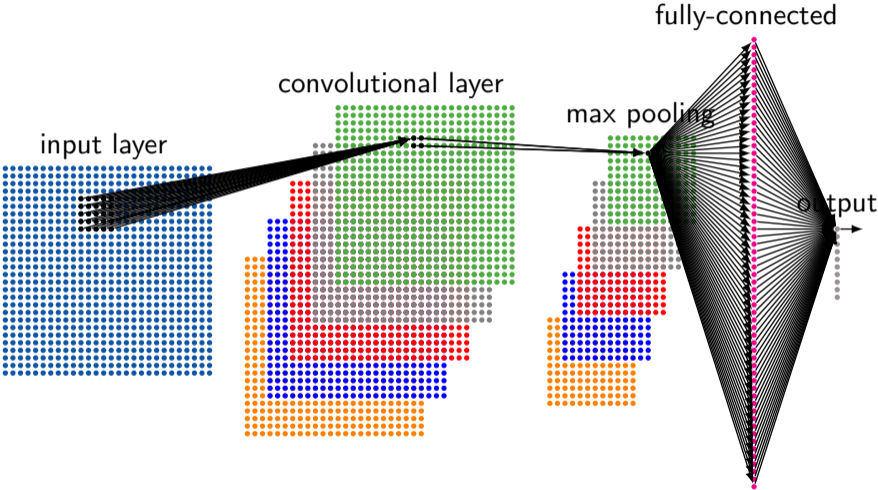
```
Out[32]: (60000, 28, 28, 1)
```

```
In [33]:
```

Our network, latest version



Our network, latest version



Our network revisited again

We specify the 'kernel' of the convolutional layer, as well as the number of feature maps. A default 'stride' of 1 is used.

We also specify the pooling layer's 'pool size', which is like the kernel for the convolutional layer.

The "Flatten" layer converts the 2D output to 1D, so that the fully-connected layer can handle it.

```
# model3.py
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl

def build_model(numfm, numnodes):

    model = km.Sequential()
    model.add(kl.Conv2D(numfm, kernel_size = (5, 5),
                        input_shape = (28, 28, 1),
                        activation = 'relu'))
    model.add(kl.MaxPooling2D(pool_size = (2, 2),
                               strides = (2, 2)))

    model.add(kl.Flatten())
    model.add(kl.Dense(numnodes, activation = 'tanh'))
    model.add(kl.Dense(10, activation = 'softmax'))

    return model
```

Our network revisited again, continued

```
In [33]: import model3
```

```
In [34]: model = model3.build_model(20, 100)
```

```
In [35]: model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 20)	520
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 20)	0
flatten_1 (Flatten)	(None, 2880)	0
dense_1 (Dense)	(None, 100)	288100
dense_2 (Dense)	(None, 10)	1010

Total params: 289,630
Trainable params: 289,630
Non-trainable params: 0



Our network revisited again, more

```
In [36]:
```

```
In [36]: model.compile(loss = "categorical_crossentropy", optimizer = "sgd",  
...:                 metrics = ['accuracy'])
```

```
In [37]:
```

```
In [37]: fit = model.fit(x_train, y_train, epochs = 30, batch_size = 128, verbose = 2)
```

```
Epoch 1/30
```

```
25s - loss: 0.4992 - acc: 0.8638
```

```
Epoch 2/30
```

```
25s - loss: 0.1973 - acc: 0.9466
```

```
⋮
```

```
Epoch 30/30
```

```
24s - loss: 0.0321 - acc: 0.9911
```

```
In [38]:
```

Our network revisited again, some more

Now check against the test data.

98.35%! Only 165 / 10000 wrong!

Not bad!

You can improve this even more by adding another convolutional layer-max pooling layer after the first pair.

