Convolutional Neural Networks

Deep Learning (DL) systems are usually have 4 main components - 1) *Feature Extractor* (responsible for capturing hierarchical data patterns; i.e edges to help the network `understand' the data - CNNs), 2) *Task Specific Head* (component tailored to the problem), 3) *Parameter Optimization.*
For ϕ, a non-constant, bounded, monotonically increasing function, for any ε > 0, and any cont. fn defined on a compact of ℝ^m, there exists an integer N, real constants vi and bi and real vectors wi, with i = 1, ..., N, such that:
$F(x) = \sum_{i=1}^{N} v_i\varphi(w_i^T x + b_i)$ *with* $|F(x) - f(x)| < \epsilon$ ε is high; CoD means for more accuracy, need more layers = exp. more compute. DNNs help
The Curse of Dimensionality: As num features/dims grows, amt of data we need to generalise accurately grows expon: to optimize a Lipschitz cont. func f: ℝ^d -> ℝ with 1-ε accuracy, we need O(ε^-d) samples. e.g if in 1D space you want to approx a func; and you need 10 samples for ε acc in 1D, you'll need 100 in 2D. For 12 Mpixels * 3 channels = 36M elems, for 0.1 ε you need 10^36M...
Mass Concentration in High Dimensions: in higher D, most mass of objects resides near the boundaries. i.e: most data points lie near the edges of the data space. Consider: in nD, the nD volume of the interior is α^n times the original shape volume when we scale or shrink by α. The volume of the rind relative to original vol is 1-α^n. As a function of α the rate of growth is d(1-α^n) = -nα^(n-1)dα. With no shrinking (α=1) and noting that α is decreasing, the initial growth rate of the rind is n. The volume of the rind initially grows much faster - n times faster than the rate at which the object is being shrunk. In higher D, tiny changes in distances cause massive changes in volume. Miniscule changes in datasets can massively change the volume in DL
Half-Length: How much do you need to shrink object in all D to have 1/2 vol: Solve α^n = 1/2. In 2D: 1-0.35; so half the data lies within 35/2 = 18% of its diameter from boundary. In 350D: 1-0.02..; half the data is in 1% of diam from b. Distance measures lose effectiveness in measuring dissimilarities in high D.
Invariance: no matter how the input is transformed, the output is invariant
Equivariance: the output is transformed in the same way as the input
Shift-Invariance: response is the same when input (like im pixels) are shifted. Important in CNNs, which have SI, which is why they're good at image tasks.
Shift-Equivariance: the response shifts by the same amount as the input shifts. ie: for a model that segments an image, marking cat pixels "1" and non cat "0", if the cat in the image moves, so should the segments in the same way.
ie: shift and then mark is the same as mark and then shift. s∘f(x) = f∘s(x)
Locality: You don't have to look far away from a location (i,j) to find info about it The assumptions of locality and translation invariance have guided model dev.
Correlation: involves sliding a subimage across a base image and comparing pixel values. While shift-invariant and local, it doesn't always find the best match. Convolutions help by focusing on important features for matching.
FCNs: Each input is connected to each node in the subseq. layer. So every single input feature influences every neuron. Very flexible, can capture *any* relation, but very expensive and easily overfits. Consider 36M inputs (image); we end with 36M^2 params which is astronomical. More sparsely connected nets are better; these paved the way for CNNs.
Weight Sharing: A subset of weights are identical and used across multiple connections. Sparsity + Weight sharing make CNNs very powerful!
$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \ \ for\ f, g : [0, \infty) \to \mathbb{R}$$
Convolutions: (Correlation has a + instead of -, and doesn't involve flipping the kernel). Both Conv and Corr are v. similar for ML models. Both aim for local pattern detection thru shared weights. We use discrete conv. on arrs: Given arr u_t, and w_t, the conv is the func s_t. Where u_t and w_t aren't defined, we assume their values to be 0.
$$s_t = \sum_{a=-\infty}^{+\infty} u_a w_{t-a}$$
Mathematically, conv = cross-correlation between f(x) and g(-x) or f(-x) and g(x)
Properties: 1. *Commutativity*: f*g = g*f 2. *Associtivity*: f*(g*h) = (f*g)*h
3. *Distributivity*: f*(g+h) = (f*g) + (f*h) 4. Assoc w/ scalar mult: a(f*g) = (af)*g
CNN Building Blocks: 1. *Convolutional Layers:* core of the CNN, where kernels and filters slide across input producing feature maps. Catches local patterns and details like edges and textures, making them very suited for image tasks
2. *Activation Functions:* Introduces non linearity, allowing the model to learn complex mappings from input to output. ReLU is commonly used as its good
3. *Pooling Layers*: Takes the input of the previous layers and summarises/ simplifies it. This reduces the spatial dimensions of the feature maps, reducing the computational burden. Max-Pooling commonly used.
4. *Fully Connected Layers*: used towards the end of the model. Each neuron conns to each in prev layer. Integrates learned features for final prediction.
5. *Flatten Layers*: Bridge b/w conv layers and fully connected layers by converting 2D feature maps into 1D vectors.
6. *Dropout*: Regularization technique; randomly zero some neuron weights. Reduces overfitting and makes the network more robust.
7. *Batch Normalization*: This keeps the distribution of each layer's outputs stable, aiding in faster and more reliable training convergence.
8. *Softmax Layer:* commonly found at the end. It converts the final layer's outputs into a probability distribution, useful for classification tasks.
9. *Loss Function:* measures training perf for backpropagation. Cross-entropy loss is commonly used for classification tasks in CNNs. Loss quantifies how well the model's predictions align with the true labels.
10. *Optimizer*. Adjusts the network weights based on gradients found during backprop. SGD (Stochastic Gradient Descent) and Adam are popular choices.

Kernels: For a typical NN you'd have took an image and flattened it into an X*Y*Z sized tensor. But these do not take advantage of our assumptions: Locality and Translation Invariance! Kernels do, and thus CNNs do by doing a [X, Y, Z] * [I, J, K] conv to make feature maps; I = kernel height, J width, K depth. The resulting feature map is of size [X-(I-1), Y-(J-1), 1] w/ stride = 1, padding = 0 and if we're using just 1 filter - conv layers can learn multiple filters or kernels at the same time if they're all of the same dimension. Each kernel's feature map can then be learnt on. To preserve the [X, Y] size we can use padding; we can zero pad the edges of the image with [I/2, J/2] pxls. This is helpful if our image is quite small and we want lots and lots of conv. layers (which otherwise decrease the size of the image).
At its core a CNN is just a seq. of conv. layers with activation funcs.
Each filter has *I*J*K+1* (bias) parameters to learn. This is a lot less than FCNs! 1x1 convs are used to reduce NN depth.
Stride - the step size a filter takes as it slides across an input image. Strides are used to reduce the spatial dimensions of the output volume, which in turn reduces the amount of parameters and computation in the network.
Padding - adding extra pixels around the edges of the input image.
Computational Complexity: assume a 5x5 convolution. It involves 25 multiply-and-add ops per input element. But if we use two consecutive 3x3 convolutions instead which can be constructed to produce the an approximation of the same output, they only take 18 total ops per element. But the second conv has to work on even fewer elements because of the shrinking thanks to the first conv as well - so the efficiency gains are even better! We also go from 25 to 18 param
Separable Convolutions: You can approximate a 5x5 conv with a 1x5 conv followed by a 5x5 conv - if the 5x5 conv is separable. It works well only if the original conv is approximable. Example: Consider an image of dim (D, M, N). If M=N=32, and D=3, the 1x5xd conv then 5x1xd conv is: 32x28x3x5x3+1 + 28x28x3x5x3+1 = 75602 ops. But the 5x5xd is: 28x28x3x5x5x3+1 = 176401.
Pooling: Permutation-Invariant - involves aggregation+downsampling. Reduces resolution, helps the network concentrate on increasingly abstract features - adding a level of hierarchy, adds shift invariance. MaxPooling, AvgPooling. Pooling is applied to each channel seperately, e.g: RGB on images.
Nyquist Sampling Theorem: a continuous signal can be completely represented by its samples and fully reconstructed if it is sampled at least twice as fast as its highest frequency component. By aliasing via pooling/convs we lose info - we no longer adhere to the theorem. It's found that Convs/Pooling only improves results when used properly and carefully.
Image Frequency: An image has high frequency if it "oscillates" a lot in colour. High freq images have lots of edges, textures and fine details (high gradients). Max Pooling breaks shift equivariance. A solution is to use a Gaussian Blur to downsample. Sadly, CNNs are not rotation invariant; but this can be achieved with spherical harmonics (or remember HoG?).
CNNs are largely deformation invariant - f(x) ≈ f(D_t x); and this makes them amazing for image classification.
Flattening Layers: connect conv and fully conv layers, as conv layers output a 3D tensor of learned features, and FC layers expect a 1D tensor. Flattening layers use 1x1 convs to do this. i.e: they take a 4x4x64 tensor into a 1024x1x1

