CMT307 Applied Machine Learning

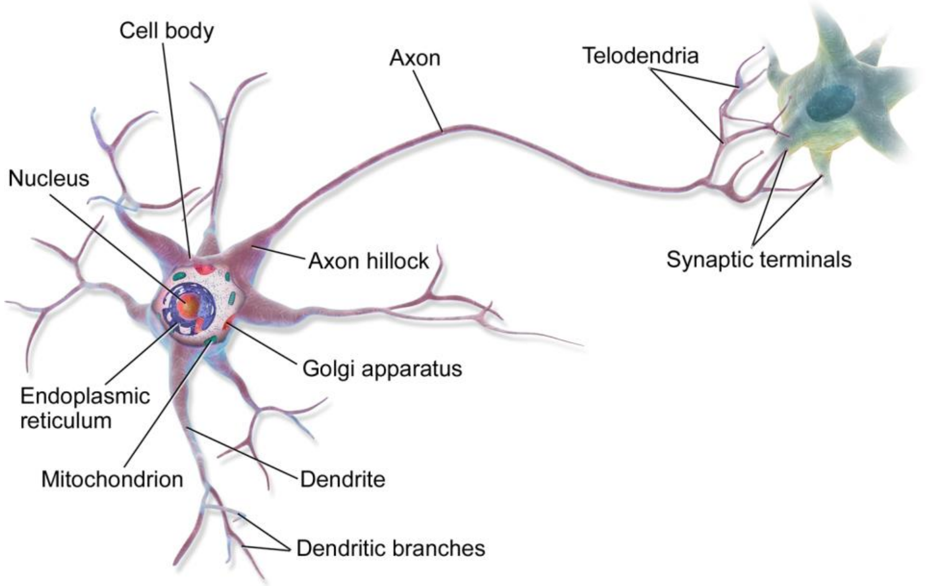
Session 12

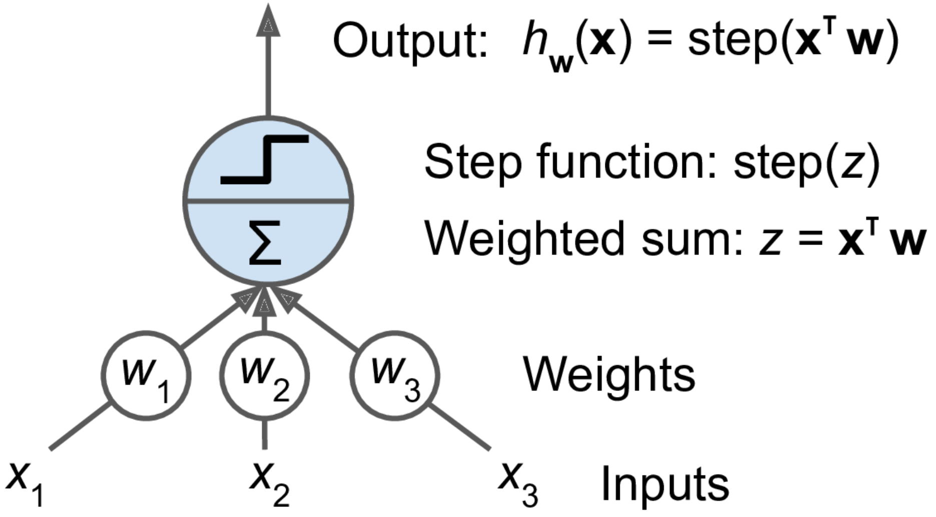
Activation Functions, Regularisation, Dropout

Introduction to Convolutional Neural Networks

Activation Functions

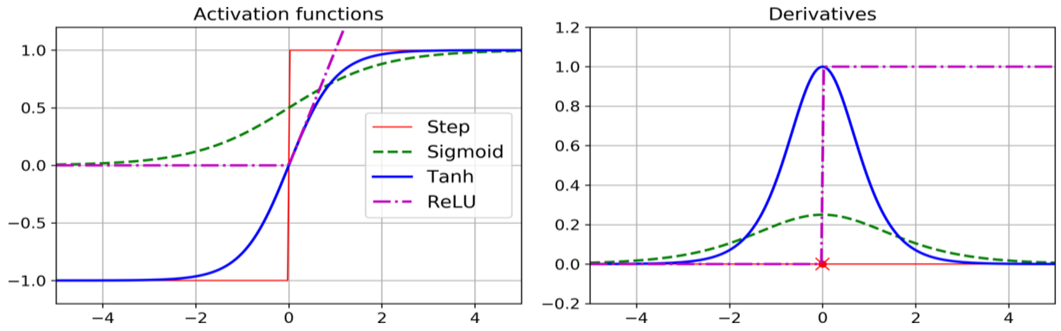
- Why needed?
  - Inspiration from bio-neurons
  - Achieving more than a simple linear mapping
    - linear mappings combined are still a linear mapping, e.g.
    - $f(x) = 2x + 3,\ g(x) = 5x - 1$
    - $f\big(g(x)\big) = 2(5x - 1) + 3 = 10x - 1$
  - Essential for neural networks to have strong learning capabilities





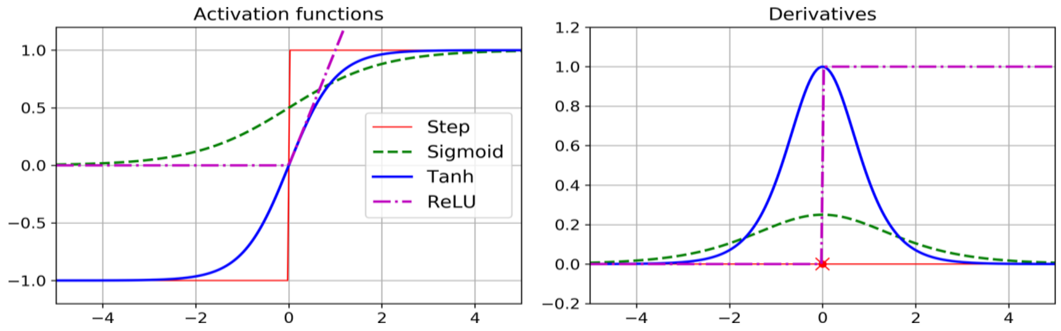
Typical Activation Function: Step Function

- Step function:
  - $\sigma(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$
  - Easy to compute
  - Disadvantages:
    - Not differentiable at $z = 0$
    - 0 gradient elsewhere: not good for gradient descent



