Fundamentals of machine learning: data, models, examples

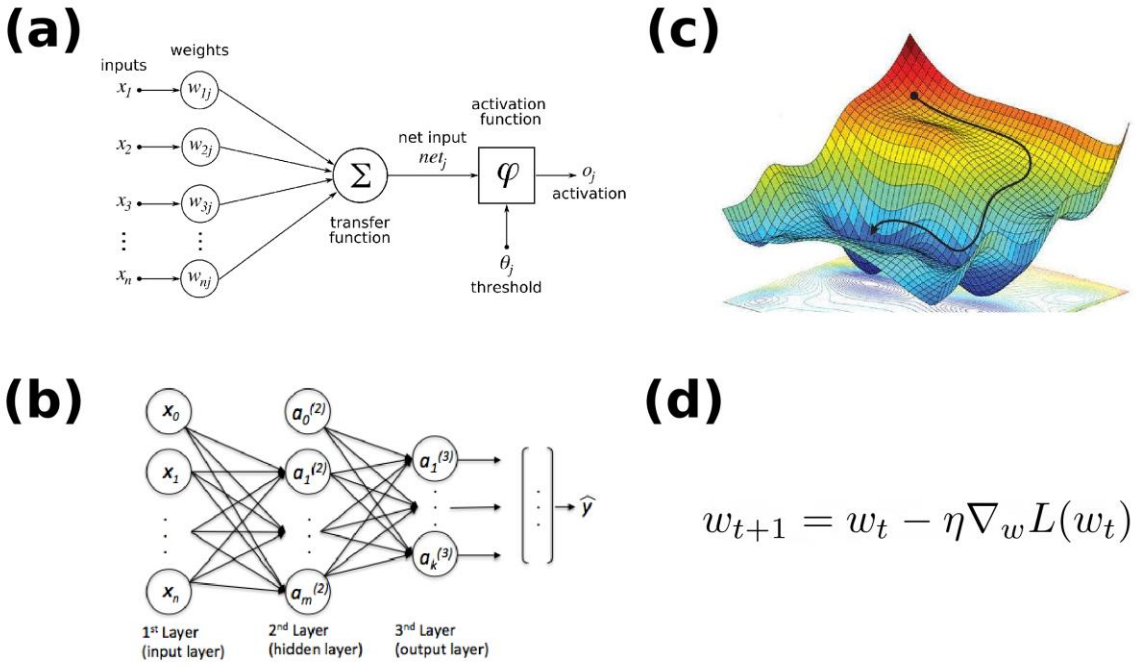
Ronan Docherty – TLDR Group – 14/07/2024

IMPERIAL

Content

Intro

- Neural networks are **universal function approximators** [FA]
- Can be as simple as $f(x) = x^2$ or complex as $f($  $) = \begin{cases} '\mathrm{cat}' \\ '\mathrm{dog}' \end{cases}$
- Flexible and powerful: can do anything from linear regression to promptable image generation



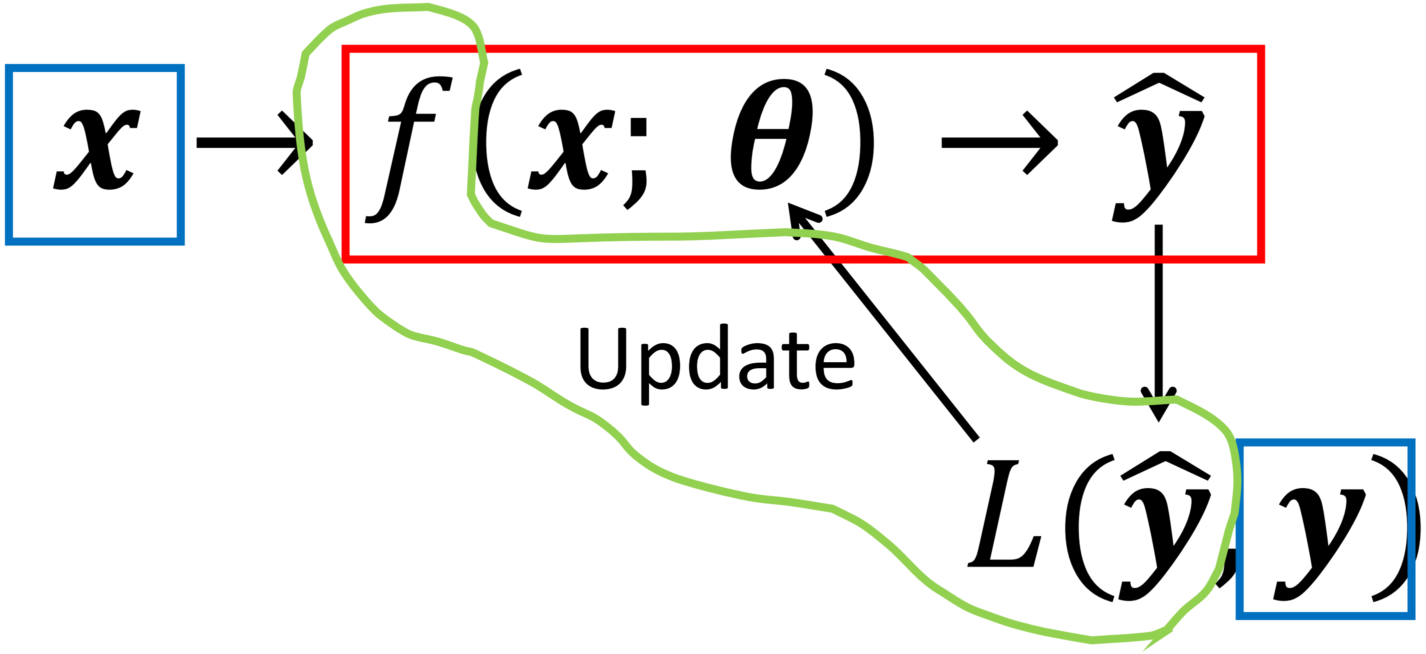
(a) Artificial neuron with weights *w* [AN], **(b)** stacking neurons into network [St], **(c)** loss landscape as a function of *w* [LL] and **(d)** an example rule to update *w* to reach the minimum in **(c)**

Content

Problem statement

Data: input to model $\boldsymbol{x}$, target $\boldsymbol{y}$

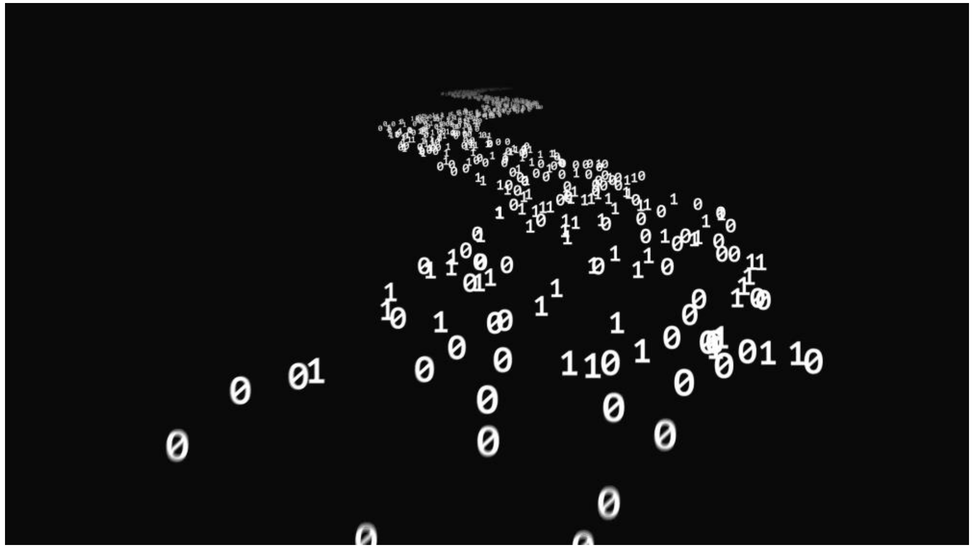
Model: predicts $\hat{\boldsymbol{y}}$ given $\boldsymbol{x}$ and $\boldsymbol{\theta}$



Task: how to structure f, compare $\hat{\boldsymbol{y}}$ to target $\boldsymbol{y}$ & update $\boldsymbol{\theta}$?