Activation Functions and Loss

Activation Functions: they determine how neurons in the network activate with a set of inputs. Introduces non linearity, allows learning complex bhviour
1. *Naive Activation*: if the input is above a specific value, declare the neuron activated (output = 1). Not differentiable, so cannot backprop.
2. *Linear Activation*: c*x. Constant grad, no relation to x during backprop.
3. *Sigmoid Activation:* output = 1 / (1+e^-input), nonlinear - allows us to stack layers. Continuous output; differentiable. Curve is steep between 2 and -2 giving a lot of feedback during gradient descent, allowing for quick learning Used a lot in foundational ML, but nowadays alternatives are chosen.
4. *Tanh*: (e^x-e^-x)/(e^x+e^-x). It's chosen when we want a scaled sigmoid with output from -1 to 1 rather than 0 to 1, and steeper gradients than sigmoid.
5. *ReLU*: max(0, x). Nonlinear, especially thanks to the sharp corner at origin. Combos of ReLU are non linear. Unbounded, useful for some types of data but causes activation explosion in others. ReLU often outputs 0, because of its response to negatives which causes some neurons to be ignored making the net more computationally efficient. But, this sparsity causes the dying ReLU problem - if a neuron output is always 0 then the gradient is too; preventing that portion of the NN from being updated through backprop. ReLU is used a lot due to efficiency and effectiveness in many applications.
6. *Leaky ReLU:* output = x for x >= 0, 0.01x for x < 0. Has a small slope for negative values (nonzero gradient) avoiding the dying ReLU problem.
7. *PReLU*: output = x for x <= 0, *a*.x for x < 0. *a* is learnable, and either one *a* is used for all channels or separate one for each input channel (more nuanced behaviour during training at the cost of extra computational complexity). All the ReLUs are scale invariant which makes them suited for some tasks.
8. *SoftPlus*: 1/β * log(1+e(β*x)). This is a smooth (easier to differentiate) approximation of ReLU, with only positive output. The higher β is, the closer to ReLU it is; and we default to linear if β.x > thresh. Smoother, differentiable alternative to ReLU that is desirable when we want positive activations and numerical stability.
9. *ELU*: max(0, x) + min(0, α * (e^x-1)). It is element wise, operating one each input element independently. ELU can go beneath 0, and is parameterized by α. It's a soft version of ReLU, with mean closer to 0 (thanks to being able to output negatives) - this helps the network converge quicker. It's good when we want a balance of smoothness, differentiability, and the ability to have a mean around 0. This is a unique blend of feature.
10. *CELU*: max(0, x) + min(0, α * (e^(x/α)-1)). ELU variant; element wise. Continually differentiable when α not equal to 1. Builds on ELU w/ more mathematical rigour to suit certain applications and optim scenarios.
11. SELU: scale * (max(0, x) + min(0, α * (e^x-1))). It comes with predefined α and scale which have been meticulously optimized. SELU allows for internal normalization. α and scale are solutions to a fixed p eq that maintain mean 0 & var 1 across layers. Normalizing activations in an NN can happen at 3 levels. The first is input norm, where we scale input features, e.g grayscale pixel vals, into a specific range, such as 0 to 1. The second level is batch normalization, a technique specifically designed for NNs to stabilize the learning process. SELU shines at the third level, which is internal normalization. SELU ensures that mean & var of activations are preserved one layer to the next. SELU has pos and neg outputs to have a mean of 0. It has gradients close to 0 which helps normalization. SELU avoids the "dying unit" problem that plagues. SELU offers a way to build DNNs without worrying too much about manual normalization. Keeps internal NN statistics stable
12. *GELU:* x*Φ(x); Φ is the Gaussian CDF. This adds probability to the activation process. This can cause regularization; good for training & perf
13. *ReLU6*: min(max(0,x), 6) - this caps the ReLU to a value. This saturates activations preventing excessive growth and numerical instability. The max value of the net can be configured for different saturation levels.
14. *LogSigmoid*: log(1/(1+e^-x). Operates elemwise. Typically used in cost functions rather than activations.
15. *SoftMin*: e^-xi / Σj e^-xj - this applies softmin on an n-dim tensor. Rescales them so the elems of the output lie between [0,1] and sum to 1. Acts as a method to convert a set of nums into a representation that resembles a prob dist; valuable in scenarios where you want relative weights or preferences among alternatives. Softmin's role lies in rescaling and transforming input tensors into prob dists, contributing to the multi-dim non-linearities of NNs and enables the modelling of various factors in the data.
16. SoftMax: e^xi / Σj e^xj - Common and important. Like softmin, rescales elems b/w [0,1] and all sum to 1. Key use case: convert raw scores (logits) generated by a NN into meaningful class probability scores.
17. LogSoftmax: log(SoftMax). Beneficial as it gives numerical stability. Good for loss functions for NNs, rather than in activations.
18. SIREN (Sinusoidal Representation Network) activation:
$$\Phi(\mathbf{x}) = \mathbf{W}_n(\phi_{n-1} \circ \phi_{n-2} \circ \ldots \circ \phi_0)(\mathbf{x}) + \mathbf{b}_n, \quad \mathbf{x}_i \mapsto \phi_i(\mathbf{x}_i) = \sin(\mathbf{W}_i\mathbf{x}_i + \mathbf{b}_i)$$
Periodic activation - useful but can be hard to use. Useful for dealing with implicit representations -i.e: finding a cont. func representing sparse input, such as images, waves. Great for spatial/temporal derivatives of signals.
Error/Loss Functions: quantify how well the network performed, allowing for backprop where we can update model params. Loss functions do this by measuring the distance between the produced output and ground truth.
1. *L2 Norm:* (MSE): Make samples from same classes close and different classes far away. Objective: Distance for the good pair has to be smaller than distance to the bad pair. Actual distance does not need to be small, just smaller. Used for metric learning and Siamese networks (minibatched), $l_n = (x_n - y_n)^2$
2. *L1 Norm*: $l_n = (x_n - y_n)$ (penalises small outliers more; good for noisy data) Not differentiable at 0, so can use smooth versions
3. *Smooth L1 Loss:* $z_i = 0.5(x_i - y_i)^2$ if $|x_i - y_i| < 1$ else $|x_i - y_i| - 0.5$. balances L1 and L2. 0.5 = scale factor (hyperparam) - when we transition b/w linear/quad.
4. *Negative Log Likelihood Loss*: Assumes net output is log likelihoods. We want to make the desired class output big, others as small as possible $l(x,y) = L = \{l_1, ..., l_N\}^T$, $l_n = -w_{y_n}x_{n,y_n}$ w_c = weight[c] . 1{c =/= $ignore_{index}$}
Reductions are used to aggregate the NLL scores per class:
   - if using mean reduction: $l(x,y) = \Sigma_{n=1\ to\ N}\ 1 / (\Sigma_{n=1\ to\ N}\ w_{y_n})\ l_n$
   - if using sum reduction: $l(x,y) = \Sigma_{n=1\ to\ N}\ l_n$

   We use weights to give diff importance to classes e.g: imbalanced dataset
5. *Cross Entropy Loss*: combines LogSoftmax and NLLLoss
- Useful for classification problems with C classes:
  loss(x, class) = -log(e^x[class] / Σj e^x[j]) = -x[class] + log(Σj e^x[j])
- Classes can be weighted. Losses avg across observations for each minibatch
- x[class] is the correct classes' logit. esries = all logits.

