

Convolutional Neural Networks

Applied Machine Learning, KU
Daniel Murnane - May 6th, 2026



Introduction & Goals

- Nice to meet you!
- I work on ML for high energy physics @ NBI
- Goals for today:
 - Understand how to represent an image
 - See how images have traditionally been processed
 - Get intuition for the power of a convolution
 - Practice “convolution arithmetic”
 - Get a feeling for where CNNs fit into the rest of ML
 - See a few examples of CNNs in action
- Have borrowed content from [MIT Intro to Deep Learning](#); and Julius Kirkegaard’s slides from 2022

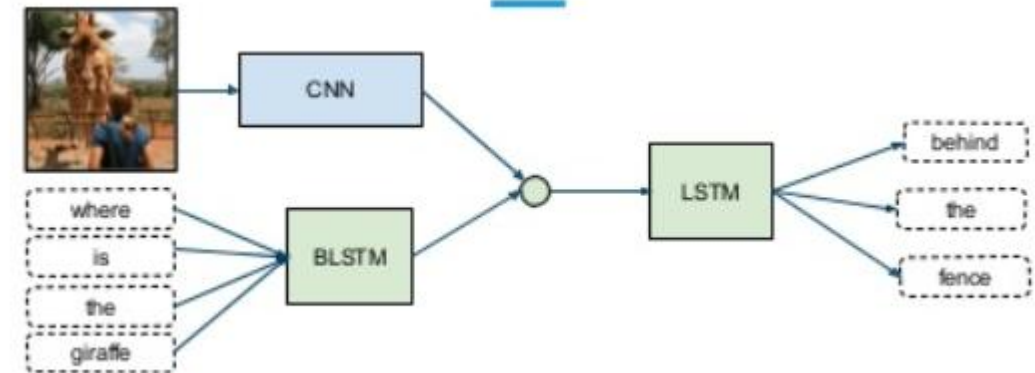


Computer Vision



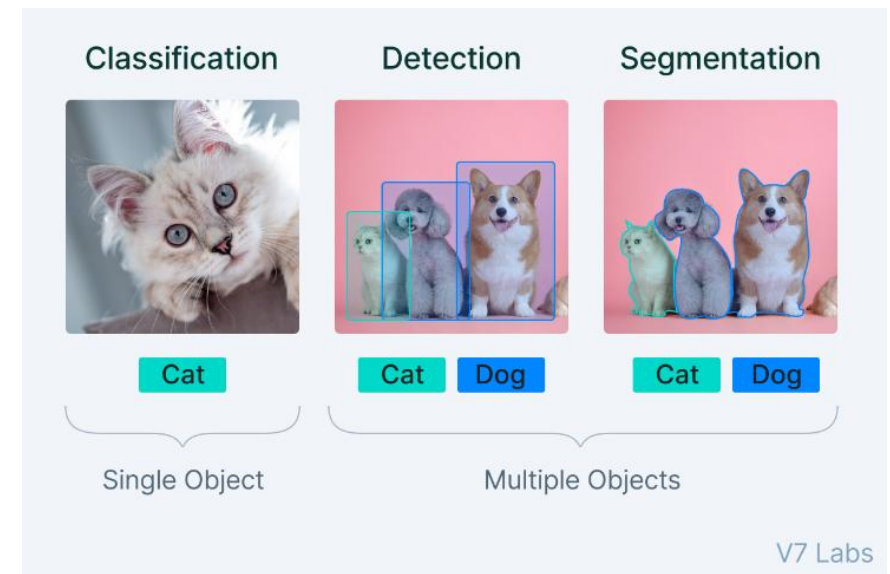
Modalities in ML

- So far, have seen “tabular data”
- Quick test for “tabularity”: Would this work well in Excel?
- The world contains more *modalities* than tabular:
 - **Images** (today!) – i.e. vision
 - Sequences (Next week!) – i.e. 1-dimensional time and/or causality
 - Point clouds and graphs (Wednesday morning!) – Multi-dimensional time and/or causality
- Most other modalities and senses can be represented as combinations of these:
 - Video = Images + Sequence
 - Robotic action = Tabular + Sequence/Graph
 - Audio = Images (waveform) or Sequence (fourier transform)
 - Biological & Chemical = Graph (Protein/DNA language models)
- Others might come along... Touch? Smell?
- The hot thing now is combining modalities – which is straightforward once you understand how to consume each individually



Computer Vision

- Today's focus is vision
- Many diverse tasks: scene reconstruction, object detection, event detection, video tracking, object recognition, 3D pose estimation, learning, indexing, motion estimation, 3D scene modelling, and image restoration
- A beautiful template for the rest of machine learning
- Conventional CV: A huge collection of hand-engineered techniques on a task-by-task basis
- ML CV: A small set of models that can be used for many tasks



Images as Numbers

- Need way to consume images and apply functions to them
- Most typical way: represent each pixel as a vector
- Greyscale images only need a single value in the vector: the brightness of the pixel
- Colour images need 3 or 4 values in the vector: [R, G, B] or [C, M, Y, K]
- As in **every other ML project**, we normalize those inputs! Conventionally given to us as values [0, 255], therefore divide by 255



157	153	174	168	150	152	129	151	172	161	155	156			
195	182	163	74	75	62	33	17	110	230	180	154			
180	180	50	14	34	6	10	33	48	106	159	181			
206	109	5	124	131	111	120	204	166	15	56	180			
194	68	137	251	237	239	239	228	227	87	71	201			
172	105	207	233	233	214	220	239	228	98	74	206			
188	88	179	209	185	215	211	158	139	75	20	169			
189	97	165	84	10	168	134	11	31	62	22	148			
199	168	191	193	155	227	178	143	182	106	36	190			
205	174	155	252	236	231	149	178	228	43	95	234			
190	216	116	149	236	187	85	150	79	38	218	241			
190	224	147	108	227	210	127	102	36	101	255	224			
190	214	173	66	193	143	95	90	2	108	249	215			
187	196	235	75	1	81	47	0	6	217	255	211			
183	202	237	145	0	0	12	108	200	138	243	236			
195	206	128	207	177	121	123	200	175	19	96	218			

157	153	174	168	150	152	129	151	172	161	155	156			
195	182	163	74	75	62	33	17	110	230	180	154			
180	180	50	14	34	6	10	33	48	106	159	181			
206	109	5	124	131	111	120	204	166	15	56	180			
194	68	137	251	237	239	239	228	227	87	71	201			
172	105	207	233	233	214	220	239	228	98	74	206			
188	88	179	209	185	215	211	158	139	75	20	169			
189	97	165	84	10	168	134	11	31	62	22	148			
199	168	191	193	155	227	178	143	182	106	36	190			
205	174	155	252	236	231	149	178	228	43	95	234			
190	216	116	149	236	187	85	150	79	38	218	241			
190	224	147	108	227	210	127	102	36	101	255	224			
190	214	173	66	193	143	95	90	2	108	249	215			
187	196	235	75	1	81	47	0	6	217	255	211			
183	202	237	145	0	0	12	108	200	138	243	236			
195	206	128	207	177	121	123	200	175	19	96	218			



A Typical Computer Vision Task

- Let's consider a typical task in CV: classification of the foreground object in an image
- How does a human do this?
- High level feature detection
- We are very sophisticated!
- Imagine if we had to look pixel-by-pixel...

High Level Feature Detection

Let's identify key features in each image category



Nose,
Eyes,
Mouth



Wheels,
License Plate,
Headlights



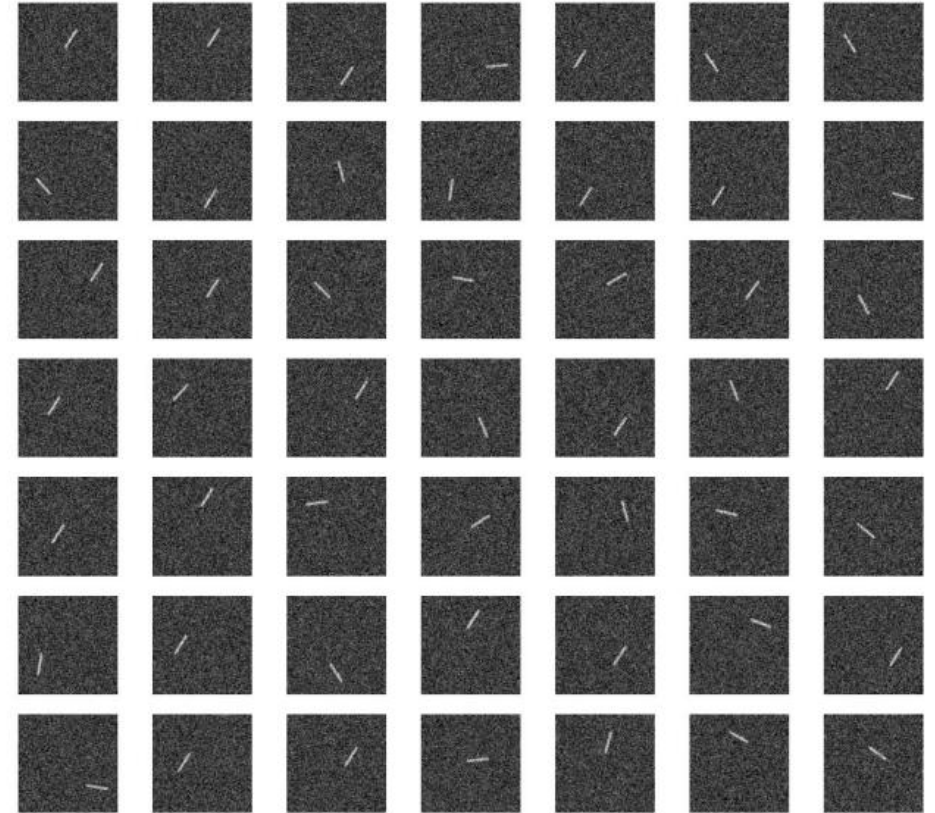
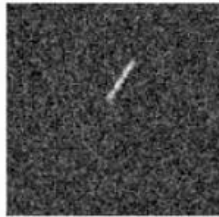
Door,
Windows,
Steps



The Almighty Filter

- Consider a very simple classification problem:

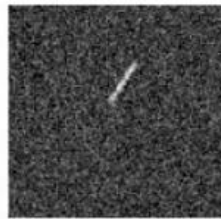
Find images with angle:



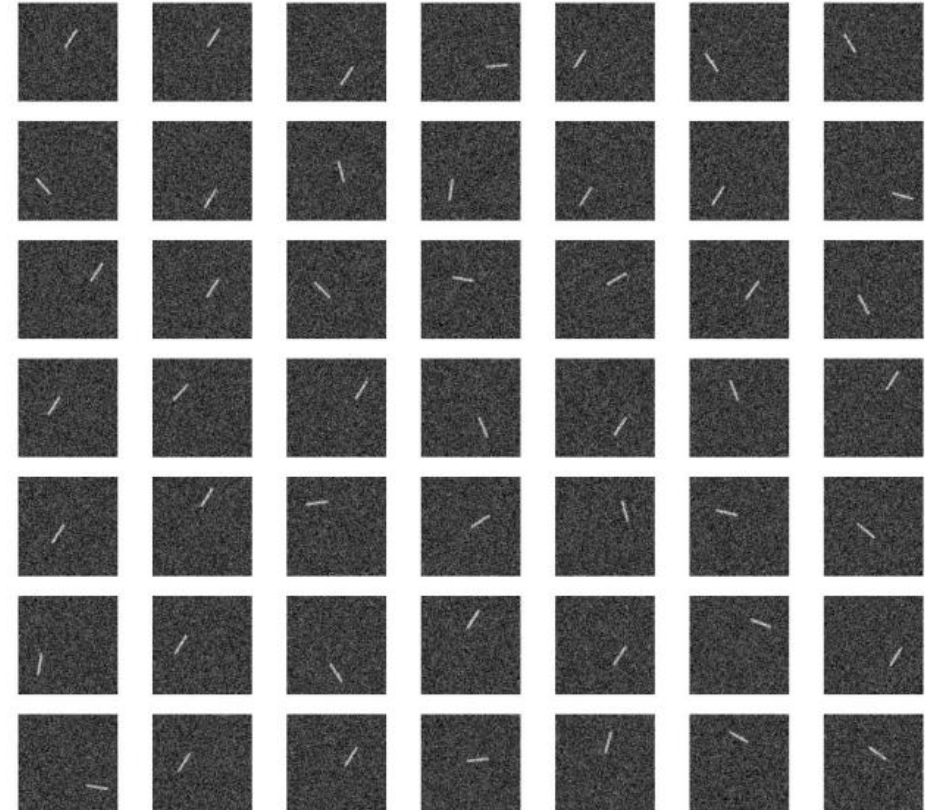
The Almighty Filter

- Consider a very simple classification problem:

Find images with angle:



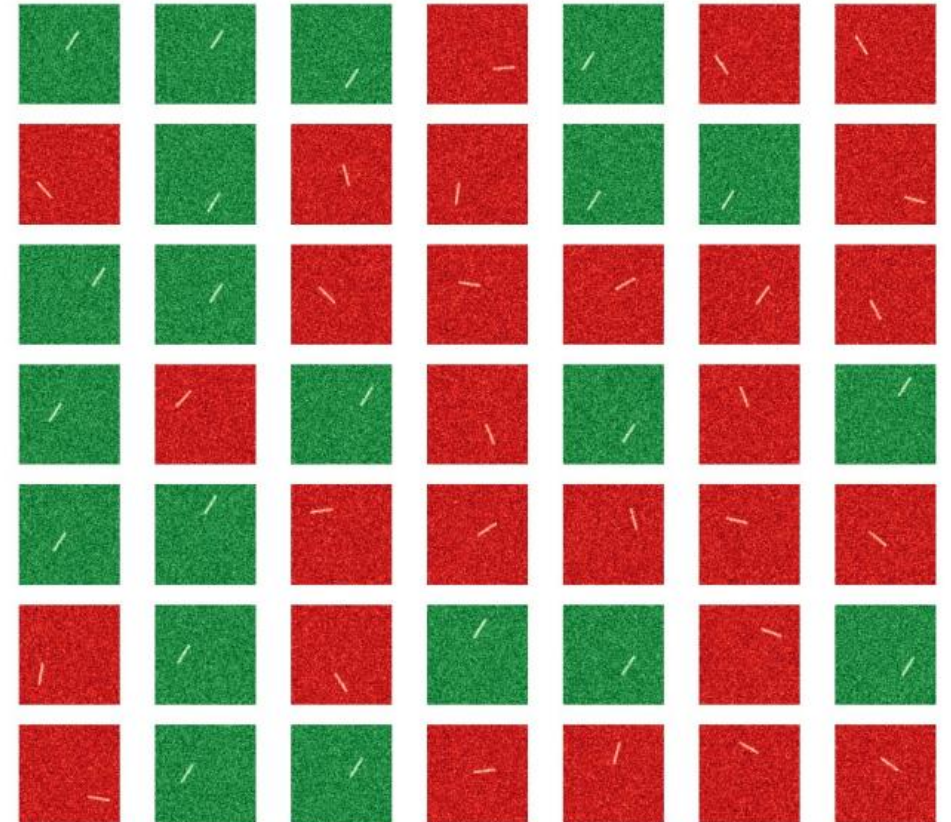
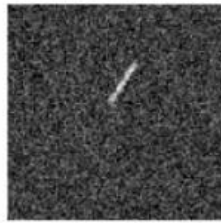
Let's try this on
the board first



The Almighty Filter

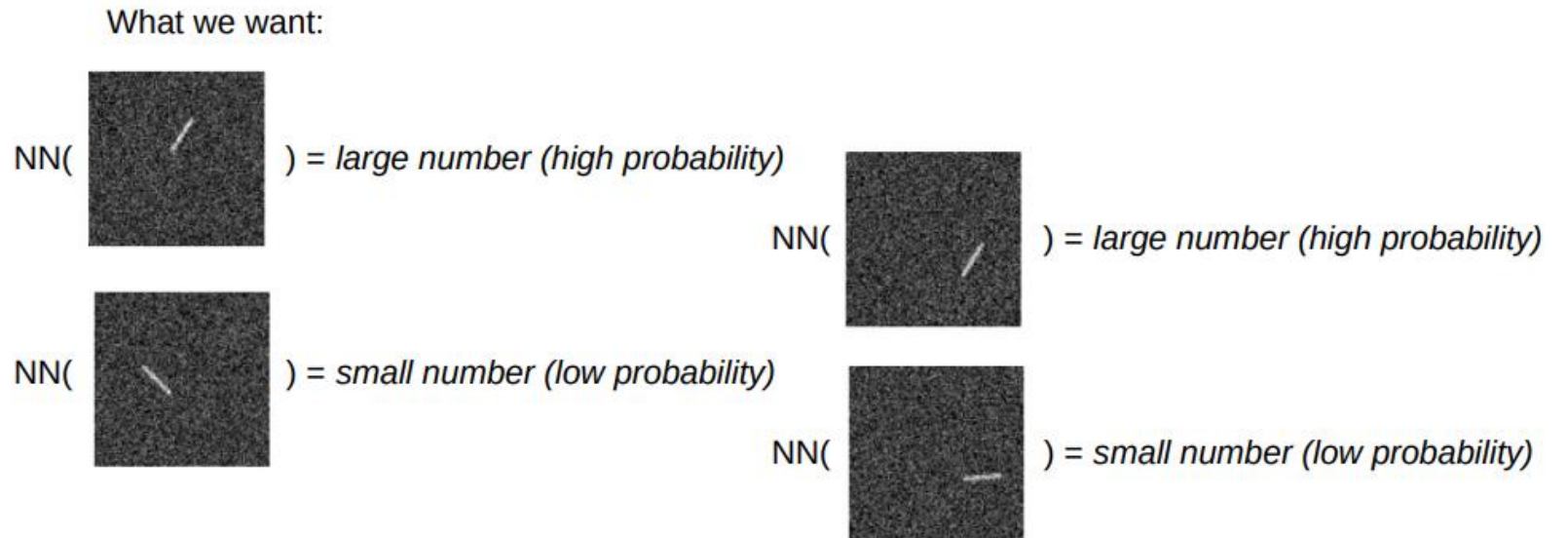
- Consider a very simple classification problem:
- Note that we don't care where the feature is *within* the image

Find images with angle:



The Almighty Filter

- Consider a very simple classification problem:
- Note that we don't care where the feature is *within* the image



The Almighty Filter

Define a *kernel*, $k =$



(color just for visualisation purposes.
Think of this a matrix of numbers)

- A “kernel”, or a “filter” is a hypothesis about a feature of interest to the task
- We are hand-engineering here, so we can use our intuition
- A kernel is a matrix of the same size of the neighbourhood of pixels we want to “test our hypothesis” on

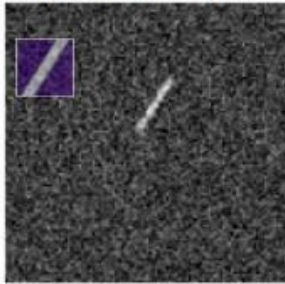


The Almighty Filter

Define a kernel, $k =$



Place kernel somewhere on image, multiply and sum:

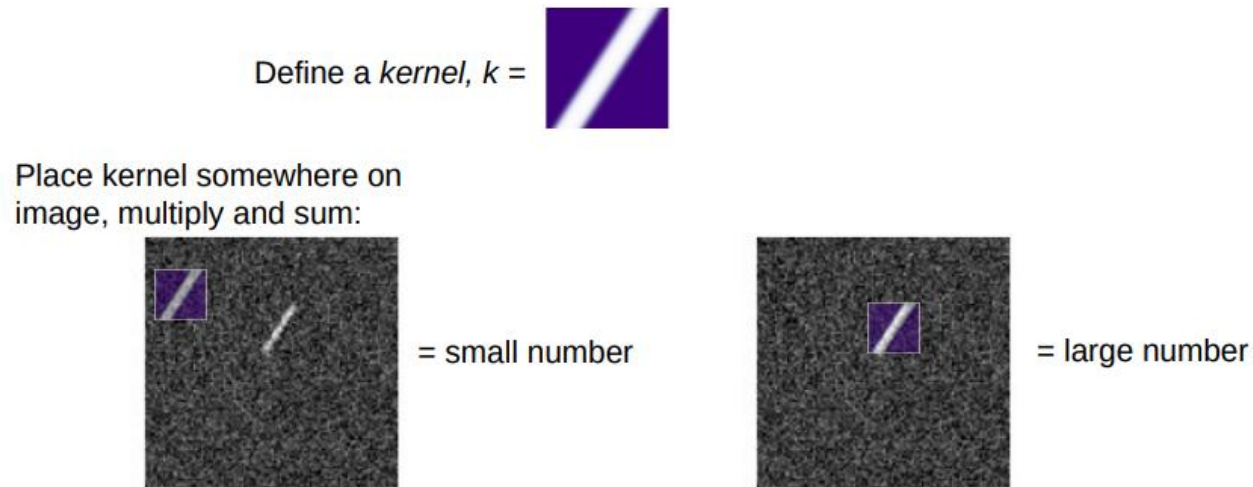


= small number

- A “kernel”, or a “filter” is a hypothesis about a feature of interest to the task
- We are hand-engineering here, so we can use our intuition
- A kernel is a matrix of the same size of the neighbourhood of pixels we want to “test our hypothesis” on



The Almighty Filter



- A “kernel”, or a “filter” is a hypothesis about a feature of interest to the task
- We are hand-engineering here, so we can use our intuition
- A kernel is a matrix of the same size of the neighbourhood of pixels we want to “test our hypothesis” on

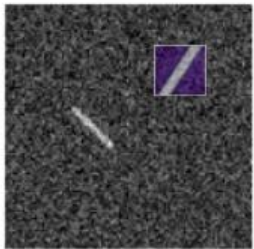


The Almighty Filter

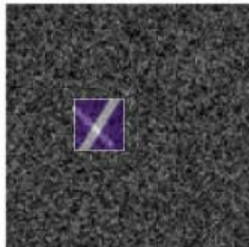
Define a *kernel*, $k =$



What happens for a “wrong” image?



= small number



= small(ish) number

- A “kernel”, or a “filter” is a hypothesis about a feature of interest to the task
- We are hand-engineering here, so we can use our intuition
- A kernel is a matrix of the same size of the neighbourhood of pixels we want to “test our hypothesis” on



The Almighty Filter

- Applying the filter is actually just a sum over elementwise multiplication:

Image neighbourhood

0.1	0.1	0.9
0.1	0.9	0.1
0.9	0.1	0.1

Filter

0.1	0.1	0.9
0.1	0.9	0.1
0.9	0.1	0.1

$$\sum (\cdot) = \sum () = 2.43$$

0.01	0.01	0.81
0.01	0.81	0.01
0.81	0.01	0.01



The Almighty Filter

- Applying the filter is actually just a sum over elementwise multiplication:

Image neighbourhood

0.9	0.1	0.1
0.1	0.9	0.1
0.1	0.1	0.9

Filter

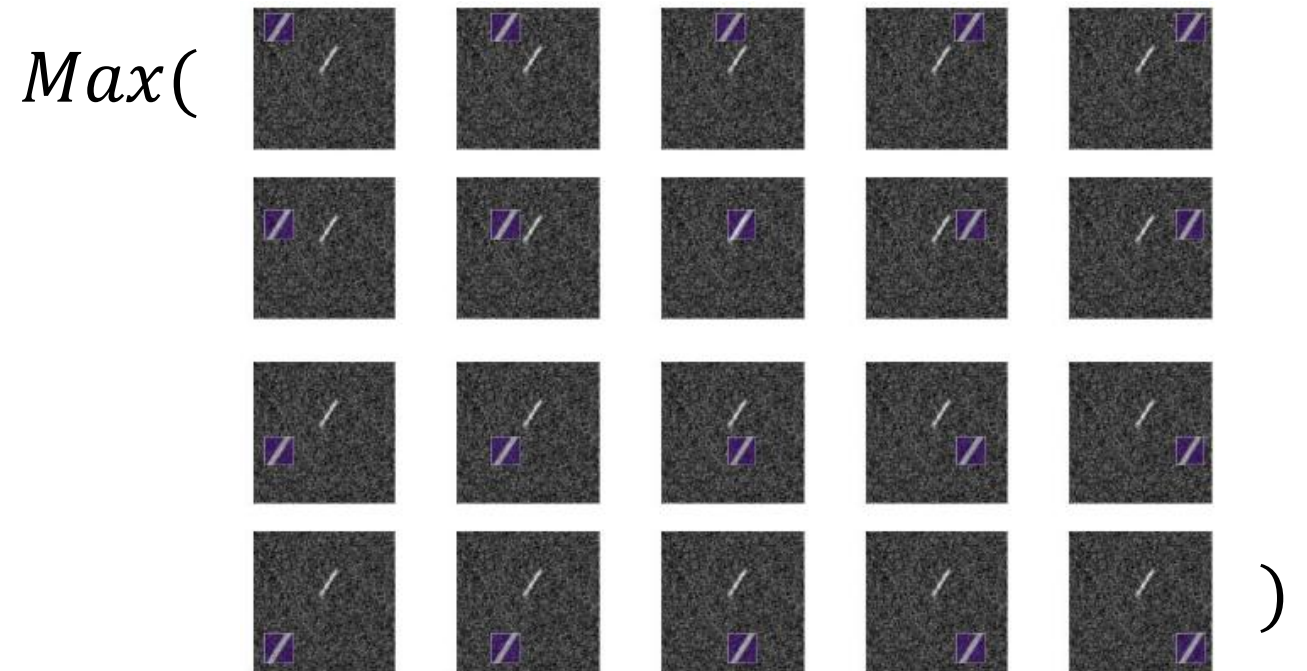
0.1	0.1	0.9
0.1	0.9	0.1
0.9	0.1	0.1

$$\sum \left(\begin{matrix} 0.9 & 0.1 & 0.1 \\ 0.1 & 0.9 & 0.1 \\ 0.1 & 0.1 & 0.9 \end{matrix} \cdot \begin{matrix} 0.1 & 0.1 & 0.9 \\ 0.1 & 0.9 & 0.1 \\ 0.9 & 0.1 & 0.1 \end{matrix} \right) = \sum \left(\begin{matrix} 0.09 & 0.01 & 0.09 \\ 0.01 & 0.81 & 0.01 \\ 0.09 & 0.01 & 0.09 \end{matrix} \right) = 1.21$$