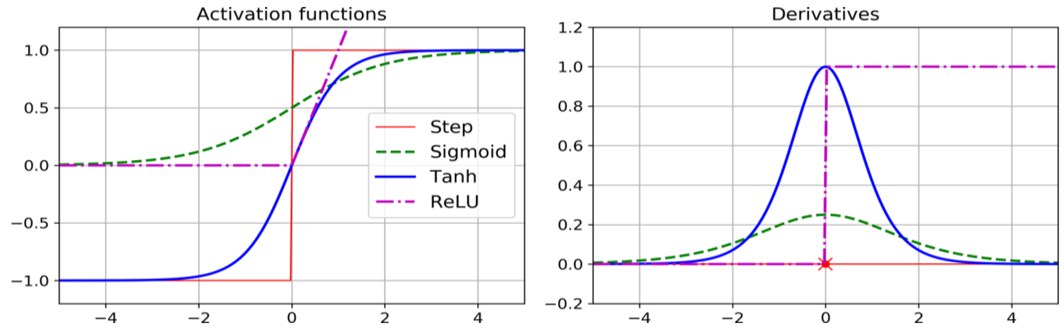
Typical Activation Function: Sigmoid Function

- Sigmoid function
  - $\sigma(z) = \frac{1}{1+e^{-z}}$
    - Well-defined, non-zero gradient everywhere
    - Slow to compute with exp
    - The output is bounded: (0, 1)
    - Especially suitable for the output layer when range (0, 1) is expected
    - But: the gradient can be small => Vanishing gradient problem



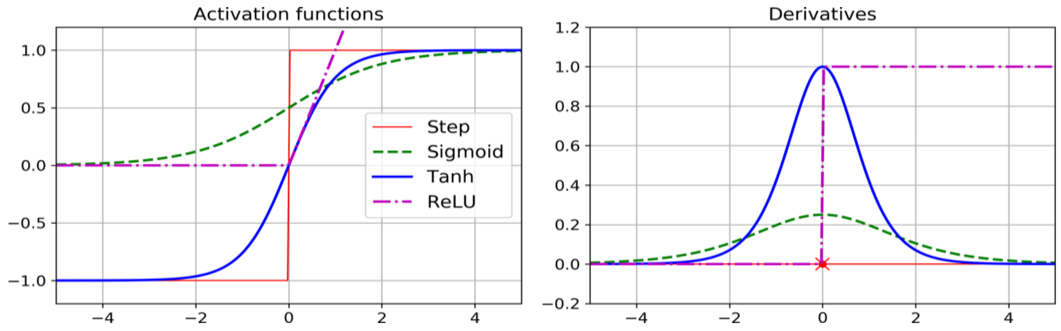
Typical Activation Function: tanh Function

- Hyperbolic tangent (tanh) function:
  - $\sigma(z) = \frac{2}{1+e^{-2z}} + 1$
  - Similar to sigmoid, well-defined non-zero gradients everywhere
  - Similar to sigmoid, slow to compute due to exp.
  - Output bound to the range (-1, 1)
  - Gradients in general stronger than sigmoid
  - Similar to sigmoid, the gradient can be small => Vanishing gradient problem
  - A popular choice, e.g. for output layer or recurrent network (covered later)



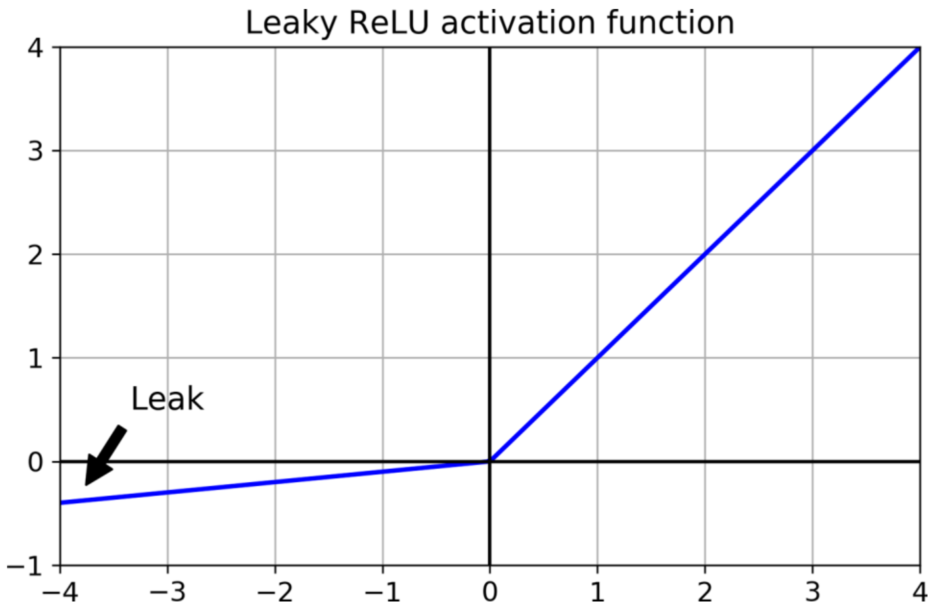
Typical Activation Function: ReLU function

- ReLU (Rectified Linear Unit)
  - $\sigma(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$
  - Zero gradients for negative *z*, and constant gradient (1) for positive *z*
  - Simple and efficient to compute
  - Works surprisingly well, often better convergence than *sigmoid* and *tanh*
  - Default choice for hidden layers, normally not used for output layers
  - Output true zeros, and unbounded positive values
  - "Dying ReLU" problem: zero gradients for negative *z => once neurons get into negative zone, unlikely to recover*