6. *Binary Cross Entropy Loss*: As above but designed for 2 classes:
- $l(x,y) = L = \{l_1, ..., l_N\}^T$. $l_n = -w_n[y_n . \log x_n + (1 - y_n) . \log(1 - x_n)]$ q
- If you don't have probabilities in range [0,1] you need to use "WithLogits"
- y_n = true label (0/1), x_n = predicted prob of class 1.

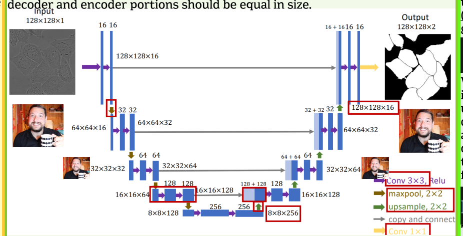
7. *Kullback-Leibler Divergence Loss*: measures distance b/w distibs y_n, x_n
$l(x,y) = L = \{l_1, ..., l_N\}^T$; $l_n = y_n . (\log y_n - \log x_n) = y_n (\log (y_n / x_n))$
Particularly useful for one-hot representations (single category is 1, rest 0) We assume probabilities are in [0, 1]. Numerical instability w/ low probabils
8. *Margin Ranking Loss*: loss(x, y) = max(0, -y . (x_1 - x_2) + margin). Useful to push classes as far away as possible (we want at least margin sized diff). y = 1 or -1 choosing which class should be larger. Measures diff. b/w emb repres. Goal: train feat extractors for similar reprs for similar inps; distinct for diff
9. *Triplet Margin Loss*: Make samples from same classes close and diff classes far. $l(x,y) = L = \{l_1, ..., l_N\}^T$, $l_n(x_a, x_p, x_n) = \max(0, m + |f(x_a) - f(x_p)| - |f(x_a) - f(x_n)|$ (x_a = actual, x_p = positive sample, x_n = negative sample)
Objective: distance for the good pair must be smaller than dist to the bad pair. Actual dist doesn't need to be small, just smaller. Used in metric learning and Siamese networks.
10. *Cosine Embedding Loss*: if y = 1: loss(x, y) = 1 - cos(x_1, x_2)
if y = -1: loss(x, y) = max(0, cos(x_1, x_2) - margin)
Measures whether two inputs are similar or not. Similar to a normalised Euclidian distance but focuses on angles b/w the inputs not magnitude.

Network Architectures

LeNet: First designed for low-res, b&w image recog, specifically for digits. Input is 32x32 pixel grayscale, then *conv1 layer* makes 6 28x28 feature maps, then *avg pool layer* reduces dimensionality to 14x14, 6 channels (pooling applied to each channel, not across them so num channels unchanged). Then *conv2 layer* -> 10x10, 16 channels. *Avg pooling* -> 5x5, 16 channels. *Fully connected layers*, first 120 units, second 84 units then uses *Gaussian Radial Basis Function (RBF)* to map features to 10 classes. LeNet showed CNNs can in both *obj localisation and recognition* (detecting numbers at post office)
MNIST: Large dataset of handwritten digits. Centred and scaled, 28x28 (low computation cost), 50k training and 10k testing imgs, 10 classes for 0-9 digits.
ImageNet: 1.2mil imgs, 1K classes, dims ranging from 496x384, RGB.
Fully connected layers used to convert features to classes, simple for 10 classes in MNIST but *complex/bottleneck* for thousand+ classes in ImageNet.
AlexNet: Outperformed non-deep learning models in ImageNet task. Similar to LeNet but w key diffs. Has more layers - 5 conv & 3 fc layers, 224x224 RGB (3 channels) image input. 11x11 filters w stride=4 (smaller filters can be better), fc layers size=4096 & last layer connects to 1000 classes via softmax (instead of Support Vector Machine tech used previously). Uses *dropout regularisation* across the layers which allows for much deeper networks - can control complexity of model more effectively, also uses *Rectified Linear Units (ReLU)* for activation funcs that help mitigate *vanishing gradient problem*, *max pooling* used instead of avg pooling - learned features more *shift-invariant*. Also used heavy *data augmentation*: *cropping, shifting* & *rotation*. *Kernel size* and *stride increase* to handle larger, more complex images. More output channels mean the network can learn a *greater variety of features*. Hidden layer size inc from 120 units to 4096 in AlexNet. AlexNet is 250x more comp'ly exp than LeNet-5, and has 10x params, tradeoff b/w comp & mem.
VGG: Visual Geometry Group, CNN evolved. Uses *parameterised block*s that expand into *large networks* rather than specifying each layer by hand. Analysis shown that using more layers of narrow convolutions outperforms using fewer wider ones ie more layers of simpler funcs generally more powerful than fewer layers of more complex funcs. VGG block has *several 3x3 convs*, padded by 1 maintains *spatial dims* from input to output layer. End of block has 2x2 *max-pooling* w stride=2 - halving resolution. VGG *stacks* these blocks combining with *dense layers*, much like AlexNet. Number of blocks denotes model name, VGG-num (like VGG-16 or VGG-19). VGG has *slower throughput* compared to AlexNet, but is way *more accurate*.
Inception (GoogLeNet): Deep network that introduces *parallel paths*. Data flow through different paths which allows to capture different types of features more effectively. Combines best of diff types of convs & pooling to enhance performance. Unique parallel struct is go-to for many complex tasks. *Inception block* will try *multiple* different convs (1x1, 3x3, 5x5, or even 3x3 max pooling) in parallel so that all types are covered (can get more complex) then are concatenated along output channel. Appropriate *conv padding* needed (3x3: half-padding by one, 5x5: half-padding by two) for aligning dimension inputs and outputs. Then stack the blocks. Different channels focussed on different tasks (one on cats, other on bird recog, etc). The spread of channels used is determined, ie sum to 256 (1x1:64, 3x3:128, 5x5:32, etc). *Inception block designed to use relatively low param number and FLOPs w/o sacrificing performance*. (256 output channels: 160k params & 128MFLOPs), only using 3x3 conv would cost 3x more in both. 8x for 5x5.
Inception block operations: Given conv kernel of size KxK, comp cost is K^2 * Cin * Cout * H * W (stride=1, inf 0-padding, omit additions for simplicity) H and W are fixed but Cin and Cout are customisable. Total comp cost for inception block is Cin * H * W * sum(K_j² * Cout_j), for different paths j. We can optimise network performance by carefully allocating resources - varying channel numbers and kernel sizes etc. So inception blocks allow for judicial allocation of comp resources across diff paths, offering more efficient sol than using single type conv throughout the arch. Combines benefits of various convs & pooling while optimising comp cost. The arch of the entire network is in 5 stages that include 9 inception blocks (stage 1&2 normal conv and pooling layers, stage 3 has 2 inception blocks with max pooling at the very end, stage 4 has 5 inception blocks with pooling at the end, stage 6 has 2 blocks with global avg pool) then there is a FC layer to the output. The first two stages are similar to AlexNet but the Inception network retains more original information due to output channels and original size feature map. Stage 3 dynam'y allocs channels across diff conv paths inc'ing from 256 to 480 channels while reducing spatial res from 28x28 to 14x14. Improves efficiency by balancing comp complexity and perf, adapting to resource constraints and data complexity. In stage 4&5 similar, inc'ing channels before maxpooling bc pooling reduces res so better to store info in channels. 1024 output channels, aligned w class num being predicted. Global avg pool over 7x7 allows net to make better choices based on essential feats in spatial dims.
Flavours: Diff versions of inception network (v2 has batchnorm, v3 asymm conv kernels - more efficient with less params per conv, v4 residual connections - ResNet inspo), *AmoebaNet* auto-gens the arch of the net in survival of fittest approach. V4 doesnt outperf ResNet.
BatchNorm: Deep nets struggle to converge (training difficult and time-cons), "deep supervision" (backprop loss from intermed stages) can be used but has limitations. Earlier layers when adjusted by loss that feed back to later layers that have already adjusted causing readjustment, destabilises next layer so slow converge. BatchNorm normalises feats within mini-batch which stabilises training and speeds up convergence.
$$\mu_B = \frac{1}{|B|}\sum_{i \in B} x_i \ and\ \sigma_B^2 = \frac{1}{|B|}\sum_{i \in B}(x_i - \mu_B)^2 + \varepsilon \qquad x_{i+1} = \gamma\frac{x_i - \mu_B}{\sigma_B} + \beta$$
(mean, variance)
We adjust mean and var of layers during training. Mu (mean) and sigma (var) for each mini-batch computed, then re-norm each input feature using them. Gamma and Beta are learned params, Gamma scales and Beta shifts the features too. Epsilon is added for numerical stability (small const.). Idea to adjust first two eqs sep from network learning. Using these eqs allows to norm each mini-batch separately, making training deep nets more stable and faster. Gamma and Beta are learned allowing some flex to undo norm if thats more beneficial for the task. Was supposed to reduce covariate shift but doesnt, but still aids convergence bc BatchNorm is just regularisation (reg) by adding noise to the model. Small sample mini-batch mu and sigma subject to noise, augmenting input based on noisy stats (so dont need other reg methods like dropout), if batch too large then not enough noise for reg. Batch too small the noise level is counterprod, affecting convergence. *At inference time* we have fixed learned Gamma and Beta and we use running avg of mu and sigma from training to normalise the features.
Dense layer: One norm for all, Conv layer: One norm per channel.
ResNet: Solves *vanishing gradients* in deep nets, ensuring *effective grad prop*. Deeper nets can model more complex funcs but may *diverge* from optim func instead of improving. Ideally func classes are *nested* meaning deeper layers should refine instead of changing learned func. ResNet helps maintain nesting making deeper nets more *stable* & *effective* in aprrox'ing the *true function*. ResNet parameterises around *identity func* f(x) = x instead of 0, so simplest func it can learn is identity - *reduces comp overhead* as model doesnt need to learn this from scratch. ResNet blocks use *residual connections* that add input to the output (*before activation*) which helps gradients flow & enables deeper nets. This *prevents degradation* issues & allows for deeper models that approx func classes more effectively. *Each block generally has 2+ convs, each followed by batchnorm and ReLU*. Sometimes we do 1x1 conv on residual connection pre-adding to output esp if need to change dims to match output. The identity mapping allows net to decide if changing from it is beneficial. The arrangement of batchnorm and relu makes the struct powerful and comp eff.
DenseNet: Densely Connected Convolutional Network, each layer connected to every following layer (unlike trad CNNs only connect to next layer). Uses higher order `Taylor Series' expansion: x_i+1 = [x_i , f_i(x_i)]. x_1 = x; x_2 = [x, f_1(x)]; x_3 = [x, f_1(x), f_2([x, f_1(x)])]. Pro of dense connectivity is feature reuse as each layer gets feature maps from all preceding layers. Reuse makes net highly eff also reqs fewer params for high perf. Comp&Mem eff. Dense conns helps w vanish grad prob. Arch acts as regularisation, reducing overfit even if model large. DenseNet is scalable, can easily adjust num layers & other hyperparams to make suitable for variety of tasks and data sizes.
Takeaways: Deeper doesnt auto-mean better, esp if func space isnt properly regularised. Reg'n techniques help constrain func space, thereby making net generalise better to unseen data. Arch is important but how you train it is equally as important as well as the problem you are tackling.

