

Introduction to TensorFlow and Deep Learning

Lecture 4: Convolutional Neural Networks, Vision and RNNs

IADS Summer School, 1st August 2022

Dr Michael Fairbank

University of Essex, UK

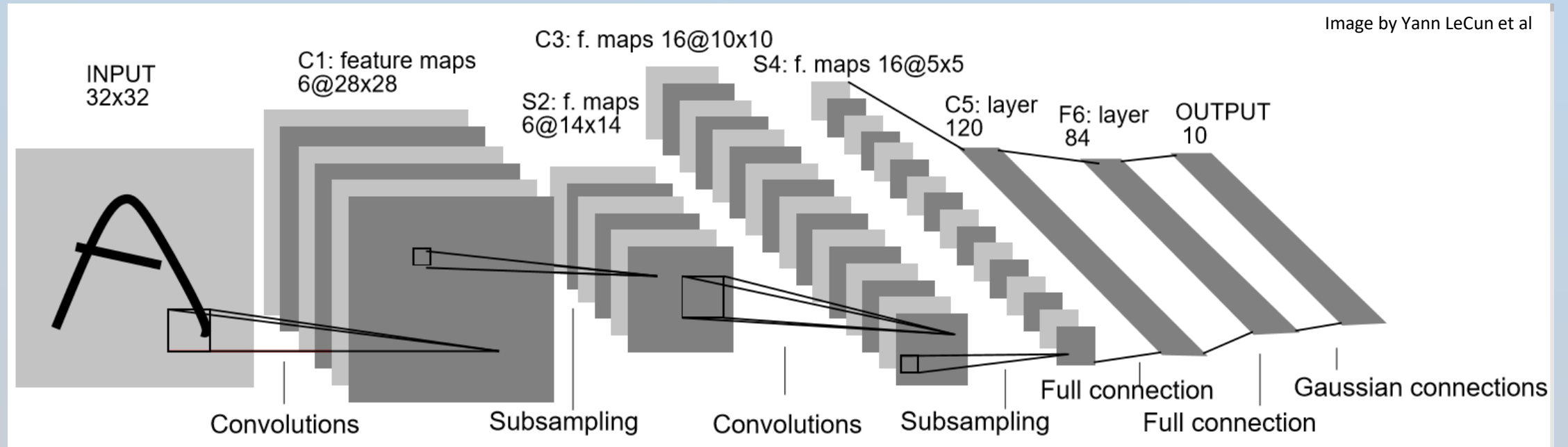
Email: m.fairbank@essex.ac.uk

Recap

- Summary so far:
 - TensorFlow basics
 - Gradient Descent and automatic gradient finding
 - Feedforward neural networks
 - MNIST vision task (scored $\approx 92\%$)
 - Loading data
 - Matplotlib for plotting training and seeing overfitting
 - Fighting Overfitting: Regularisation and dropout
 - Saving the learned neural networks
 - Minibatches
- This lecture (3.30pm-5.00pm):
 - Keras “Fit” Loop
 - CNNs (Convolutional Neural Networks)
 - MNIST revisited (will score $\approx 97.5\%$)
 - Introduction to Recurrent Neural Networks (* If time permits)

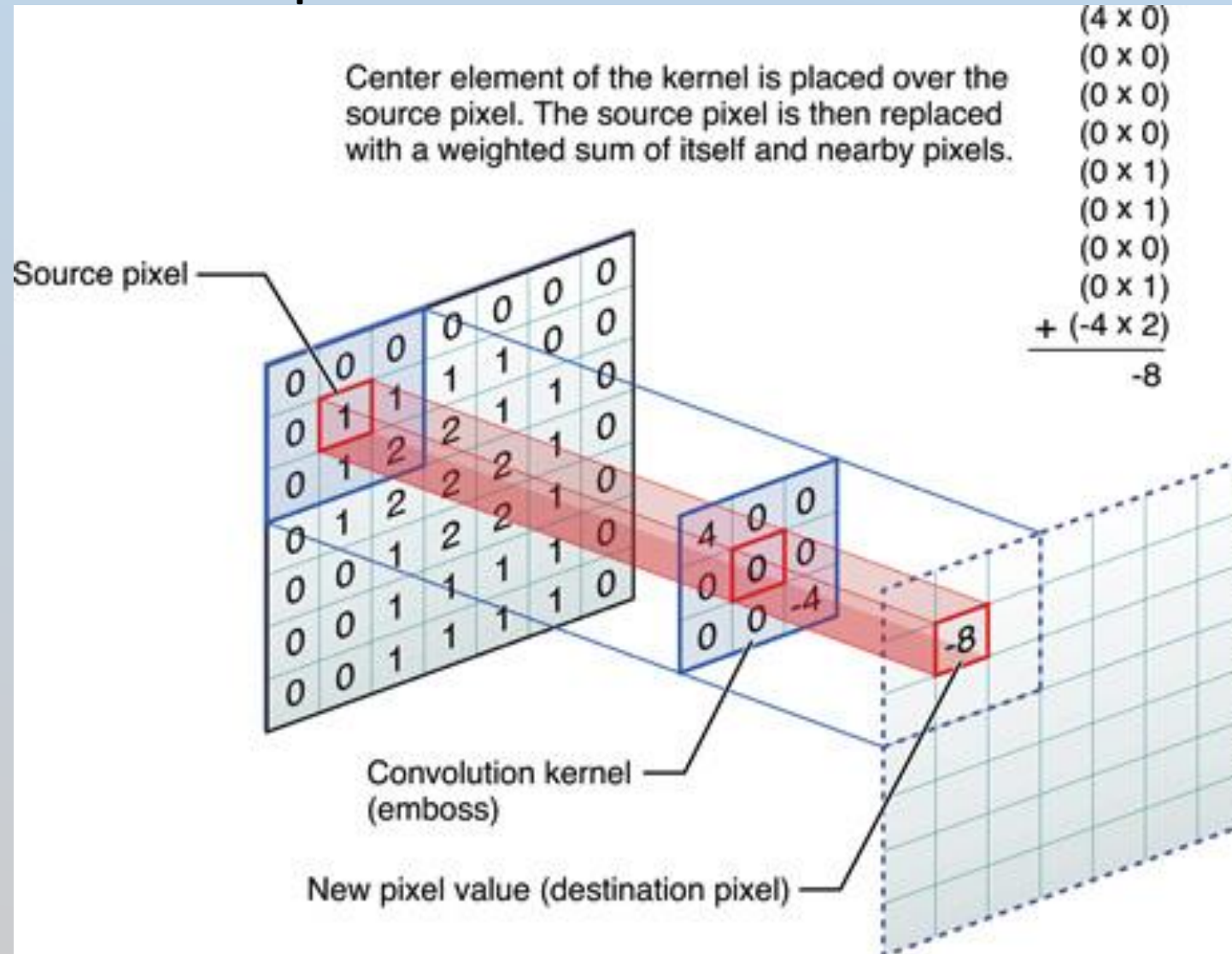
Convolutional Neural Networks

CNNs: LeNet-5



“LeNet-5” by Yann LeCun et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324

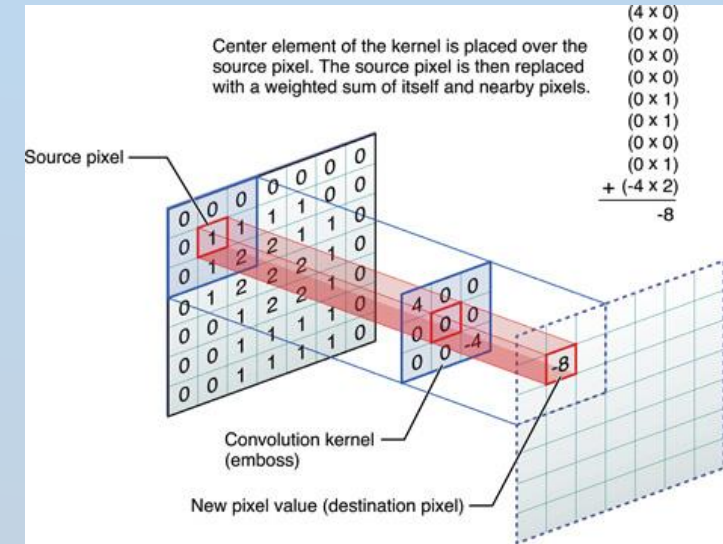
Convolution operation:



CNNs

Innovation of the Convolution Operation:

1. Allows the input layer to be rearranged into 2D (or 3D) form – so we can have an input *matrix* instead of an input *vector*
 - Previously for MNIST, we had to use the 28*28 image lineated into a length-784 input vector
2. Apply the weight matrix to many different small portions of the input space
 - Instead of one large matrix that looks at the whole input vector



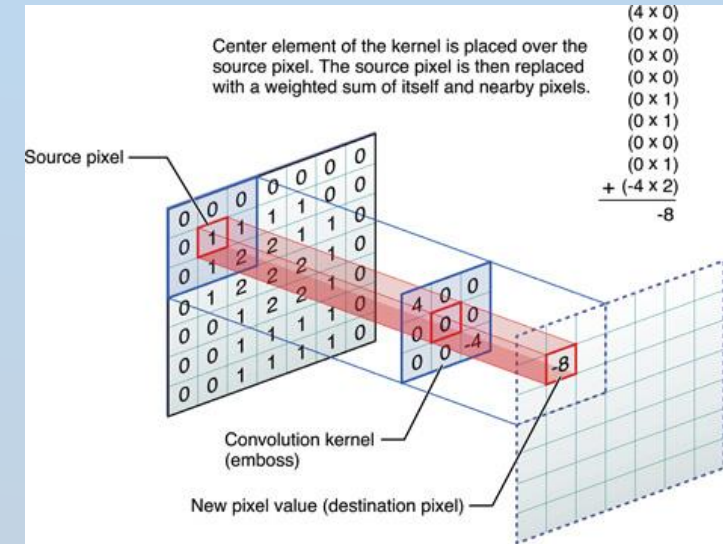
Convolution operation:

The 3*3 matrix is the convolutional “weight matrix” or “filter” or “kernel”

The kernel in this image is moved over the source image in steps of “stride_length” (Usually use stride_length=1)

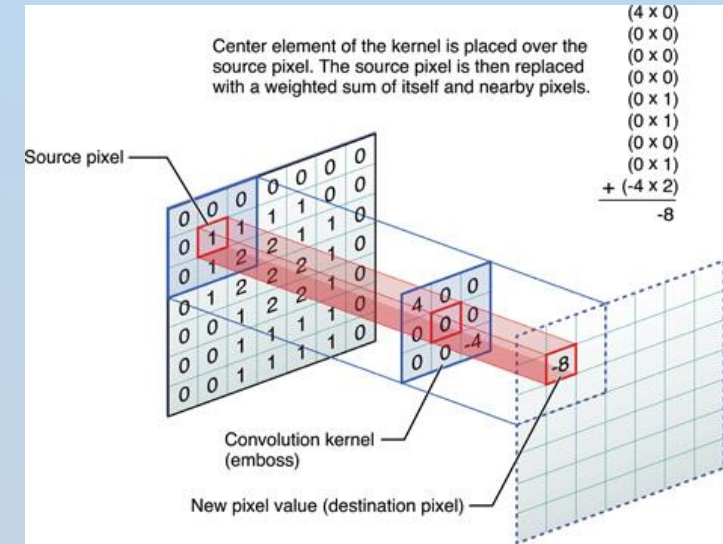
Using a convolution has the following benefits:

1. Reduced number of weights, compared to a fully connected layer
2. Translational invariance
3. Shared weights



Convolution operation:

- In this figure there are just 9 weights connecting to the “-8” node. You could say all other weights connecting to that “-8” node are zero
 - (Sparseness)
- The same 9 weights are used repeatedly to connect each 3*3 portion of the image to a node in the next layer
 - (shared weights)
- “Shared weights” mean those 9 weights get several training patterns from the same single image
 - Reuses limited training data



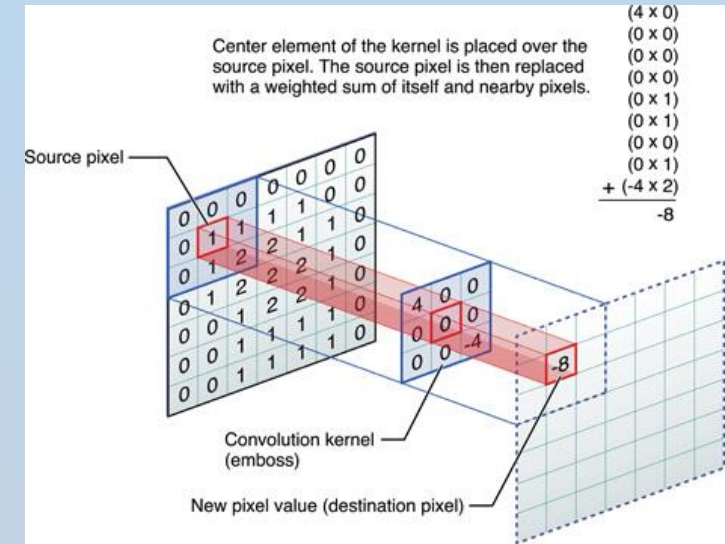
Convolution operation:

- TensorFlow code:

Our first tensor
with rank>2

```
# input tensor is shape (n,28,28,1)
```

```
layer=tf.keras.layers.Conv2D(1, kernel_size=(3,3), strides=(1,1), padding="same", activation='relu')
```



Convolution operation:

- The source image could have 3 layers, e.g. RGB components.
- The convolutional kernel would also be 3D
- Here 5 kernels are swept over an image with 3 channels.
- Each kernel is hopefully looking for different useful image features (e.g. edges, or higher level shapes)

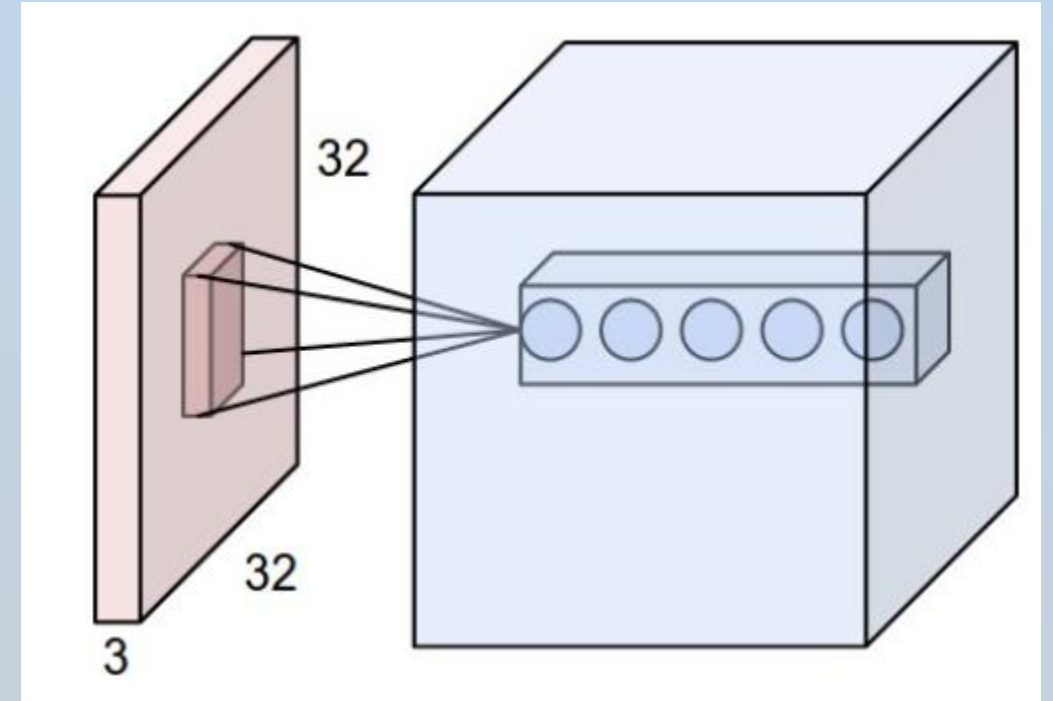
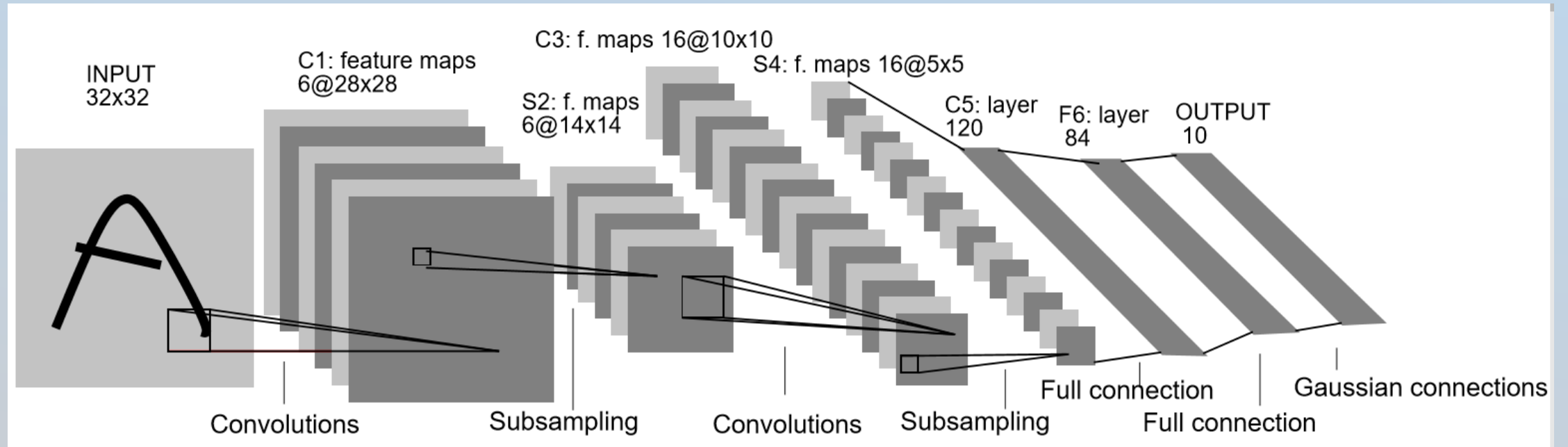