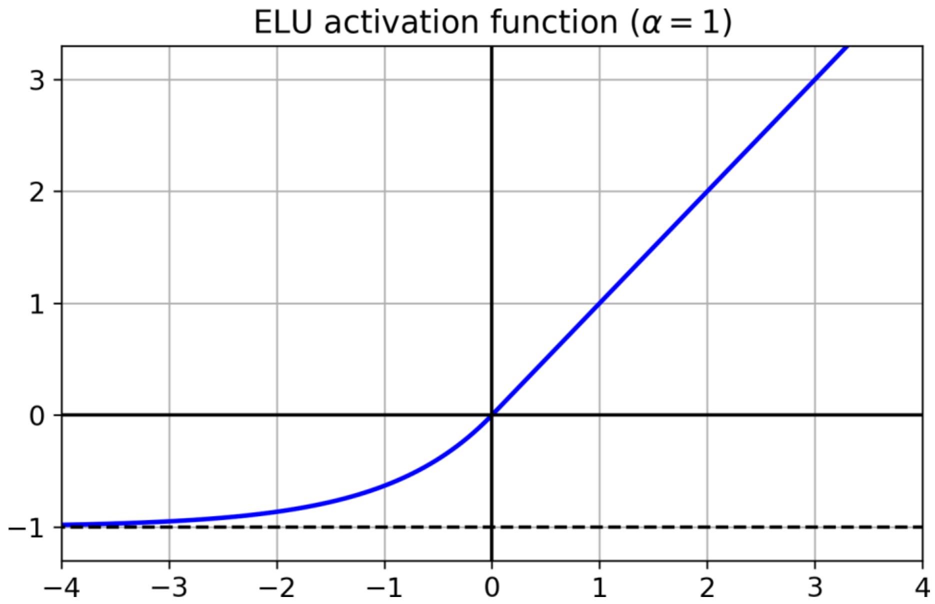


Variants of ReLU: Leaky ReLU

- To fix the "Dying ReLU" problem
  - Add a small slope for negative input
  - $LeakyReLU_{\alpha}(z) = \max(az, z)$
  - Hyperparameter α: how much to leak, e.g. 0.3
  - Variants:
    - Randomized Leaky ReLU (RReLU): α is randomised during training and fixed to the average during testing
    - Parametric Leaky ReLU (PReLU): α is a learning parameter during training.
      - Better performance for large datasets, risk of overfitting for small datasets



Variants of ReLU: ELU



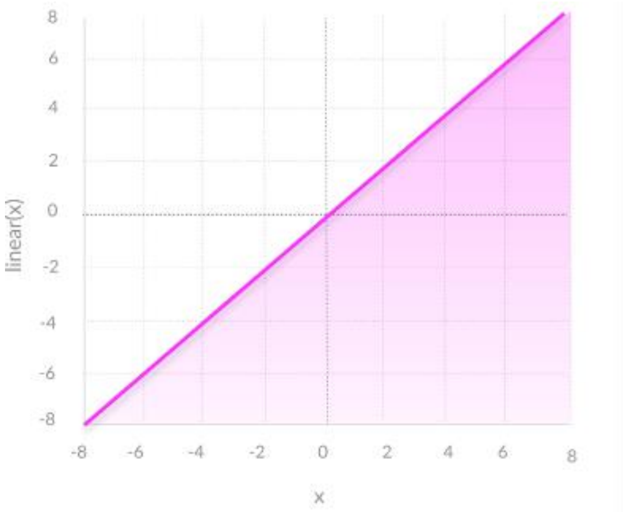
- ELU (Exponential Linear Unit):
  - $ELU_\alpha(z) = \begin{cases} \alpha(e^z - 1) & z < 0 \\ z & z \geq 0 \end{cases}$
  - Same as ReLU for positive *z*
  - Non-zero gradients for $z < 0$
  - If $\alpha = 1$, ELU is differentiable everywhere
  - Disadvantage: ELU is slower to compute
  - Results: Training time is fine as it has better convergence; evaluation time longer with ELU
- SELU (Scaled ELU):
  - self-normalise: output of each layer will tend to preserve a mean of 0 and standard deviation of 1 during training => addressing vanishing/exploding gradients problem

Activation Function: softmax

- Softmax activation:
  - Processes a vector $\mathbf{z} = (z_1, z_2, \ldots, z_K)$
  - $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
  - Each value in the range of (0, 1) and add up to 1: can be interpreted as probabilities
  - Larger input leads to larger output
  - Often used in output layer for multiclass classification

Activation Function: linear

- Linear activation:
  - $\sigma(z) = z$
  - Input/output both unbounded
  - Not suitable for hidden layers (as equivalent to one layer)
  - Often used for output layer



Activation Functions with Keras

- Two ways:
  - Through activation argument when creating layers

```
model.add(keras.layers.Dense(100, activation="relu"))
```

    - Here, `relu` can be changed to any supported activation functions: `sigmoid`, `tanh`, `relu`, `elu`, `selu`, `softmax`, `linear`, etc.
    - For more advanced options, including Leaky ReLU, use:

```
model.add(keras.layers.Dense(100, activation=keras.layers.LeakyReLU()))
```

  - Add as activation layers

```
model.add(keras.layers.Dense(100))
model.add(keras.layers.Activation("relu"))
```

- Second approach: allows other layers such as BatchNormalization to be applied *before* non-linear activation

```
model.add(keras.layers.Dense(100))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
```

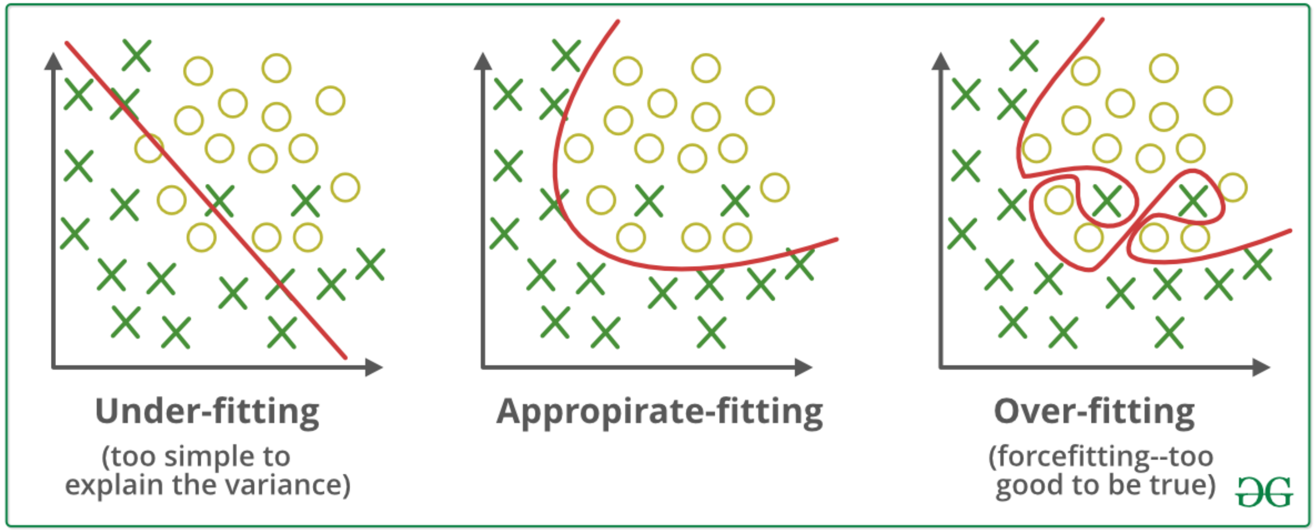
`[Demo] Comparison of activation functions`

