Compute Ontario Summer School: Artificial Neural Networks: convolutional neural networks

Erik Spence

SciNet HPC Consortium

9 June 2025

Today's code and slides

You can get the slides and code for today's class at the 2025 Compute Ontario Summer School web site:

`https://training.computeontario.ca/courses/course/view.php?id=130`

Scroll down to this session. You can find the slides and code there.

The code for this session is in "nn2_code.tar.gz".

Today’s class

This afternoon we will cover the following topics:

- Review some of the available neural network frameworks,
- Redo the MNIST example using Keras,
- Dropout,
- Different activation functions, cost functions,
- Updated Keras network,
- Hands-on,
- Convolutional neural networks,
- Feature maps, pooling layers,
- The latest (and best) versions of our MNIST neural network,
- Hands-on.

We will be using Keras for the rest of today’s class. Install it by installing Tensorflow.

A review of last class

Recall what we did this morning.

- We built a neural network, consisting of three layers: an input layer, a single hidden layer, and an output layer.
- We defined a cost function, which measured the inaccuracy of the neural network's predictions.
- We used backpropagation to calculate the derivatives of the cost function with respect to the weights and biases.
- Using gradient descent, we trained the network to identify images from the MNIST data set.

However, we built all the parts of the network by hand. There are better ways to do this.

Neural network frameworks

Now that we have a sense of how neural networks work, we're ready to switch gears and use a 'framework'. Why would we do that?

- Coding your own networks from scratch can be a bit of work. (Though it's easier and cleaner if you use classes.)
- Neural network (NN) frameworks have been specifically designed to solve NN problems.
- Python, of course, is not a high-performance language.
- The neural networks which are built using frameworks are compiled before being used, thus being much faster than Python.
- The NN frameworks are also designed to use GPUs, which make things significantly faster than just using CPUs.

The training of neural networks is particularly well suited to GPUs.

TensorFlow

TensorFlow is Google's NN framework.

- Released as open source in November 2015.
- The second-generation machine-learning framework developed internally at Google, successor to DistBelief.
- More flexible than some other neural network frameworks.
- Capable of running on multiple cores and GPUs.
- Provides APIs for Python, C++, Java and other languages.
- Used to be quite a challenge to learn (many ways to do the same thing).
- With Tensorflow 2.0, Keras has become the main high-level API. A significant consolidation of the API was performed.
- Will be succeeded by JAX?

This framework is popular, though not necessarily the fastest.

JAX

JAX is a numpy-like machine learning framework.

- Released by Google as open source in 2018.
- Designed for high-performance numerical computing.
- Easily takes numpy commands and runs them on GPUs, using a just-in-time compiler.
- Very fast.
- A number of libraries have been built on top of JAX, to extend its capabilities: Flax, Equinox, RLax, jraph, and others.

This framework is growing in popularity.

PyTorch

Another framework used for neural networks is PyTorch.

- Based on Torch, which was first released in 2002. Quite mature at this point.
- PyTorch was released by Facebook in January 2018. This is now the most-commonly used interface to Torch, though there is also a C++ interface.
- PyTorch is more flexible than just NN. It is more of a generic scientific computing framework.
- Very strong on GPUs.
- Very fast. Often the fastest depending on the problem being considered.
- Used and maintained by Facebook, Twitter and other high-profile companies.
- PyTorch Lightning was released recently. This gives a more Keras-like interface to PyTorch, making it easier to use.

This framework is the other major player, other than Tensorflow.

Keras

We will use Keras for the rest of this course.

- 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

Because it is so commonly used, the MNIST data set is built into most NN frameworks.

Keras is no different, so we'll just grab it from Keras directly.

```
In [1]:
In [1]: from keras.datasets import mnist
In [2]:
In [2]: (x_train, y_train), (x_test, y_test) =
                    mnist.load_data()
In [3]:
In [3]: x_train.shape
Out[3]: (60000, 28, 28)
In [4]:
```