```
In [38]:
```

```
score = model.evaluate(x_test, y_test)
```

```
In [39]:
```

```
score
```

```
Out[39]: [0.053592409740015862, 0.983500000000000004]
```

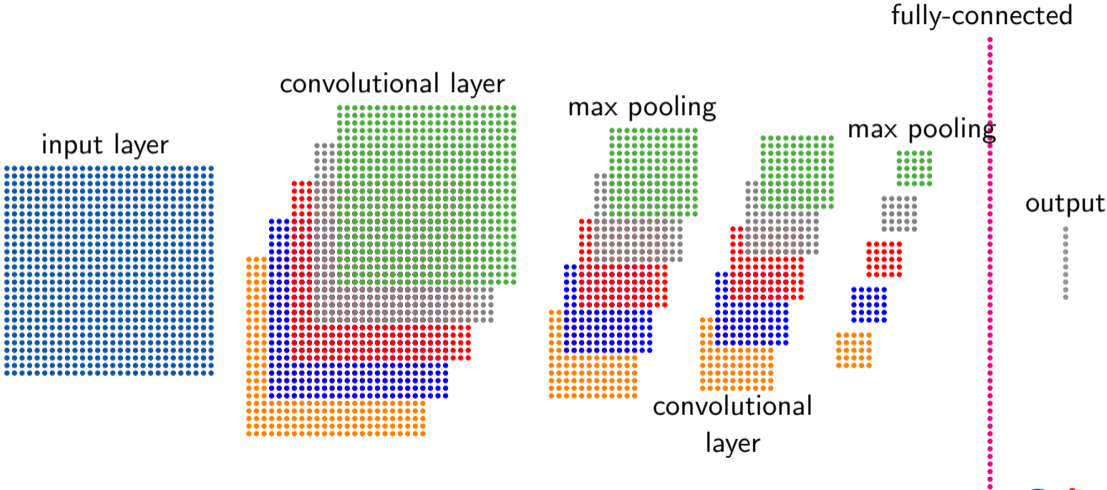
```
In [40]:
```

Notes on Convolutional Networks

The previous network is called a Convolutional Neural Network (CNN), and is quite common in image analysis.

- Often more than a single convolutional layer-pooling layer combination will be used.
- This will lead to improved performance, in this case.
- In practice people come up with all manner of combinations of convolutional, pooling and fully-connected layers in their networks.
- Trial-and-error is a good starting point. Again, hyperparameter optimization techniques should be considered. Reviewing the literature, you will find themes, but also much art.

Our latest network, version 2



Our network, version 2, continued

```
# model4.py
import tensorflow.keras.models as km, tensorflow.keras.layers as kl

def build_model(numfm, numnodes):

    model = km.Sequential()
    model.add(kl.Conv2D(numfm, kernel_size = (5, 5), input_shape = (28, 28, 1),
                        activation = 'relu'))
    model.add(kl.MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

    model.add(kl.Conv2D(2 * numfm, kernel_size = (3, 3), activation = 'relu'))
    model.add(kl.MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))

    model.add(kl.Flatten())
    model.add(kl.Dense(numnodes, activation = 'tanh'))
    model.add(kl.Dense(10, activation = 'softmax'))

    return model
```

Our latest network, version 2, summary

```
In [40]: import model4
```

```
In [41]: model = model4.build_model(20, 100)
```

```
In [42]: model.summary()
```

```
-----
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 20)	520
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 20)	0
conv2d_2 (Conv2D)	(None, 10, 10, 40)	7240
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 40)	0
flatten_1 (Flatten)	(None, 1000)	0
dense_1 (Dense)	(None, 100)	100100
dense_2 (Dense)	(None, 10)	1010

```
-----
```

Total params: 108,870
Trainable params: 108,870
Non-trainable params: 0

Understanding successive Convolutional Layers

As you noticed, we previously had to change the input data to have dimension (28, 28, 1) rather than (28, 28).

- All convolutional layers assume that its input has a 'channel', which is a third dimension.
- For colour images the three colours of the image (RGB) are the three channels.
- The channel is usually put in the third dimension, though sometimes in the first.
- When the output of one convolutional layer is fed into another layer, the feature maps are the channels.
- How are the channels read by the feature maps?
- If the filter size is, say (3 x 3), and there 20 channels, as in this example, then the number of weights in a given feature map will be (3 x 3) x 20, plus 1 bias.
- Thus, the number of trainable parameters in the second convolutional layer is $((3 \times 3) \times 20) + 1 \times 40 = 7240$.