U-Net

Image Segmentation: the task of taking an image and assigning each pixel to a class label representing the "segment" of the image it is a part of.
U-Net: CNN architecture built for image segmentation. Input: RGB im.
Architecture:
1) We have a contracting path (encoder): built from convolutional and max pooling layers to extract image features.
2) We have an expanding path (decoder): built from convolutional and upsampling layers to build the segmentation
3) Crucially we have "skip connections" which concatenate the contracting path's layer outputs to the corresponding level layer on the decoder side. Note that we have padding to ensure these features match in size - the concated decoder and encoder portions should be equal in size.



- Doesn't have to be grayscale image - can use 3 input channels.
- We can also use more/less features, and change depth of net. But the network should be symmetric (or rather, the concatted sizes should be the same) - and the input and output should be the same size. Pad if necessary!

Variants of U-Net: 1) 3D-UNet: replaces the 2D ops with 3D ones.
2) H-DenseUNet: segments liver and tumors. Connects 2D & 3D unet.
3) UNet++: uses densely connected decoder nested sub-networks
4) nnU-Net: Automatically configs itself (including pre-processing, building, network architecture, training, post-processing) for any task. Mainly medical
5) GraphU-Net: For graphical data

Input Augmentation and Anomaly Detection

Input Augmentation: Artificially inflate training data size through applying expected transformations during training. Helps regularize against overfitting
- Example augmentations for images: flipping, scaling, rots, intens/contrast variations, crop/pad, noise, affine and perspective transformations
- Augs we do should only produce items that'd plausibly be in the test set.
- *Considerations*: assess impact on validation perf, consider real-time vs computing augmentations in advance.
- *Test-time Aug*: applying augs to test data during inference and aggregating preds. Improves robustness, acc, an ensemble model effect, and sampling posterior. Takes more computational resources; diminishing returns. Behaves like MC-*dropout*, TTA causes aleatoric (data) uncertainty, dropout causes epistemic (model) uncertainty.

Anomaly Detection: From samples of one class, pick out the unusual ones.
- Approach: build an autoencoder than can predict the continuation of the distribution function. We measure the dist b/w our input and continuation in latent space, if it's beyond a thresh then we've found an out-of-dist item.
- Classifying the different types of subtle variation helps the model.
- *Deep Learning*: can learn from a lot of data to build its own feature representation robust to noise and capable of learning cross domain patterns.
  - *Unsupervised* - Use autoencoder reconstruction error and use moving averages, use dropout with a set time window
  - *Supervised* - RNNs: learn from a set of yes/nos in a time series. RNNs can learn from series of time steps to predict when anomalies will occur.
  - Use streaming/minibatches (all neural nets can learn like this)
- *Anomalies in images:*
  - Encode, find outliers in latent space
  - Reconstruct and build difference to input (AnoGAN)
  - Interpolate sample patches into image and learn interpolation factor (Foreign patch interpolation) Done in medical image o-o-dist channelling

Misc NNs:
- *Mask R-CNN*: builds on faster R-CNN adding a branch to output seg masks for each found object. For detecting individual instances e.g: people or cars
- *You-only-look-once*: designed for real-time: processes images in 1 forward pass. Performs localization and classification in 1 go. Divides im into grid, predicts bounding boxes and class probabilities for all cells simultaneously
- *Attention U-Net*: Attention Gates help UNet focus on relevant regions; suppresses background noise. No longer need multi-stage cascaded CNNs
- *Trans U-Net*: Transformers help capture long range dependencies

Normative Representation Learning: variant of unsupervised single-class learning with a special focus on OOD detection where we need high specificity *Dense Representation Models* encode data into dense, ideally continuous low dim vector spaces. Few zeroed elems, allows us to interp complex samples, and detect areas with low support - unknowns which imply OOD data. We build them without labels with transformers, GANs, diffusion models, LDMs...

Tutorials/extra

CNNs

Calculate Image Size: O = ((Input - kernel size + (2xpadding))/stride) + 1
Calculate Parameters: conv = (Kh * Kw * ChannelsIn + (bias=1)) * ChansOut
Calc receptive field = (previous receptive field) + (product of strides before current layer) * (current layer kernal size)
Reduce overfit: Data aug, dropout, BatchNorm
- Primary role of stride is downsampling
- Compute deriv of (p_i) softmax ->

$$\text{If } i = j,\ \frac{\partial p_i}{\partial c_j} = \frac{e^{c_i}\sum_k e^{c_k} - e_{c_i}e^{c_j}}{(\sum_k e^{c_k})^2} = \frac{e^{c_i}}{\sum_k e^{c_k}}\frac{\sum_k e^{c_k} - e^{c_j}}{\sum_k e^{c_k}} = p_i \cdot (1 - p_j)$$
$$\text{If } i \neq j,\ \frac{\partial p_i}{\partial c_j} = \frac{-e^{c_i}e^{c_j}}{(sum_k e^{c_k})^2} = \frac{e^{c_i}}{\sum_k e^{c_k}}\frac{-e^{c_j}}{\sum_k e^{c_k}} = p_i \cdot -p_j$$

Invariance vs equivariance: invariance means func f is invar wrt T if f(T(x))=f(x), output dont change even when input has. Equivar f is when f(T(x))=T(f(x)). Convolution is equivar to translation (shift equivar). Pooling introd's approximate shift invariance.
- Skip conns help solve vanish grad problem