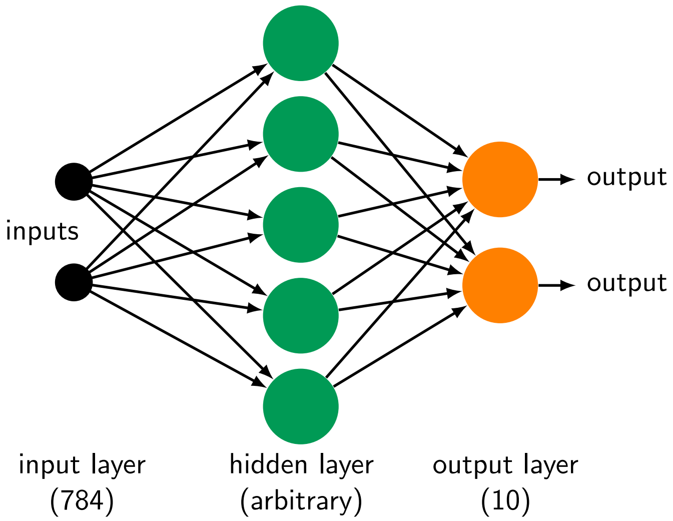
Prepping the data

As with last time, we need the data in a specific format:

- Instead of 28 x 28, we flatten the data into a 784-element vector.
- We only take the first 500 data points for training, to be consistent with last class.
- The labels must be changed to a categorical format (one-hot encoding).

```
In [4]:
In [4]: import keras.utils as ku
In [5]:
In [5]: x_train2 = x_train[0:500, :, :].reshape(500, 784)
In [6]: x_test2 = x_test[0:100, :, :].reshape(100, 784)
In [7]:
In [7]: y_train2 = ku.to_categorical(y_train[0:500], 10)
In [8]: y_test2 = ku.to_categorical(y_test[0:100], 10)
In [9]:
In [9]: y_train2.shape
Out[9]: (500, 10)
In [10]:
```

Our neural network



Our network using Keras

Let us re-implement our second network using Keras.

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

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

def get_model(numnodes):
  model = km.Sequential()
  model.add(kl.Dense(numnodes, input_dim = 784,
    activation = 'sigmoid', name = 'hidden'))
  model.add(kl.Dense(10, name = 'output',
    activation = 'sigmoid'))
  return model
```

```
In [10]: import model1 as m1

In [11]: model = m1.get_model(30)

In [12]: model.output_shape
Out[12]: (None, 10)

In [13]:
```

Our network using Keras, continued

```
In [13]:
```

```
In [13]: model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
hidden (Dense)               (None, 30)                23550
_________________________________________________________________
output (Dense)               (None, 10)                310
=================================================================
Total params:  23,860
Trainable params:  23,860
Non-trainable params:  0
_________________________________________________________________
```

```
In [14]:
```

A different optimization flag

We will set the optimization flag to "sgd".

- This stands for "Stochastic Gradient Descent".
- In practice, regular gradient descent is never used, stochastic gradient descent is used instead, since it's so much faster.
- It is similar to regular gradient descent that we used previously.
  - Regular gradient descent is ridiculously slow on large amounts of data.
  - To speed things up, SGD uses a randomly-selected subset of the data (a "batch") to update the weights and biases.
  - This is repeated many times, using different batches, until all of the data has been used. This is called an "epoch".
- The only real advantage of regular gradient descent is that it's easier to code, which is why I used it this morning.
- There are many variations on SGD that are also used.

Our network using Keras, compilation

Some notes about the compilation of the model.

- We must specify the loss (cost) function with the "loss" argument.
- We must specify the optimization algorithm, using the "optimizer" flag.
- The optimizer can be generic ('sgd'), as in this example, or you can specify parameters using the optimizers in the keras.optimizers module.
- I sometimes specify the optimizer explicitly so that I can specify the value of η. Otherwise a default value is used.
- The 'metrics' argument is optional, but is needed if you want the accuracy to be printed.

We now have all the pieces to compile and train the model.

Our network using Keras, continued more

```
In [14]:
In [14]: model.compile(optimizer = 'sgd', metrics = ['accuracy'], loss = "mean_squared_error")
In [15]:
In [15]: fit = model.fit(x_train2, y_train2, epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
100/100 - 0s - 6ms/step - accuracy: 0.1170 - loss: 0.1963
Epoch 2/1000
100/100 - 0s - 2ms/step - accuracy: 0.1720 - loss: 0.1338
.
.
.
Epoch 999/1000
100/100 - 0s - 2ms/step - accuracy: 0.8440 - loss: 0.0394
Epoch 1000/1000
100/100 - 0s - 2ms/step - accuracy: 0.8440 - loss: 0.0394
In [16]:
```

Our network using Keras, continued even more

Now check against the test data.

We see the over-fitting rearing its head (84% versus 62%).

We can do better!