Our latest network, version 2, more

```
In [43]:
```

```
In [43]: model.compile(loss = "categorical_crossentropy", optimizer = "sgd",  
...:                 metrics = ['accuracy'])
```

```
In [44]:
```

```
In [44]: fit = model.fit(x_train, y_train, epochs = 100, batch_size = 128, verbose = 2)
```

```
Epoch 1/100
```

```
33s - loss: 0.7378 - acc: 0.7966
```

```
Epoch 2/100
```

```
33s - loss: 0.2010 - acc: 0.9486
```

```
⋮
```

```
Epoch 99/100
```

```
33s - loss: 0.0028 - acc: 0.9996
```

```
Epoch 100/100
```

```
32s - loss: 0.0028 - acc: 0.9996
```

```
In [45]:
```

Our latest network, version 2, even more

Now check against the test data.

99.1%! Only 88 / 10000 wrong! Not bad!

```
In [45]:
```

```
score = model.evaluate(x_test, y_test)
```

```
In [46]:
```

```
score
```

```
Out[46]: [0.028576645734044722, 0.9911999999999997]
```

```
In [47]:
```

Using GPUs

An important note. Graphical Processing Units (GPUs) are particularly good at running NN-training calculations.

data size	CPU only		CPU-GPU	
	epoch time	total time	epoch time	total time
50000	41 s	21 min 4 s	4 s	2 min 43s
250000	198 s	100 min	26 s	15 min

These numbers are for today's first convolutional network. These were run on a Power 8 CPU, and a P100 GPU.

Multi-GPU functionality is available in Keras running on TensorFlow, though it can be a bit of work to set up.

Deep Learning

You've probably heard the term. What is Deep Learning?

- Quite simply: a neural network with many hidden layers.
- Up until the mid-2000s neural network research was dominated by "shallow" networks, networks with only 1 or 2 hidden layers.
- The breakthrough came in discovering that it was practical to train networks with a larger number of hidden layers.
- But it only became practical with the advent of sufficient computing power (GPUs) and easily-accessible huge data sets.
- State-of-the-art networks today can contain dozens of layers.

Other neural network results

Neural networks have been applied to all manner of situations. There are too many other areas to cover in detail. We will finish the workshop by reviewing some areas which have been tackled, and the current state of the art.

- Image classification.
- Object detection.
- Image segmentation.
- Generative networks.
- Style transfer.
- Text generation.

There are many many other applications which have been developed.

Other neural network results: image classification

Neural networks have revolutionized image classification.

- This is the problem of: given a photo, what is in it?
- This is similar to working with MNIST data, but there are important differences:
 - ▶ the images are much bigger,
 - ▶ the images are colour,
 - ▶ there are thousands of categories.
- The networks which solve these problems are deep, and often contain a number of architectural innovations.

Early in the deep-learning revolution this was a very active area of research, strongly driven by the ILSVRC competition.

Image classification, top-5 example



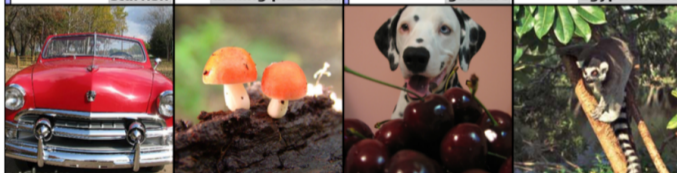
mite

container ship

motor scooter

leopard

mite	container ship	motor scooter	leopard
black widow	lifeboat	go-kart	jaguar
cockroach	amphibian	moped	cheetah
tick	fireboat	bumper car	snow leopard
starfish	drilling platform	golfcart	Egyptian cat



grille

mushroom

cherry

Madagascar cat

convertible	agaric	dalmatian	squirrel monkey
grille	mushroom	grape	spider monkey
pickup	jelly fungus	elderberry	titi
beach wagon	gill fungus	ffordshire bullterrier	indri
fire engine	dead-man's-fingers	currant	howler monkey

ILSVRC

From 2010 - 2017 the most pre-eminent image classification competition was ILSVRC.