RNNs

Grad ∇_Wh L contains a (W_H)^T (dh1/dWh) term whose norm depends on the largest abs eigenvalue of W_H. Since this is > 1, this means the gradient contains a term that will explode when T → +∞, so gradient will explode. Which of the following sols can help address the gradient vanishing/explosion problem in practice? (Following are TRUE): Truncated back-propagation through time; Use GRU network instead of simple RNN; Use orthonormal matrix for Wh. (Normalise the ROW vectors in Wh to have `2 norm 1)
Question: Consider the following RNN to process the input sequence x_{1:T}: $h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h), \quad t = 1, \ldots, T.$
With the convention that h+0 = 0, and sigma(·) is the activation function. Assume the final output is y = h_T and there is a loss function L(y) applied to it. Find the fully expanded form of dL(y)/db_h.
$$\frac{dL}{db_h} = \frac{dL}{dy}\sum_{i=1}^{T}\left(\prod_{j=i}^{T-1}\sigma'(W_h h_j + W_x x_{j+1} + b_h) \cdot W_h\right) \cdot \sigma'(W_h h_j + W_x x_{j+1} + b_h)$$
LSTMs are sequential, so can process arb length sentences by continually updating their hidden state. Decoders similarly will continually produce output until hitting the eos token.

GAN

Question: Consider training text-to-image generation models using GANs. Here you are given a labelled image dataset D_L = {(t_n, x_n)}^N_{n=1} where x_n is an image and t_n is the image's text description (as s sequence of text tokens). We also assume the access of a pre-trained language model p(t), which allows us to sample text tokens and extract text features (via an intermediate layer F(t)).
(i) Provide your design for a text-to-image GAN, including the arch of the generator G & discriminator D, as well as the training adversarial loss.
$$\min_\theta \max_\phi \mathbb{E}_{x,t\sim p_{data}(x,t)}[\log D_\phi(x, F(t))] + \mathbb{E}_{x,t\sim p_\theta(x|t)p(t)}[\log 1 - D_\phi(x, F(t))]$$
Where x is dist'd by p_theta(x|t) => x=G_theta(t), which is our generator, i.e. a NN. (ii) In addition, assume you are given an extra dataset of unlabelled image dataset D_U={x_m}^M_{m=1}. How can you leverage this dataset to further train the text-to-image GAN? Provide your design for an extra adversarial loss. Answer: introduce an unconditional discriminator D_eta, where p_data now has imgs from both datasets for this loss. We can join this unconditional loss with the original loss function, and control the tradeoff with hyperparams like lambda_1 etc.
$$\mathbb{E}_{x\sim p_{data}(x)}[\log D_\eta(x)] + \mathbb{E}_{x\sim p_\theta(x|t)p(t)}[\log 1 - D_\eta(x)]$$

Transformers

- Time complexity for patches with stride 1, no padding, stride size w*h*c, and input of W*H*C: O((H-h+1)^2*(W-w+1)^2*whC)
- Dot product attention is permutation equivariant, sinusoid encoding is not 1-1 if input pos is in reals, so if we want 1-1 we need min/max

Question: Applying self-attention over a sequence is faster than using an RNN on modern hardware, why? Answer: RNNs have to iterate through every word in the sequence - this is not parallelizable, which makes it slower than huge matrix multiplications on GPU/TPU hardware.
Question: Capturing relationships between elements of a sequence which are far apart is easier with self-attention than RNNs, why? Answer: Firstly, RNNs have limited capacity in their "memory", and there is always some loss of information between subsequent time steps. Neither of these apply to the key-query operation which is applied across the entire sequence. Secondly, it is easier for self-attention mechanisms to learn long-range dependencies as the gradient path is much shorter (especially through stacked attention layers, as in transformers) . Even if your RNN doesn't have "vanishing gradients", long-range effects will still be washed out.

VAE:

Given a VAE with structure p(x, z1, z2, z3) and q(z1, z2, z3|x), and factorisation *p*(*x*,*z*1,*z*2,*z*3)=*p*(*z*3)**p*(*z*2|*z*3)**p*(*z*1|*z*2)**p*(*x*|*z*1) and *q*(*z*1,*z*2,*z*3|*x*) = *q*(*z*1|*z*2,*x*)**q*(*z*2|*z*3,*x*)**q*(*z*3 |*x*), the corresponding ELBO objective is:
$$\mathbb{E}_{x\sim p_{data}(x)}[\mathbb{E}_{z_1,z_2,z_3\sim q_\phi(z_1,z_2,z_3|x)}[\log p_\theta(x|z_1,z_2,z_3)] - \mathrm{KL}[q_\phi(z_1,z_2,z_3|x)||p(z_1,z_2,z_3)]]] = \mathbb{E}_{x\sim p_{data}(x)}[\mathbb{E}_{z_1\sim q_\phi(z_1|z_2,x)q_\phi(z_2|z_3,x)q_\phi(z_3|x)}[\log p_\theta(x|z_1)] - \mathrm{KL}[q_\phi(z_1|z_2,x)q_\phi(z_2|z_3,x)q_\phi(z_3|x)||p(z_1|z_2)p(z_2|z_3)p(z_3)]]]$$

Common formulae:

accuracy=(TP+TN)/(TP+TN+FP+FN), precision=TP/(TP+FP), recall=TP/(TP+FN), f1= (2*p*r)/p+r
Chain Rule: dy/dx = dy/dt * dt/dx
Product Rule: d(uv)dx = u dv/dx * v du/dx
Quotient Rule: d/dx (f(x) / g(x)) = (g(x) f′(x) - f(x) g′(x)) / ((g(x))^2
Convs: Assoc (w/each other & scalar mult), Commut, Distrib, derivative
Divergence: a function D[.][.]: PxP ∊ R s.t: D[P||Q] >= for all P,Q ∊ R and D[P||Q] = 0 iff. P=Q.
Bayes Theorem: P(A) * P(B|A) / P(B)
log(m^X) = X log m, log(m/n) = logm - logn, log(m*n) = logm + logn
dsinx = cosx, dcosx = -sinx, dtanx = sec^2x, d(a^x) = a^x lna, d(lnx) = 1/x, d/dx log_a x = 1/xlna

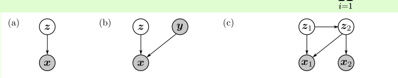
GPU for Poverty: Can ignore lots of the smaller layers if unimportant (e.g: for FLOPs, embedding, layer norm, RELU/GELU often not so important - not a hard rule though!). Keep track of input at each stage! Consider batch size (multiplies input) and other layer input sizes. Consider "num layers" as a multiplier too.

VAEs

Divergences: Compares two probability distribution. Given a set of probability distributions P on a random variable X, a divergence is defined as a function D[.|.] : P x P -> R such that D[P||Q] >= 0 and D[P||Q] = 0 iff P=Q.

- Kullback-Leibler (KL) Divergence: $\mathrm{KL}[p \parallel q] = \int p(x) \log \frac{p(x)}{q(x)}\, dx, \quad p, q \in \mathcal{P}$
- Jensen-Shannon (JS) Divergence: JS[p||q] = 1/2 KL[p||m] + 1/2 KL[q||m] where m(x) = 1/2 (p(x) + q(x))

Probabilistic Graphical Models (PGMs): Directed acyclic graphs that describe the factorisation structure of a joint probability distribution. $p(x_1, \ldots, x_D) = \prod_{i=1}^{D} p(x_i \mid pa(x_i))$
pa(x_i) = parents of x_i in DAG.



Jensen's Inequality: For a convex function: *f* then for any distribution p(x). $\mathbb{E}_{p(x)}[f(x)] \geq f(\mathbb{E}_{p(x)}[x])$: with equality iff *f* is linear or p(x) is a delta measure.
Law of the Unconscious Statistician (LOTUS): Given a distribution p_X(x) and a function y = g(x) such that E_pX(x) [g(x)] < +∞, the random variable Y = g(X) has its distribution p_Y(y) satisfying E_pY(y) [y] = E_pX(x) [g(x)].
Generalised Jensen's Inequality: E_pX(x) [f(g(x))] ≥ f(E_pX(x) [g(x)])
Maximum Likelihood Estimation (MLE): To fit a generative model p_θ(x) to a dataset {(x_n)} ~ p_data(x) we can minimise the KL divergence between p_data and p_θ. The optimal parameters: θ* = argmin KL[p_data(x)||p_θ(x)].

$$\mathrm{KL}[p_{\text{data}}(x) \| p_\theta(x)] = \mathbb{E}_{p_{\text{data}}(x)} \underbrace{[\log p_{\text{data}}(x)]}_{\text{constant w.r.t. } \theta} - \underbrace{\mathbb{E}_{p_{\text{data}}(x)}[\log p_\theta(x)]}_{\text{dependent on } \theta}.$$

- Ignore terms constant to θ, we get θ* = argmax E_pdata (x)[log p_θ(x)].
- Approx. using dataset: θ* = arg max (1/N) Σ (from n=1 to N) log p_θ(x_n).