```
In [16]:

In [17]: score = model.evaluate(x_test2, y_test2)

In [18]:

In [18]: score
Out[18]: [0.056402873396873471, 0.62]

In [19]:
```

Over-fitting

Over-fitting occurs when a model is excessively fit to the noise in the training data, resulting in a model which does not generalize well to the test data.

It commonly occurs when there are too many free parameters (23,860) relative to the number of training data points (500).

This can be a serious issue with neural networks since it's trivially easy to have multitudes of weights and biases. How do we deal with this?

- More data! Either real (original), or artificially created.
- Regularization.
- Dropout.

The first is self-explanatory. We'll go over dropout today.

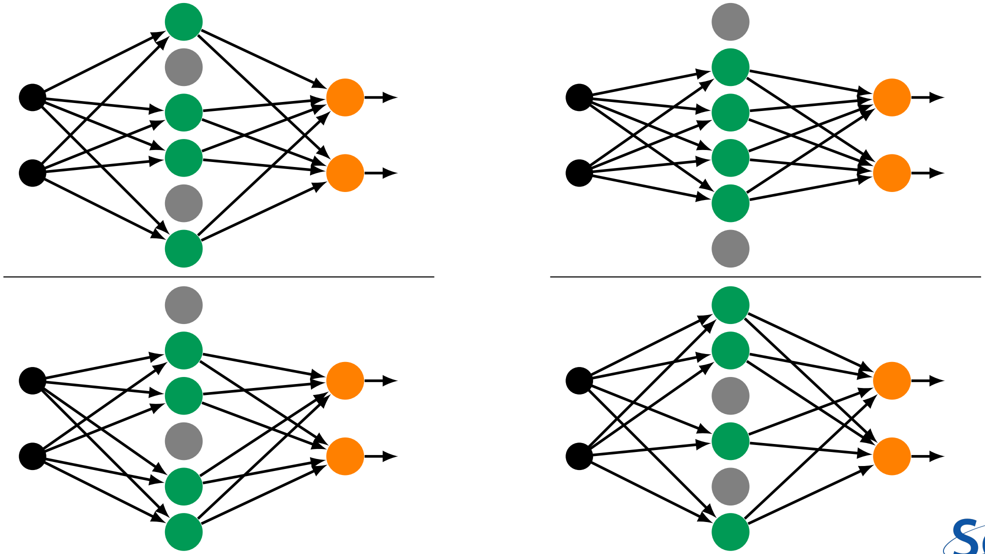
Dropout

Dropout is a uniquely-neural-network technique to prevent over-fitting.

- The principle is simple: randomly "drop out" neurons from the network during each batch of the stochastic gradient descent.
- Like regularization, this results in the network not putting too much importance on any given weight, since the weights keep randomly disappearing from the network.
- It can be thought of as averaging over several different-but-similar neural networks.
- Different fractions of different layers can be specified for dropout. A general rule of thumb is 30 - 50%.
- In the final model (after training):
  - the neurons in the dropout layer are no longer dropped out, and
  - the output from the neurons in the dropout layer is scaled by $(1 - p)$, where p is the probability of being dropped out.

This form of over-fitting control is quite common to encounter.

Dropout, visualized



Dropout using Keras

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

def get_model(numnodes, d_rate = 0.4):

  model = km.Sequential()

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

  model.add(kl.Dropout(d_rate,
    name = 'dropout'))

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

```
In [19]: import model2 as m2

In [20]: model2 = m2.get_model(30, d_rate = 0.2)

In [20]: model2.compile(loss = "mean_squared_error",
    ...:  optimizer = 'sgd', metrics = ['accuracy'])

In [21]: fit = model2.fit(x_train2, y_train2,
    ...:  epochs = 1000, batch_size = 5, verbose = 2)
Epoch 1/1000
100/100 - 0s - 1ms/step - accuracy: 0.1600 - loss: 0.1727
:
:
Epoch 1000/1000
100/100 - 0s - 1ms/step - accuracy:  0.6860 - loss: 0.0533

In [22]:

In [22]: model2.evaluate(x_test2, y_test2)
100/100 [==============================] - 0s 11us/step
Out[22]: [0.05041521415114403, 0.6899999976158142]