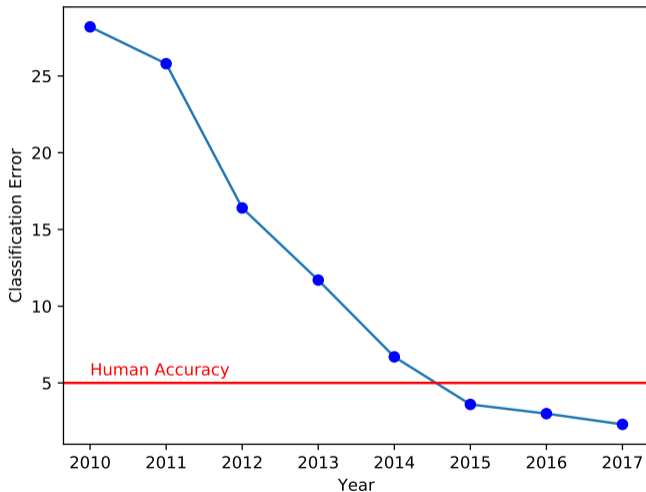
- Stands for ImageNet Large Scale Visual Recognition Challenge (the "ImageNet Competition").
- Millions of training images, 1000 categories.
- More-recent competitions have also had object-detection challenges, which involves also locating objects within images, and now also within videos.
- The classification competitions allowed the top-5 classifications to be submitted for any given image, along with the associated bounding boxes for each object.
- In 2017, 29 of 38 teams had greater than 95% accuracy. This is probably why the challenge does not appear to be running anymore.

This competition saw significant innovations in neural network architectures.

ILSVRC, continued

ILSVRC winners introduced new ideas to image classification.

- AlexNet (2012): first network to use successive convolutional layers in the competition.
- GoogLeNet (2014): introduced the "Inception Module".
- ResNet (2015): introduced the "ResNet Module".



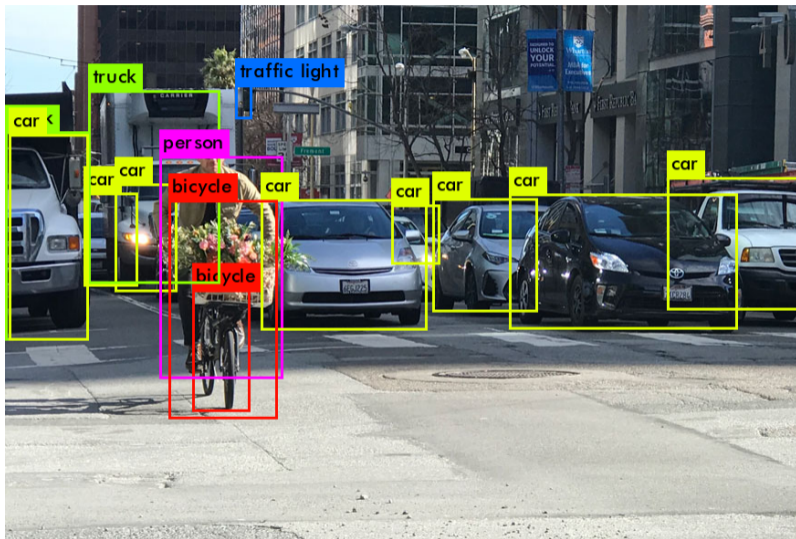
Other neural network results: object detection

The use of neural networks in object detection is an active area of research.

- This is the problem of: given a photo, what is in it, and WHERE is it?
- This is more complicated than simple image classification, obviously:
 - ▶ the images usually have multiple objects within them,
 - ▶ a bounding box must be put around each object,
 - ▶ each object must simultaneously be classified,
 - ▶ sometimes the whole object is not visible in the image.
- The networks which solve these problems are deep, and often contain a number of architectural innovations.
- The networks are also optimized for speed, since real-time object detection is necessary for self-driving cars.

This is not a solved problem on the most-difficult data sets.

Object detection, example



Other neural network results: image segmentation

Obviously, not all applications of object detection have their needs sufficiently met with bounding boxes. This leads to one of the hardest image-analysis problems: image segmentation.