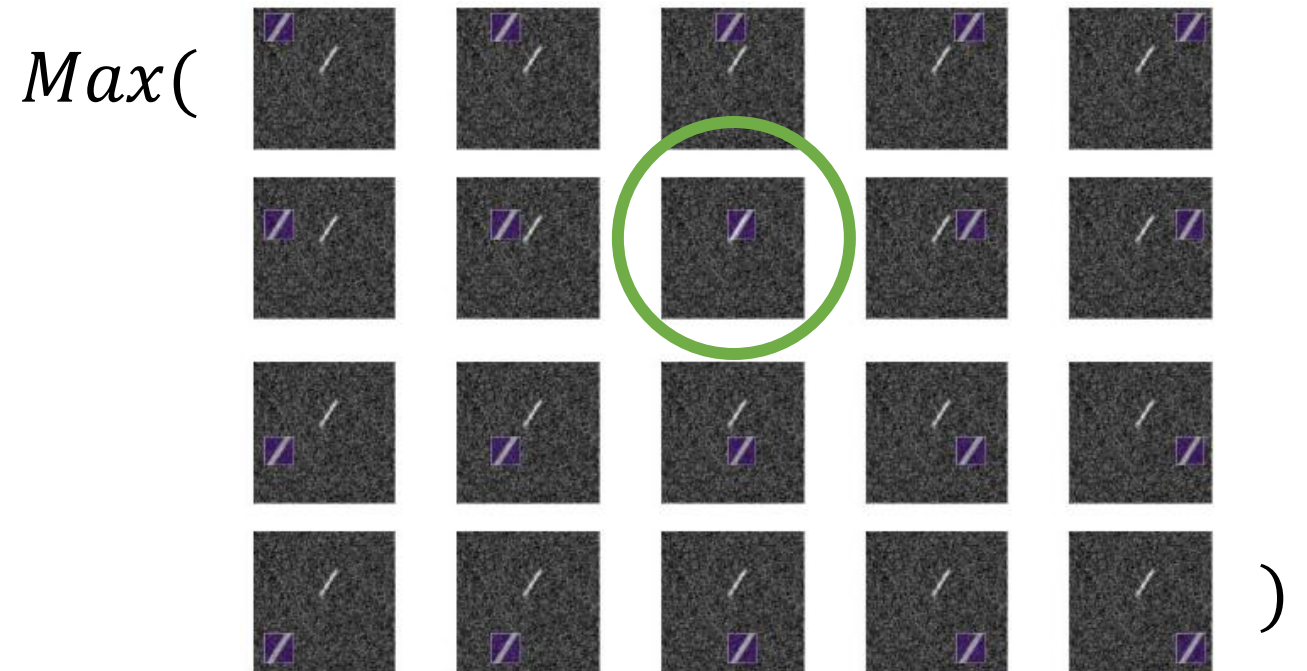
Max Pooling

- Let's slide the filter across all neighbourhoods in the image
- Keep track of the filter score for each neighbourhood



Max Pooling

- Let's slide the filter across all neighbourhoods in the image
- Keep track of the filter score for each neighbourhood
- Find the maximum score across the whole image



Convolutional Filter

This is precisely what a convolution does!

$$\text{NN}(\text{img}) = \max(\text{conv2D}(\text{img}, \text{filter}))$$

We have our function!

Indeed:

$$\begin{aligned} \text{NN}(\text{img}_1) &= \text{large} & \text{NN}(\text{img}_2) &= \text{large} \\ \text{NN}(\text{img}_3) &= \text{small} & \text{NN}(\text{img}_4) &= \text{small} \end{aligned}$$

- But, we haven't trained anything!
- This is a "hand-crafted" filter:

0.1	0.1	0.9
0.1	0.9	0.1
0.9	0.1	0.1

- What if we made the 9 values in the matrix learnable parameters?
- Then we could find the filter that minimises a loss function



Trainable Convolutional Filter

$$\text{NN}(\text{img}) = \max(\text{conv2D}(\text{img}, \text{kernel}))$$

We handcrafted the kernel. In CNNs we train to choose the best kernels

```
net = nn.Sequential(nn.Conv2d(1, 1, 11),  
                  nn.AdaptiveMaxPool2d(1),  
                  nn.Linear(1, 1))
```

```
# Soft-max probabilities -> cross entropy  
p = torch.exp(pred) / (torch.exp(pred) + torch.exp(-pred))  
loss = -torch.mean(labels * torch.log(p) + (1 - labels) * torch.log(1 - p)) # (this can be done smarter!,  
                                                                           # no reason to take exp, then log)
```

Run code!

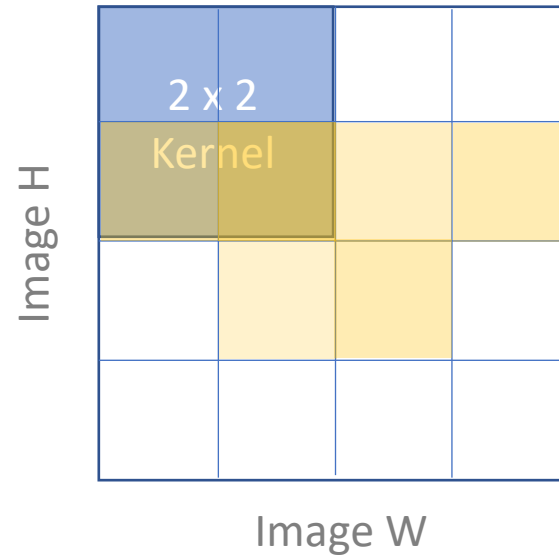


(slightly smarter than ours!,
albeit noisy)



Adding a Latent Space

- Currently everything is “flat” – each pixel multiplied by a single learnable parameter

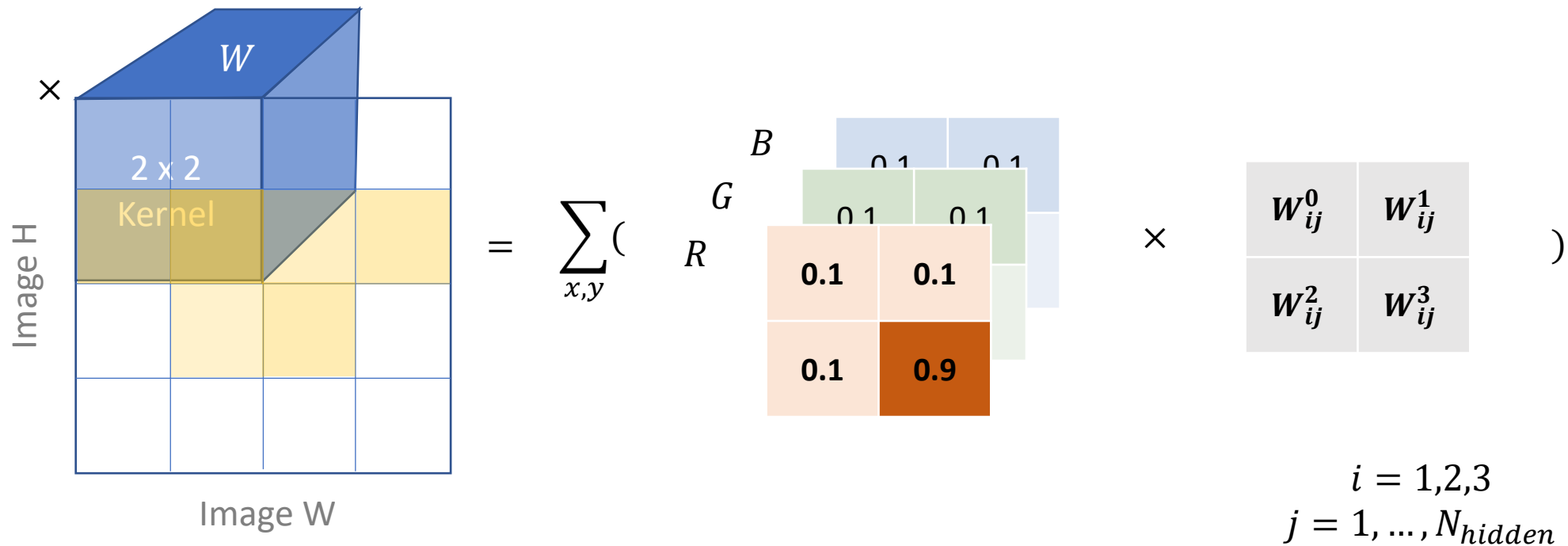


- But recall neural networks can learn *many* parameters



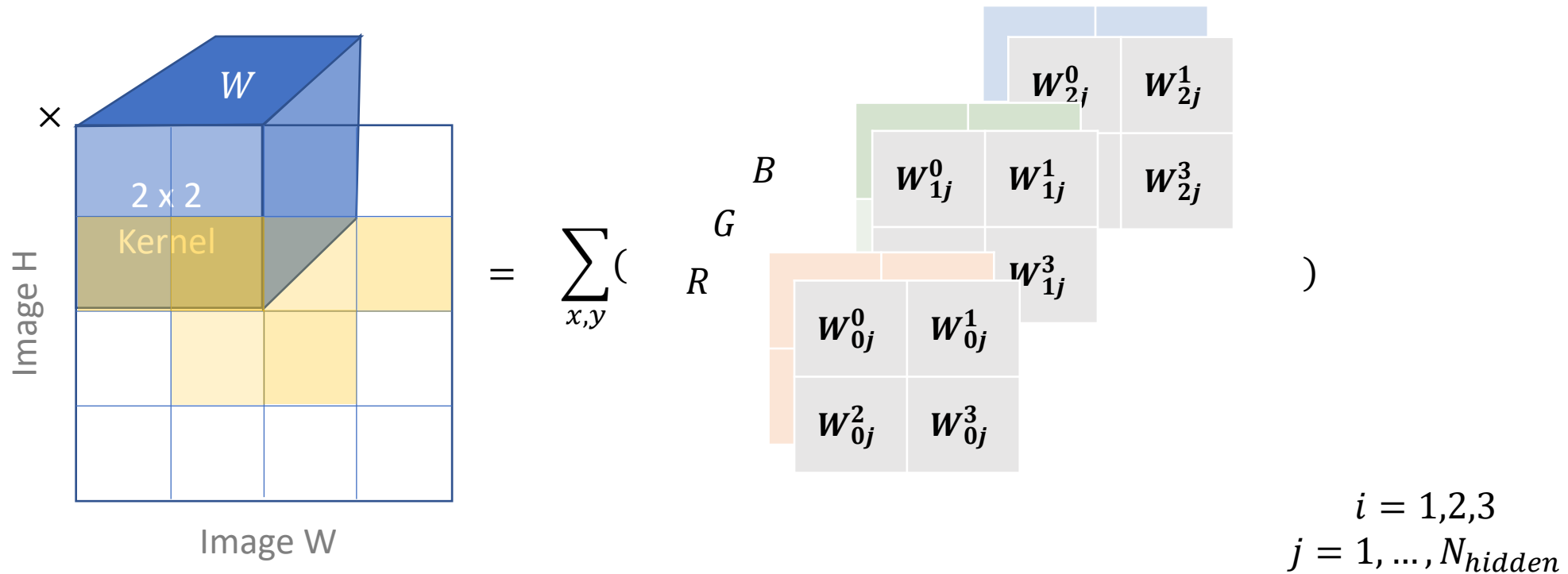
Adding a Latent Space

- We can add as much depth to the convolution as we want
- This is a bit hard to understand! There are two levels of matrices...



Adding a Latent Space

- We can add as much depth to the convolution as we want
- This is a bit hard to understand! There are two levels of matrices...



Why not just use a feedforward layer?