Choice of Activation Functions

- Tips for choosing activation functions:
  - Hidden layers:
    - ReLU is often a good choice
    - If "DyingReLU" becomes a problem, consider ELU, SELU or Leaky ReLU
  - Output layers:
    - If normalised outputs are expected, consider tanh, sigmoid, etc.
    - If output is like probabilities, softmax is often used
    - If output is unbounded, no activation function (equivalent to Linear activation)
  - Use Batch Normalisation to help address vanishing/exploding gradient problems

Regularisation

- Why using regularisation?
  - Deep models often have many parameters (typical millions)
  - This can easily lead to *overfitting* problem
  - The model works well for the training set, but does not generalise well to validation/test sets



Regularisation: Constraining Connection Weights

- Constraining connection weights:
  - L2 regularisation: adding a loss term corresponding to the L2 norm of the connection weights
  - L1 regularisation: adding a loss term corresponding to the L1 norm of the connection weights
    - => favours sparse connection weights (i.e. more connection weights that are close to 0)

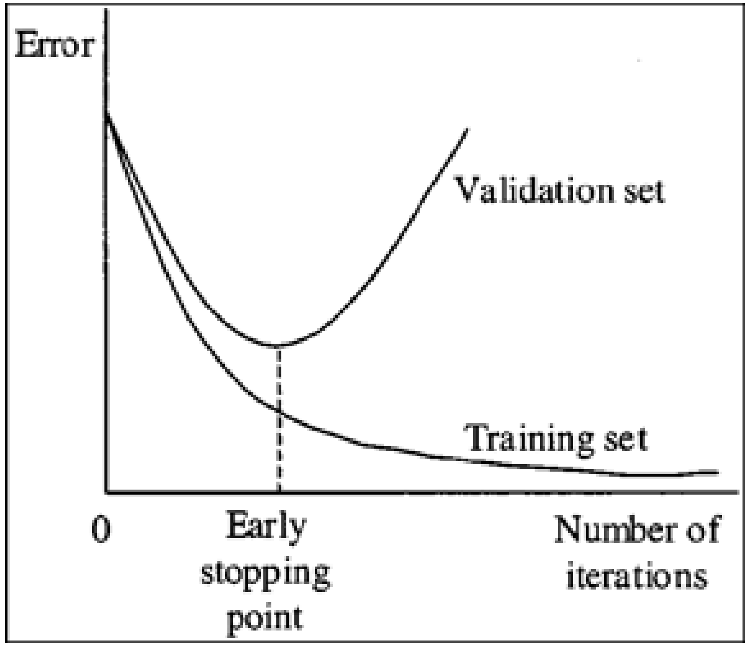
  - Regularisation factor: controls how much regularisation to add
- Keras example:
  - (`l2` can be changed to , 0.01 is the regularisation factor)

```
model.add(keras.layers.Dense(300, activation="relu", kernel_r
egularizer=keras.regularizers.l2(0.01)))
```

```
[Demo] Regularisation
```

Regularisation: Early Stopping

- More training epochs do not always lead to better models
  - The learned model may overfit the training data
  - This can be monitored using the performance on the validation set
  - Can be seen as a powerful regularisation



Keras Implementation: Saving and Loading Models

- To save a trained model, using
  - `model.save("my_keras_model.h5")`
  - my_keras_model => can be changed to the name of choice
  - .h5 => standard HDF5 (Hierarchical Data Format) format
- To load the model back, using
  - `model = keras.models.load_model("my_keras_model.h5")`
  - Change the filename to match the model file

Keras Implementation: Checkpoint

- Training can take very long time
- It is good practice to save the model once in a while

```
[...] # build and compile the model
checkpoint_cb =
keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10,
callbacks=[checkpoint_cb])
```

- This will save the model after each epoch of training
- The model can be later reloaded to continue training

Keras Implementation: saving the best model

- To avoid overfitting, the best model on the validation set can be saved, and loaded back later (setting `save_best_only` to `True`):

```
checkpoint_cb =
keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                        save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5")
# roll back to best model
```

Keras Implementation: early stopping

- Alternatively, stopping early if no progress after a few epochs (defined by `patience`)

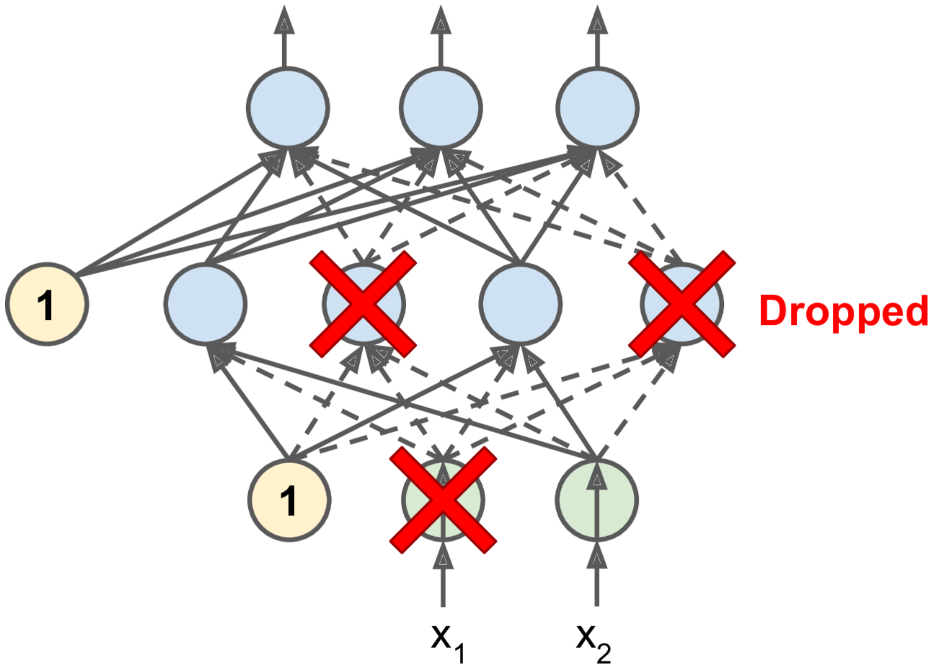
```
early_stopping_cb =
keras.callbacks.EarlyStopping(patience=10,
                                restore_best_weights=True)


history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[early_stopping_cb])
```