Image source: <https://cs231n.github.io/convolutional-networks/>

```
layer=tf.keras.layers.Conv2D(5, kernel_size=(3,3), strides=(1,1), padding="same", activation='relu')
```

See https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D

Convolution vision networks



- Convolutional layers are usually alternated with operations that attempt to lower the resolution of the image: Subsampling (Downsampling, max-pooling)

Max-pooling (Subsampling)

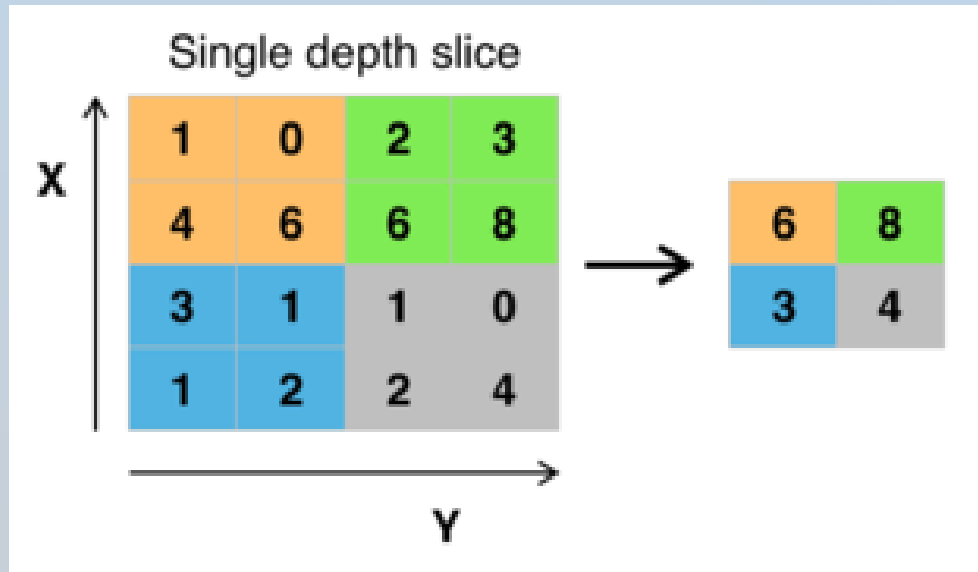


Image credit: Aphex34, https://commons.wikimedia.org/wiki/File:Max_pooling.png

Here, $8 = \max(2, 3, 6, 8)$

Subsampling, with kernel size=2,
stride_length=2

- This has halved the width and height of image

Max-pooling (Subsampling)

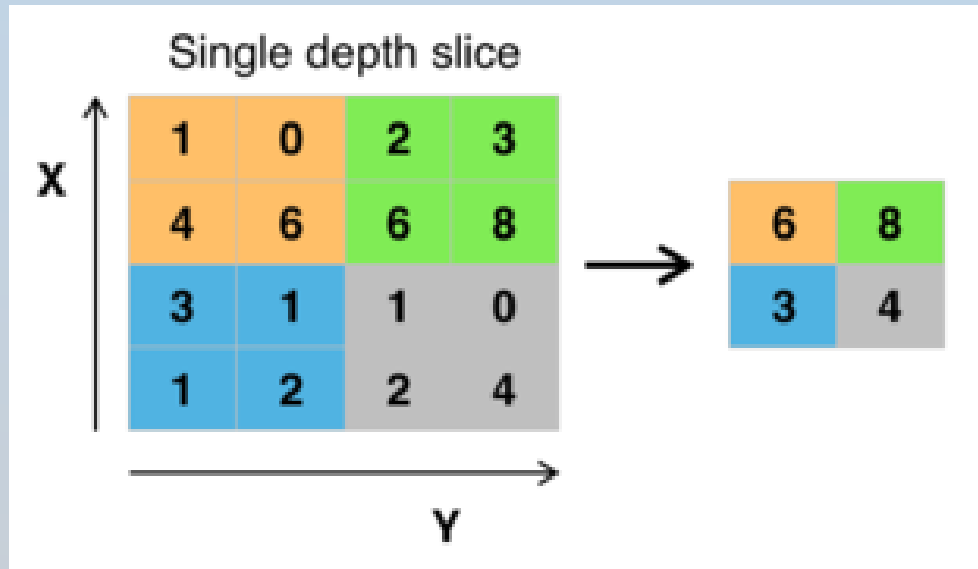


Image credit: Aphex34, https://commons.wikimedia.org/wiki/File:Max_pooling.png

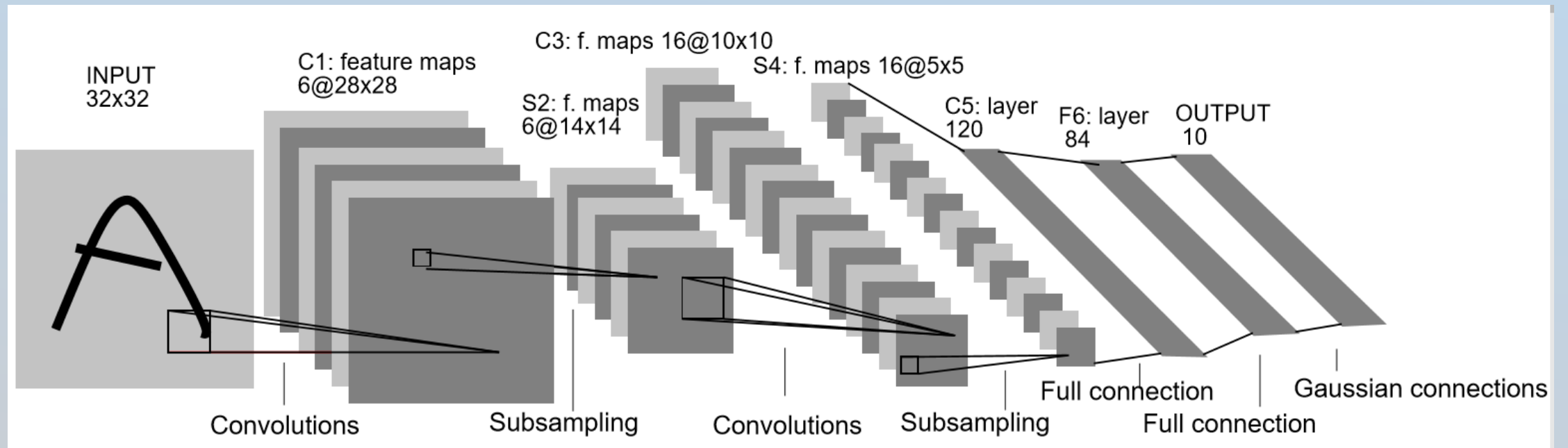
Here, $8 = \max(2, 3, 6, 8)$

Subsampling, with kernel size=2,
stride_length=2

```
layer=tf.keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding="same")
```