In [23]:
```

The next steps

We can do better. What's the plan? There are a few simple approaches:

- Use more data.
- 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 the over-fitting.

We'll try some of these next, but there are also some not-so-simple approaches:

- Completely overhaul the network strategy.

We'll take a look at this last approach after the break.

Other activation functions: relu

Two commonly-used functions:

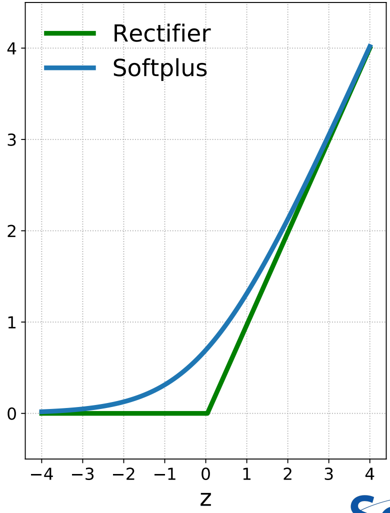
- Rectifier Linear Units (or RELUs):

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

- 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).

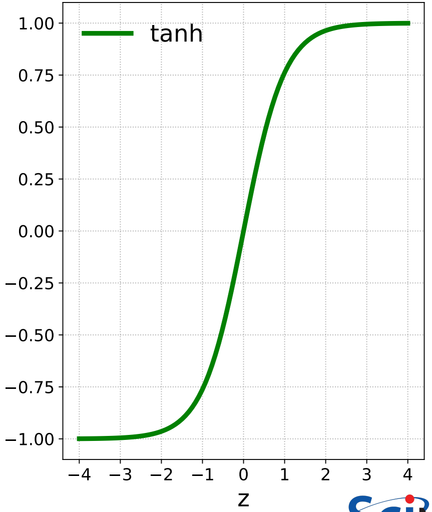


Other activation functions: tanh

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.
- Bad: the function saturates at large or small values of z.



Other activation functions: softmax

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

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

where j is the jth neuron, and the sum is over all N neurons in the layer. The advantage of this function is that it converts the output to a probability.

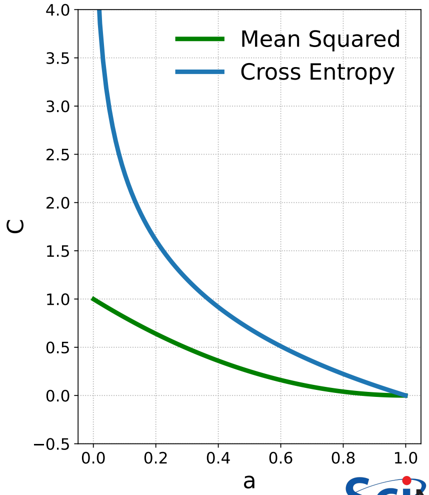
This is the activation function that is always used on the output layer when doing classification.

Other cost functions: cross entropy

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

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

- Good: the gradient of cross entropy is directly proportional to the error; learning is faster than with mean squared error.
- Because the output of the network, a, $0 \le a \le 1$, 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?

- Use all of the data.
- 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.
- We won't add regularization or dropout, as the data set is larger than the number of parameters in the model.

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

Our Keras network, revisited

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

def get_model(numnodes):

  model = km.Sequential()

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

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

  return model
```

```
In [23]:
In [23]: x_train.shape
Out[23]: (60000, 28, 28)
In [24]: x_test.shape
Out[24]: (10000, 28, 28)
In [25]:
In [25]: x_train = x_train.reshape(60000, 784)
In [26]: x_test = x_test.reshape(10000, 784)
In [27]:
In [27]: y_train = ku.to_categorical(y_train, 10)
In [28]: y_test = ku.to_categorical(y_test, 10)
In [29]:
```

Our Keras network revisited, continued

```
In [29]: import model3 as m3
```

```
In [30]: model3 = m3.get_model(30)
```

```
In [31]:
```

```
In [31]: model3.compile(loss = "categorical_crossentropy", optimizer = "adam",
...:           metrics = ['accuracy'])
```

```
In [32]:
```

```
In [32]: fit = model3.fit(x_train, y_train, epochs = 100, batch_size = 128, verbose = 2)
Epoch 1/100
469/469 - 2s - 4ms/step - accuracy:  0.6907 - loss:  0.9934
Epoch 2/100
469/469 - 2s - 2ms/step - accuracy:  0.8362 - loss:  0.5447
.
.
.
Epoch 100/100
469/469 - 2s - 2ms/step - accuracy:  0.9322 - loss:  0.2291
```

```
In [33]:
```

Our Keras network revisited, continued more

Now check against the test data.