[Demo] Early stopping

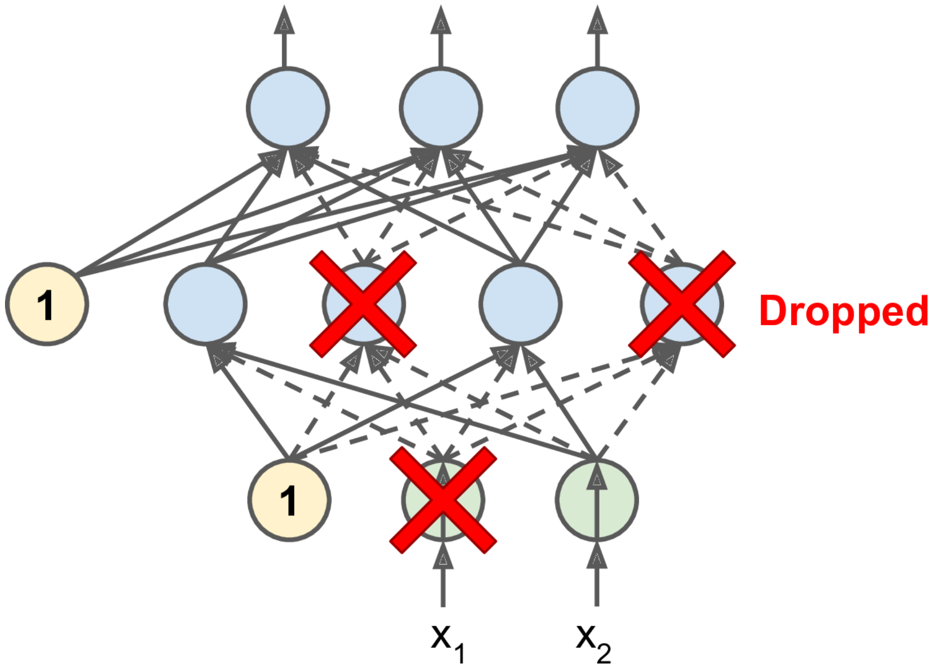
Dropout

- Dropout
  - A popular regulariser
  - Simple idea but effective
  - Often leads to performance gain
- How does Dropout work?
  - At every training step
  - Each involved neuron has a probability *p* of being temporarily “dropped out” (output 0)
  - Different random neurons are dropped out in different steps
  - Hyperparameter *p:* typically 10%-50%
  - May apply to the input layer but *not* the output layer
  - Often applied to top 1-3 layers (excluding the output layer)



Dropout

- Why Dropout works?
  - Every step, a different (but related) network is trained
  - The network needs to be effective, even with part of it disabled
  - It cannot rely on specific neurons to work effectively
  - Better robustness and resilience
  - Imagine a team with random members not attending…



Dropout

- Dropout: training and testing
  - Dropout is only applied in training, and disabled in testing
  - Suppose $p = 0.5$, on average, during testing, each neuron will be connected to twice as many input neurons
  - To address this, we need to perform either of the following
    - Do nothing during training. After training, multiply connection weight of each neuron input by 0.5 (1-*p, i.e. keep probability*)
    - During training, divide each neuron's output by (1-*p*). Do nothing during testing
  - These two approaches are not identical, but both work well.
  - Keras implements Dropout layers that use the second option (so no adjustment needed once the model is trained)

Dropout Keras Implementation

- Create a Dropout layer using e.g.

```
keras.layers.Dropout(rate=0.2)
```

  - 0.2: dropout rate, can be adjusted
  - Not used after the output layer (otherwise you lose part of the output!)

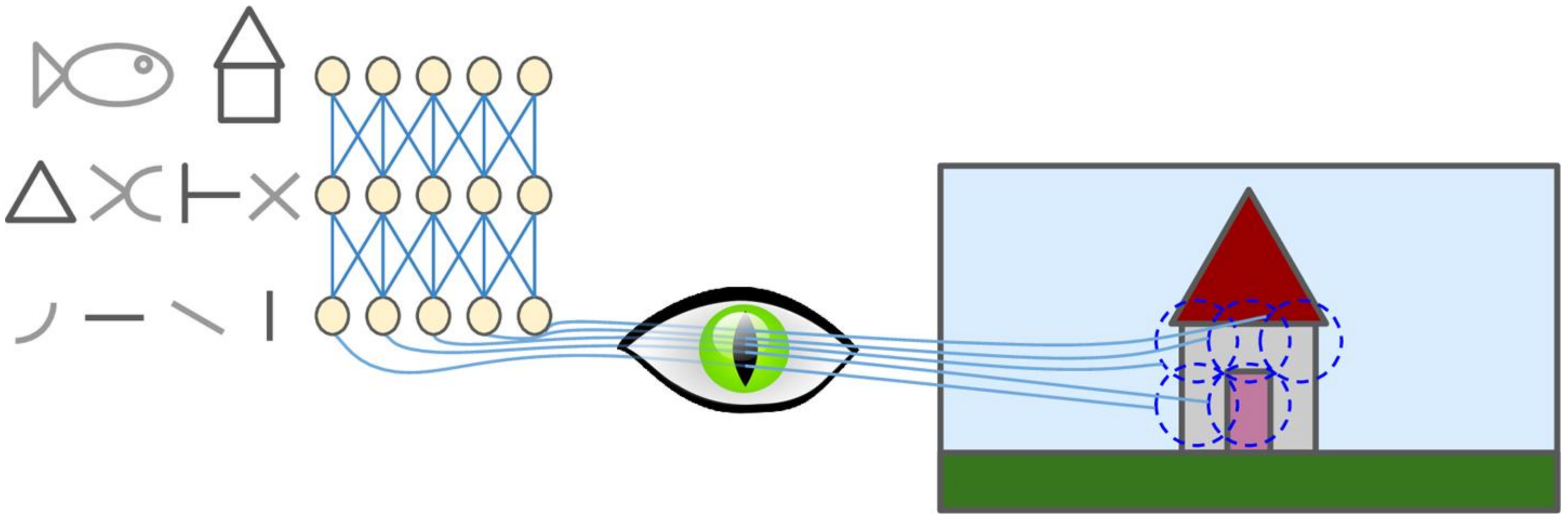
- `[Demo] Using Dropout`

Convolutional Neural Networks (CNNs)

- Problem with fully connected layers:
  - Too many connection weights, especially for images
    - E.g. 100×100 image input, with only 1,000 first layer neurons => 10M connection weights
    - More with more layers and neurons
  - Harder to train, does not generalise well
  - Does not work except for very small images