- **It is our job** to pick training data, model and task to best show the mapping/function we want the NN to learn

Data

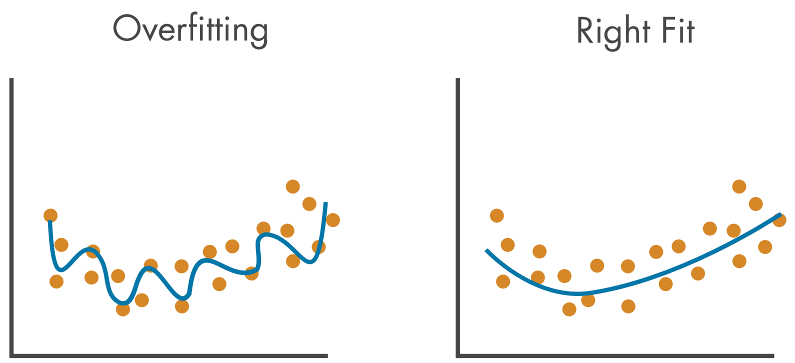


Just a cool graphic. From [Qt]

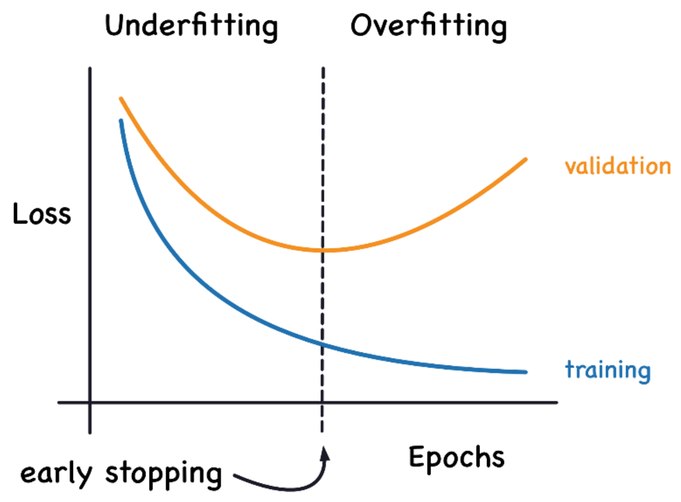
- Anything we can represent as numbers: measurements, text, images, audio, video, graphs, … can be an input $\boldsymbol{x}$ or target $\boldsymbol{y}$
- Our training data are samples from some underlying distribution
- NN learns the mapping in the data (not always the mapping we want!)
- More data = better!

Overfitting

- We know classical overfitting & NNs can have millions of parameters -> prone to overfitting
- How do we detect it? Data splits!
  - **Train split** (70%): we train model on this and backpropagate loss
  - **Validation split** (20%): evaluate model loss during training, if increasing then we are overfitting
  - **Test split** (10%): for comparing to other models
- How do we fix it? More data and model regularization



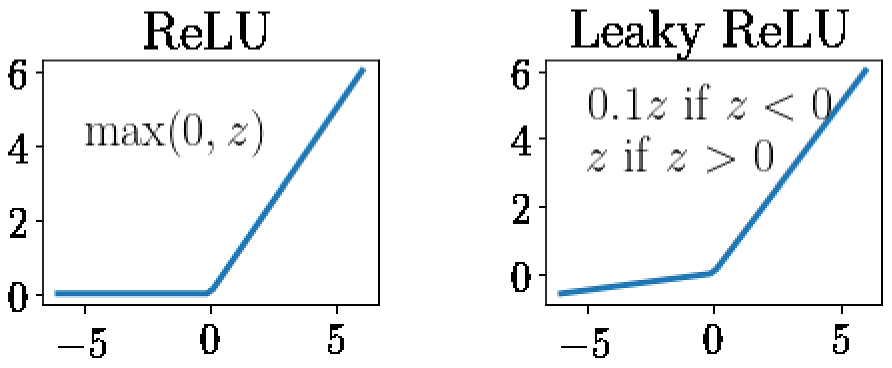
From [OF]



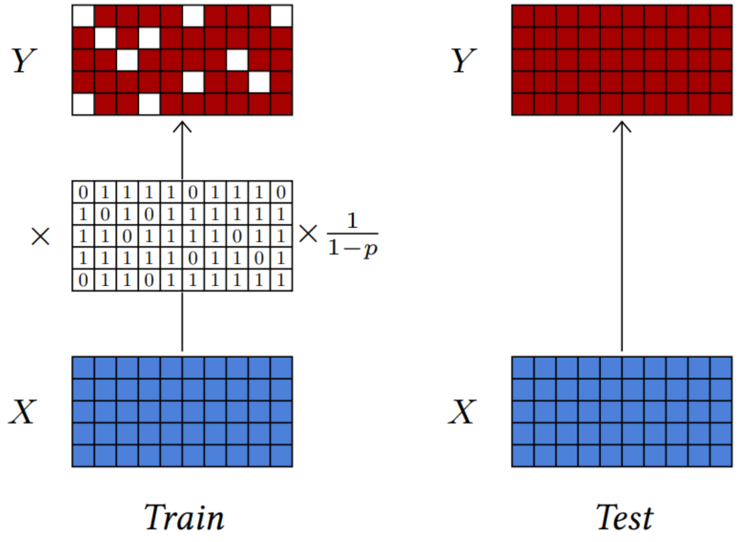
From [KO]

Network

- Layers stacked like Lego, output of previous layer as input of next layer
- Non-linear activation layers => allows learning non-linear functions
- Regularization stops overfitting & speeds up training:
  - Normalization layers: normalizes activation values over a batch of data or layer
  - Skip connections: adding/appending output of previous layer to a future one
  - Dropout: ignore connection from one layer to another with probability *p*



From [AF]



From [LB]

Loss function

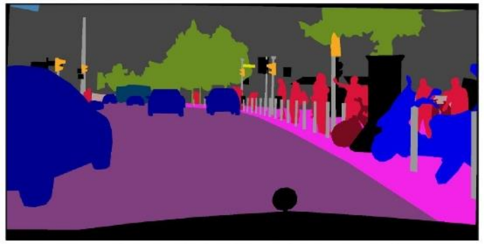
- Measures how wrong our model is going
- Can be as simple as least-squares loss, a weighted sum of other loss functions or structured to reflect your problem
- An example:
  - Input image, target are numbers/labels (0-10). Say 0=bg, 1=car, 2=bike, …, 5=truck
  - Network’s goal is to predict labels for each pixel in image
  - If we used least squares loss, that says classes 0 and 1 are more similar than 1 and 5
  - This would cause network to learn poorly, so we choose a different loss

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

From [MSE]

$\boldsymbol{x}$

$\boldsymbol{y}$

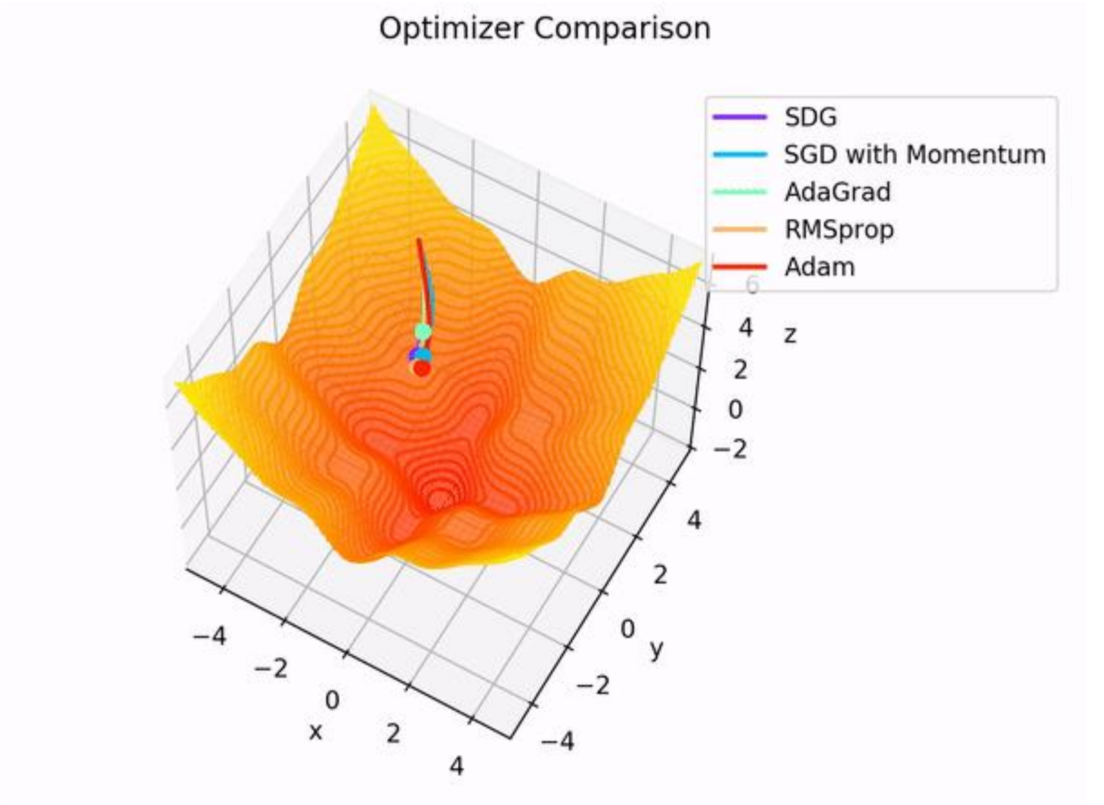


From [SS]

$$l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})}$$

Optimizers and gradient descent

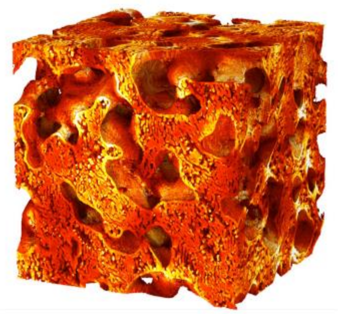
- We have fed $\boldsymbol{x}$ into f and compared its output $\widehat{\boldsymbol{y}}$ to $\boldsymbol{y}$ with L
- Backpropagation & **chain-rule** gives us gradient of loss w.r.t each parameter in $\boldsymbol{\theta}$
- Update these parameters to ‘move’ in direction that minimizes loss
- SGD is the simplest update rule, other ways exist, incorporating ideas like ‘momentum’
- Most common is Adam