93%! Better!

```
In [33]:
In [33]: score = model3.evaluate(x_test, y_test)
In [34]:
In [34]: score
Out[34]: [0.2364978790283203, 0.9286000132560730]
In [35]:
```

Hands-on

Let's take a break, and spend some time practicing. What things can we modify to explore what we've learned so far?

- Change the number of neurons in our hidden layer.
- Add more hidden layers.
- Change our activation functions.
- Add Dropout.
- Change our batch size.
- Change the number of training epochs.

How does changing these things affect the overall accuracy, training speed?

A review of what we've done this class

Our story so far:

- We've built a neural network, using Keras, consisting of an input layer, a hidden layer, and an output layer, to classify the MNIST data:
  - We used the tanh function as the activation for the hidden layer.
  - We used the softmax function as the activation for the output layer.
- We used cross entropy as the cost function, and Stochastic Gradient Descent as the optimization algorithm.
- We trained on the full data set, and achieved an accuracy of 93% on the test data.

We can do better, but it requires a different approach.

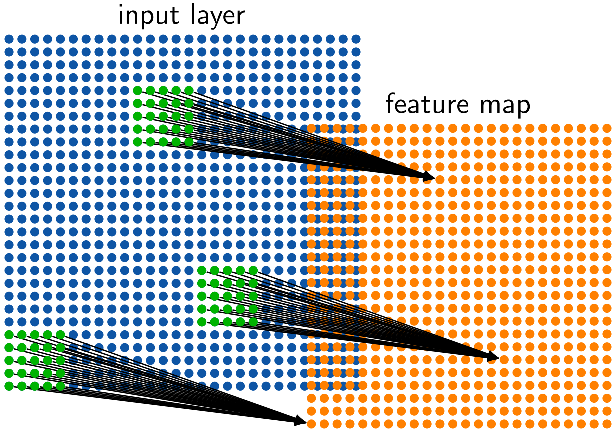
What next?

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 x 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 x 28, not 1 x 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".



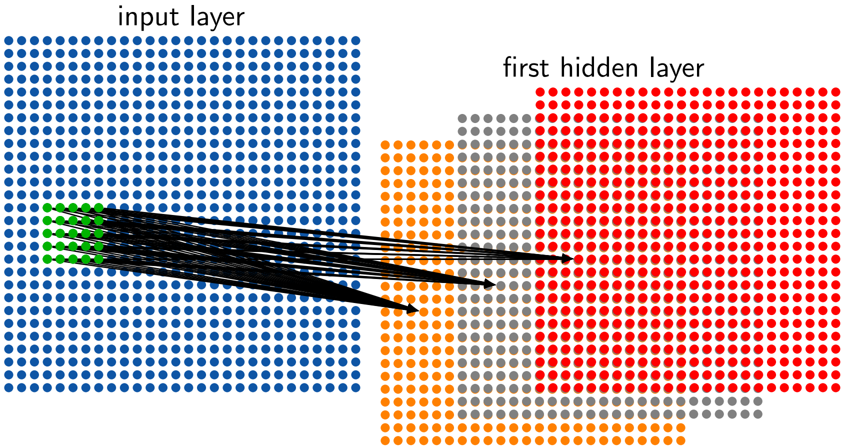
Feature maps

Some notes about feature maps.

- Notice that the feature map is smaller (24 x 24) than the input layer (28 x 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".

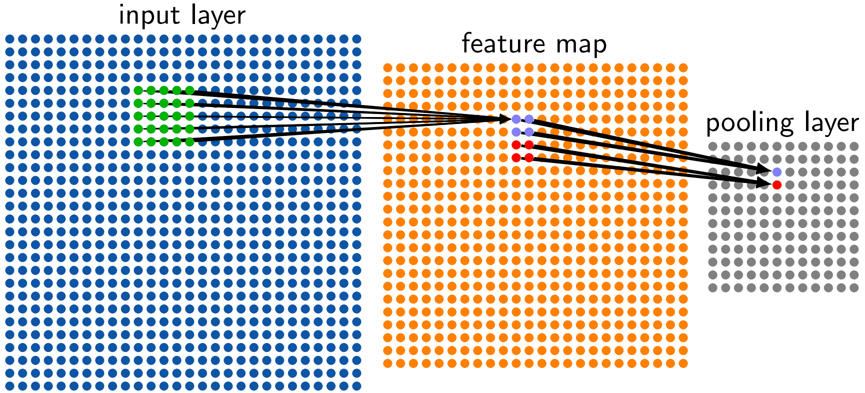
Convolutional layers, continued more

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 x 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 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.