Latent Variable Model (LVM): Statistical models in which observations are based on some unobserved "latent" variables. Generally of form: p(x|z) where z is the latent variable.
Variational Inference: Assuming a LVM p(x|z) we can marginalise to get p(x) as p(x) = ∫p(x|z) p(z) dz. We can construct a deep LVM as follows:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}), \quad p_\theta(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}(\mathbf{x}; G_\theta(\mathbf{z}), \sigma^2\mathbf{I})$$

where G_θ is a neural network. To optimise θ using the MLE requires computing the marginalisation integral which is intractable. We can however provide a lower bound to this using another distribution q(z) as:

$$\log p_\theta(\mathbf{x}) = \log \int p_\theta(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})\, d\mathbf{z} = \log \int q(\mathbf{z}) \frac{p_\theta(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q(\mathbf{z})}\, d\mathbf{z}$$
$$\geq \int q(\mathbf{z}) \log \frac{p_\theta(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q(\mathbf{z})}\, d\mathbf{z} \quad \text{(Jensen's inequality)}$$
$$= \mathbb{E}_{q(\mathbf{z})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] - \mathrm{KL}[q(\mathbf{z}) \| p(\mathbf{z})] := \mathcal{L}(\mathbf{x}, q, \theta).$$

We quantify the error between the lower bound and log_pθ(x) as follows

$$\log p_\theta(\mathbf{x}) - \mathrm{KL}[q(\mathbf{z}) \| p_\theta(\mathbf{z} \mid \mathbf{x})] = \log p_\theta(\mathbf{x}) - \mathbb{E}_{q(\mathbf{z})}\left[\log \frac{q(\mathbf{z})}{p_\theta(\mathbf{z} \mid \mathbf{x})}\right]$$
$$= \log p_\theta(\mathbf{x}) + \mathbb{E}_{q(\mathbf{z})}\left[\log \frac{p_\theta(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z})}{q(\mathbf{z}) p_\theta(\mathbf{x})}\right] \quad \text{(Bayes' rule)}$$
$$= \mathbb{E}_{q(\mathbf{z})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] - \mathrm{KL}[q(\mathbf{z}) \| p(\mathbf{z})] = \mathcal{L}(\mathbf{x}, q, \theta).$$

By improving lower bound q(z) goes to the exact posterior p_θ(z|x). q_ϕ(z|x) is a tractable posterior which we use the KL divergence to fit, since p_θ(z|x) is intractable. Optim var lower bound wrt model params doesn't ALWAYS optim log likelihood, as KL div'gence could grow, so we optim wrt ϕ too.
Variational Autoencoder: We define the q distribution as q(z) := q_ϕ(z|x) generally using a Neural Network:

$$q_\phi(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_\phi(\mathbf{x}), \mathrm{diag}(\boldsymbol{\sigma}^2_\phi(\mathbf{x}))), \quad \boldsymbol{\mu}_\phi(\mathbf{x}), \log \boldsymbol{\sigma}_\phi(\mathbf{x}) = \mathrm{NN}_\phi(\mathbf{x}).$$

The optimisation object is then given by: $\phi^*, \theta^* = \arg\max \mathcal{L}(\phi, \theta)$

$$\mathcal{L}(\phi, \theta) = \mathbb{E}_{p_{\text{data}}(\mathbf{x})}\Big[\underbrace{\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] - \mathrm{KL}[q_\phi(\mathbf{z} \mid \mathbf{x}) \parallel p(\mathbf{z})]}_{\mathcal{L}(\mathbf{x}, \phi, \theta)}\Big]$$

Analytical KL term: If q_ϕ(z|x) and p(z) are assumed to be factorized gaussians we can write the KL divergence term analytically (d = latent dimension) and <x, 1> is simply the sum of all values in x:

$$\mathrm{KL}[q_\phi(\mathbf{z} \mid \mathbf{x}) \parallel p(\mathbf{z})] = \frac{1}{2}\left(\|\boldsymbol{\mu}_\phi(\mathbf{x})\|_2^2 + \|\boldsymbol{\sigma}_\phi(\mathbf{x})\|_2^2 - 2\langle \log \boldsymbol{\sigma}_\phi(\mathbf{x}), \mathbf{1}\rangle - d\right)$$

Monte Carlo Estimation: the optim obj' above relies on evaluating NN transforms for all possible z to calc expectation - can be replaced by MC approximations. 1) Draw samples z(i) from q_ϕ(z|x) (factorised Gaussian) 2) feed these to the decoder p_θ(x|z) which gives the params of the likelihood dist of x (i.e. mean/var for Gaussian), then 3) plug those params and some x into pmf/pdf to calc prob of observing that x, do this for each z^(i). 4) take log of each prob, avg of those is the MC estimate. Reparametrisation trick:

$$z \sim q_\phi(z \mid x) \iff z = \mu_\phi + \sigma_\phi \odot \epsilon, \quad \epsilon \sim \mathcal{N}(\epsilon; 0, \mathbb{I})$$

This allows us to sample from Gaussian dist in a way that is differentiable - used for step 1 above. Conditional VAE: is to gen images conditioned on other other discrete/continuous info. Corresponds to learning gen model p_θ(x|y) to approx p_data(x|y), x is random var for i.e. img, y is random var for i.e. label/view angle. For gen model, us cond LVM: $p_\theta(x|y) = \int p_\theta(x|z, y)p(z)dz$
Often, p(z) is standard normal, which gives us:
$p_\theta(x|z, y) = \mathcal{N}(x; G_\theta(z, y), \sigma^2\mathrm{I})$ Learning maximises this var lower bound:

$$\mathcal{L}(\phi, \theta) = \mathbb{E}_{p_{\text{data}}(\mathbf{x}, \mathbf{y})}\left[\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{y})}[\log p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{y})] - \mathrm{KL}[q_\phi(\mathbf{z}|\mathbf{x}, \mathbf{y}) \| p(\mathbf{z})]\right]$$

ELBO equation: $E_{q(z|x)}[\log p(x|z)] - KL[q(z|x)||p(z)]$

Tutorials/extra

Misc

L1 regularisation: encourages feature sparsity -> wants fewer weights, and can push some to 0, enabling feature selection. Is more robust to outliers in training data. Adds penalty term prop' to sum of abs vals of weights.
L2 regularisation: penalises high value coefficients, but never to exactly 0, can help with making a smoother output, and deals with multicollinearity. Is more sensitive to outliers. Adds penalty term prop' to sum of sq'd weights.
Covariate shift: dist of input vars (covariates) changes between training data and test/real world data, but relation between inputs and outputs remains the same. This can make model perform badly on new data. Data augmentation could help if you know how input distribution might change (i.e. augment with flips/crops in dataset).
Skip Conns: help resolve vanishing gradients by allowing gradients to flow through to much earlier layers.
If GANs reject samples with high confidence: (D(G(z) → 0)) then we get vanishing gradients as D(x) => 0.
Finding Optimal Discriminator: If in integral form you can still take derivative of the inside and optimize w.r.t D(x).
VAE MLE obj is intractable because the data pθ is represented by NN
Minibatches more volatile (in gradient and thus loss update) than fullbatch.
Rank of a Seperable Convolution Matrix = 1
Aux losses in inception help predict the same output as the final layer by using model output. Helps mitigate vanishing grad to improve training.
Train on large corpus of unlabelled data: semi-sup GAN/VAE for dim reduction and then assign closest labels for unlabelled data then classify.
RNNs have hidden state, LSTMs also have cell state (which doesn't have a vanishing gradient). Reset gate also helps against vanishing grad. GRUs only pass 1 hidden state between cells.