- This is the problem of: given a photo, what is in it, and draw an outline around the boundaries of the objects.
- This is more complicated than simple object detection, obviously:
 - ▶ the images usually have multiple objects within them,
 - ▶ the object's 'mask' must outline each object,
 - ▶ each object must simultaneously be classified,
 - ▶ distinct objects, even of the same type, must be uniquely identified.
- The use of such networks in medicine, especially radiology, is a very active area of research.

These networks are moving toward the localizing of tumours, cancer, etc. in diagnostic images.

Image segmentation, example

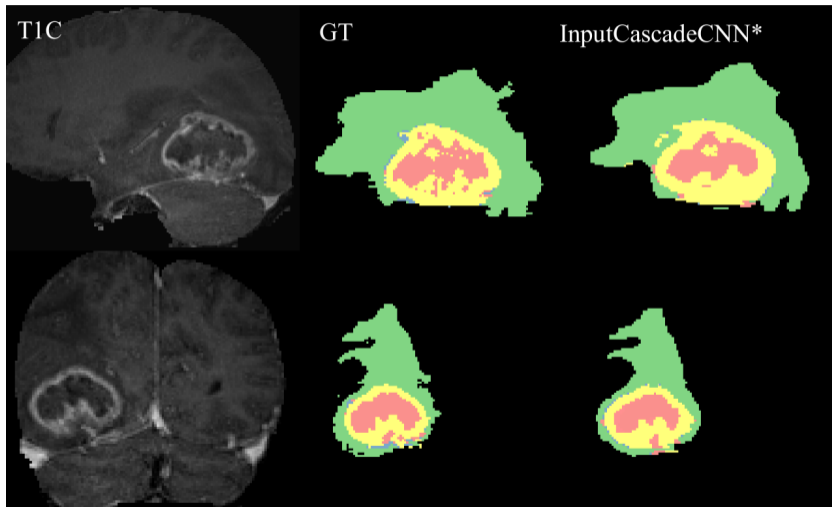


Image stolen from <https://arxiv.org/abs/1505.03540>

Discriminative versus generative networks

Let's examine a distinction we haven't yet made: discriminative versus generative networks.

- A discriminative network is trained to detect whether some input data is a member of a given class. Examples include the standard networks we've come to know and love, such as fully-connected and CNNs.
- In probabilistic terms, given the input data \mathbf{x} , and a desired label \mathbf{y} , the discriminative network calculates the conditional probability $P(\mathbf{y}|\mathbf{x})$.
- In contrast, a generative network is trained to calculate the probability of the data directly, $P(\mathbf{x})$.
- Once we have $P(\mathbf{x})$, we can sample from this distribution to create new data.

The ability to generate fake, authentic looking data has a number of applications.

Generative networks

There are several types of generative networks you may run into.

- PixelCNN: an auto-regressive model, the conditional distribution of each pixel is modeled given the left and above pixels.
- Variational Autoencoders (2014): one network (the encoder) casts the input data into a lower-dimensional representation; a second network (the decoder) reconstructs the input from the low-D representation.
- Generative Adversarial networks (2014): two networks are trained simultaneously, one to generate fake data, and one to identify the fake data, when compared to real data.
- Boltzmann Machines, Fully Visible Belief Networks, Generative Stochastic Networks, and others.
- Diffusion models: noise is gradually 'denoised' to create images.

Today we won't go into these in detail, but you need to know they exist.

GANs can do amazing things



<https://thispersondoesnotexist.com>

Style transfer (2015)

Let's look at an interesting application of neural networks.