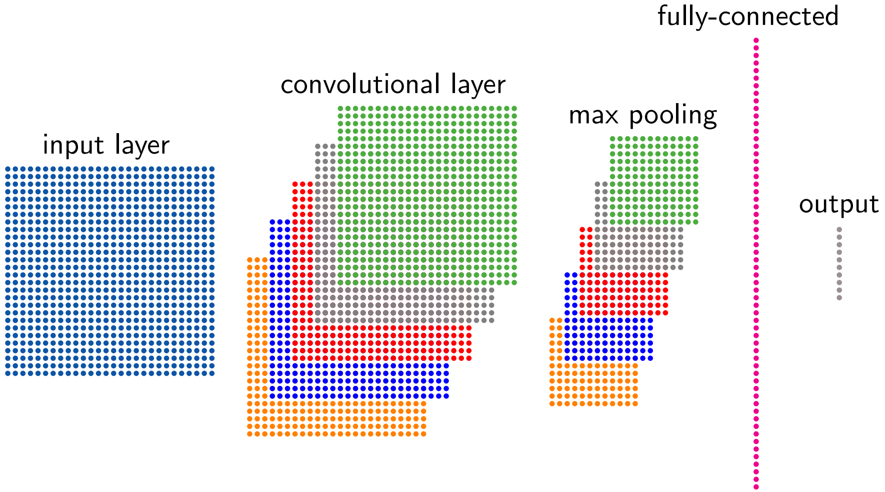
2D data formatting

```
In [35]: from keras.datasets import mnist

In [36]: import keras.utils as ku

In [36]: import keras.backend as K

In [37]:

In [37]: (x_train, y_train), (x_test, y_test) =
 ...:        mnist.load_data()

In [38]:

In [38]: x_train.shape
Out[38]: (60000, 28, 28)

In [39]:
```

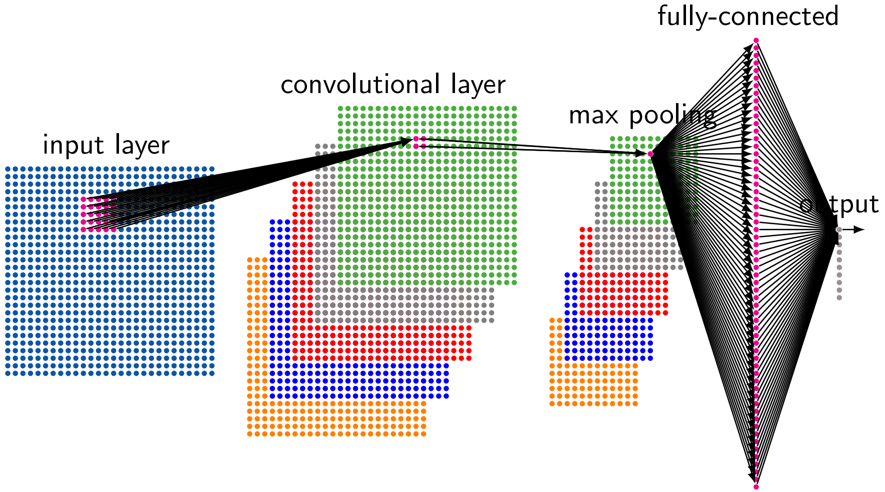
2D images are actually 3D, due to colours. The third dimension (the "channels") shows up in the dimension given by the "image_data_format" function.

```
In [39]: K.backend()
Out[39]: 'tensorflow'

In [40]:

In [40]: K.image_data_format()
Out[40]: 'channels_last'

In [41]:

In [41]: x_train = x_train.reshape(60000,28,28,1)

In [42]: x_test = x_test.reshape(10000,28,28,1)

In [43]:

In [43]: x_train.shape
Out[43]: (60000, 28, 28, 1)

In [44]:

In [44]: y_train = ku.to_categorical(y_train, 10)

In [45]: y_test = ku.to_categorical(y_test, 10)

In [46]:
```

Our network, latest version



Our network, latest version



Our network revisited again

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

def get_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
```

```
In [46]:
In [46]: import model4 as m4
In [47]:
In [47]: model = m4.get_model(20, 100)
In [48]:
```

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

Our network revisited again, continued