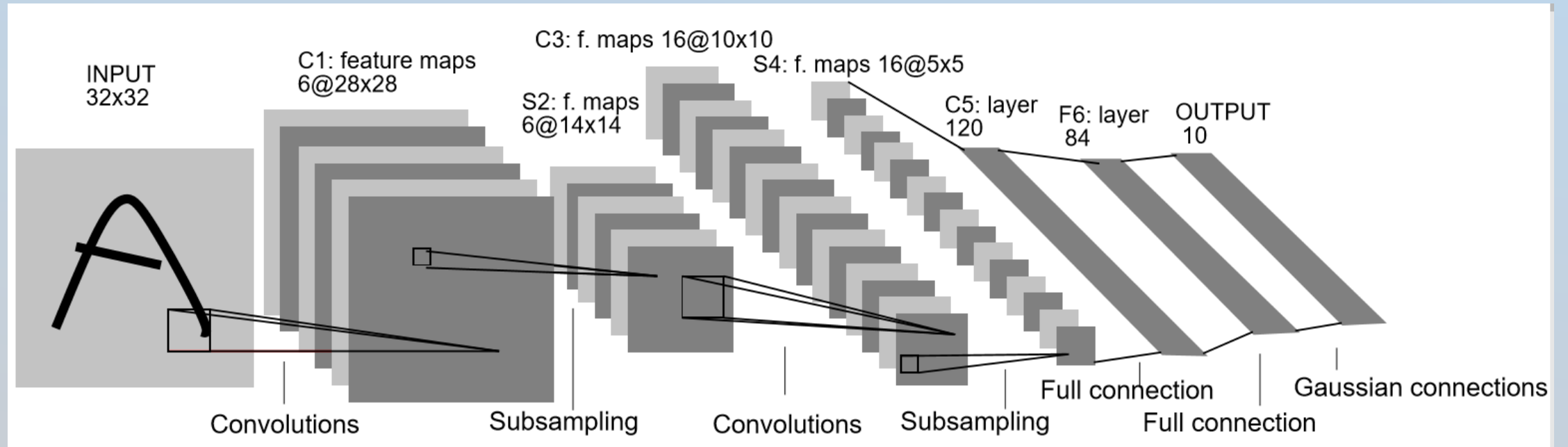
See https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D

Convolution vision networks



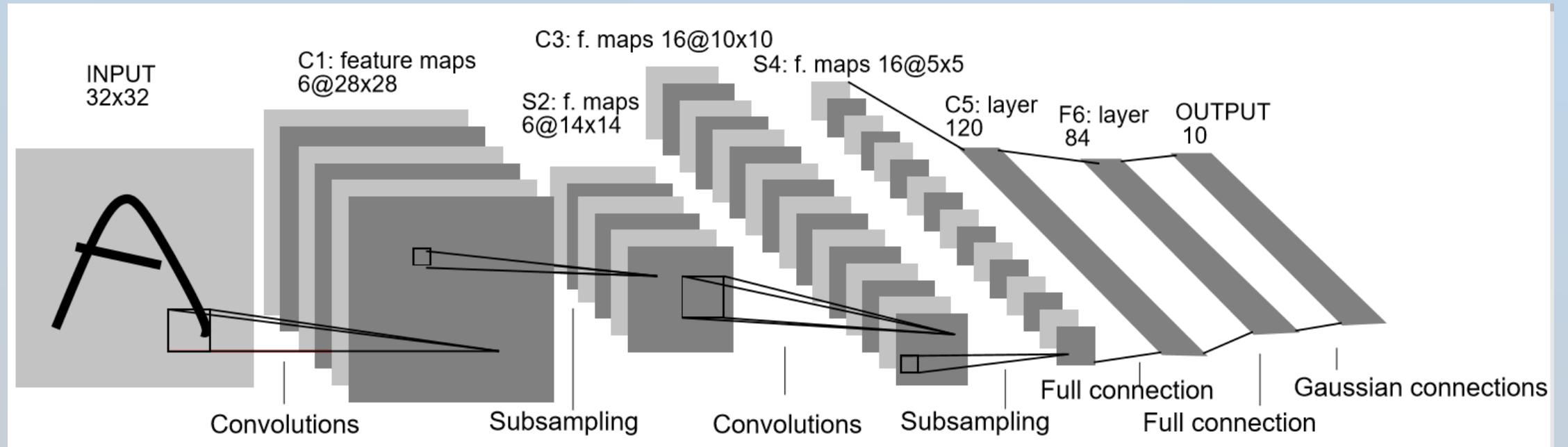
- The alternating convolutional and max-pooling layers iteratively attempt to:
 1. Convolution to extract features of increasingly higher level (e.g. edges... outlines... components)
 2. Max_pooling to Lower the resolution to something manageable (and higher-level)

Convolution vision networks



- After alternating convolutional and max-pooling layers, there will be at least one fully connected layer
- Final layer is a softmax layer classifier (trained by cross-entropy)

Convolution vision networks



- While we might have one (B+W) or 3 (RGB) “channels” in the input image, by the c3 or c1 layer we have many channels appearing.
- The next convolutional layer looks at all of the channels of the preceding layer to transform the features further.

Convolution vision networks

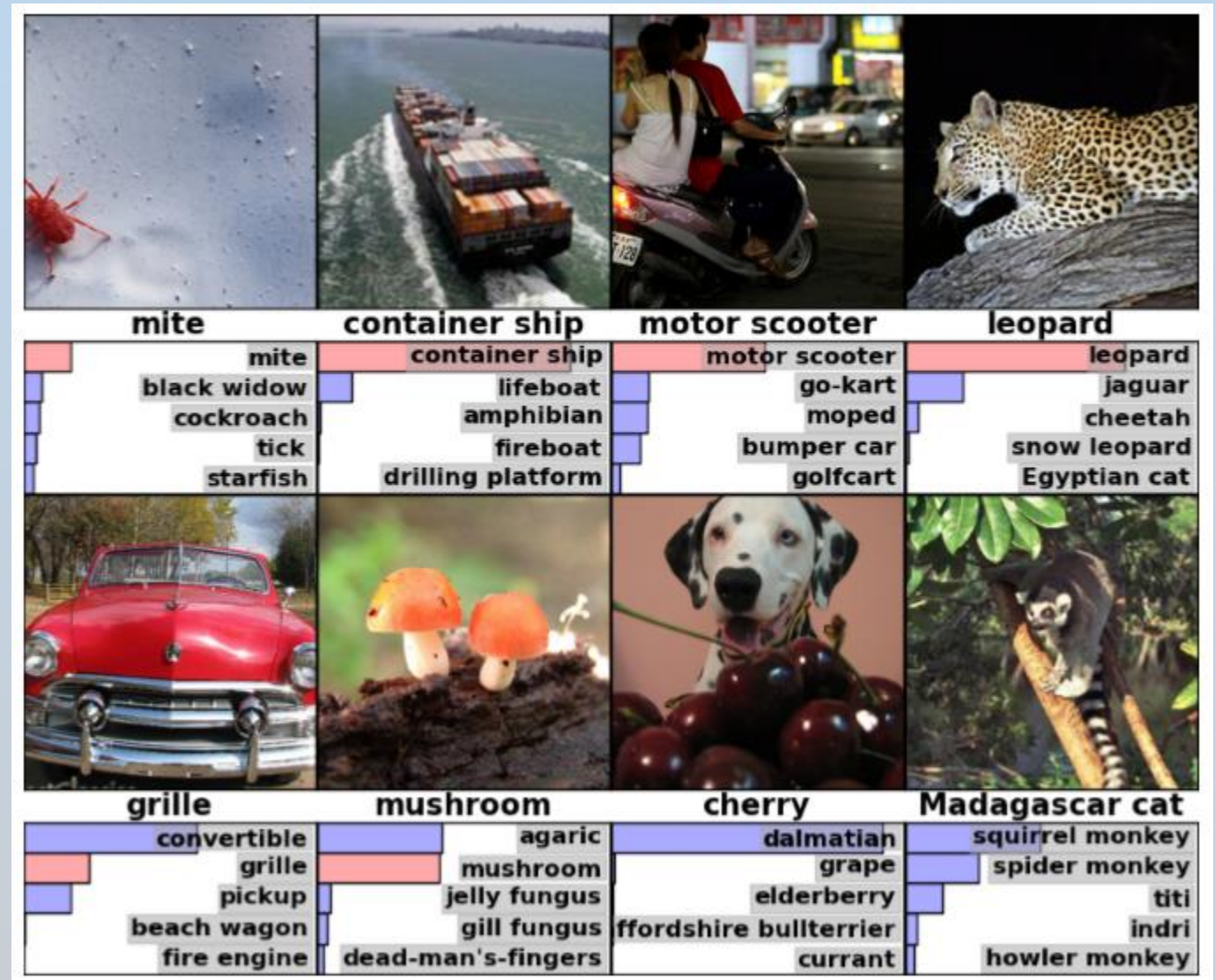
Krizhevsky et al



- Convolutional weights learned by
 - Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.
- Shows that the early CNN layers have learned to detect edges at various orientations

Krizhevsky et al 2012:

- Results trained on 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest.
- 1000 different classes.
- On the test data, they achieved top-1 and top-5 error rates of 37.5% and 17.0%
- Training from scratch took “five to six days on two NVIDIA GTX 580 3GB GPUs.”