From [OPT], ignore typo

Implementation

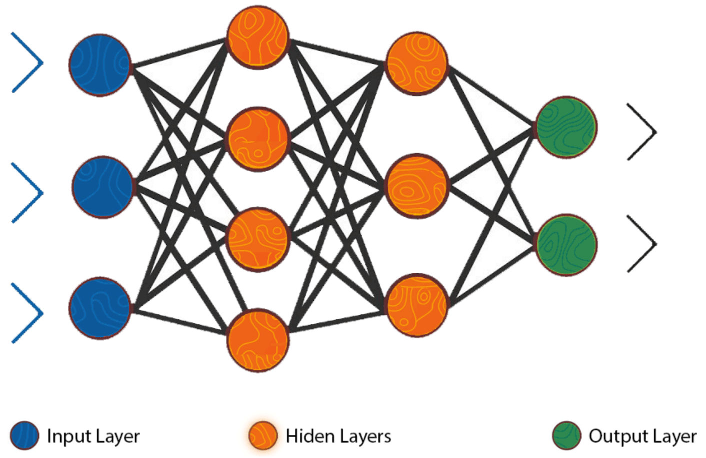
- Implement these ideas in code with **Pytorch**
- Structure:
  - Data and parameters are **tensors** (=multidimensional arrays)
  - Pack training data into **batches** (*i.e,* many 2D images into 3D array)
  - This is because a) matrix multiplication is very efficient on GPUs b) to reduce number of data copies to CPU and c) smooths our gradient descent
- Autodifferentation: track operations on tensors in a computational graph to work out loss gradients
- Side note: Pytorch not just for deep learning, is also a GPU accelerated optimizer and matrix multiplier



Content

Fully connected layers

- Also called ‘Feed Forward’, ‘Dense’, or ‘Linear’ layer
- Does the affine operation W @ $\boldsymbol{x}$ + $\boldsymbol{b}$ on input $\boldsymbol{x}$
- W is **weights matrix** of shape D’ x D, where D is the dimension of vector $\boldsymbol{x}$ and D’ is the ‘hidden dimension’ of the layer [LB]
- Input must be 1D/Vector, output is also a vector
- Can model geometric transformations, projections, similarities [LB]
- All-to-all nature of connections means it **scales poorly as dimension of input increases**



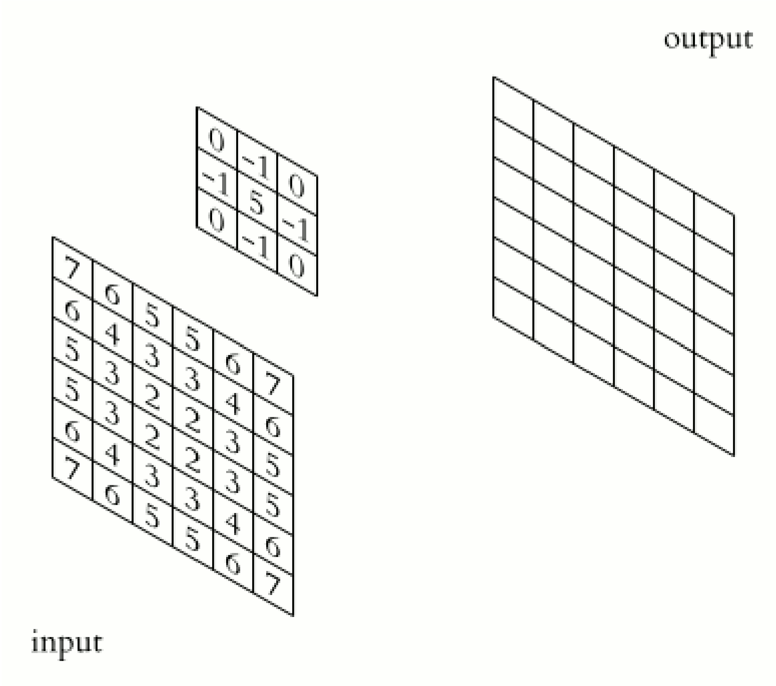
GIF of information flow through series of fully-connected layers. From [FF]

'Convolutions'

- FC layers scale badly with input size – can we reuse same set of weights across different parts of input? Yes!
- Not a real convolution – is cross-correlation or sliding dot product
- Same set of weights slid across input – more efficient & **learns general image features** (edges, textures)
- What we slide is the kernel, length *K* which we move stride *S* 'pixels' at a time
- Input (& output) can be N-dimensional (unlike FC layer)

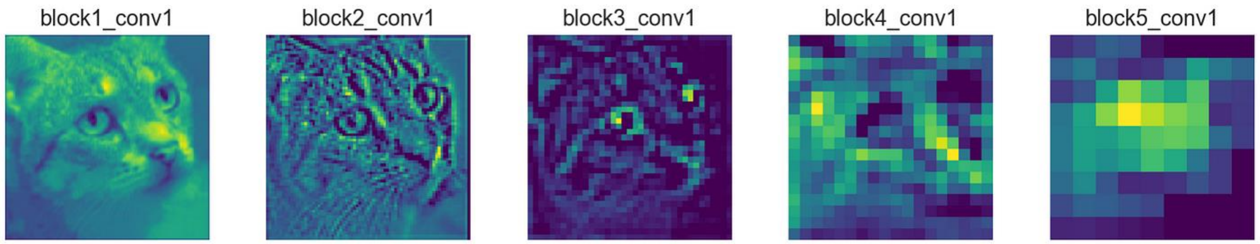
$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

From [PC]

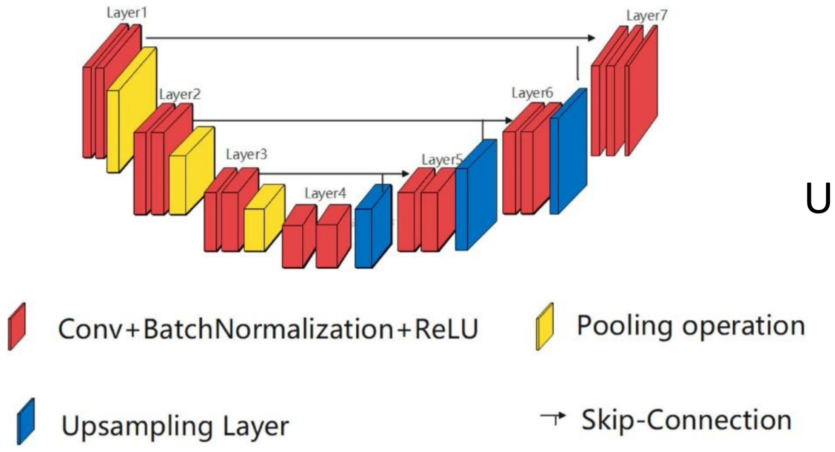


A sharpening kernel. From [WC]

Convolutional neural nets (CNNs)



Successive convolutional layers
learn ‘deeper’ features. From [VG]



U-Net diagram.
From [UN]

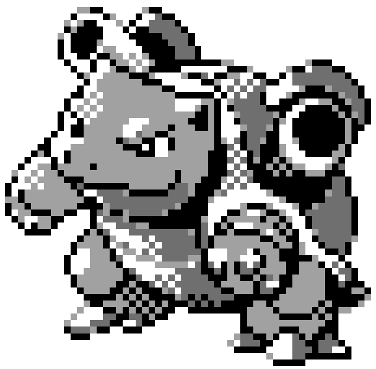
- FC layer maps D-dimension vector to D’-dimensions, conv maps D-channel tensor to D’-channel tensor *i.e,* a 3-channel RGB image (3x 2D arrays) to 32-channel ‘image’ (32x 2D arrays)
- Common to decrease spatial dimensions (pooling/downsampling) and increase channels (hidden information)
- U-Net: downsamples spatially then upsamples, creating ‘**information bottleneck**’

Why use CNNs? A worked example

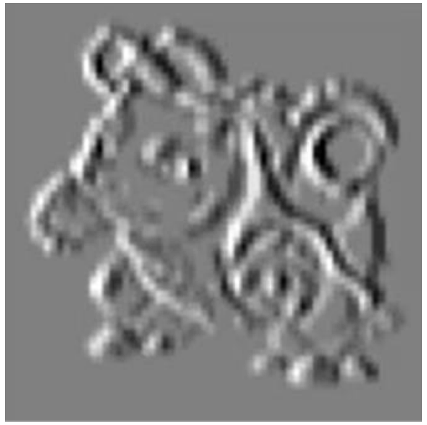
Sobel vertical edges. From [SW]

$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$\boldsymbol{x}$



$\boldsymbol{y}$



$\boldsymbol{x}_{\text{val}}$

$\boldsymbol{y}_{\text{val}}$

- Simple task: learn to detect vertical edges in an image
- Two networks: a fully connected layer with 16781312 parameters and a convolutional layer with 9 parameters
- Include validation image, never seen by network – how well does it end up working?