```
In [48]: model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 20)        520
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (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 [49]:
In [49]: model.compile(loss = "categorical_crossentropy", optimizer = "sgd",
...:             metrics = ['accuracy'])
In [50]:
In [50]: fit = model.fit(x_train, y_train, epochs = 30, batch_size = 128, verbose = 2)
Epoch 1/30
469/469 - 12s - 26ms/step - accuracy:  0.8638 - loss:  0.4992
Epoch 2/30
469/469 - 13s - 27ms/step - accuracy:  0.9466 - loss:  0.1973
⋮
Epoch 29/30
469/469 - 14s - 30ms/step - accuracy:  0.9917 - loss:  0.0313
Epoch 30/30
469/469 - 14s - 30ms/step - accuracy:  0.9925 - loss:  0.0285
In [51]:
```

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.

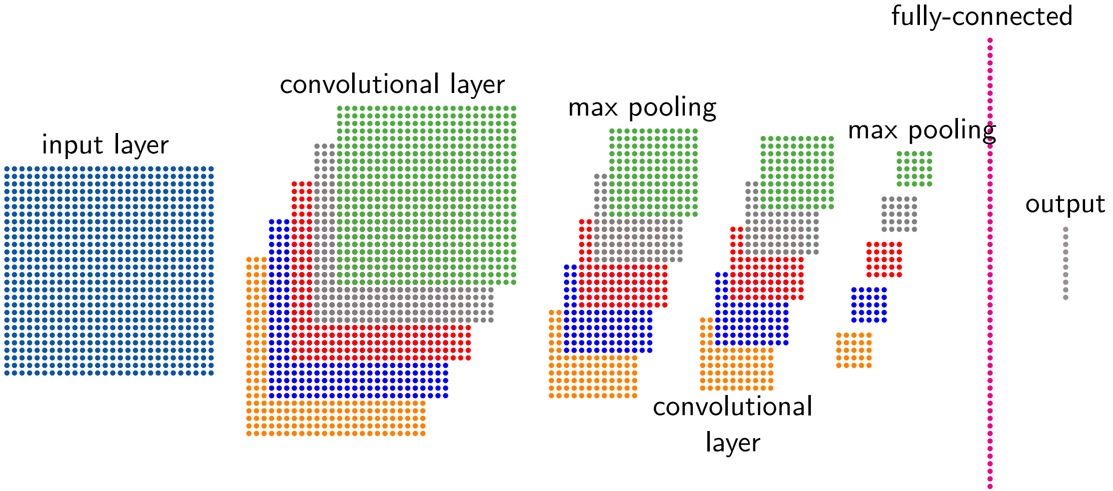
```
In [51]:
In [51]: score = model.evaluate(x_test, y_test)
In [52]:
In [52]: score
Out[52]: [0.053592409740015862, 0.98350000000000004]
In [53]:
```

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 latest network, version 2, continued

```
# model5.py
import keras.models as km, keras.layers as kl

def get_model(numfm, numnodes, input_shape = (28, 28, 1)):

  model = km.Sequential()
  model.add(kl.Conv2D(numfm, kernel_size = (5, 5), input_shape = input_shape, 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 [53]: import model5 as m5
```

```
In [54]: model = m5.get_model(20, 100)
```

```
In [55]:
```

```
In [55]: model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 20)        520
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 20)        0
conv2d_2 (Conv2D)            (None, 10, 10, 40)        7240
max_pooling2d_2 (MaxPooling2 (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 x 3) x 20) + 1) x 40 = 7240.

Our latest network, version 2, more

```
In [56]:
In [56]: model.compile(loss = "categorical_crossentropy", optimizer = "sgd",
...:             metrics = ['accuracy'])
In [57]:
In [57]: fit = model.fit(x_train, y_train, epochs = 100, batch_size = 128, verbose = 2)
Epoch 1/100
469/469 - 18s - 38ms/step - accuracy:  0.7966 - loss:  0.7378
Epoch 2/100
469/469 - 18s - 37ms/step - accuracy:  0.9486 - loss:  0.2010
⋮
Epoch 99/100
469/469 - 17s - 37ms/step - accuracy:  0.9996 - loss:  0.0028
Epoch 100/100
469/469 - 17s - 37ms/step - accuracy:  0.9996 - loss:  0.0028
In [58]:
```

Our latest network, version 2, even more

Now check against the test data.