- You might ask...
- We have this magical block box: fully-connected, feedforward neural network
- We know it can approximate any function, provided it is wide and deep enough
- Why not “flatten” the image, then simply feed in the list of pixels as “tabular data” and run through a FFNN?
- First answer: Translational invariance

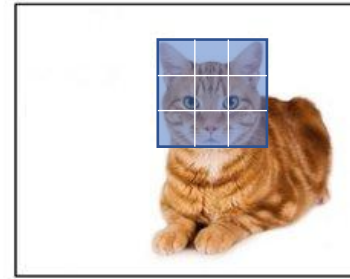
[Hornick et al, 1989](#)



Symmetries, Invariances & Inductive Bias

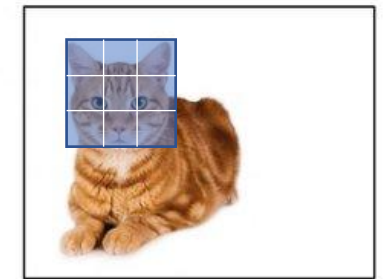
- We *know* that the location of a feature shouldn't matter in an image – we can translate a cat in x and y and it's still a cat
- In a CNN, the *same* convolutional filter is applied in each neighbourhood (aka “shared weights”), so will find a cat in any part of the image: it is translationally invariant
- A FFNN, on the other hand, has a dedicated parameter for *each* pixel – it has no concept of “nearness” or a “neighbourhood”
- We have introduced an “inductive bias” by choosing a function (the filter) that is translationally invariant
- A CNN is able to train with *much* less data than a FFNN, and is less prone to overfitting: Because we used our human intuition of translational symmetry

“Cat-face” filter



Cat

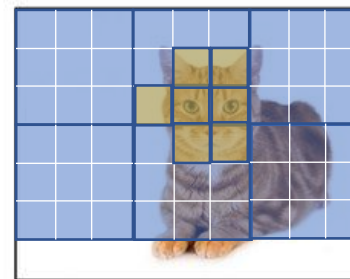
Same “Cat-face” filter



Cat

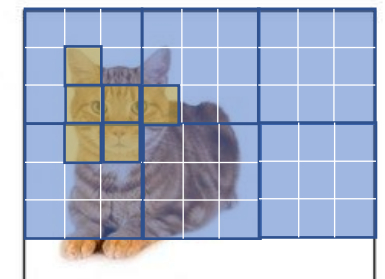
CNN

“Cat-face” neurons



Cat

Different “Cat-face” neurons



Cat

FFNN

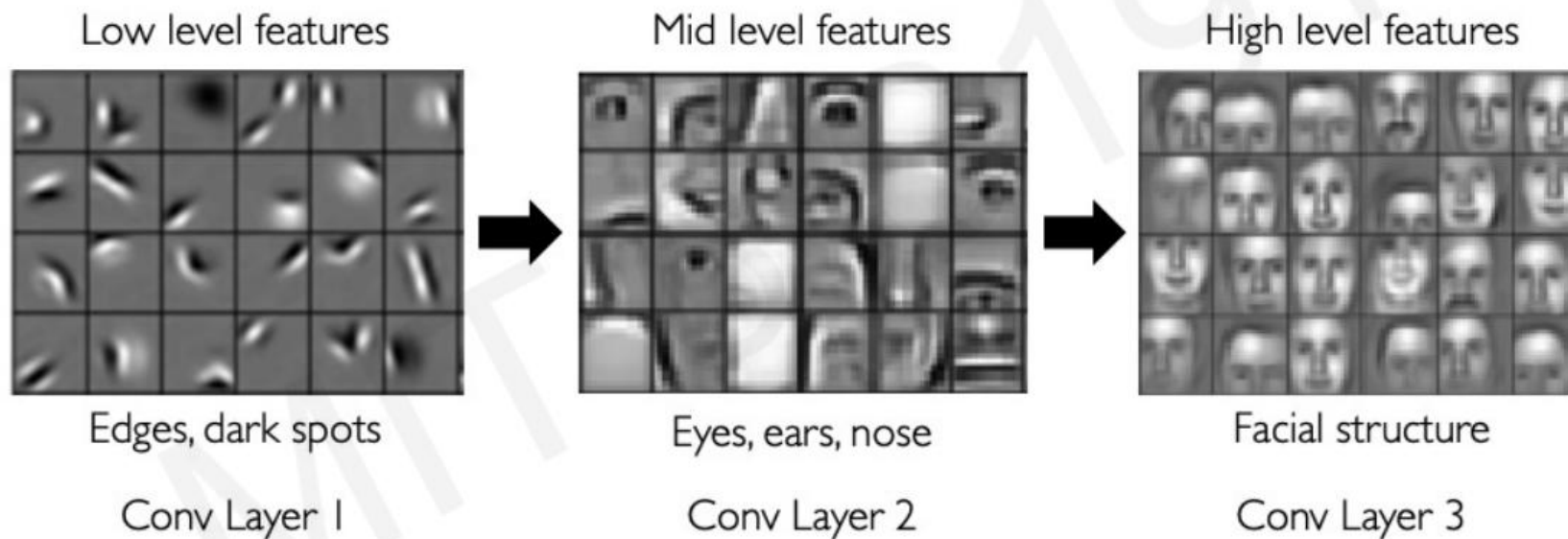


Convolutional Neural Networks



Hierarchy and Scale-dependent Features

- Our filter from earlier didn't seem to represent any recognisable object
- But it might be an “edge” or a “corner”, which are important low-level features in many objects
- Our knowledge of scale is another inductive bias that we can introduce into the CNN architecture



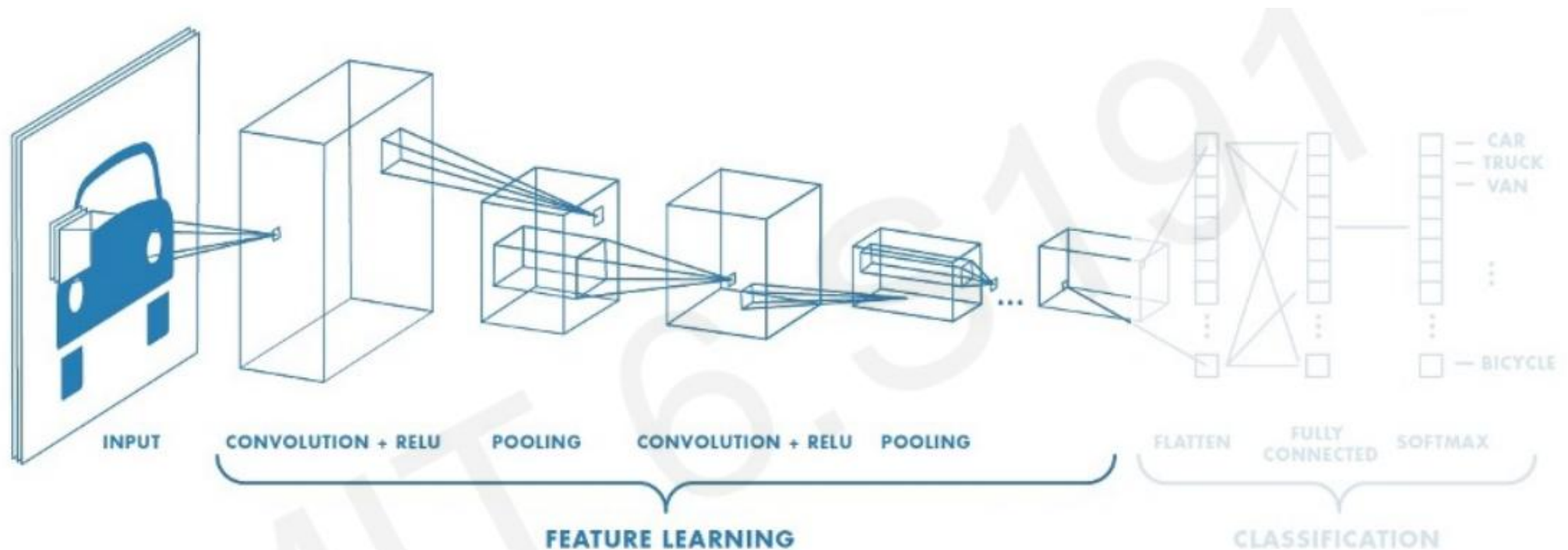
A Typical CNN Architecture

- We can handle scale by having convolutions of increasing size, each one applied to the output of the previous convolution
- Recall the pattern: $MaxPool \left(ReLU \left(Conv2D(x_{ij}) \right) \right)$: encoder
- Note: We added a non-linearity here – this allows the NN to universally approximate
- We stack multiple encoders to learn features



A Typical CNN Architecture

- We stack multiple encoders to learn features



Convolution Arithmetic

- Let's look at a couple of examples to understand the specifics

CONV2D



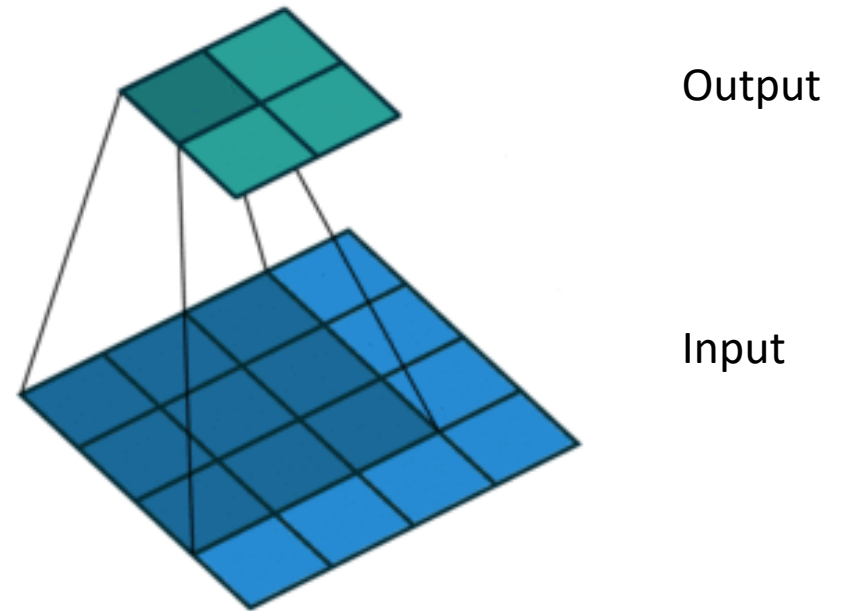
```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1,  
groups=1, bias=True, padding_mode='zeros', device=None, dtype=None) \[SOURCE\]
```

- The Conv2D class is simply an implementation of the kernel/filter that we invented earlier, but it has a few bells and whistles that you need to understand



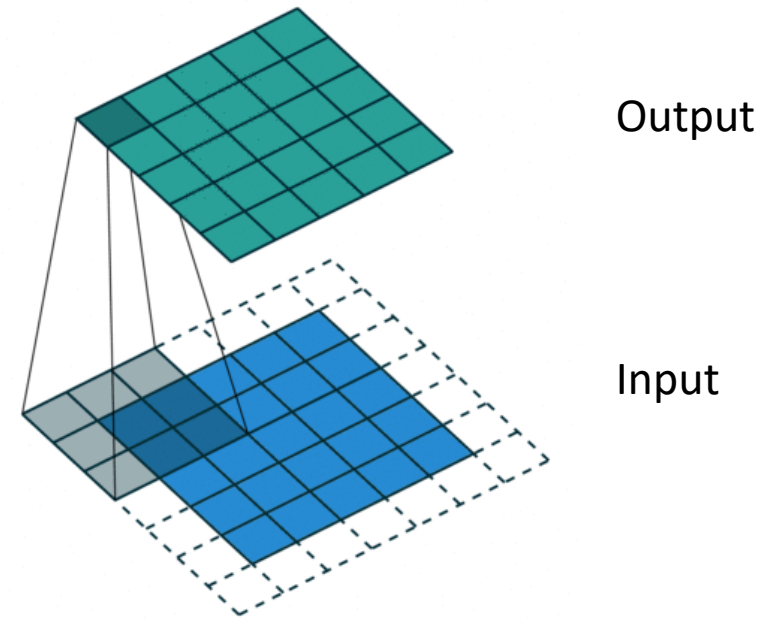
Kernel Size

- Consider the convolution on the right
- Our input image is 4 x 4
- The “kernel size” is the “receptive field” of the convolution: how big is the matrix?
- The kernel size is 3 x 3 (it is conventionally square)



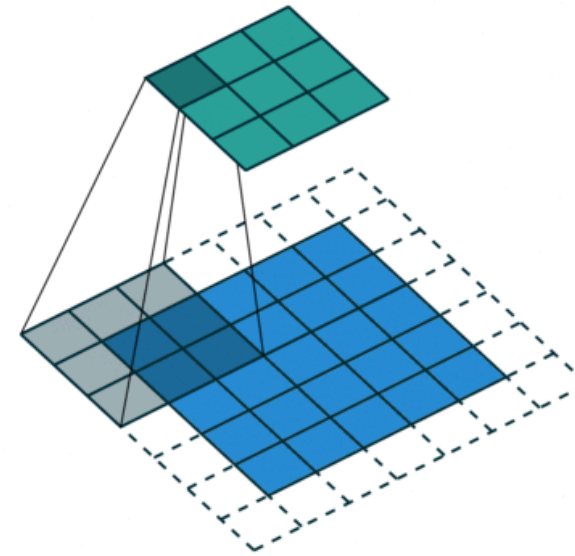
Padding

- Notice in the previous slide that our output is smaller than the input: *we don't necessarily want this!*
- To give us back our precious pixels, we can add some dummy pixels to the input
- See the example on the right has output size equal to input size



Stride

- Pixels side-by-side are likely to contain very similar information
- It may be a waste of computational resources to consider the neighbourhood of *every single pixel*
- Assuming it's enough that a pixel is in the neighbourhood of *some* convolution, we can set a stride: the number of pixels to slide along
- Here we stride by 2 pixels



Output

Input



Example 1

- Consider the following:
 - Convolution of kernel size = 3, stride = 2, padding = 0; applied to
 - An image of 5 x 5 pixels
- What is the output size?