GANs

Binary Classification: Given dist $p_{data}(x, y)$ with $y \in \{0, 1\}$, fit binary classifier $p_\phi(y|x)$ to cond dist $p_{data}(y|x)$. MLE of params ϕ obtained by solving optim task:

$$\phi^* = \arg\max_\phi \mathbb{E}_{p_{\text{data}}(x,y)}[\log p_\phi(y|x)] \quad (1)$$

Assume dataset balanced, ie $p_{data}(y)$ = Bern(0.5) then above objective same as

$$\phi^* = \arg\max_\phi \mathbb{E}_{p_{\text{data}}(x,y=1)}[\log p_\phi(y=1|x)] + \mathbb{E}_{p_{\text{data}}(x,y=0)}[\log(1 - p_\phi(y=1|x))] \quad (2)$$

Negation of above maximum likelihood objective is a.k.a cross-entropy loss.
OG GAN: Makes binary class'n task to help learning of gen've model dist $p_\theta(x)$ fit data dist $p_{data}(x)$. Done by labelling datapoints from data dist "real" and from model as "fake". So joint dist $\tilde{p}(x, y)$ as follows for binary class'n task:

$$\tilde{p}(\mathbf{x}, y) = \tilde{p}(\mathbf{x}|y)\tilde{p}(y), \quad \tilde{p}(y) = \mathrm{Bern}(0.5), \quad \tilde{p}(\mathbf{x}|y) = \begin{cases} p_{\text{data}}(\mathbf{x}), & y = 1 \\ p_\theta(\mathbf{x}), & y = 0 \end{cases} \quad (3)$$

Fitting this "*discriminator*" w $p_\phi(y = 1|x) = D_\phi(x)$ to $\tilde{p}(y|x)$ by maxing MLE obj

$$\phi^*(\theta) = \arg\max_\phi \mathcal{L}(\theta, \phi), \quad \mathcal{L}(\theta, \phi) := \mathbb{E}_{p_{\text{data}}(x)}[\log D_\phi(\mathbf{x})] + \mathbb{E}_{p_\theta(x)}[\log(1 - D_\phi(\mathbf{x}))] \quad (4)$$

There is dependence on generative model parameter θ, since "data dist" $\tilde{p}(x, y)$ of bin class'n task depends on $p_\theta(x)$. Training of this gen model tries to fool discriminator by *minimising* log probability of making the right decisions:

$$\theta^*(\phi) = \arg\min_\theta \mathbb{E}_{p_\theta(x)}[\log(1 - D_\phi(\mathbf{x}))] \quad (5)$$

2-player GAN gen&discr train obj: $\min_\theta \max_\phi \mathcal{L}(\theta, \phi)$ (6)

Term in obj for $p_\theta(x)$ is $\mathbb{E}_{p_\theta(x)}[\log(1 - D_\phi(\mathbf{x}))]$, in practice approx by Monte Carlo:

$$\mathbb{E}_{p_\theta(x)}[\log(1 - D_\phi(x))] \approx \log(1 - D_\phi(x)), \quad x \sim p_\theta(x) \quad (7)$$

Comp of dist $p_\theta(x)$ not required for eval of obj, instead can define directly the sampling process of $p_\theta(x)$ which also defines dist $p_\theta(x)$ in an *implicit* way:
$x \sim p_\theta(x) \iff z \sim p(z), \quad x = G_\theta(z)$ (8) , often we set $p(z) = \mathcal{N}(z; 0, I)$

Jenson-Shannon divergence minimisation: To justify 2-player obj (6) we show that with infinite capacity for both gen and discr, the global optimum of gen is $p_\theta(x) = p_{data}(x)$. For fixed gen $p_\theta(x)$, we compute gradient of GAN obj wrt ϕ:

$$\nabla_\phi \mathcal{L}(\theta, \phi) = \int \left(\frac{p_{\text{data}}(x)}{D_\phi(x)} - \frac{p_\theta(x)}{1 - D_\phi(x)}\right) \nabla_\phi D_\phi(x)\, dx \quad (9)$$

Given infinite capacity of the discr, setting $\nabla_\phi \mathcal{L}(\theta, \phi) = 0$ results in

$$\frac{p_{\text{data}}(x)}{D_\phi(x)} = \frac{p_\theta(x)}{1 - D_\phi(x)} \implies D_{\phi^*(\theta)}(x) = \frac{p_{\text{data}}(x)}{p_\theta(x) + p_{\text{data}}(x)} \quad (10)$$

Plugging in the optimal discriminator to the GAN objective:

$$\mathcal{L}(\theta, \phi^*(\theta)) = \mathbb{E}_{p_{\text{data}}(x)}\left[\log \frac{p_{\text{data}}(x)}{p_\theta(x) + p_{\text{data}}(x)}\right] + \mathbb{E}_{p_\theta(x)}\left[\log \frac{p_\theta(x)}{p_\theta(x) + p_{\text{data}}(x)}\right]$$
$$= \mathbb{E}_{p_{\text{data}}(x)}\left[\log \frac{p_{\text{data}}(x)}{\frac{1}{2}(p_\theta(x) + p_{\text{data}}(x))}\right] + \mathbb{E}_{p_\theta(x)}\left[\log \frac{p_\theta(x)}{\frac{1}{2}(p_\theta(x) + p_{\text{data}}(x))}\right] - 2\log 2$$
$$= 2\underbrace{\left(\frac{1}{2}\mathrm{KL}\left[p_{\text{data}}(x) \middle\| \frac{1}{2}(p_{\text{data}}(x) + p_\theta(x))\right] + \frac{1}{2}\mathrm{KL}\left[p_\theta(x) \middle\| \frac{1}{2}(p_{\text{data}}(x) + p_\theta(x))\right]\right)}_{:= \mathrm{JS}[p_{\text{data}}(x) || p_\theta(x)]} - 2\log 2 \quad (11)$$

Where $\mathrm{JS}[p_{\text{data}}(x)||p_\theta(x)]$ is the Jenson-Shannon divergence between $p_{data}(x)$ and $p_\theta(x)$. Since JS divergence is a valid divergence measure, this means with infinite capacity for the generator, $\mathcal{L}(\theta, \phi^*(\theta))$ is minimised iff $p_\theta(x) = p_{data}(x)$
Alternative Loss for Generator: Proposed optimised alt "*non-saturated*" obj for generator, given a fixed discriminator: $\max \mathbb{E}_{p_\theta(x)}[\log D_\phi(x)]$ (12)
Compared w (5) which mins log prob of true$^\theta$predictions, the alt obj maxes log prob of making *false* predictions. Notice discr has near perfect pred perf at start of GAN training ("fake" data quality bad atp) so $D_\phi(x) \approx 0$ for $x \sim p_\theta(x)$. Also assume gen model implicitly defined by $z \sim p(z), x = G_\theta(z)$. $D_\phi(x)$ often defined using sigmoid activation sigmoid(t) = (1 + exp[-t])^-1 at the last layer, ie $D_\phi(x)$ = sigmoid($d_\phi(x)$) with $d_\phi(x)$ param'd by a neural net. So $D_\phi(x) \approx 0$ when $d_\phi(x) \to -\infty$ (so at the start of GAN training $d_\phi(x) \to -\infty$ for $x \sim p_\theta(x)$). So gradients of the 2 objectives wrt Theta are

$$\nabla_\theta \mathbb{E}_{p_\theta(x)}[\log(1 - D_\phi(x))] = -\nabla_\theta \mathbb{E}_{p(z)}[\log(1 + \exp[d_\phi(G_\theta(z))])] = -\mathbb{E}_{p(z)}[\underbrace{D_\phi(G_\theta(z))}_{\approx 0} \nabla_\theta d_\phi(G_\theta(z))] \quad (13)$$

$$\nabla_\theta \mathbb{E}_{p_\theta(x)}[\log D_\phi(x)] = -\nabla_\theta \mathbb{E}_{p(z)}[\log(1 + \exp[-d_\phi(G_\theta(z))])] = \mathbb{E}_{p(z)}[\underbrace{(1 - D_\phi(G_\theta(z)))}_{\approx 1} \nabla_\theta d_\phi(G_\theta(z))] \quad (14)$$

The alt obj addresses the *vanishing gradient problem* of the OG one (5) at the start of training, hence the name "*non-saturated objective*".