Image credit: Nintendo

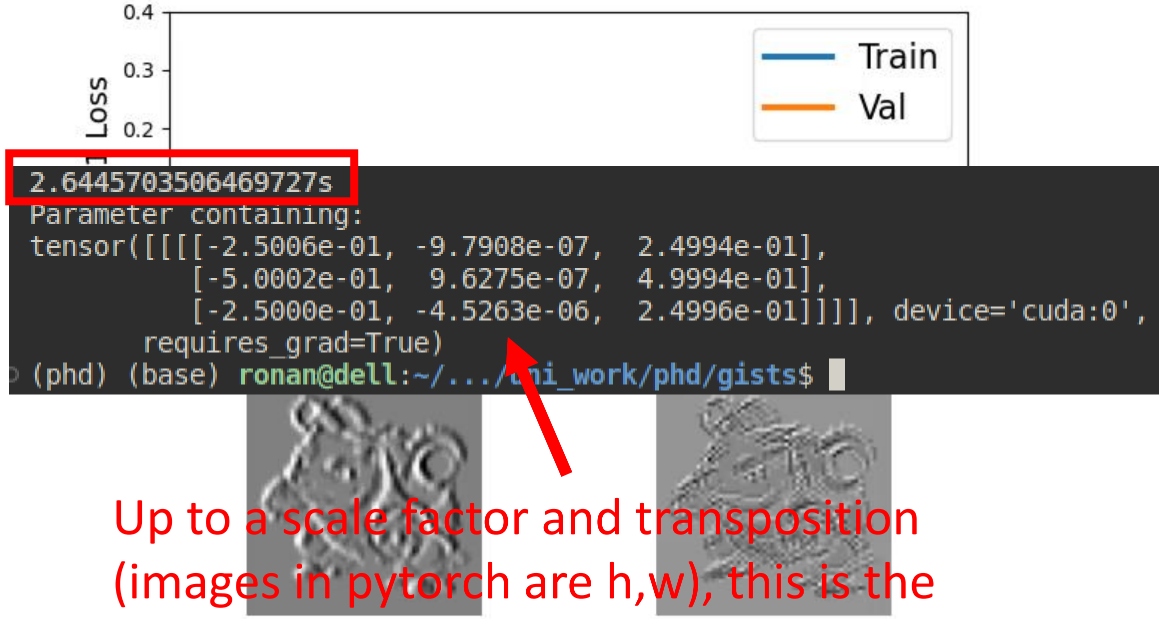
Why use CNNs? A worked example (contd.)

```python
import torch
from torch import nn
import numpy as np
from skimage.filters import sobel_v
from PIL import Image

IMG_H, IMG_W = 64, 64
N_EPOCHS = 1000
LR = 1e-3
GPU = torch.device("cuda:0")


class Linear(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        n_pix = IMG_H * IMG_W
        self.fully_connected = nn.Linear(in_features=n_pix, out_features=n_pix)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        B, H, W = x.shape
        x = x.reshape((1, -1))  # flatten B,H,W -> B,H*W
        x = self.fully_connected(x)
        x = x.reshape((B, H, W))  # reshape B,H*W -> B,H,W
        return x

```

```python
# open image -> greyscale
img = Image.open("pisa/train_img.png").convert("L")
# set image array between 0-1 - we need a numpy arr to convert to tensor
arr = np.array(img) / 255.0
# sobel is an edge detecting kernel
edges = sobel_v(arr)

x, y = torch.Tensor(arr), torch.Tensor(edges)
# unsqueeze adds extra dimension i.e, vector shape (4) -> (1, 4)
x_batch, y_batch = x.unsqueeze(0), y.unsqueeze(0)

# setup net, loss and optimizer
net = Linear()
loss_function = nn.L1Loss()
optimizer = torch.optim.Adam(net.parameters(), lr=LR)
# move everything to GPU
x_batch, y_batch = x_batch.to(GPU), y_batch.to(GPU)
net = net.to(GPU)

for i in range(N_EPOCHS):
    optimizer.zero_grad()  # clear old gradients
    y_pred = net(x_batch)  # looks like a function!
    loss_value = loss_function(y_pred, y_batch)  # compute loss
    loss_value.backward()  # backpropagate
    optimizer.step()  # update weights
```

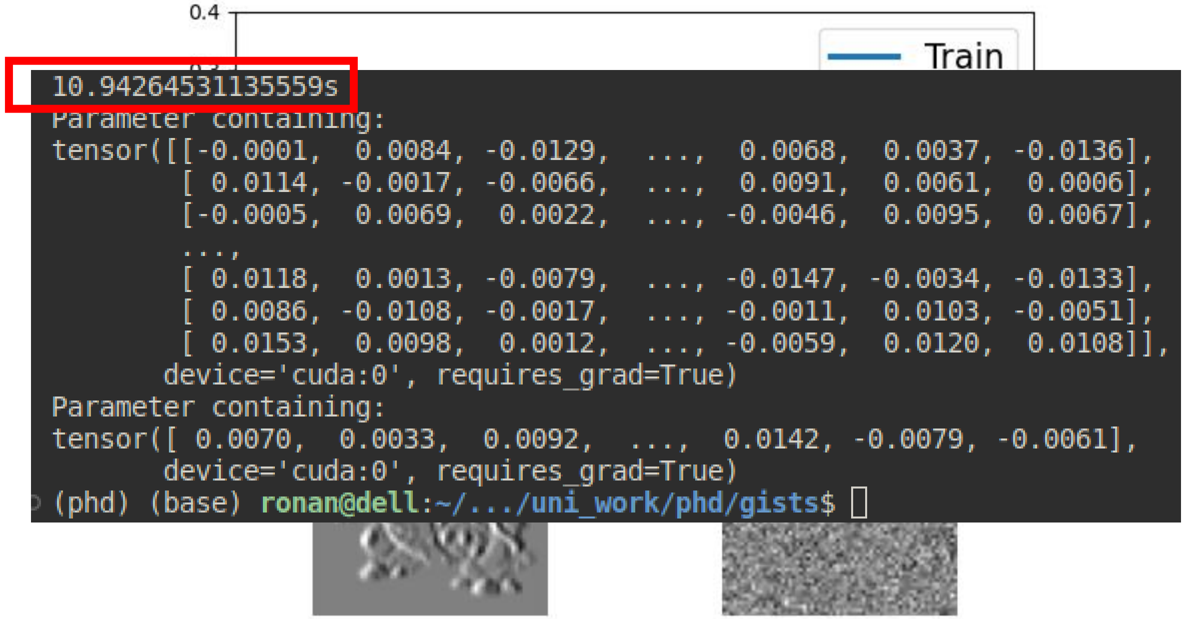
Why use CNNs? Results!

Convolutional layer

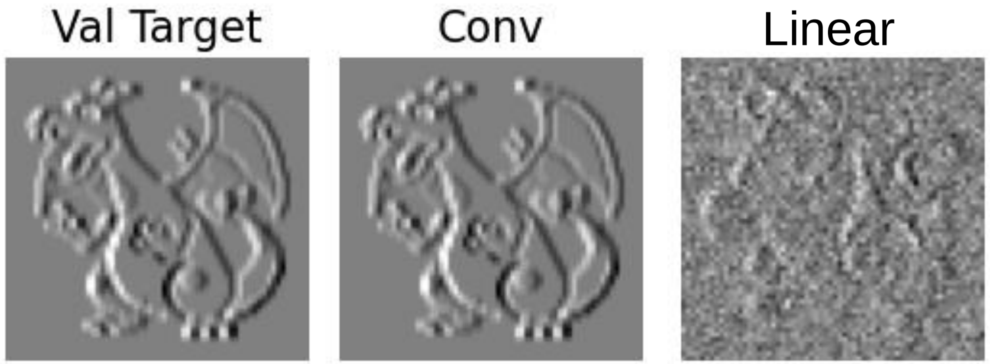


Up to a scale factor and transposition (images in pytorch are h,w), this is the Sobel Kernel!

Fully Connected layer



How do they generalize?