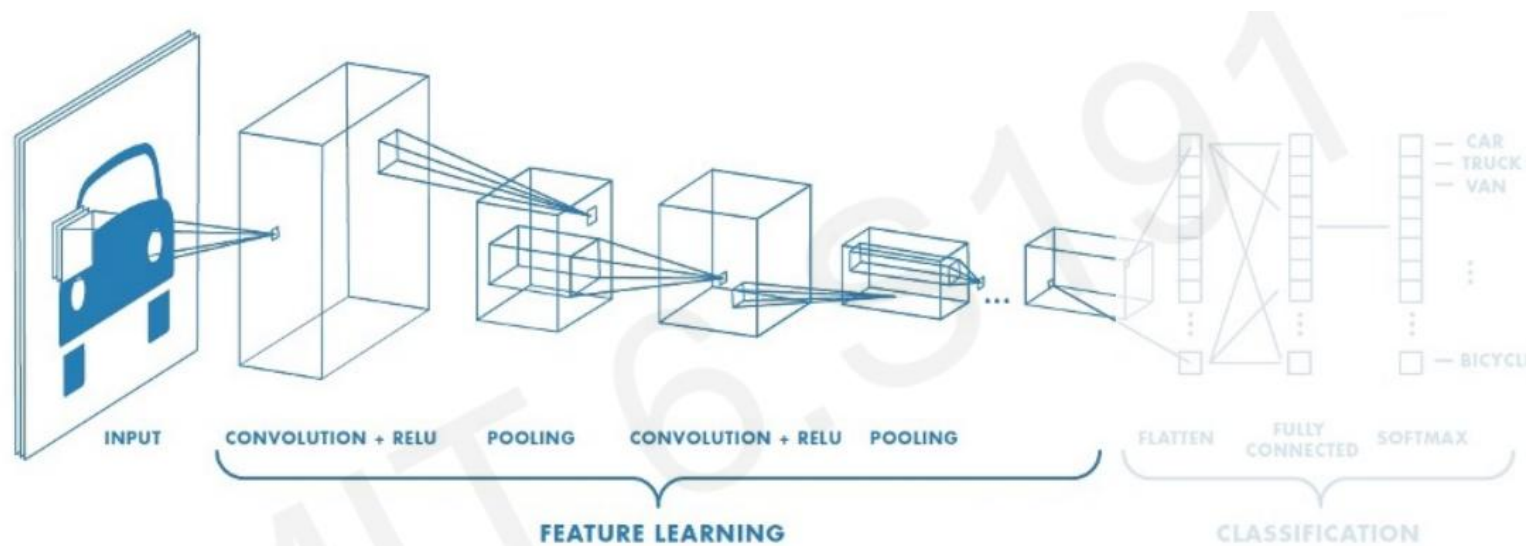
Example 2

- Consider the following:
 - Max Pool of kernel size = 4, stride = 1, padding = 0; applied to
 - Convolution of kernel size = 3, stride = 2, padding = 1; applied to
 - An image of 256 x 256 pixels
- What is the output size?



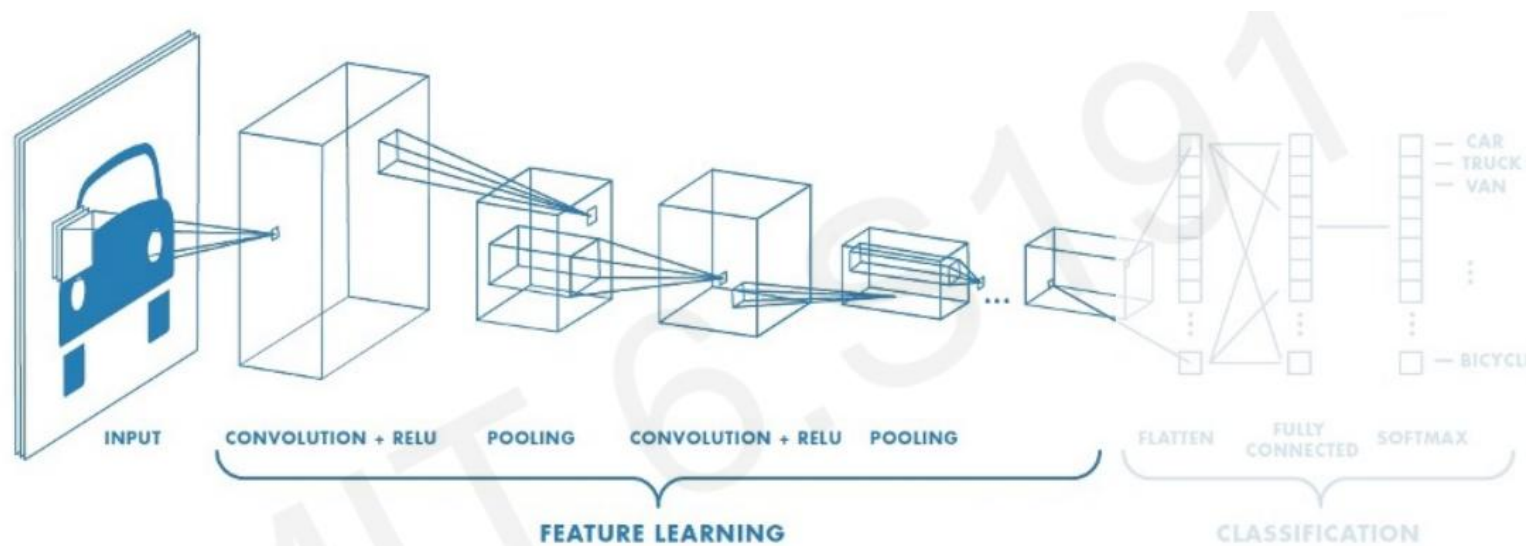
Back to the Typical Architecture

- Typically, we want to grow our receptive field (by using stride and max-pooling to down-sample the image), and deepen the latent space
- This is based on the intuition that there are a small number of complex ideas/features that can fully capture an image



Back to the Typical Architecture

- Okay, so we have convolved and pooled our way to a tiny image with a big latent space
- What happens at the end??
- It depends...

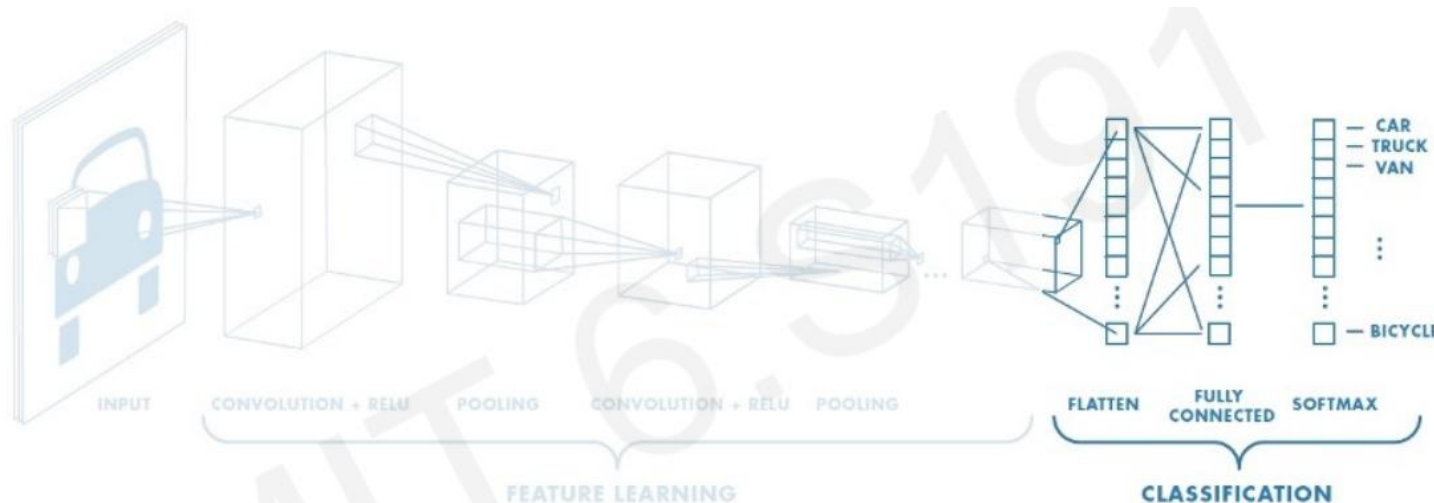


Case Studies



Classification

- The classic task: classification (or regression)
- As in all classification, we need to transform our $x \times y \times h$ encoded image into a prediction vector (for example, where the entries are class probabilities)
- Simple idea: just line up all the rows of pixels into one big vector (aka “flatten the image”) and pass through a FFNN



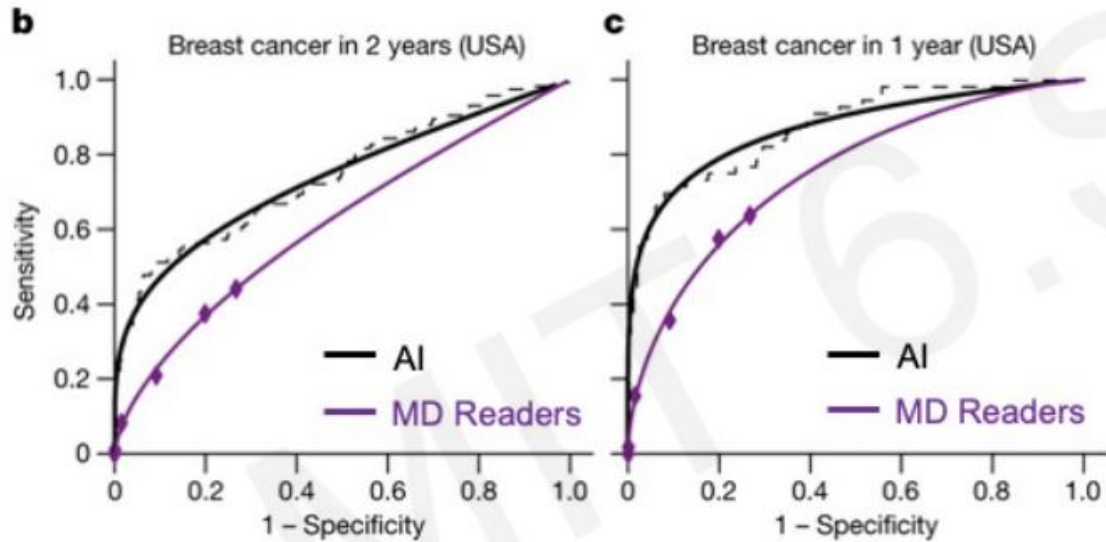
Classification

- The classic task: classification (or regression)
- As in all classification, we need to transform our $x \times y \times h$ encoded image into a prediction vector (for example, where the entries are class probabilities)
- Simple idea: just line up all the rows of pixels into one big vector (aka “flatten the image”) and pass through a FFNN
- But didn’t you say this doesn’t work very well? Once we have learned features with convolutions and downsampled the image, it works extremely well. Doesn’t work well on the raw input pixels

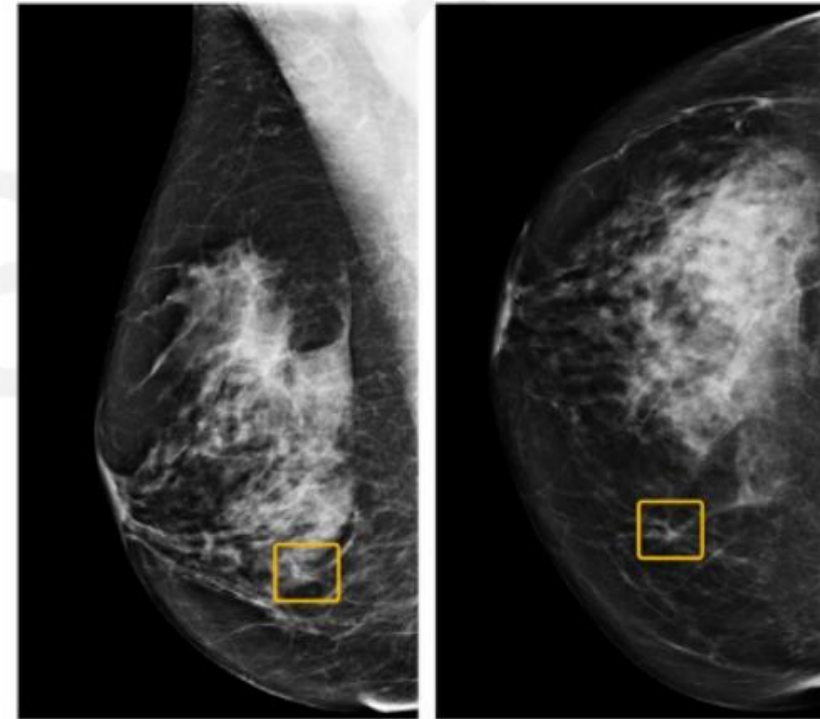


Classification: Breast Cancer Screening

International evaluation of an AI system for breast cancer screening nature



CNN-based system outperformed expert radiologists at detecting breast cancer from mammograms