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

```
In [58]:
In [58]: score = model.evaluate(x_test, y_test)
In [59]:
In [59]: score
Out[59]: [0.028576645734044722, 0.99119999999999997]
In [60]:
```

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 the previous 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.

Our original network using Keras, different syntax

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

def get_model(numnodes):

  model = km.Sequential()

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

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

  return model
```

```python
# model1_v2.py
import keras.models as km
import keras.layers as kl

def get_model(numnodes):

  input_image = kl.Input(shape = (784,),
    name = 'input')

  x = kl.Dense(numnodes, name = 'hidden',
    activation = 'sigmoid')(input_image)

  x = kl.Dense(10, name = 'output',
    activation = 'sigmoid')(x)

  model = km.Model(inputs = input_image, outputs = x)

  return model
```

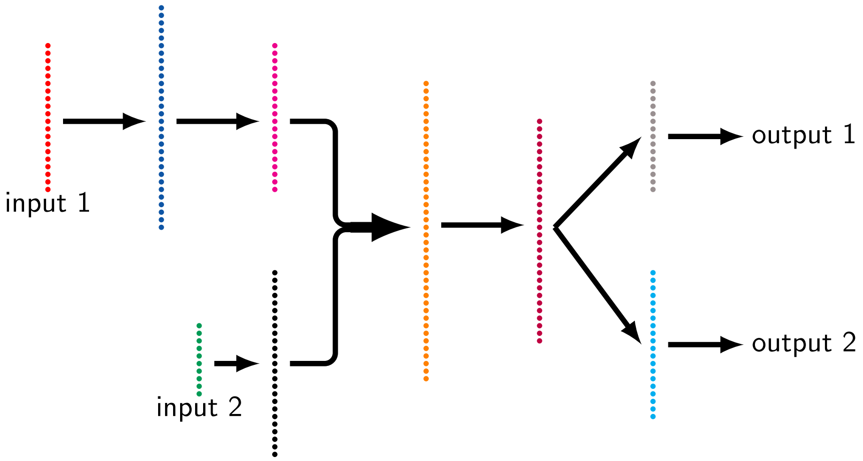
Keras has two network-building syntaxes.

Keras' functional syntax

The second syntax is known as the 'functional' syntax. Why would such a thing be available?

- The original syntax, using model = km.Sequential(), works fine assuming you have a single input, a single output, and all the layers lay in a sequence.
- But what do you do if you have multiple inputs or outputs?
- In a later class, we'll do an example of a network which takes two inputs: input data, and a requested number. The sequential networks can't handle this sort of input without combining the input together at the input stage, which may not make sense.
- The functional syntax allows you to create networks which have multiple inputs and outputs.
- It also gives you the ability to combine the output of layers within the network.
- Options for combining layer outputs include concatenating, multipling/dividing, adding/subtracting, etc.

Keras' functional syntax



Training with multiple outputs

As you might imagine, if you have multiple outputs the training of your network is going to get complicated. There are several things that must be done to train such networks.

- The first is to specify multiple loss functions, one for each output. This is done by putting a list of loss functions into your compile command (loss = ['mean_squared_error', 'categorical_crossentropy'])
- You can also scale how much emphasis to put on one output versus another, using the 'loss_weights' argument to the compile function (loss_weights = [1.0, 0.2]).
- You can also define your own custom loss functions. These take two arguments (y_true and y_pred), and return the loss value.

There is lots of flexibility available for setting up the loss functions. We will use this functionality later in the course.

Deep Learning

What is Deep Learning?

- Quite simply: a neural network with many hidden layers.
- Our last network probably qualified as Deep Learning, though barely.
- 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.

Models with hundreds of billions of free parameters are now routinely trained.

Hands-on

Let's take a break, and spend some time practicing. What things can we modify to explore what we've learned so far?

- Once again, we can change the number of hidden layers, and associated neurons.
- We can now change the number of convolutional layers, and pooling layers.
- We can vary the number of feature maps, kernel sizes, strides and activation functions of our convolutional layers.
- We can still add Dropout.
- We can change our batch size, and number of epochs.

This allows us to achieve near state-of-the-art on this data set.

Linky goodness

Frameworks:

- `https://keras.io`
- `https://www.tensorflow.org`
- `https://pytorch.org`
- `https://github.com/jax-ml/jax`

Convolutional neural networks:

- `http://www.cs.utoronto.ca/~fidler/teaching/2015/CSC2523.html`
- `https://cs231n.github.io/convolutional-networks`
- `https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8`