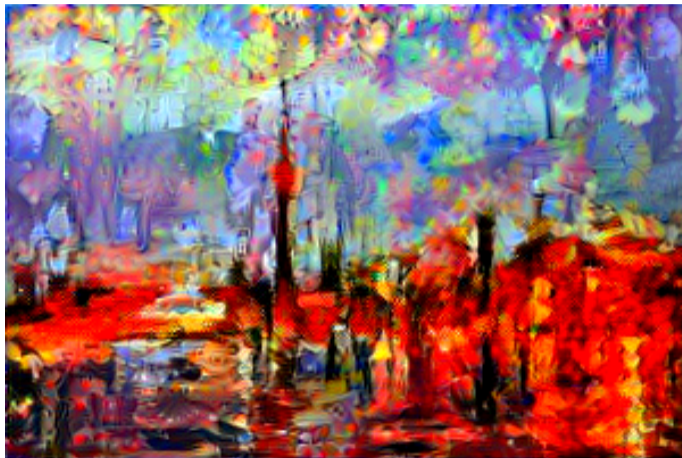
This application leverages Convolutional Neural Network's strengths in visual pattern recognition and object localization.

Interestingly, CNNs can also distinguish between "content" of an image, and "style". This can be used to impose the style of one image onto another. This is known as "style transfer".

Style transfer, an example



Style transfer, an example, continued



Style transfer, continued

So how does it work? The technique proceeds as follows.

- Start with two images, the 'content' image (the photo), and the 'style' image (the painting).
- We create a loss function, which measures how close the 'content' of the generated image is to the 'content' image.
- We create a second loss function, which measures how close the 'style' of the generated image is to the 'style' image.
- We combine these two loss functions into a single loss function.
- We use scipy, rather than Keras, to minimize the combined loss function, in the process creating our generated image.
- Note that the minimization is done, not by adjusting the weights and biases of a neural network, but rather by choosing the input to the network which minimizes the loss function.

Image transformation

This is part of a broader area of research known as "image transformation". In this field, CNNs are used to perform a number of different applications:

- super-resolution,
- colourization,
- surface-normal prediction,
- depth prediction,
- style transfer.

This is an active area of research.

Style transfer + GANs



Text generation

You've heard of ChatGPT by now, probably. What is that?

- ChatGPT, and its predecessors (GPT, GPT-2, GPT-3), are in the 'Transformer' family of neural networks.
- These are, more generally, sequence-to-sequence networks.
- That means, given a sequence as input, what is the output sequence?
- This has obvious applications in text generation problems, but also summarization and translation.
- These networks are also used in DNA analysis.

Any time you have a sequence as an input a Transformer is probably the type of neural network that you want to use.

Linky goodness: CNNs

Convolutional neural networks:

- <http://scs.ryerson.ca/~aharley/vis/conv/flat.html>
- <http://www.cs.utoronto.ca/~fidler/teaching/2015/CSC2523.html>
- <https://cs231n.github.io/convolutional-networks>
- <http://deeplearning.net/tutorial/lenet.html>
- [https://medium.com/technologymadeeasy/
the-best-explanation-of-convolutional-neural-networks-on-the-internet-fb](https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fb)

Linky goodness: image classification

ILSVRC:

- <http://www.image-net.org/challenges/LSVRC>

Image classification networks:

- AlexNet: <http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf>
- ResNet: <https://arxiv.org/abs/1512.03385>
- GoogLeNet: <https://arxiv.org/abs/1409.4842>
- Xception: <https://arxiv.org/abs/1610.02357>

Linky goodness: GANs

GANs:

- <https://arxiv.org/abs/1701.00160>
- <https://blog.openai.com/generative-models>
- <https://deephunt.in/the-gan-zoo-79597dc8c347>
- <http://arxiv.org/abs/1511.06434>
- <https://medium.com/towards-data-science/gan-by-example-using-keras-on-tensorflow-backend-1a6d515a60d0>
- <https://arxiv.org/abs/1606.03498>

WGAN:

- <https://arxiv.org/abs/1701.07875> (original WGAN paper)
- <https://arxiv.org/abs/1704.00028>
- <http://www.alexirpan.com/2017/02/22/wasserstein-gan.html>

Linky goodness: style transfer

Style transfer:

- <https://arxiv.org/abs/1508.06576> (the original paper)
- <https://medium.com/mlreview/making-ai-art-with-style-transfer-using-keras-8bb5fa44b216>
- <https://github.com/titu1994/Neural-Style-Transfer>
- <https://chrisrodley.com/2017/06/19/dinosaur-flowers>

Photo style transfer:

- <https://arxiv.org/abs/1703.07511>