Krizhevsky et al

Convolution vision networks

- Run `lecture4-notebook-cnn.ipynb` on MNIST digits
- While running, study the code and try to work out types and dimensions of the following layer types used:
 - Convolution
 - Max-pooling
 - Fully connected
 - Softmax
- Should see recognition rate on test set increase to around 98-99%
- Note this would run much faster on a GPU

Image Shape

```
Reshaped images from (60000, 28, 28) to (60000, 28, 28, 1) so that 'channel'  
dimension exists
```



For MNIST digits, each image is 28*28 size
with just 1 colour channel (black and white image)

Main network definition

We could build our network like this:

```
# Create model (CNN network)
model = keras.Sequential()
model.add(layers.Conv2D(32, kernel_size=(5,5), strides=(1,1), padding="same", activation='relu'))
model.add(layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding="same"))
model.add(layers.Conv2D(64, kernel_size=(5,5), strides=(1,1), padding="same", activation='relu'))
model.add(layers.MaxPool2D(pool_size=(2,2), strides=(2,2), padding="same"))
model.add(layers.Flatten()) # will reshape each pattern from shape [7,7,64] to rank 1
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dropout(rate=0.25))
model.add(layers.Dense(n_classes, activation='softmax'))
```

- Q: How many convolutional layers are present?
- Q: How many Max Pooling?
- Q: What is size of image after each max pooling?
- Q: How many fully connected layers?
- Q: How many inputs to first fully-connected layer?

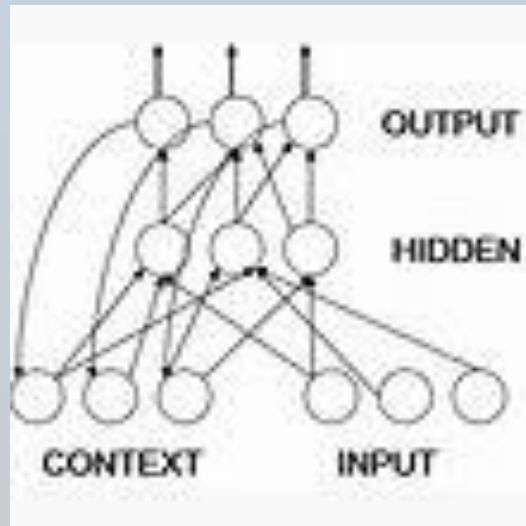
Convolution vision networks

- Work on the exercises at the bottom of the notebook
 - Try the other datasets, CIFAR10 and Fashion
- For further reading on CNNs, see
 - <https://www.tensorflow.org/tutorials/images/cnn>
 - Particularly, see how to use existing pre-trained vision networks and customise them to new tasks (“transfer learning”)

Introduction to Recurrent Neural Networks

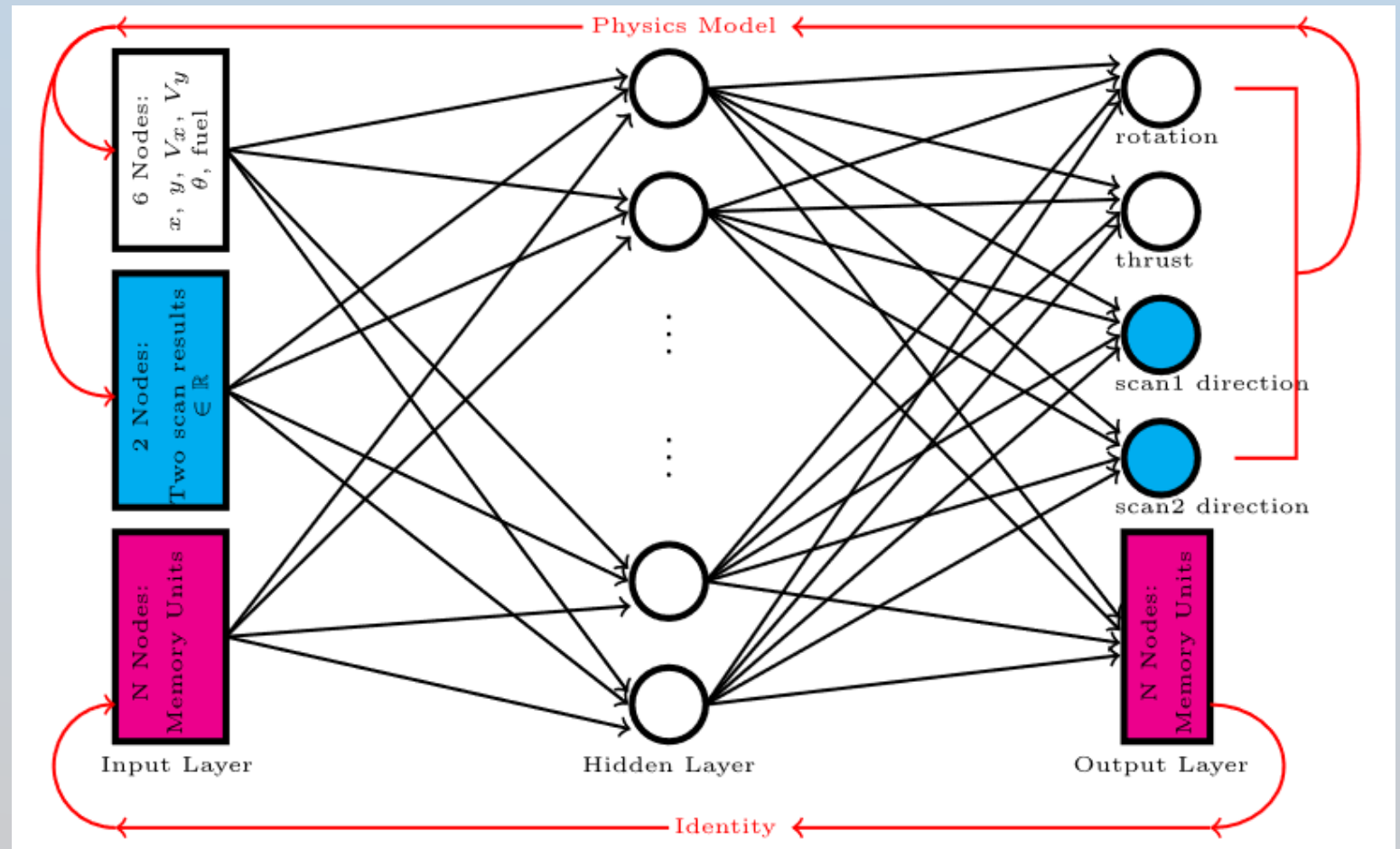
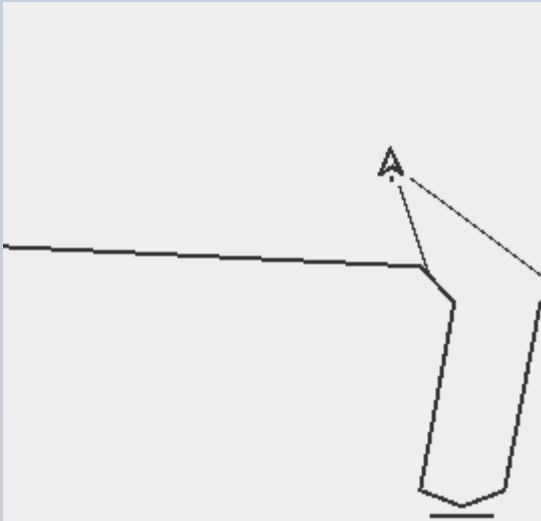
Recurrent Neural Networks

- In a Recurrent Neural Network (RNN) one or more connections point backwards from later layers to earlier layers.
- This allows the RNN to form *memories* of past inputs



Recurrent Neural Networks

- A RNN is what I used to create my Neuropilot demos in 2001-2008
 - We will look at “control problems” more in tomorrow’s course