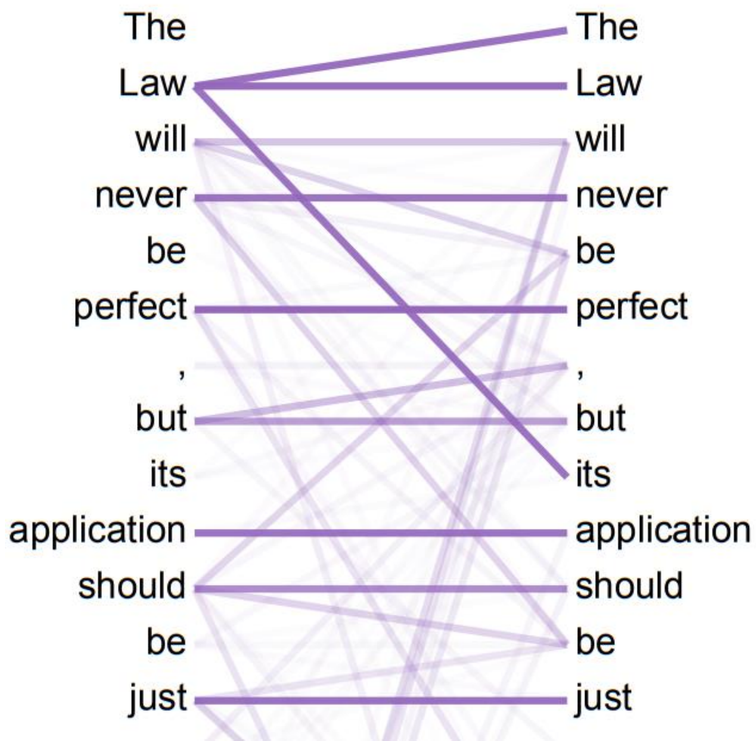
Clear overfitting from the linear layer

Attention

- For translation tasks, inputs were sequences of (embedded) words called ‘tokens’ [AI]
- Modelling whole context (adjective-noun, pronouns, …) important
- Q, K, V different learned projections of our (embedded) input token sequence (via FCs)
- Computes **pairwise similarity** of Q & K and matches them with V
- ‘How important is each bit of context to each token and how should I update its representation?’

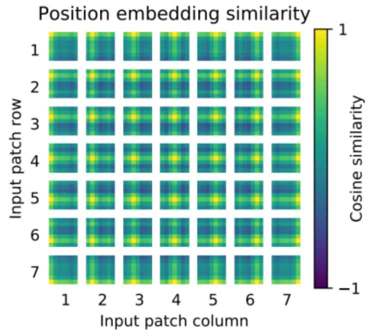
Top: attention equation. Bottom: attention map for a sentence. From [AI]

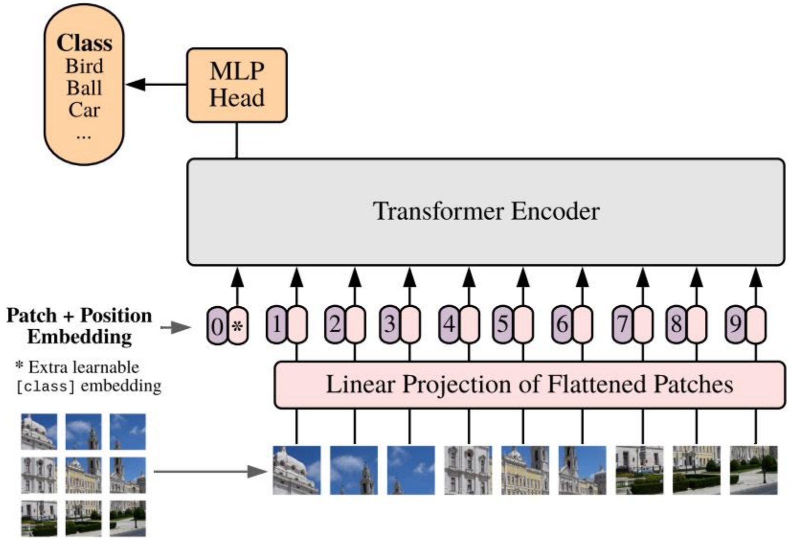
$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$



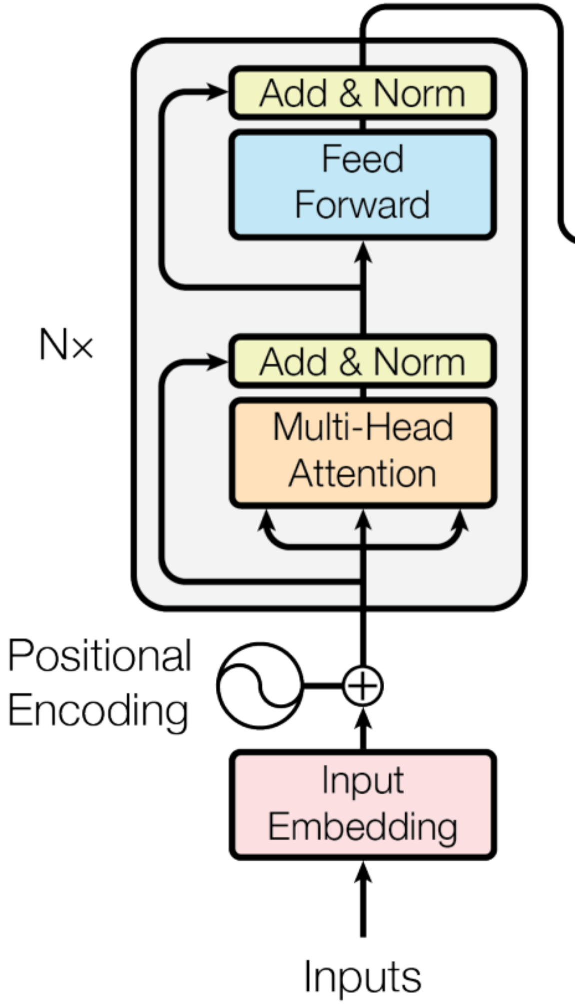
Transformers

- **Positional encoding** says where words are in sentence (i.e 1^{st}, 2^{nd}, …). Also works for images or anything encoded as a sequence
- All-to-all attention -> learns global features & propagates info easily
- $O(n^2)$ operations for *n* tokens – **expensive computationally**





Right: a simple Vision Transformer (ViT) [VT]. **Top:** cosine positional encoding of image patches [PE]



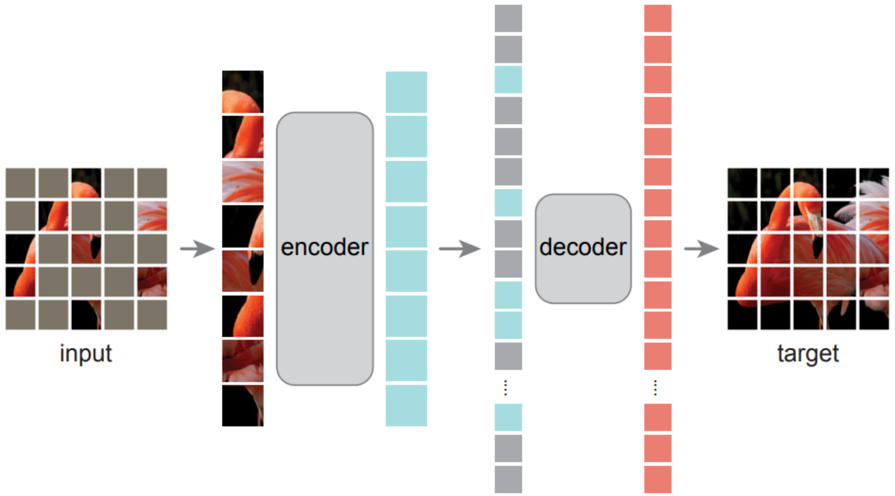
Transformer block. From [AI]

Content

1) Intro
2) A quick overview
3) The layers
4) Foundation models
5) Conclusion

Autoencoders & pretraining

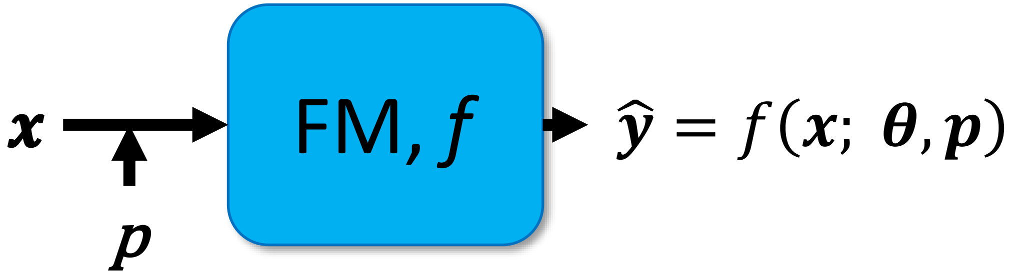
- Autoencoder: encoder makes hidden representation of input $\boldsymbol{x}$, decoder decodes to reconstruct original
- Masked Autoencoder: cover 70% of $\boldsymbol{x}$, encode, decode then compare to original
- Needs to learn general image features to do this over many images
- Super easy to get training data: download images and cover them with program
- ‘Self-supervised learning’