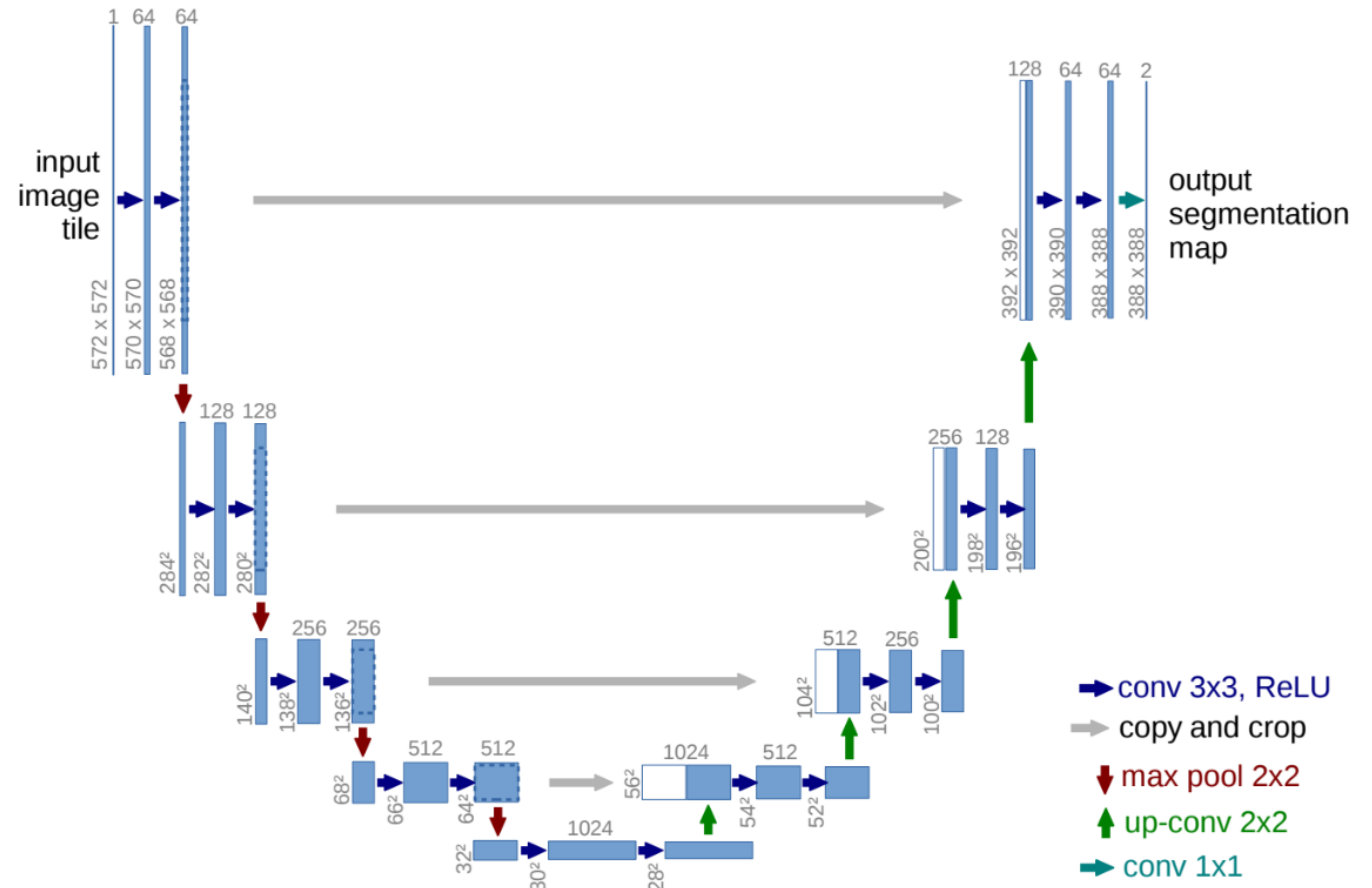
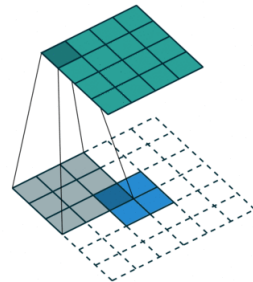


Breast cancer case missed by radiologist but detected by AI



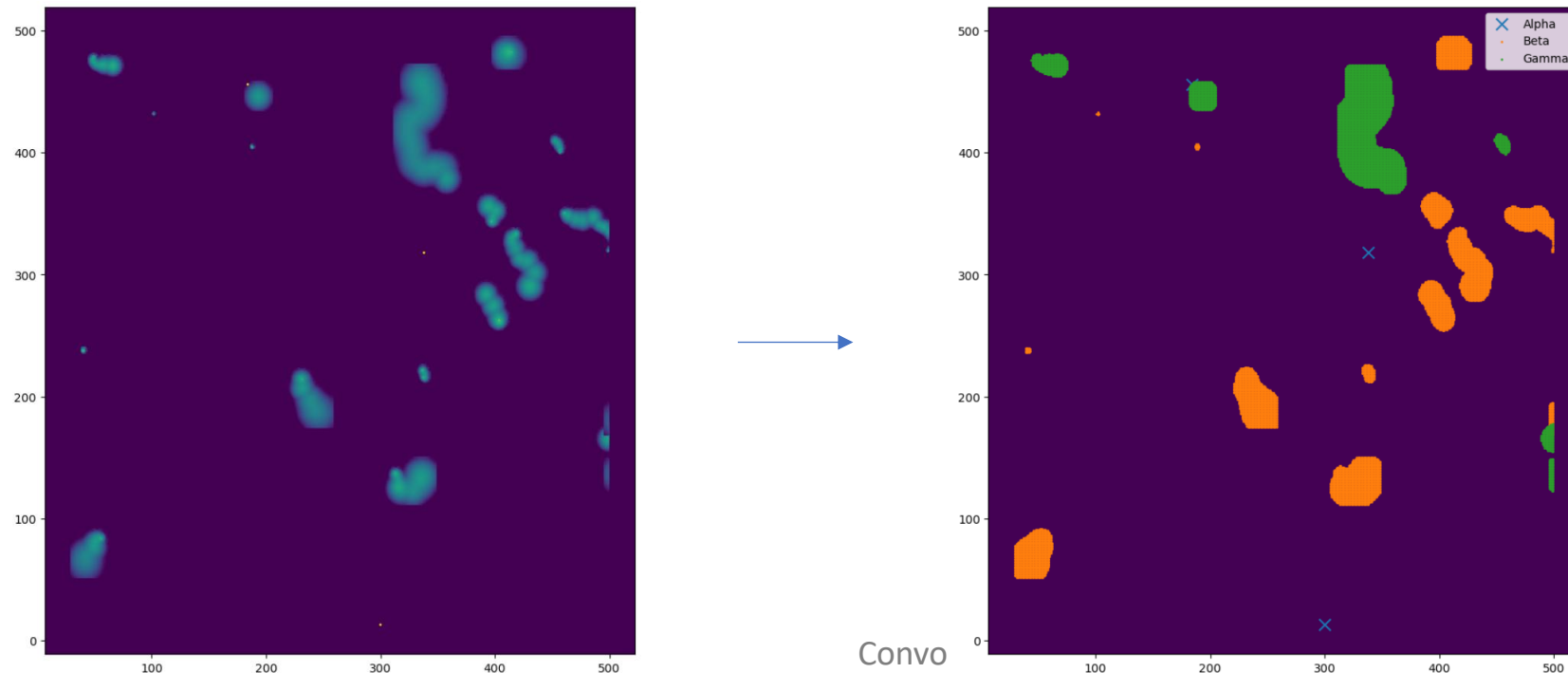
Instance and Semantic Segmentation

- What if we want to label all the objects in an image, and find their location?
- This is segmentation: rather than a classification for the image, give a classification for *each pixel*
- But this requires returning an output the same size as the input image
- Enter the “Unet”: allows a large receptive field, while still returning the original dimensions
- Contains a “transposed convolution”



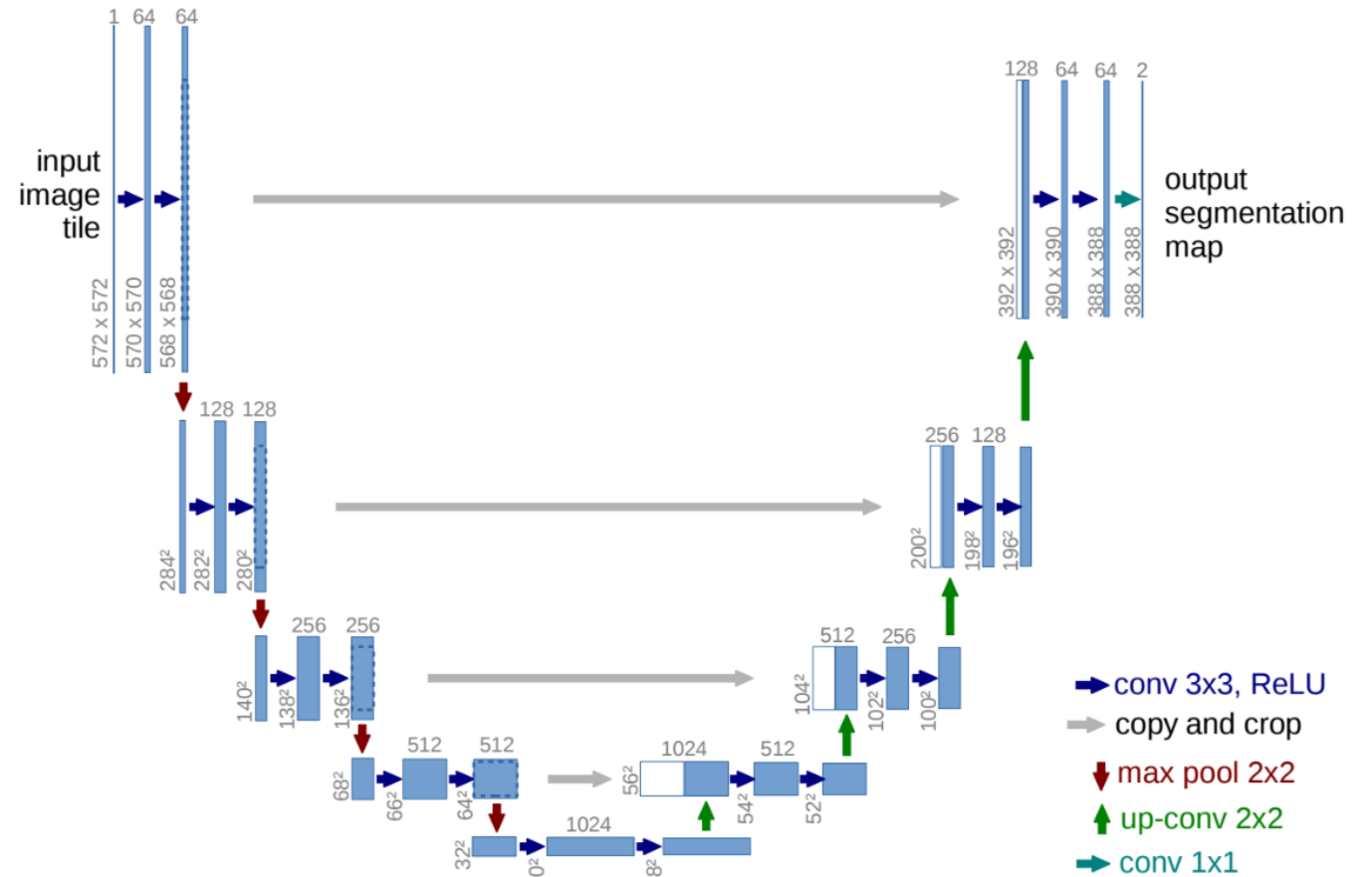
Semantic Segmentation: Tritium Detection

- Given the fallout of a nuclear disaster, can you detect radioactive decays from soil in a silicon charge-couple device?
- We hope so! This is a semantic segmentation problem:



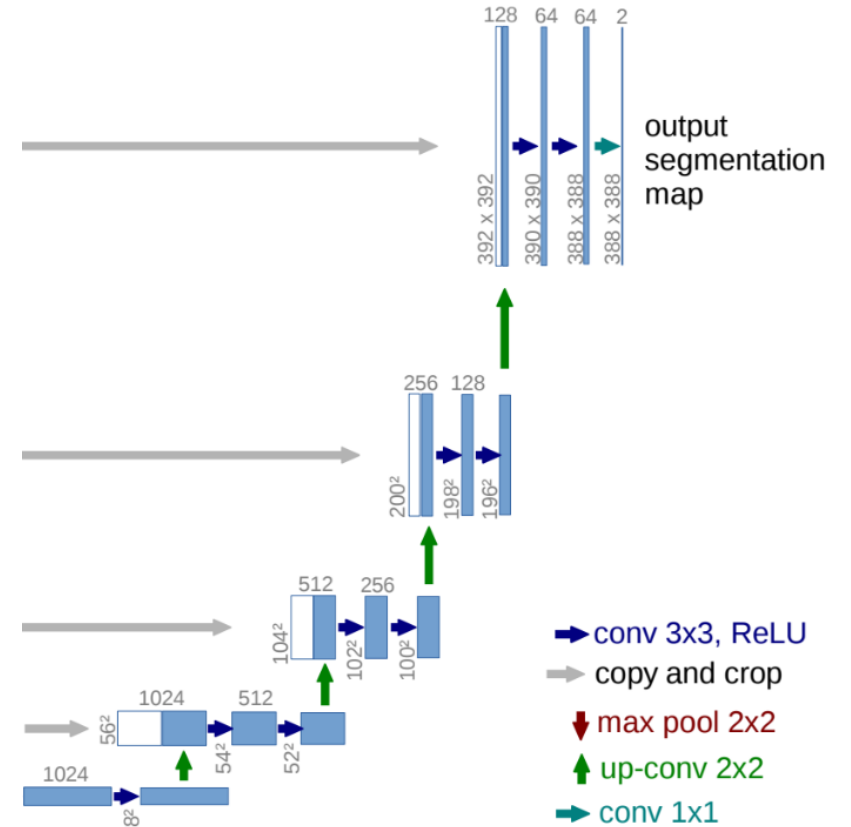
Generative Models

- Recall the Unet: it goes all the way down to a single “pixel” of very large latent space
- Called the “bottleneck” – used to capture information about the whole image
- The grey arrows are “residual connections”: simply concatenate/sum two latent spaces together! Simple but powerful



Generative Models

- Recall the Unet: it goes all the way down to a single “pixel” of very large latent space
- Called the “bottleneck” – used to capture information about the whole image
- We can train a network that *starts* at the bottleneck to perform transposed convolutions and *generate* an image
- **Many** different ways to train, but they almost always rely on bottleneck → transposed convolution



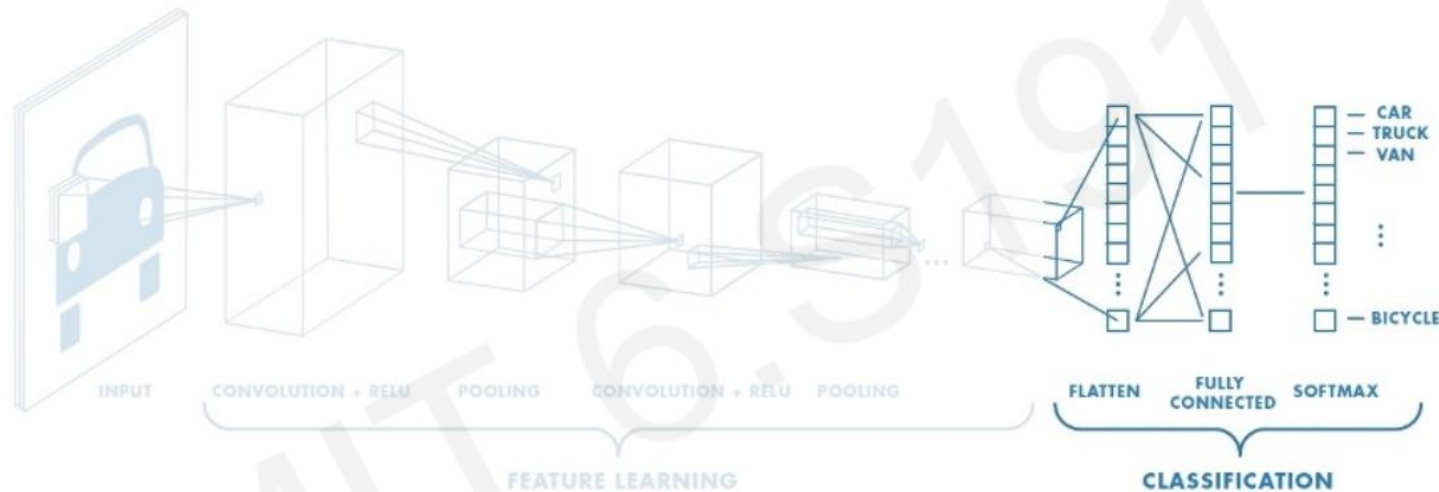
Generative Models

- CycleGAN is a sophisticated sequence of transposed convolutions and bottlenecks



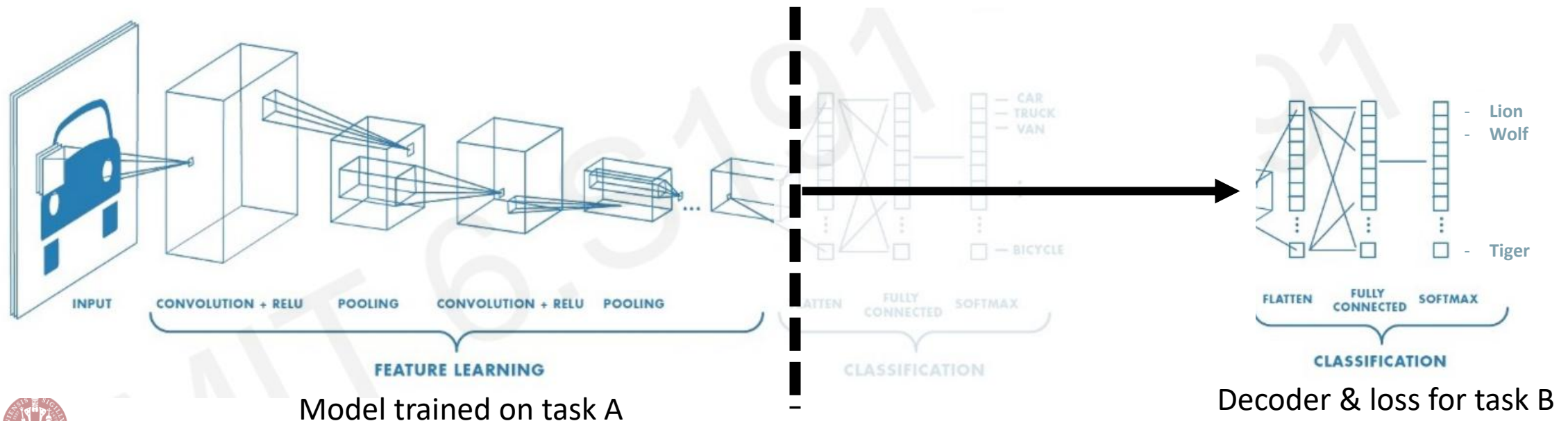
Transfer Learning with Feature Extraction

- What if we spend \$1million training a cat/dog classifier, then realise we actually want to classify lions and wolves – or worse: aeroplanes, boats and cars



Transfer Learning with Feature Extraction

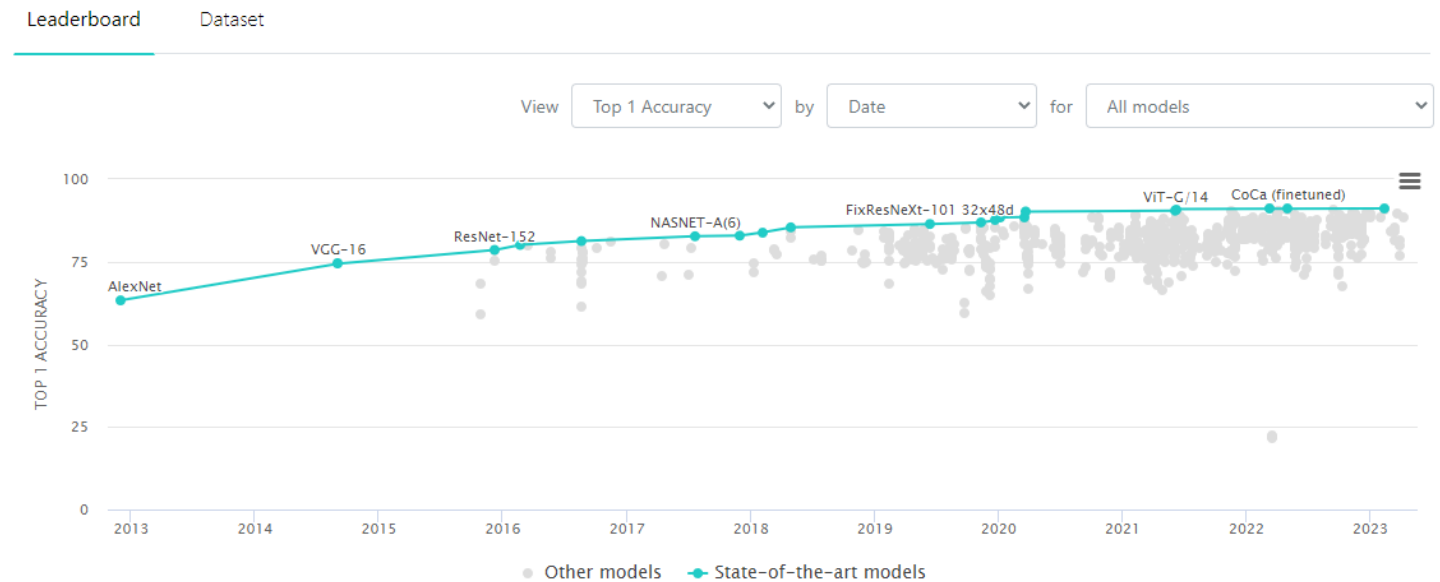
- What if we spend \$1million training a cat/dog classifier, then realise we actually want to classify lions and wolves – or worse: aeroplanes, boats and cars
- Intuition: the features learned from a cat picture (edges for example) are relevant for other image classifiers
- We can slice off the final task-specific layer of a model and still use its feature layers



Transfer Learning: ImageNet

- ImageNet has 14 million images, with 20,000 (!!!) classes
- There are many publicly available CNNs trained on ImageNet that have feature layers for transfer learning
- ResNet is a classic model that combines convolutions and residual connections

Image Classification on ImageNet



```
import torch
model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18', pretrained=True)
```



Other Dimensions of Convolution: Sequences & Videos

- We have only talked about 2D convolutions, but →
- If we have a sequence of scalar measurements (e.g. histogram), where nearby measurements should be related somehow, then we can use a 1D convolution
- If we have a sequence of vector measurements, then we can use a 2D convolution
- If we have a sequence of 2D measurements (e.g. video), then we can use a 3D convolution
- In principle, could go ever higher!

 PyTorch

Convolution Layers

```
nn.Conv1d
```

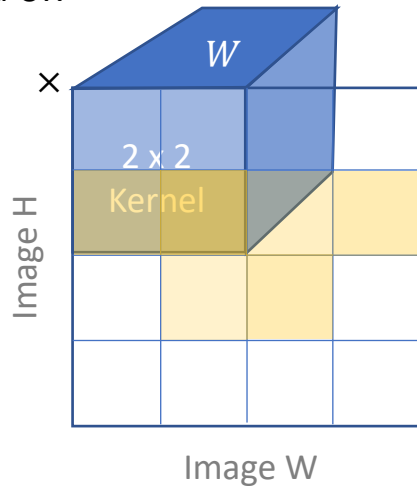
```
nn.Conv2d
```

```
nn.Conv3d
```



Other Dimensions of Convolution: Sequences & Videos

- We can perform a 3D convolution across $[t, x, y]$
- That is, instead of:



- We need to use a $t \times x \times y \times h$ tensor to multiply each time-pixel
- This might capture concepts that are not static, e.g. emotions...



Data Augmentation & Learned Symmetries

- We know we have translational invariance (the sliding kernel!)
- Do we have rotational invariance? Dilation invariance? [*Lorentz invariance?*](#)
- The MIT course certainly seems to think so...

X or X?