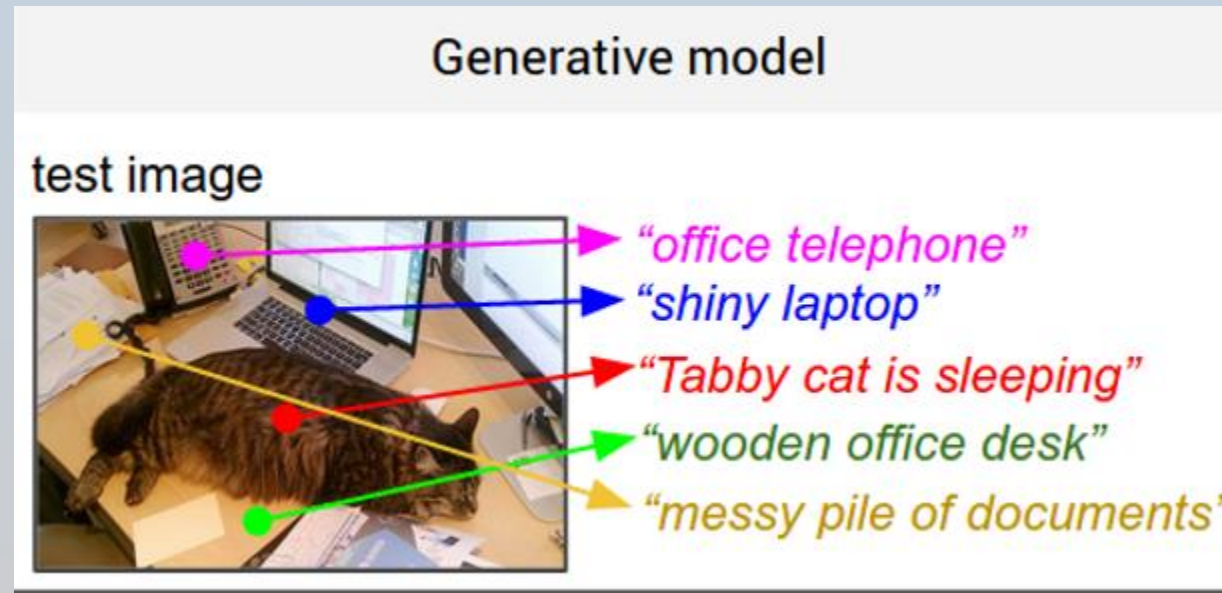


Recurrent Neural Networks

- RNNs are currently used a lot in language tasks:
 - Language Translation
 - Image Captioning
 - Question and Answering
 - Speech recognition
- For more info, see “The Unreasonable Effectiveness of Recurrent Neural Networks” [“http://karpathy.github.io/2015/05/21/rnn-effectiveness/](http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Recurrent Neural Networks

- Image Captioning
 - Karpathy, Andrej, and Li Fei-Fei. "Deep visual-semantic alignments for generating image descriptions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015.



Recurrent Neural Networks

- Question and Answering
 - Samothrakis, S., Vodopivec, T., Fairbank, M., & Fasli, M. Convolutional-Match Networks for Question Answering.
- Sentence: MARY MOVED TO THE BATHROOM. SANDRA JOURNEYED TO THE BEDROOM. **MARY GOT THE FOOTBALL THERE.** JOHN WENT BACK TO THE BEDROOM. MARY JOURNEYED TO THE OFFICE. JOHN JOURNEYED TO THE OFFICE. JOHN TOOK THE MILK. DANIEL WENT BACK TO THE KITCHEN. JOHN MOVED TO THE BEDROOM. DANIEL WENT BACK TO THE HALLWAY. DANIEL TOOK THE APPLE. JOHN LEFT THE MILK THERE. JOHN TRAVELLED TO THE KITCHEN. SANDRA WENT BACK TO THE BATHROOM. DANIEL JOURNEYED TO THE BATHROOM. JOHN JOURNEYED TO THE BATHROOM. MARY JOURNEYED TO THE BATHROOM. SANDRA WENT BACK TO THE GARDEN. SANDRA WENT TO THE OFFICE. DANIEL WENT TO THE GARDEN. SANDRA WENT BACK TO THE HALLWAY. DANIEL JOURNEYED TO THE OFFICE. MARY DROPPED THE FOOTBALL. JOHN MOVED TO THE BEDROOM.
- Question: WHERE WAS THE FOOTBALL BEFORE THE BATHROOM?
- Answer: OFFICE

Recurrent Neural Networks

Data loops around the network in time ticks t :

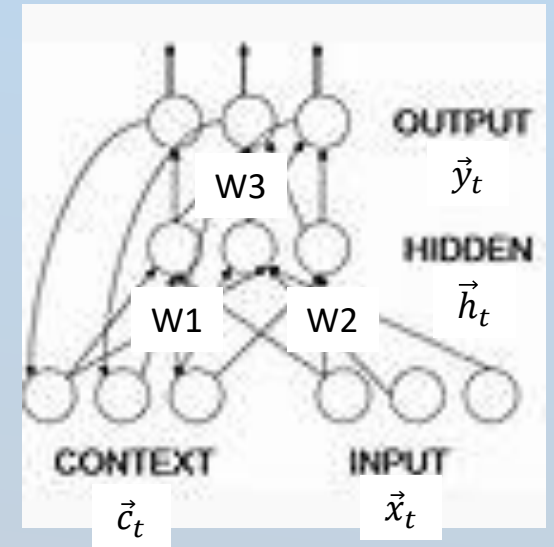
$\vec{c}_0 = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h}_t = g(\vec{c}_t W1 + \vec{x}_t W2 + b1)$ # Hidden layer calculated

$\vec{y}_t = g(\vec{h}_t W3 + b2)$ # Output layer at time t

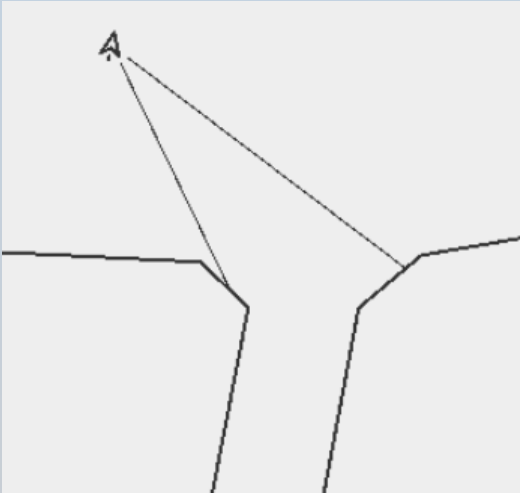
$\vec{c}_{t+1} = \vec{y}_t$ # Output units are copied back to context units ready for next time step
output