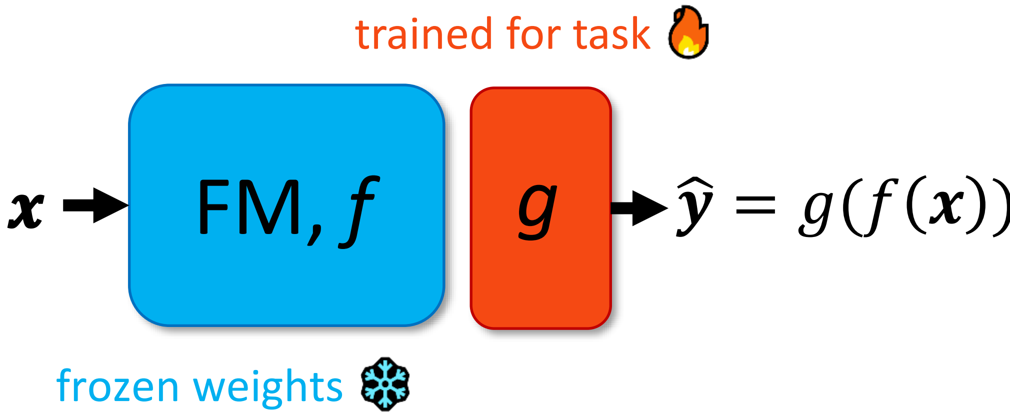


ViT MAE diagram for pre-training task. A similar principle was applied to text MAE for GPT. From [MAE]

What is a Foundation Model (FM)?

- Large (many parameters) model trained on lots of data for a long time
- Different training stages: self-supervised -> supervised -> reinforcement learning
- Designed to be applied to variety of tasks:
  - **Prompts:** additional user inputs that change output *e.g,* text in ChatGPT or DALL-E
  - **Adaptors:** train small head network to use rich FM representations for specific task
  - **Fine-tuning:** retrain all/some of the network (expensive!)





Example: ‘Segment Anything Model’

$\boldsymbol{x}$

$\boldsymbol{y}$

From [SS]

Left: example of a segmentation for self-driving cars.
Right: video of SAM producing instance segmentations ‘prompted’ at the mouse cursor.
Made using [SAM]

- Segmentation = assigning class to each pixel (*i.e*, 0=background, 1=foreground or 0=chamber, 1=catalyst, 2=bed, …)
- ‘Segment Anything Model’ = heavy autoencoder + promptable decoder
- Produces fg/bg segmentation given prompt (mouse click, bounding box)
- Decoder fast enough to run in real-time in browser!

Demo of my work!

Content

1) Intro
2) A quick overview
3) The layers
4) Foundation models
5) Conclusion

Key takeaways

$$
\begin{tikzcd}
\boldsymbol{x} \arrow[r] & f(\boldsymbol{x};\ \boldsymbol{\theta}) \arrow[r] & \widehat{\boldsymbol{y}} \arrow[d] \\
& & L(\widehat{\boldsymbol{y}}, \boldsymbol{y}) \arrow[ul, "\text{Update}"]
\end{tikzcd}
$$

1. NNs approximate the underlying function in our dataset $D = \{(\boldsymbol{x_1}, \boldsymbol{y_1}), (\boldsymbol{x_2}, \boldsymbol{y_2}), \ldots\}$ – they are **statistical models**
2. **Always** withhold part of D to evaluate (train/val/test split) model on, otherwise can't trust or compare results
3. **More data**, larger model (in proportion) => better results [BL] …
4. … but we can be **flexible & clever**, c.f single image models like SliceGAN [SG] or N2F [NF]

Field guide

- Use datasets, dataloaders (will need to write your own)
- Use Adam optimizer with default learning rate
- Easiest way to diagnose problems is to look at tensor shapes as they pass through layers
- See if you can adapt/finetune/integrate an existing network rather than train one from scratch (cheaper!)
- Recommended networks for problems:
  - **Image problems:** U-Net, Vision Transformer
  - **Text problems:** Transformers, Recurrent Neural Network (RNN)
  - **Predicting on tabular data:** Random Forests (XGBoost, LGBM)
  - **Time series prediction:** Long Short-Term Memory (LTSM) network

More reading

- Little Book of Deep Learning
- Pytorch intro/tutorial
- 3Blue1Brown deep learning series, especially his attention video
- Sam's (my supervisor) Coursera
- deep learning for molecules & materials
- Deep Learning Book (very rigorous)

Any questions?

Thanks to:

Supervisors: Dr Samuel J. Cooper, Dr Antonis Vamvakeros
Collaborators:

- Amir Dahari
- Lei Ge
- Dr Isaac Squires
- Dr Steve Kench

Website:
https://tldr-group.github.io/#/

Github:
https://github.com/tldr-group

Funders:
Centre for Doctoral Training in the Advanced Characterisation of Materials (CDT-ACM)
EPSRC and SFI

References

- [FA] K. Hornik *et al.* ‘Multilayer feedforward networks are universal approximators’. *Neural Network,* 1989 https://www.sciencedirect.com/science/article/pii/0893608089900208
- [AN] C. lb, ‘Artifical Neuron Model Diagram’ https://commons.wikimedia.org/wiki/File:ArtificialNeuronModel_english.png
- [St] R. Sebastian, Multi-Layer Perceptron https://rasbt.github.io/mlxtend/user_guide/classifier/MultiLayerPerceptron/
- [LL] A. Amini *et al.*, ‘Spatial Uncertainty Sampling for Endto-End Control 2019’, arXiv: 1805.04829.
- [Qt] A. Ananthaswamy, ‘How to Turn a Quantum Computer Into the Ultimate Randomness Generator’ https://www.quantamagazine.org/how-to-turn-a-quantum-computer-into-the-ultimate-randomness-generator-20190619/
- [OF] https://www.mathworks.com/discovery/overfitting.html
- [KO] R. Holbrook, https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting
- [AF] N.S. Johnson *et al.*, ‘Invited Review: Machine Learning for Materials Developments in Metals Additive Manufacturing.’ *Additive Manufacturing 36* https://doi.org/10.1016/j.addma.2020.101641.
- [FF] I. Khan, ‘From ANNs to RNNs’ https://medium.com/unpackai/from-anns-artificial-neural-networks-to-rnns-recurrent-neural-networks-93b638772fd1
- [LB] F. Fleuret, ‘The Little Book of Deep Learning’ https://fleuret.org/public/lbdl.pdf
- [SS] W. Gu *et al.*, ‘A review on 2D instance segmentation based on deep neural networks’, *Image and Vision Computing*, 2022, doi: 10.1016/j.imavis.2022.104401.
- [WC] M. Plotke, https://en.m.wikipedia.org/wiki/File:2D_Convolution_Animation.gif

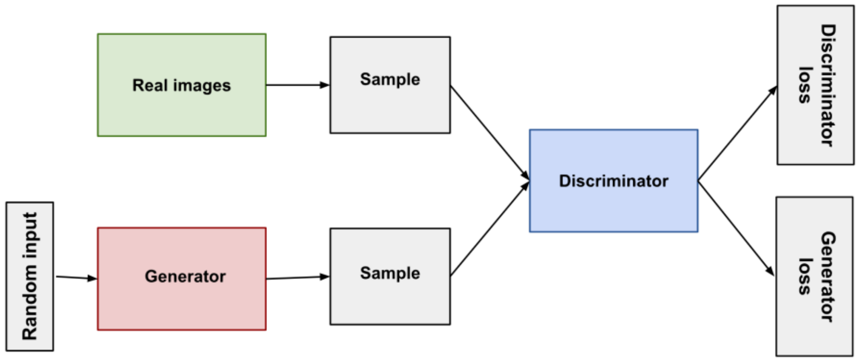
References (contd.)