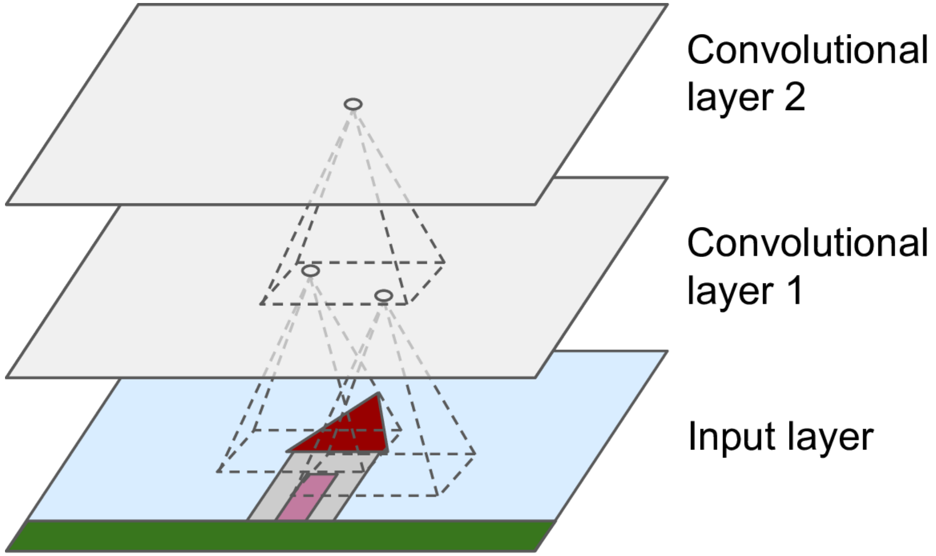
CNN: inspired by biological neurons

- Inspired by biological neurons for visual cortex
  - Patterns in small regions (receptive fields)
  - Higher layer neurons correspond to more complex patterns with larger receptive fields



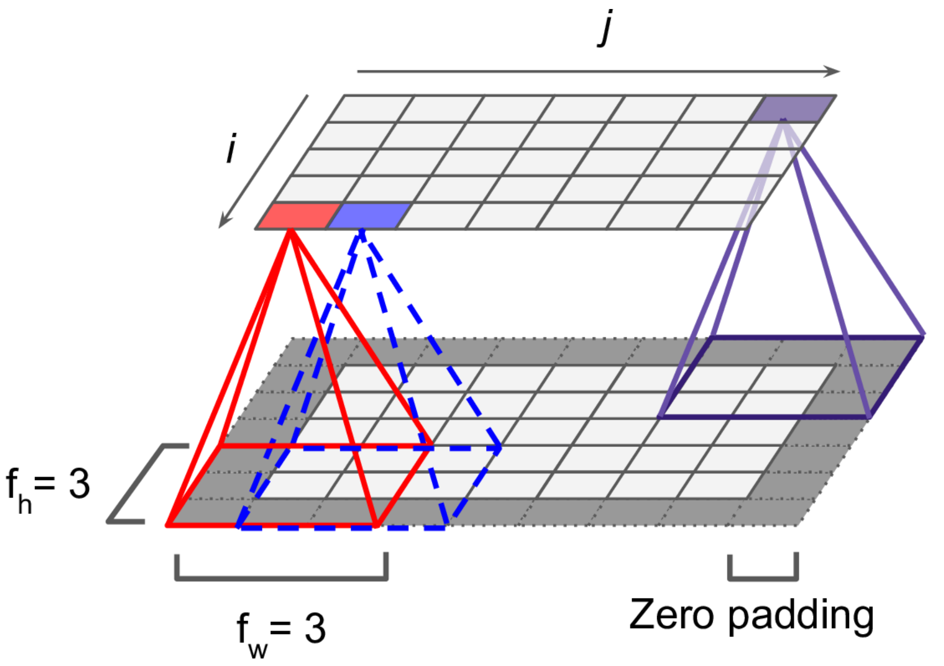
CNN

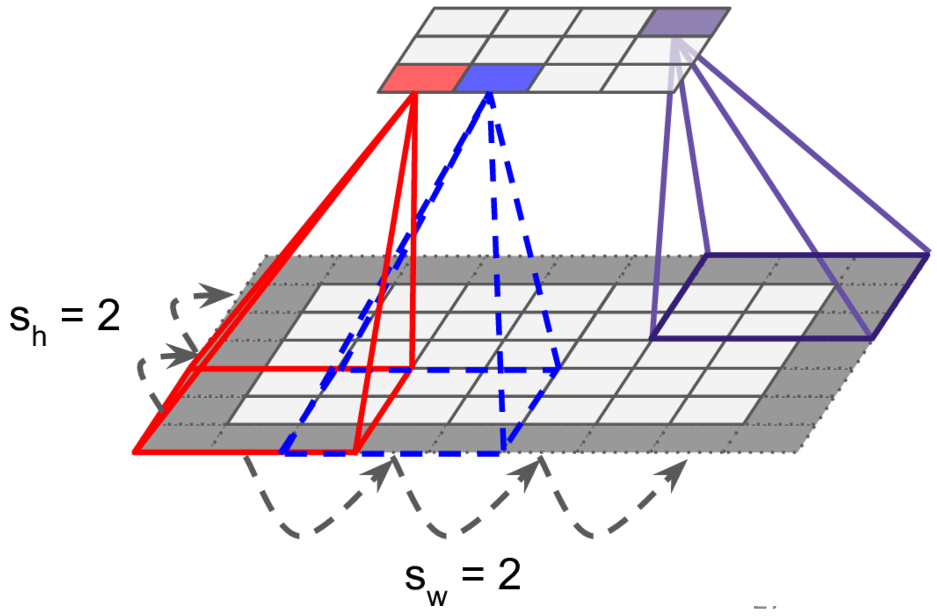
- Convolutional Neural Networks
  - Widely used for image processing
  - Also useful for speech recognition, natural language processing, etc.
  - Key: convolutional layers
- Convolution layer:
  - Local receptive fields: increasing as it goes deeper
  - Shared weights [corresponding to ‘convolution’ operation]
    - Substantial reduction of connection weights
    - Translation equivariance: an object should be recognised the same even moving around