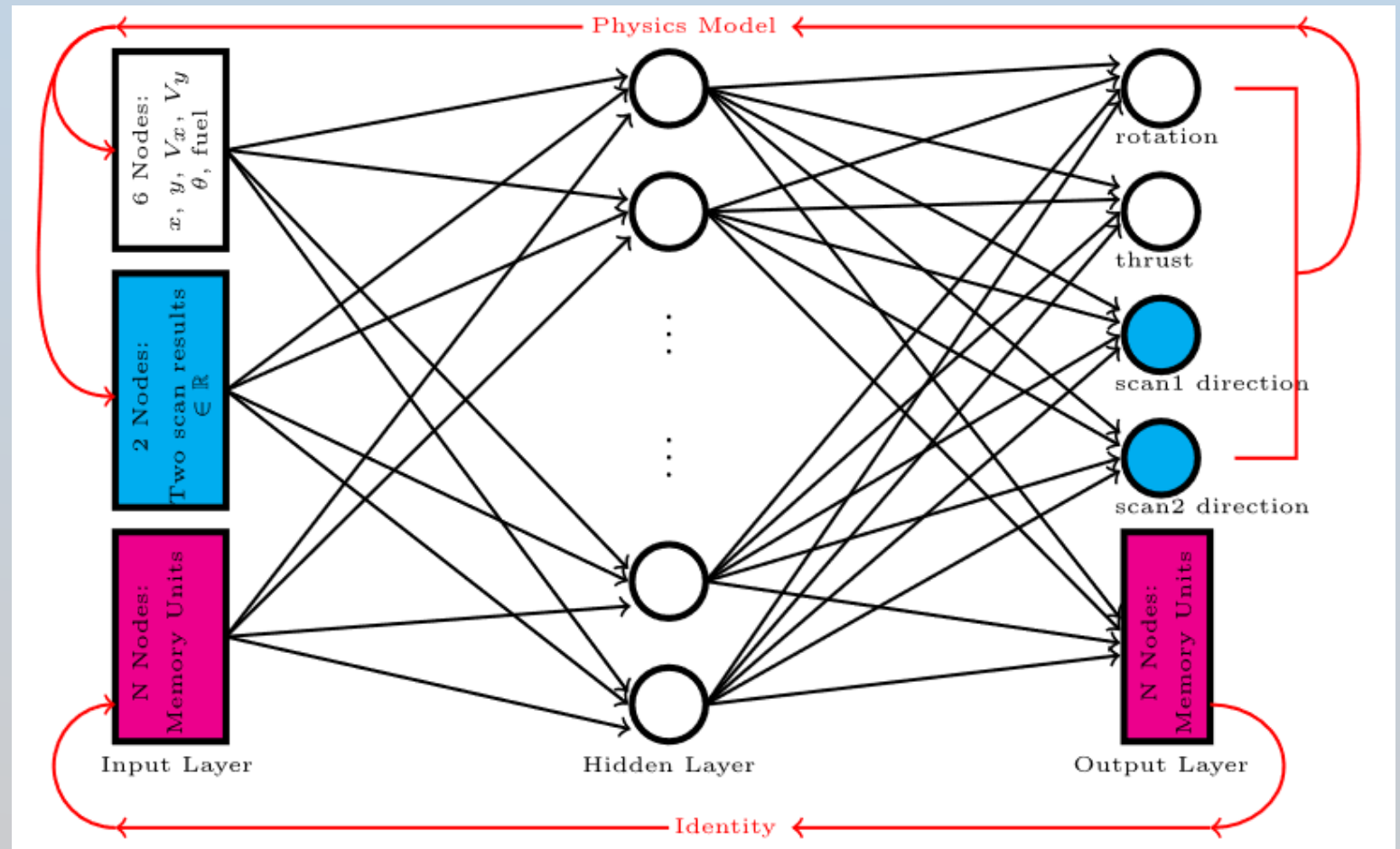
- This results in T input vectors \vec{x}_t being consumed,
 - and T outputs vectors \vec{y}_t being produced
- The context units are also hidden units. These hold the internal state or memories of the RNN.

Recurrent Neural Networks

- Here the pink nodes are the “memory” the spacecraft forms as it flies down tunnels.



- A FFNN cannot solve this



Recurrent Neural Networks

Data loops around the network in time ticks t :

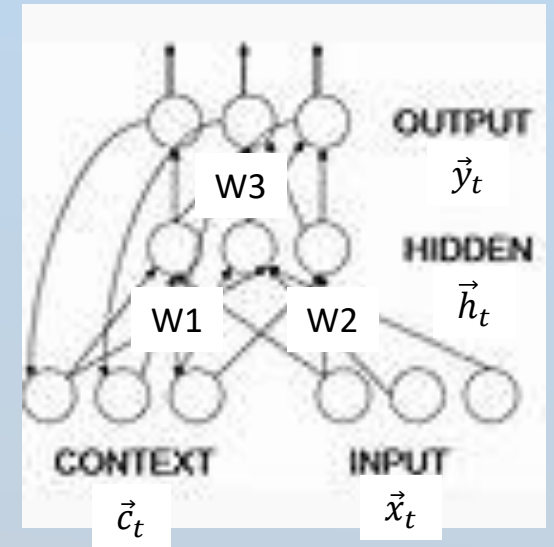
$\vec{c}_0 = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h}_t = g(\vec{c}_t W1 + \vec{x}_t W2 + b1)$ # Hidden layer calculated

$\vec{y}_t = g(\vec{h}_t W3 + b2)$ # Output layer at time t

$\vec{c}_{t+1} = \vec{y}_t$ # Output units are copied back to context units ready for next time step



- Each input vector \vec{x}_t must be the same size (for all t)
- Also if you push through a batch of vectors simultaneously (i.e. n rows of a matrix), then each sequence T needs to be the same length for all n (during training)

Recurrent Neural Networks

Data loops around the network in time ticks t :

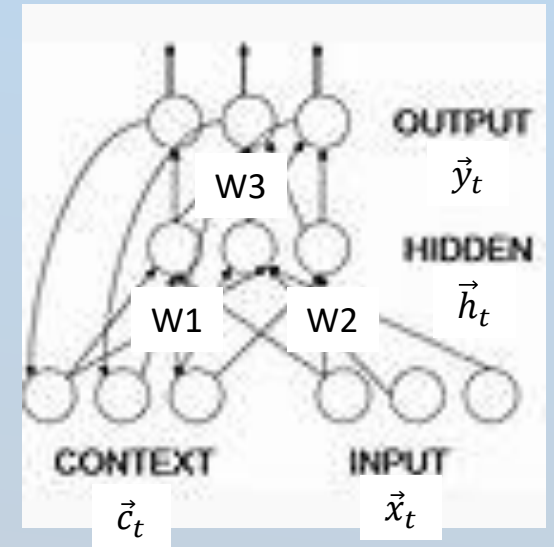
$\vec{c}_0 = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h}_t = g(\vec{c}_t W1 + \vec{x}_t W2 + b1)$ # Hidden layer calculated

$\vec{y}_t = g(\vec{h}_t W3 + b2)$ # Output layer at time t

$\vec{c}_{t+1} = \vec{y}_t$ # Output units are copied back to context units ready for next time



- Many other architectures are possible, e.g. recurrence could go from Hidden back to context instead of from output to context.
- Same weights are used in each tick, so it promotes the “shared weights” philosophy

Q- What change would need making to the pseudocode above if recurrence went back from hidden layer to context units?

Recurrent Neural Networks

Lets consider the case where the **hidden nodes** \rightarrow **context nodes**

$\vec{c}_0 = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h}_t = g(\vec{c}_t W1 + \vec{x}_t W2 + b1)$ # Hidden layer calculated

$\vec{y}_t = g(\vec{h}_t W3 + b2)$ # Output layer at time t

$\vec{c}_{t+1} = \vec{h}_t$ # hidden units are copied back to context units ready for next time step

- We can rename \vec{c}_t as \vec{h}_{t-1} :

$\vec{h}_{-1} = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h}_t = g(\vec{h}_{t-1} W1 + \vec{x}_t W2 + b1)$ # Hidden layer calculated

$\vec{y}_t = g(\vec{h}_t W3 + b2)$ # Output layer at time t

Recurrent Neural Networks

- Since we never need the historic \vec{h}_t vectors (they are hidden), we can rename \vec{h}_t as \vec{h} :

$\vec{h} = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h} \leftarrow g(\vec{h}W1 + \vec{x}_tW2 + b1)$ # Hidden layer updated

$\vec{y}_t \leftarrow g(\vec{h}W3 + b2)$ # Output layer at time t

Recurrent Neural Networks

$\vec{h} = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$$\vec{h} \leftarrow g(\vec{h}W1 + \vec{x}_tW2 + b1)$$

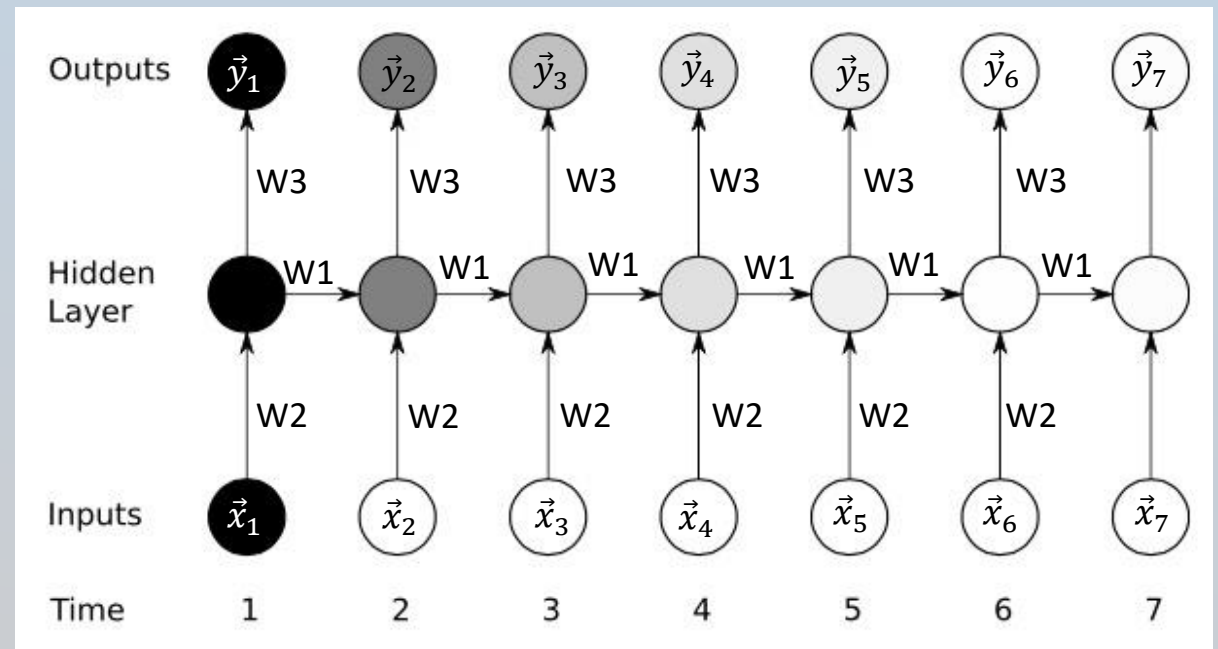
$$\vec{y}_t \leftarrow g(\vec{h}W3 + b2)$$