- [VG] A. Dertat, https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2
- [UN] S. Cai *et al.*, 'A Novel Elastomeric UNet for Medical Image Segmentation', *Frontiers in Aging Neuroscience*, 2022, doi: https://doi.org/10.3389/fnagi.2022.841297
- [AI] A. Vaswani *et al.*, 'Attention Is All You Need'. arXiv, 2017. doi: 10.48550/arXiv.1706.03762.
- [GG] https://developers.google.com/machine-learning/gan/gan_structure
- [ST] G. Kogan, 'Experiments with style transfer', https://genekogan.com/works/style-transfer/
- [SG] S. Kench and S. J. Cooper, 'Generating three-dimensional structures from a two-dimensional slice with generative adversarial network-based dimensionality expansion', *Nature Machine Intelligence*, 2021, doi: 10.1038/s42256-021-00322-1.
- [PM] J. Stuckner *et al*. 'Microstructure segmentation with deep learning encoders pre-trained on a large microscopy dataset'. npj Comput Mater 8, 200 (2022). https://doi.org/10.1038/s41524-022-00878-5
- [VT] A. Dosovitskiy *et al.* 'An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale', *arXiv*, 2021. doi: 10.48550/arXiv.2010.11929.
- [PE]] S. Paul and P.-Y. Chen, 'Vision Transformers are Robust Learners', *arXiv*, 2021. doi: 10.48550/arXiv.2105.07581.
- [MAE] K. He, *et al.* 'Masked Autoencoders Are Scalable Vision Learners'. *arXiv*, 2021. doi: 10.48550/arXiv.2111.06377.
- [SAM] https://segment-anything.com/demo#
- [N2F] J. Lequyer *et al.* 'A fast blind zero-shot denoiser'. *Nature Machine Intelligence,* 2022, https://doi.org/10.1038/s42256-022-00547-8
- [LLM] Ge, Lei *et al.* 'Materials science in the era of large language models: a perspective', *arXiv*, 2024. doi: https://arxiv.org/abs/2403.06949

Extra slides

Example 1: GANs

- First real generative model
- **Game** between two networks: generator *G* makes fake data (from seed), discriminator *D* tries to distinguish between real and fake data
- Updating weights of *G* based on how *D* detected fake samples means it makes better fake data in future
- Two networks training at once -> **unstable!**
- Applications: face generation, style transfers, *etc.*

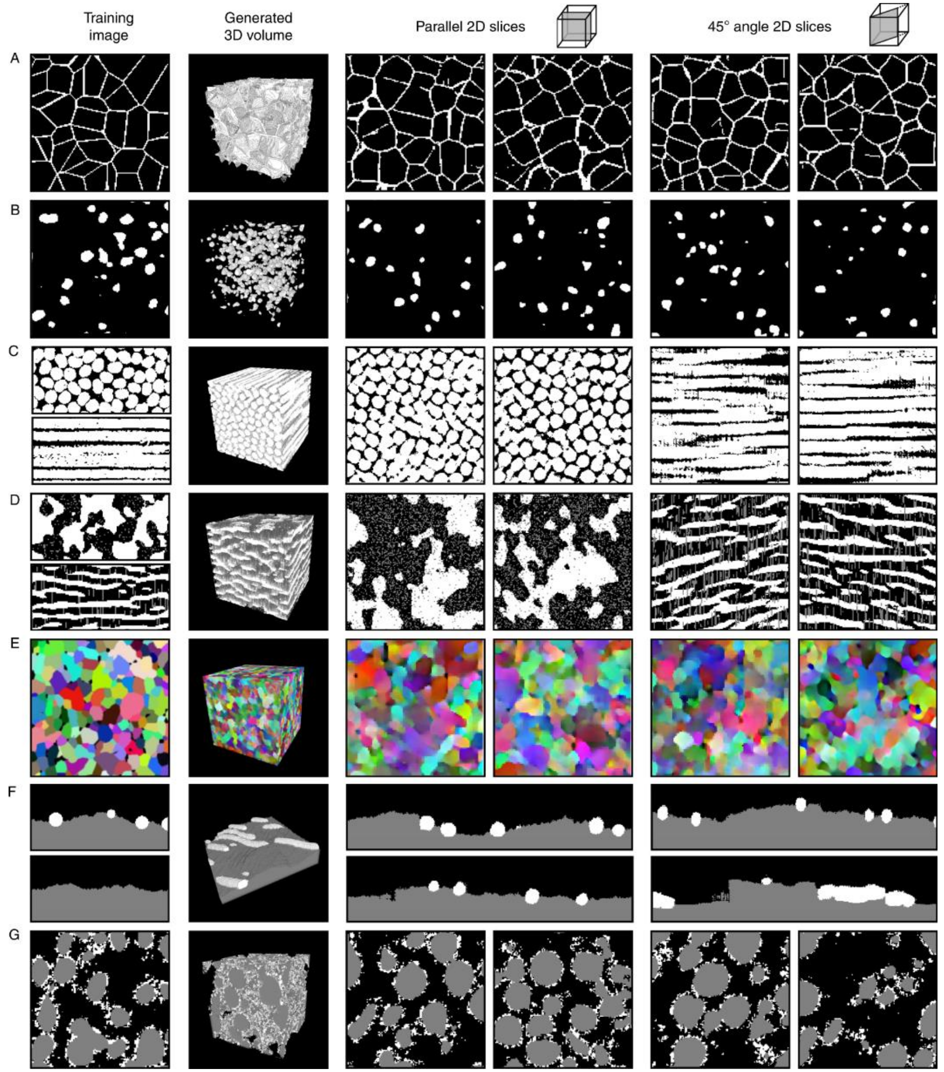


GAN architecture. From [GG]

HD style transfer on a tree. From [ST]

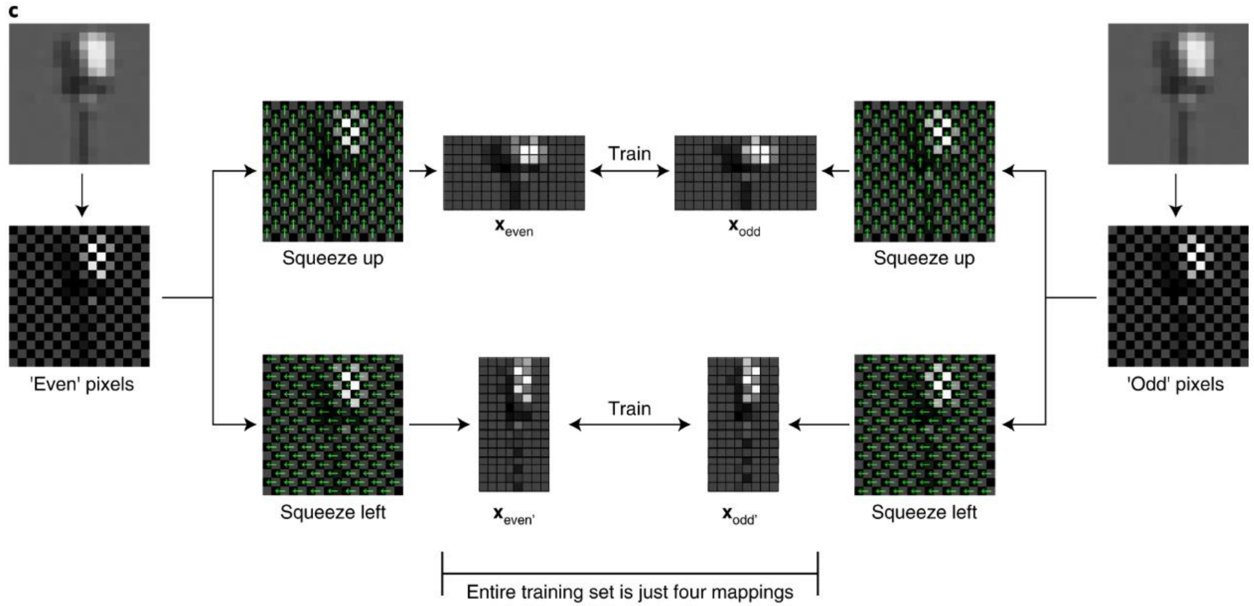
Example 1: 'SliceGAN'

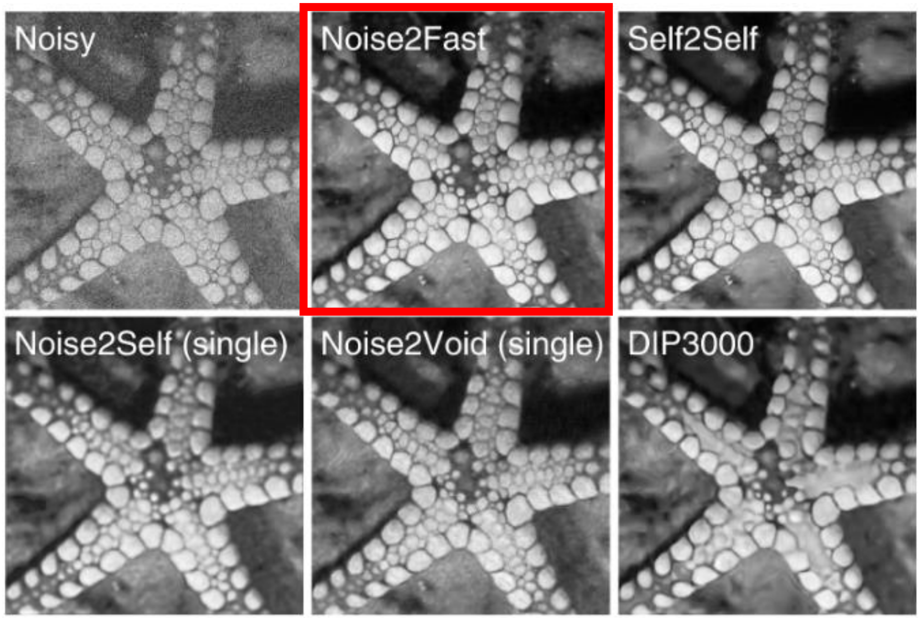
- 3D experimental data expensive (FIB-SEM) or not high resolution (μ-CT)
- Can we use a GAN to go from 2D -> 3D?
- Yes – *G* makes 3D volume which we slice in 2D and give to *D* alongside real 2D patches
- Key assumption: **homogeneity**
- When trained, *G* can many different volumes at any size
- Trained fresh on a single experimental image – 'material agnostic'