Convolutional Layer

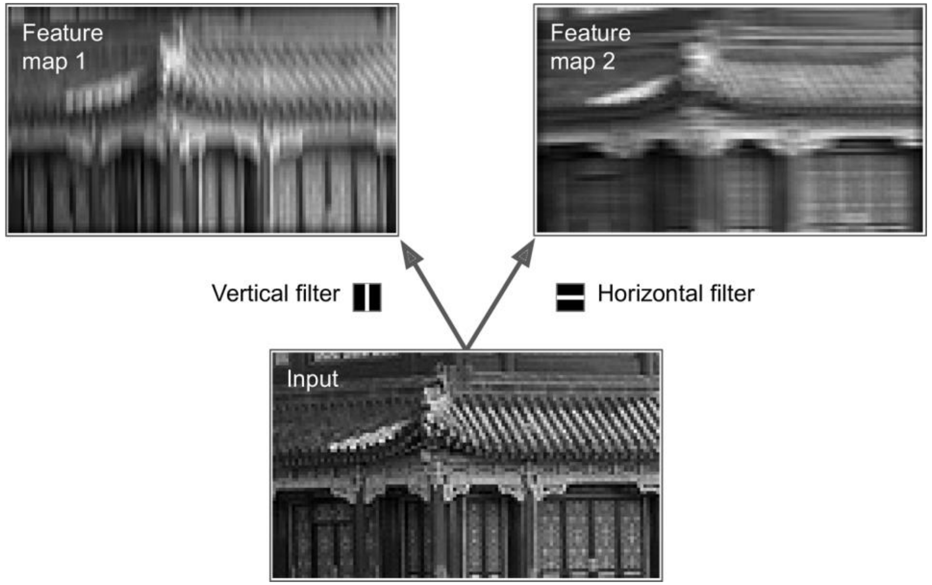
- Convolutional layer
  - Keep the 2D image structure (no Flatten layer)
  - Zero Padding
    - Without padding, the output is smaller than the input (padding = “valid”)
    - Zero padding: add zeros such that the output size is not lost at image boundaries (padding = “same”)
  - Stride
    - The shift from one receptive field to the next
    - Stride = 1: the output is roughly the same size as input
    - Stride = 2: the output is approx. half the size (in each dimension)





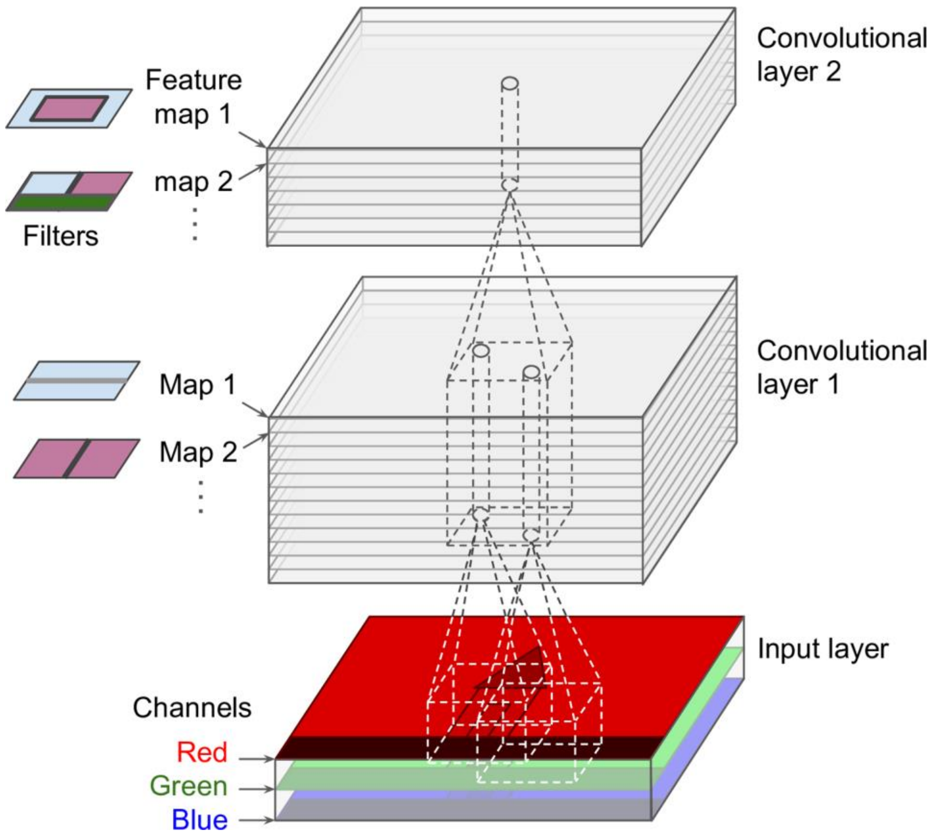
Convolutional Filters

- Each filter is useful to extract certain local image characteristic
  - A feature map is created when a filter is applied
  - Multiple filters are often used to extract different information
- In this example, the filters are hard-coded, but in practice, filters are learnable parameters.



Stacking Multiple Feature Maps

- Power of CNN
  - Each layer: 3-dimensional (2D for images, third dimension for channels/filters)
  - Input: colour images (red, green, blue channels)
  - Convolutions apply local receptive fields and across all input channels



Convolutional Layer: Keras implementation

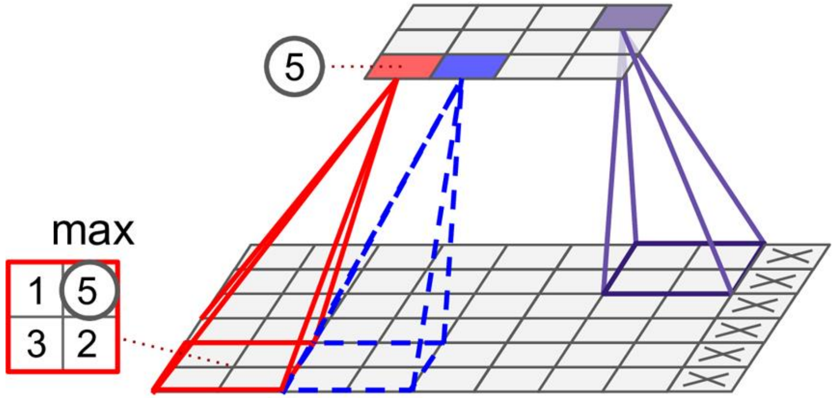
- keras.layers.Conv2D: (2D convolutional layer).
- Example:

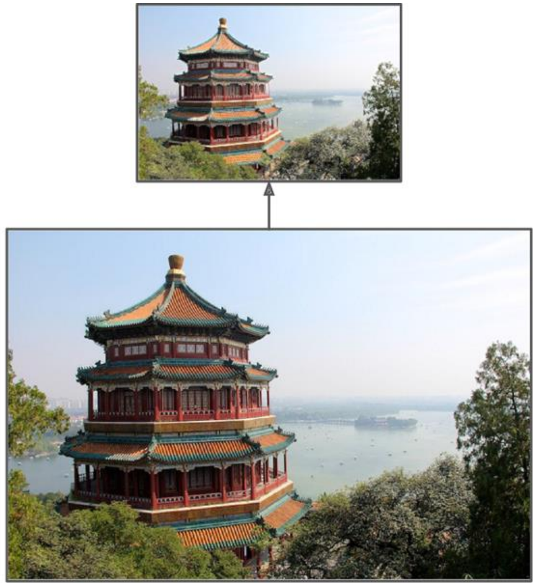
```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

- Memory usage:
  - Convolutional layers are better than fully connected layers
  - May still use a large amount of memory depending on number of filters, kernel size, number of layers, and batch size (during training)
  - If running out of memory, try reducing the batch size, or using multiple GPUs, etc.

Pooling Layers

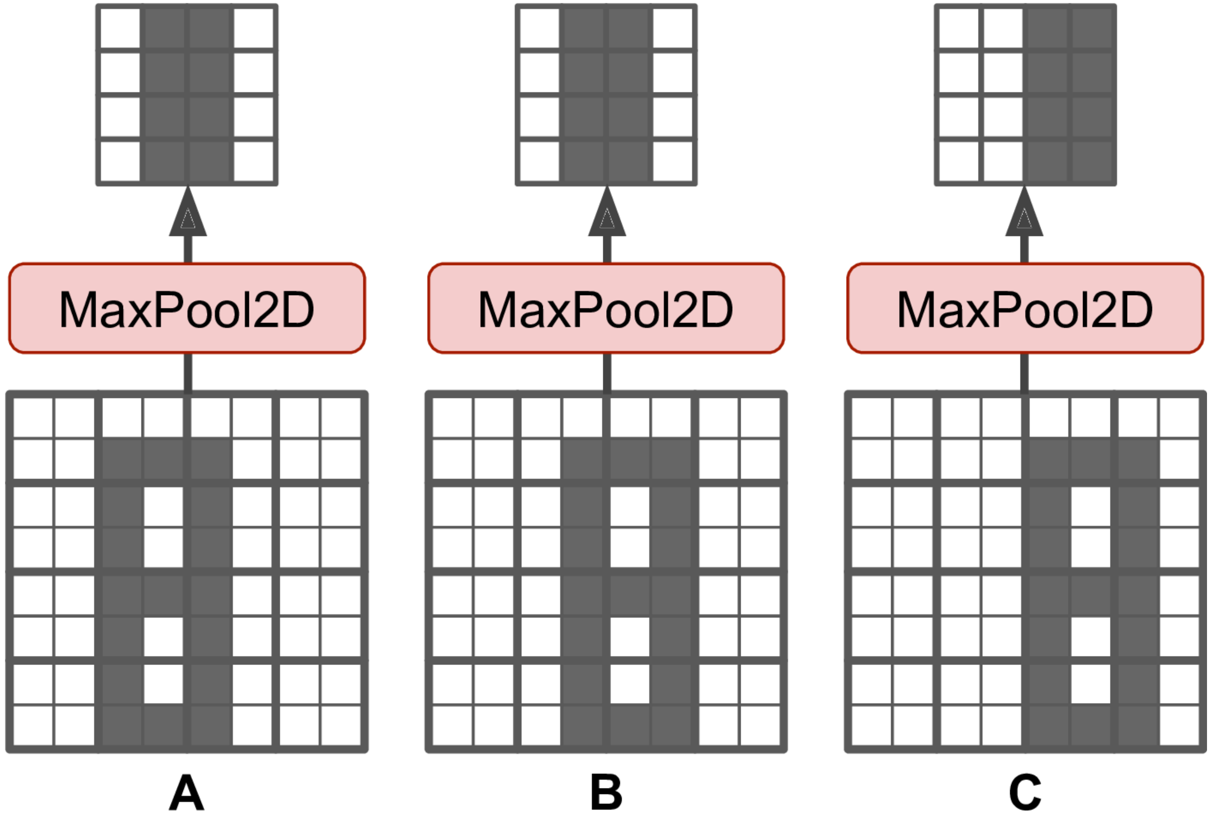
- Subsample the input
  - Reduce computational load, memory usage, and number of parameters
  - Increase robustness to some (small) transformations
  - Example: max pooling; pool_size = 2; stride = 2 (usually match)





Pooling Layers

- Max pooling: invariance to small translation

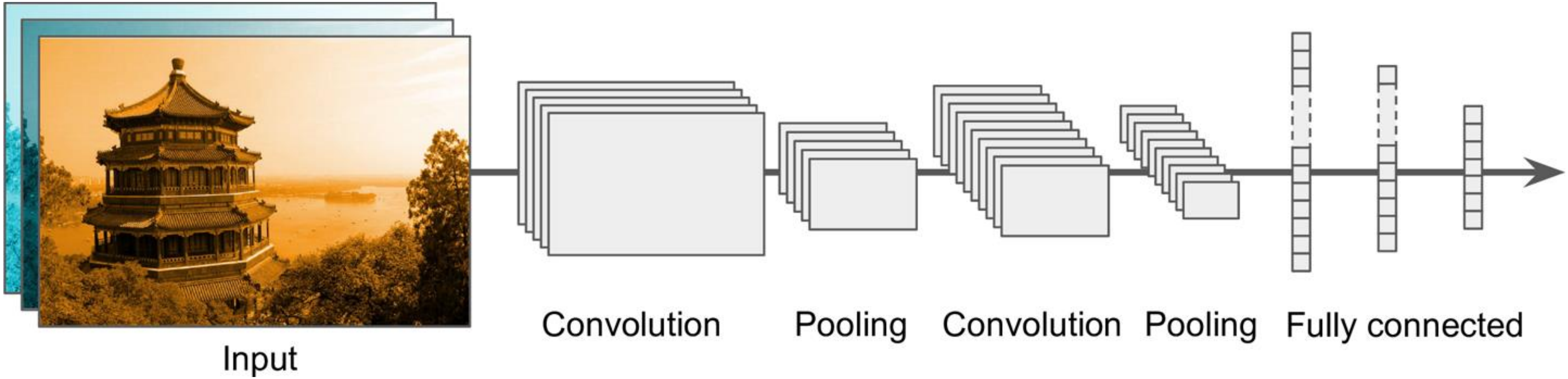


Types of Pooling Layers

- Types of pooling
  - Max pooling, e.g. `keras.layers.MaxPool2D (pool_size = 2)`
  - Average pooling, e.g. `keras.layers.AvgPool2D (pool_size = 2)`
  - Max pooling is more often used/often more effective:
    - keeping the strongest features; getting rid of meaningless features => cleaner signal to process

  - Global average pooling: `keras.layers.GlobalAvgPool2D()`
    - Output a single average over the whole spatial locations per filter (feature map)
    - Useful to obtain global aggregation

Typical CNN Architecture



- [Demo] CNN-based approach for image recognition

Intermediate Report

- 1-page report due next Thursday
  - Not contributed to your project marks
  - A chance to get feedback
  - Email it to your supervisor

Talk next Monday

- I will give a talk at 1:30pm on Monday 2nd March (here at the Turing suite)

- Title: Deep Generative Models for Images and 3D Shapes

- Time for Practice!