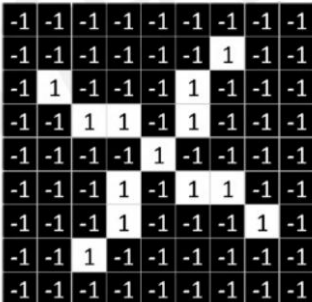
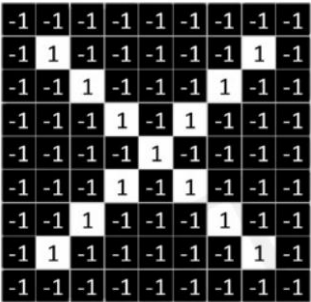
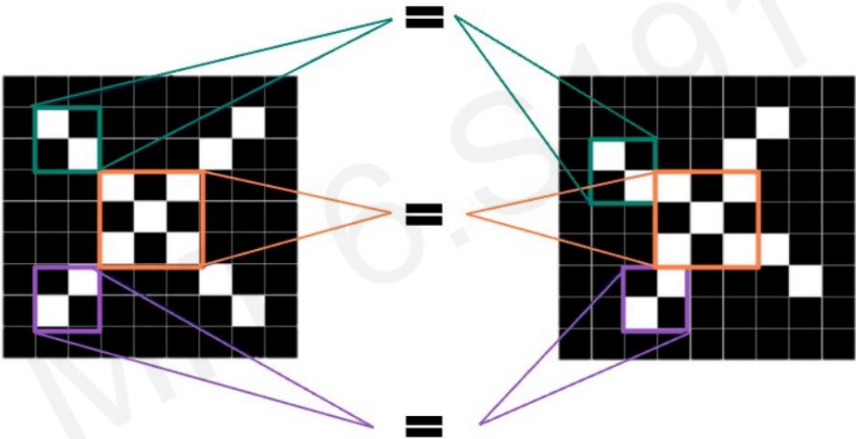


Image is represented as matrix of pixel values... and computers are literal!
We want to be able to classify an X as an X even if it's shifted, shrunk, rotated, deformed.

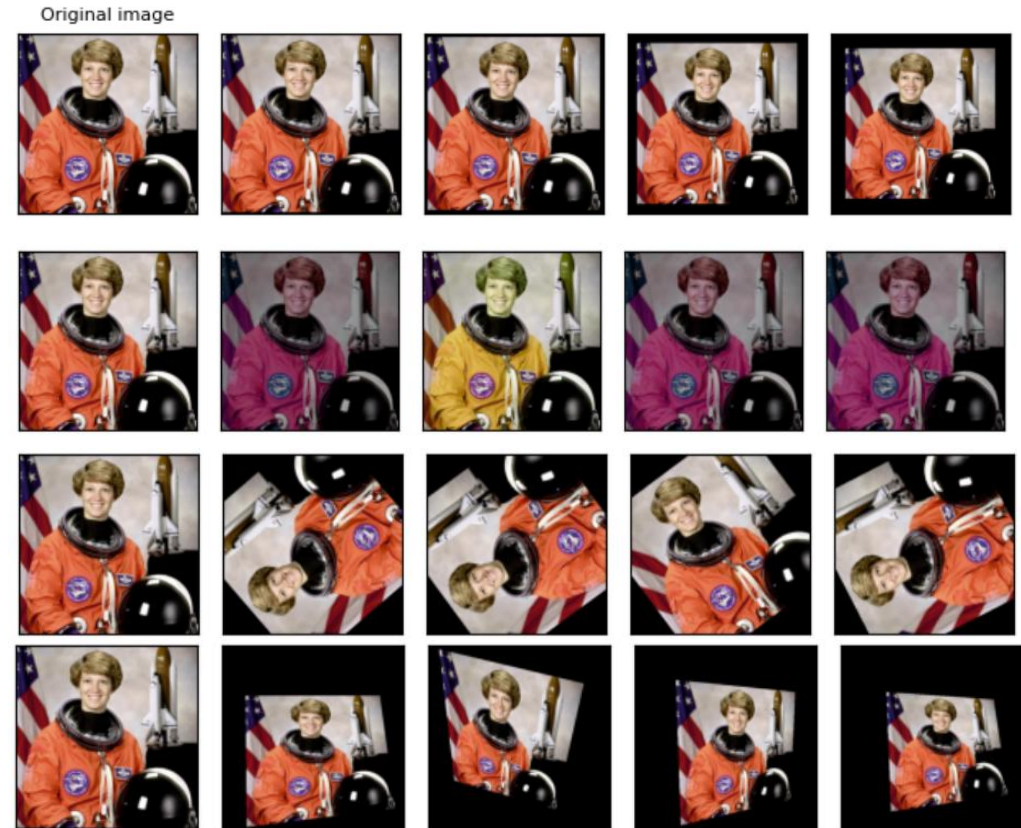
Features of X



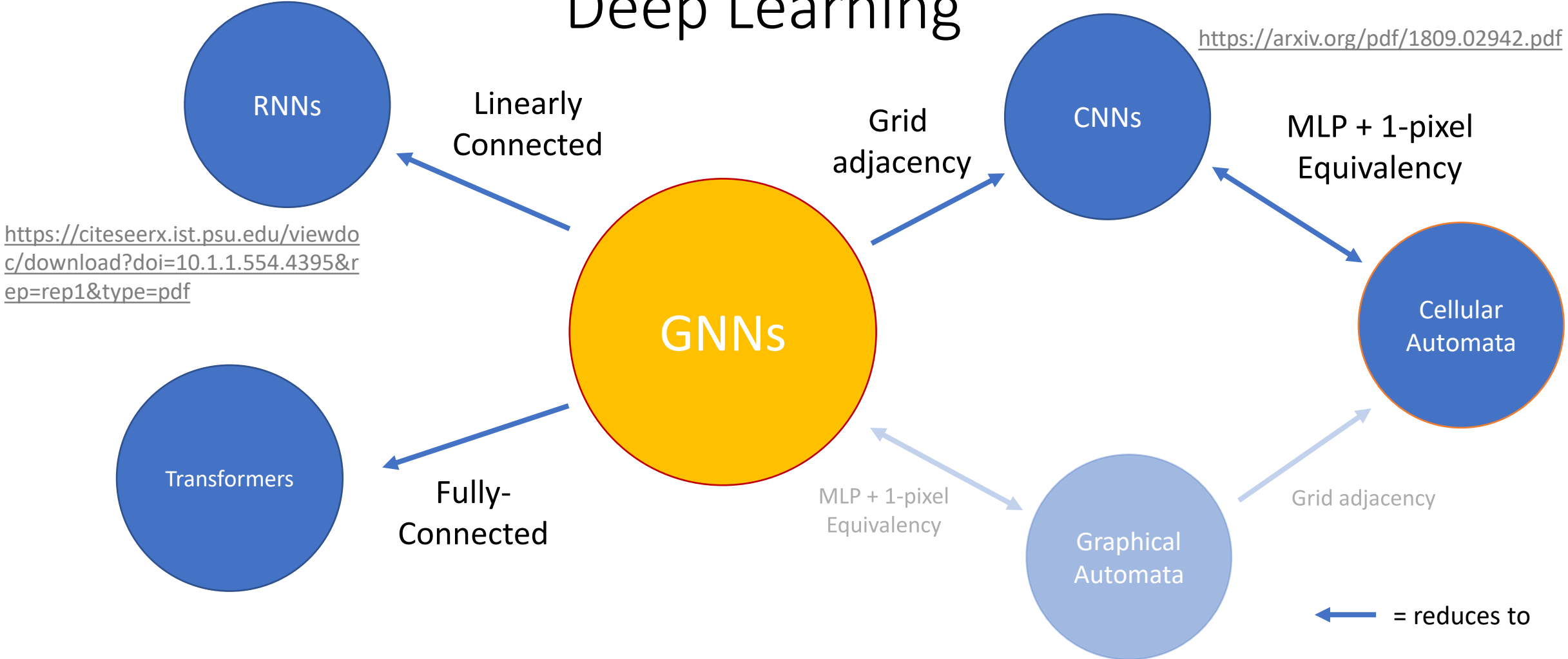
Data Augmentation & Learned Symmetries

- In fact, we don't have these other symmetries "hard-coded" in, like we do with translation
- If we want to include them, we need to learn them: Enter data augmentation
- Principle: In training, randomly transform image samples with rotations, noise, dilations, colour changes, etc.
- Practice: Use a Transform function

```
transformed_dataset = FaceLandmarksDataset(csv_file='data/faces/face_landmarks.csv',
                                           root_dir='data/faces/',
                                           transform=transforms.Compose([
                                               Rescale(256),
                                               RandomCrop(224),
                                               ToTensor()
                                           ]))
```



CNNs in The Landscape of Geometric Deep Learning



<https://arxiv.org/pdf/1809.02942.pdf>

<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.554.4395&rep=rep1&type=pdf>

[https://github.com/murnanedaniel/GNN-as-Transformer-as-GNN/blob/main/0-Transformer vs GNN Annotated.ipynb](https://github.com/murnanedaniel/GNN-as-Transformer-as-GNN/blob/main/0-Transformer%20vs%20GNN%20Annotated.ipynb)
<https://arxiv.org/pdf/2012.09699.pdf>

Key Take-aways

- The convolution is a building block in [almost any architecture](#) applied to vision and video
- Convolution arithmetic is kind of a pain, but that's what `print(tensor.shape)` statements are for
- Convolutions can also be applied to sequential data, can learn symmetries in augmented data, and can segment images
- These days: Always start with a pre-trained model – it's a free lunch!
- The humble CNN proves the value of many ideas in ML: symmetry, hierarchy, inductive bias, data augmentation, generative models, etc.



CNN FAQ



Where do the hidden channels come in?

- Recall our toy model

Image neighbourhood Filter

$$\sum \left(\begin{array}{|c|c|c|} \hline 0.1 & 0.1 & 0.9 \\ \hline 0.1 & 0.9 & 0.1 \\ \hline 0.9 & 0.1 & 0.1 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline 0.1 & 0.1 & 0.9 \\ \hline 0.1 & 0.9 & 0.1 \\ \hline 0.9 & 0.1 & 0.1 \\ \hline \end{array} \right) = \sum \left(\begin{array}{|c|c|c|} \hline 0.01 & 0.01 & 0.81 \\ \hline 0.01 & 0.81 & 0.01 \\ \hline 0.81 & 0.01 & 0.01 \\ \hline \end{array} \right) = 2.43$$



Where do the hidden channels come in?

- Recall our toy model

Image neighbourhood Filter

$$\sum \left(\begin{array}{|c|c|c|} \hline 0.1 & 0.1 & 0.9 \\ \hline 0.1 & 0.9 & 0.1 \\ \hline 0.9 & 0.1 & 0.1 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline 0.1 & 0.1 & 0.9 \\ \hline 0.1 & 0.9 & 0.1 \\ \hline 0.9 & 0.1 & 0.1 \\ \hline \end{array} \right) = \sum \left(\begin{array}{|c|c|c|} \hline 0.01 & 0.01 & 0.81 \\ \hline 0.01 & 0.81 & 0.01 \\ \hline 0.81 & 0.01 & 0.01 \\ \hline \end{array} \right) = 2.43$$

- The window (“kernel size”) is 3x3, but the input shape for each pixel is just one number (black-white brightness), and the output shape (for each pixel) is just one number (a hidden “latent” feature)



Where do the hidden channels come in?

- Recall our toy model

Image neighbourhood Filter

$$\sum_{ij} \left(\begin{array}{ccc} 0.1 & 0.1 & 0.9 \\ 0.1 & 0.9 & 0.1 \\ 0.9 & 0.1 & 0.1 \end{array} x_{ij} \cdot \begin{array}{ccc} 0.1 & 0.1 & 0.9 \\ 0.1 & 0.9 & 0.1 \\ 0.9 & 0.1 & 0.1 \end{array} W_{ij} \right) = \sum_{ij} \left(\begin{array}{ccc} 0.01 & 0.01 & 0.81 \\ 0.01 & 0.81 & 0.01 \\ 0.81 & 0.01 & 0.01 \end{array} m_{ij} \right) = 2.43$$

- The window (“kernel size”) is 3x3, but the input shape for each pixel is just one number (black-white brightness), and the output shape (for each pixel) is just one number (a hidden “latent” feature)



Where do the hidden channels come in?

- To make a CNN more powerful, we need to be able to take in any number of features (e.g. color) and we want *many* filter channels in a convolution. That is, we need input length and output length
- Our general convolution is thus

$$\sum_{ij} x_{ij}^M W_{ij}^{MN} = \sum_{ij} m_{ij}^N = h^N$$

where x_{ij}^M is the input window, W_{ij}^{MN} is the learnable kernel with the ij^{th} entry an $M \times N$ matrix. M is input shape (e.g. 3 colors), N is output shape, i is the window rows, j is the window columns. h^N is a length N vector, a different one for each pixel in the image



When do I do max pooling?

- Max pooling and the convolutional kernel are **completely separate**, and we don't even need to do pooling
- They both use a “window” system, of looking at neighborhoods of pixels, but the window size of the kernel does not have to be the same size as the max pooling window size
- **Max** pooling is just one choice of pooling. Any kind of aggregation will work: mean, sum, min, or even more exotic kinds like variance



But does the pooling reduce the image size?

- If we pad our image on the border, and slide the kernel or pooling window one pixel at a time, **our output array will be the same shape as the input array**: the kernel and pooling do not inherently down-sample an image
- It is only when we increase the stride (the number of pixels we move the window each time) that the output shape gets reduced
- Kernels and pooling both use windows, so they can both use stride, and thus can both be used to down-sample an image. Which one you use to down-sample is part intuition, part trial-and-error