SliceGAN trained on different inputs. From [SG]

Example 2: ‘Noise 2 Fast’



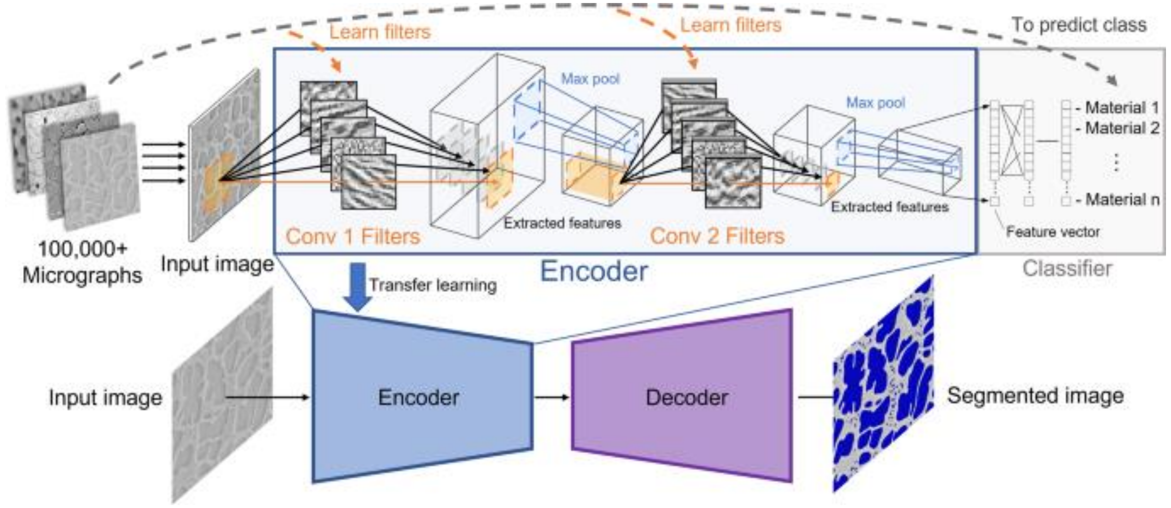


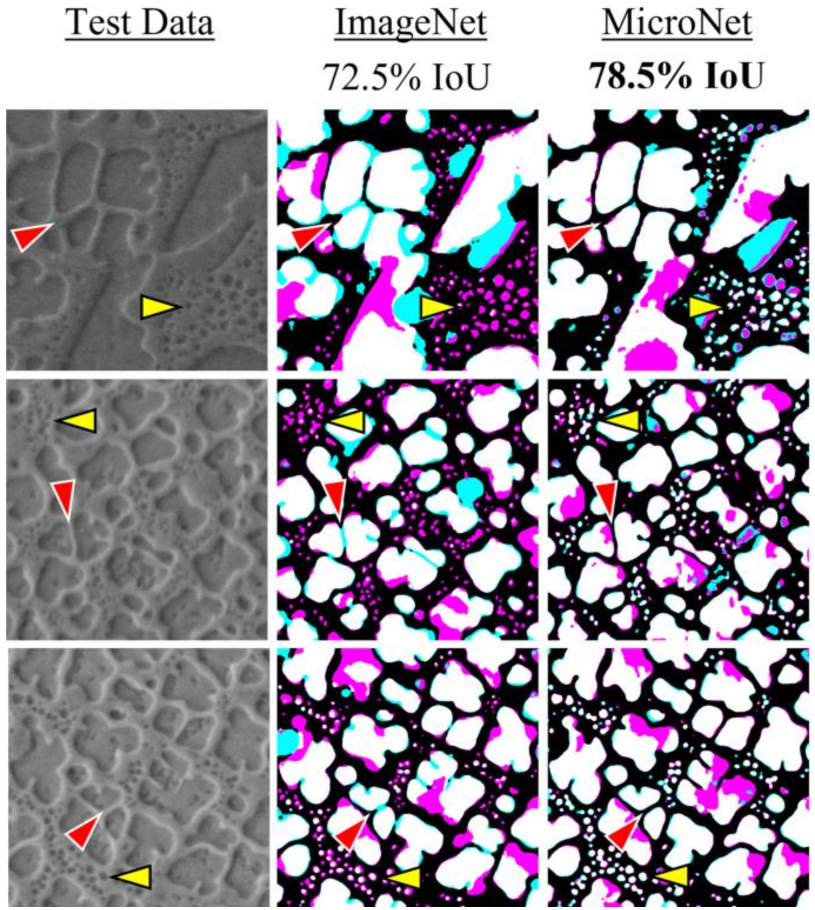
Left: N2F training process. **Right:** its application. From [N2F]

- In microscopy, often imaging something completely new (with noise!) – motivates models that denoise using a single image
- N2F trains CNN to map between ‘checkerboard downsamples’ of image
- Key assumption: **noise is spatially uncorrelated**
- Works well and trains fast, but must be trained for each image

Example 3: ‘MicroNet’

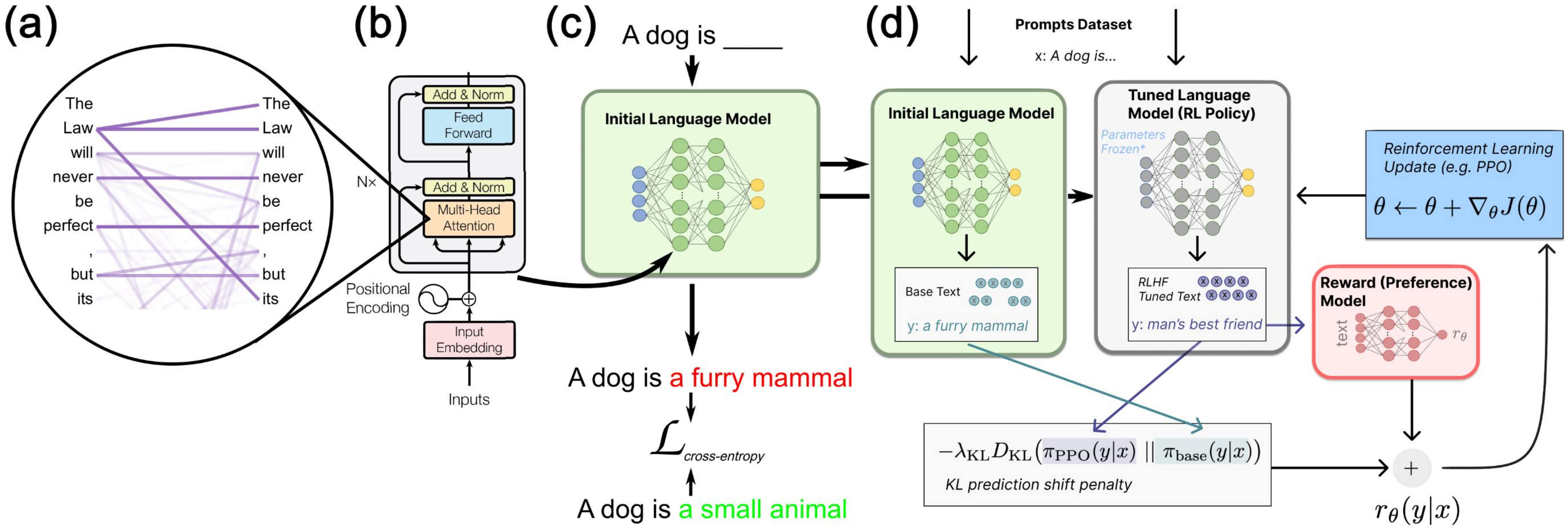
- U-Net/autoencoder architectures trained on large (100,000) micrographs to do multi-phase segmentation – useful for finding structure-property relationships
- Shows importance of using relevant training data and of feature-learning for downstream tasks





Left: model diagram – feature learning + classifier. **Top:** performance on test data, model trained on micrographs performs better. From [PM]

Example 4: ‘ChatGPT’



A multi-scale diagram of LLMs like ChatGPT. **(a)** attention mechanism on tokens, which forms part of the attention layer in the transformer block in **(b)**. Many of these are put into a ‘Large Language Model’ in **(c)** which is pretrained on masked language modelling, via cross entropy loss of predicted tokens. These models are aligned with human preferences via reinforcement learning in **(d)**. From [LLM]