- You'll often see a RNN presented like this:

- “Unrolled in time”

- This is a “many-to-many” RNN.
- If \vec{y}_1 to \vec{y}_6 were ignored then we'd have a “many to one” RNN.

Q. Are these the same exactly?



Recurrent Neural Networks

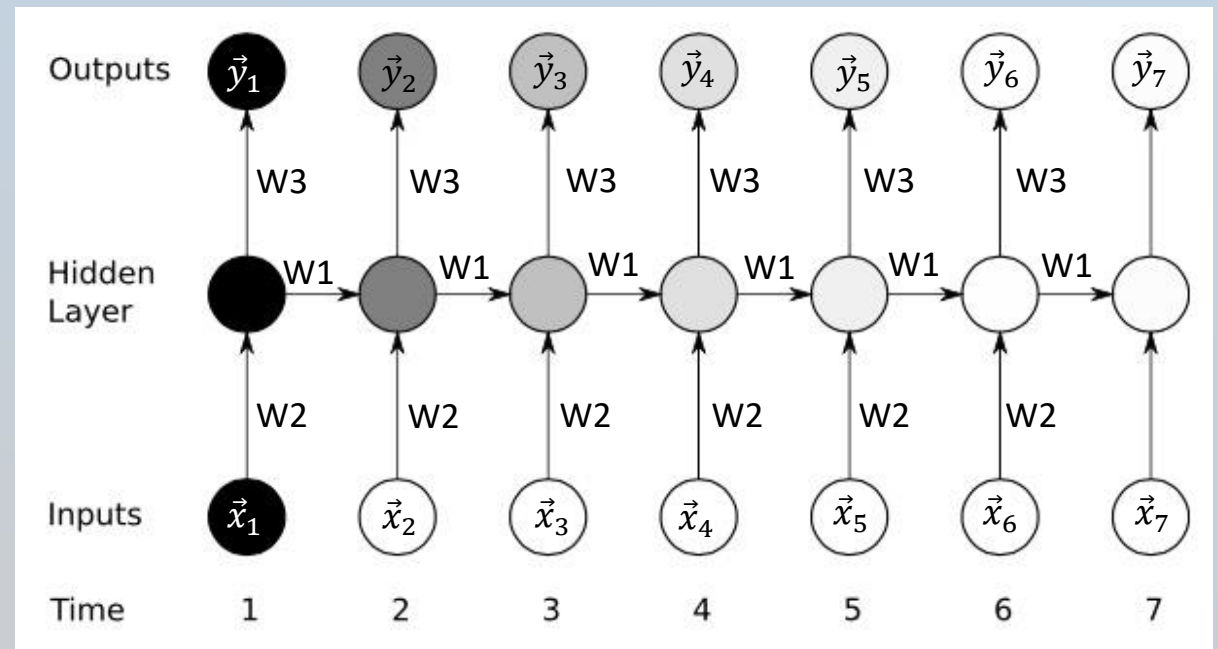
$\vec{h} = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$$\vec{h} \leftarrow g(\vec{h}W1 + \vec{x}_tW2 + b1)$$

$$\vec{y}_t \leftarrow g(\vec{h}W3 + b2)$$

- You'll often hear a RNN is "Trained by backpropagation through time"
 - We don't need to worry about this as again autodiff takes care of it for us, behind the scenes.



Recurrent Neural Networks

$\vec{h} = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h} \leftarrow g(\vec{h}W1 + \vec{x}_tW2 + b1)$ # Hidden layer updated

$\vec{y}_t \leftarrow g(\vec{h}W3 + b2)$ # Output layer at time t

TensorFlow code:

```
layer_recurrent=tf.keras.layers.Dense(numHiddenNodes, activation=tf.nn.sigmoid)
layer_output_layer=tf.keras.layers.Dense(num_output_nodes, activation=tf.nn.sigmoid)
y_list=[]
hiddenState= tf.constant(0., shape=[mini_batch_size, numHiddenNodes])
for i in range(0,seqLength):
    x_t=input_sequence[:,i,:] # pulls out the ith time-step input
    x=tf.concat([x_t, hiddenState],axis=1) # appends side-by-side
    hiddenState=layer_recurrent (x)
    y_t=layer_output_layer(hiddenState)
    y_list.append(y_t)
```

Note: Input_sequence should have shape
[batch_size, sequence_length, num_inputs]

Recurrent Neural Networks

$\vec{h} = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h} \leftarrow g(\vec{h}W1 + \vec{x}_tW2 + b1)$ # Hidden layer updated

$\vec{y}_t \leftarrow g(\vec{h}W3 + b2)$ # Output layer at time t

Note: Output sequence y will have shape
[batch_size, sequence_length, num_outputs]

Or, using higher-level API functions:

```
layer_recurrent=tf.keras.layers.SimpleRNN(numHiddenNodes, return_sequences=True)
layer_output_layer=tf.keras.layers.Dense(num_output_nodes, activation=tf.nn.sigmoid)
full_recurrent_keras_model=tf.keras.Sequential([layer_recurrent,layer_output_layer])

y=full_recurrent_keras_model(input_sequence)
```

Recurrent Neural Networks

$\vec{h} = \vec{0}$ #Context units initialised with zeros

for t in range(T):

$\vec{h} \leftarrow g(\vec{h}W1 + \vec{x}_tW2 + b1)$ # Hidden layer updated

$\vec{y}_t \leftarrow g(\vec{h}W3 + b2)$ # Output layer at time t

- As data loops around the network each tick, old memories tend to decay.
- It was difficult for RNNs to remember things for more than 10 timesteps.
- GRU and LSTM nodes were invented to improve on this problem

- Just change:

`layer_recurrent=tf.keras.layers.SimpleRNN(numHiddenNodes,return_sequences=True)`

- To:

`layer_recurrent=tf.keras.layers.LSTM(numHiddenNodes,return_sequences=True)`

- See https://en.wikipedia.org/wiki/Long_short-term_memory

Recurrent Neural Networks

- Example: Bit-sequence memoriser problem
 - Run notebook [lecture4-notebook-rnn-bit-sequence](#)

In this task the RNN receives a sequence of random bits

- e.g. 0,1,0,1,1,1,0,0,1,0,0,0

And must output the same sequence of bits delayed by a set amount, e.g.

- e.g. 1,1,1,0,0,1,0,0,0
- Vary the variable delayLength to make the problem harder/easier.
- In Fig. 8 of our recent paper on [Target Space](#), we show how to solve this problem up to delay lengths approaching 200.
 - LSTM only manages 60!

Recurrent Neural Networks

- Example: RNN with LSTM units applied to the MNIST dataset
 - Run notebook [lecture4-notebook-rnn-mnist](#)
- The MNIST images are 28*28 pixels
- The RNN takes in one row at a time of each image, and then 28 rows fed in sequentially to the RNN.
 - n_steps=28
 - n_inputs=28

Q: Is this RNN many-to-many or many-to-one?

- Impressively, this RNN scores $\approx 98\%$ on MNIST without any regularization or dropout.

Further Reading

TensorFlow:

- <https://www.tensorflow.org/tutorials/>

RNNs:

- Word2Vec + Embeddings – a really important concept in training NNs to understand language.
 - https://www.tensorflow.org/text/guide/word_embeddings
- “The Unreasonable Effectiveness of Recurrent Neural Networks” by Andrej Karpathy <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

Thank you for your attention

And, if you find any bugs in the slides please email me.