Another justification of the alt obj is given by deriving *optimal solution* of the generator, given *optimal discriminator*. Define $f(t) = \log(1 + t^{-1}) - \log 2$, in which $f(t)$ is convex and $f(1) = 0$. Then can define an f-divergence as

$$D_f[p_\theta(x)||p_{\text{data}}(x)] := \int p_\theta(x) f\left(\frac{p_{\text{data}}(x)}{p_\theta(x)}\right) dx$$
$$= \int p_\theta(x) \log\left(1 + \frac{p_\theta(x)}{p_{\text{data}}(x)}\right) dx - \log 2 \quad (15)$$
$$= -\mathbb{E}_{p_\theta(x)}\left[\log D_{\phi^*(\theta)}(x)\right] - \log 2.$$

Shows that maxing alt "non-saturated obj" is equiv to min'ing an f-divergence b/w model and data dist. Therefore again w infinite capacity of gen, optimal sol of gen model is $p_\theta(x) = p_{data}(x)$.

Conditional GAN: Similar to cond VAEs, cond GAN is particular param'n of $p_\theta(x|y)$ which approx's the data dist $p_{\text{data}}(x|y)$. Again x is the random var for data and y is the random var corresponding to the additional info. For design of the gen model $p_\theta(x|y)$, we use cond LVM (latent var model) as follows:

$$p_\theta(x|y) = \int p_\theta(x|z, y)p(z)dz \quad (16)$$

and often we set $p(z) = \mathcal{N}(z; 0, \mathrm{I})$. But diff from cond VAE which explicitly specifies the dist form of $p_\theta(x|z, y)$, for cond GAN its defined *implicitly* by the following sampling process:

$$x \sim p_\theta(x|z, y) \iff z \sim p(z), x = G_\theta(z, y) \quad (17)$$

with $G_\theta(z, y)$ defined by a neural net that takes both z and y as inputs.
Similar to GANs, learning is done by optimising an adversarial objective:

$$\min_\theta \max_\phi \mathbb{E}_{p_{\text{data}}(x,y)}[\log D_\phi(x, y)] + \mathbb{E}_{p_\theta(x|y)p_{\text{data}}(y)}[\log(1 - D_\phi(x, y))] \quad (18)$$

In practice the component related to the gen params Theta is computed by

$$\mathbb{E}_{p_\theta(x|y)p_{\text{data}}(y)}[\log(1 - D_\phi(x, y))] \approx \log(1 - D_\phi(G_\theta(z, y), y)), \ z \sim p(z), y \sim p_{\text{data}}(y) \quad (19)$$

Using similar techniques, can derive optimal discr w fixed gen w param Theta:

$$D_{\phi^*(\theta)}(x, y) = \frac{p_{\text{data}}(x, y)}{p_\theta(x, y)p_{\text{data}}(y) + p_{\text{data}}(x, y)} \quad (20)$$

and w the optimal discr, minimising $\mathcal{L}(\theta, \phi)$ wrt Theta is equivalent to minimising the Jensen-Shannon divergence $\mathrm{JS}[p_{\text{data}}(x, y)||p_\theta(x|y)p_{\text{data}}(y)]$

Recurrent Neural Networks

RNNs excel at handling sequential data.
Given a data sequence (x_1, ..., x_T) we want to predict future values: (x_1, ..., x_T) -> x_{T+1}, x_{T+2}, ... useful for time series pred, object tracking, machine translation

We have a supervised learning task aiming to learn the mapping from inputs $x_{1:T}$ to outputs $y_{1:T}$ = (y_1, ..., y_T). A simple RNN computes the following mapping for t = 1, ..., T: $h_t = \phi_h(W_h h_{t-1} + W_x x_t + b_h)$, $y_t = \phi_y(W_y h_t + b_y)$
The params are W_h, W_x, b_h, W_y, b_y and ϕs are the activations (i.e: they take in as input the input x, and we maintain a recurrent state h which we also input)
Back-propagation through time (needs to be done at each time step):

- Loss: L(θ) = $\Sigma_{t=1 \text{ to } T}$ L(y_t), and dLoss/dθ = $\Sigma_{t=1 \text{ to } T}$ d/dθ L(y_t) obviously.
  - dL(y_t)/dW_y=(dL(y_t)/dy_t)(dy_t/dW_y); dL(y_t)/db_y = (dL(y_t)/dy_t)(dy_t/db_y)
  - dL(y_t)/dW_x=(dL(y_t)/dy_t)(dy_t/dh_t)(dh_t/dW_x)
  - dL(y_t)/db_h=(dL(y_t)/dy_t)(dy_t/dh_t)(dh_t/db_h)
- dh_t/dW_x = ∂h_t / ∂W_x + dh_t/dh_{t-1} * dh_{t-1} / dW_x
- dh_t/db_h = ∂h_t / ∂b_h + dh_t/dh_{t-1} * dh_{t-1} / db_h
  - By the chain rule, obviously: dL(y_t)/ dW_h = (dL(y_t) / dh_t) * (dh_t/dW_h)
    - and dL(y_t)/dh_t = dL(y_t)/dy_t * dy_t/dh_t
    - The entries in the Jacobian dh_t / dW_h contains the total gradient h_t[i] w.r.t W_h[m, n]. dh_t / dW_h = (∂h_t / ∂W_h) + dh_t/dh_{t-1} * dh_{t-1} / dW_h
      - note the partial deriv - we treat h_{t-1} as a const w.r.t W_h. By expanding dh_{t-1} / dW_h & using the chain rule we obtain:
      $$\frac{d\boldsymbol{h}_t}{dW_h} = \frac{\partial \boldsymbol{h}_t}{\partial W_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{\partial \boldsymbol{h}_{t-1}}{\partial W_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{d\boldsymbol{h}_{t-2}}\frac{d\boldsymbol{h}_{t-2}}{dW_h} = \ldots = \sum_{\tau=1}^{t}\left(\prod_{l=\tau}^{t-1}\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}\right)\frac{\partial \boldsymbol{h}_\tau}{\partial W_h},$$
      - Convention: when τ is t, above product ↑ = 1;
      - i.e: the chain rule of the gradients needs to be computed in an reversed order from time t = T to time t = 1, hence the name Back-propagation through time (BPTT)
      - We can truncate; in the above formula;

Gradient Vanishing & Explosion:

- Simple RNNs suffer from gradient vanishing and gradient explosion issues
- Notice: dh_{l+1}^T / dh_l = ϕ_h'($W_h h_l$ + $W_x x_{l+1}$ + b_h) ⊙ W_h - a vector containing element-wise derivatives. A⊙B is "broadcasting elem wise product" - repeat A enough times to equal B in dim; and then Hadamard.
  - So the Π term contains products of t-τ copies of W_h and the derivative ϕ_h'(.) at time steps l = τ ... t-1.
    - Then assume ϕ_h'(.) is an identity mapping with derivative 1
    - And assume that the hidden states are scalar; e.g: dim(h_t) = 1
      - Then Π = (W_h^t-τ)^T - this explodes or vanishes when t-τ is large, depending on W_h < 1 or not
        - Generally, this depends on if the largest eigenV < 1 or not.
  - Also happens for sigmoid or tanh - if h_t is close to +-1 then ϕ_h'(.) gets close to 0; the derivative is saturated. Multiplying multiple saturated derivatives leads to a vanishing gradient.
- Solutions: Gradient Clipping: with a fixed hyper-parameter γ, a gradient g is clipped when ||g|| > η: g <- (γ/||g||) g - i.e: we have a max norm
  - Good weight initialisation: initialise W_h = I, b_h = 0. This makes ϕ_h'(t) = δ(t > 0) and dh_{l+1} / dh_l = δ($W_x x_{l+1}$ + h_l > 0) at initialisation. Doesn't guarantee problem removal but usually achieves LSTM level results.
  - Else make W_h orthogonal or unitrary.

Long Short Term Memory (LSTM): Solves vanishing/exploding gradients.

- Doesn't remove it - but minimizes the impact. Outperform normal RNNs
- Adds memory cell states and gates to control error flow
- The parameters of LSTM are therefore θ = {W_f, W_i, W_o, W_c, b_f, b_i , b_o, b_c}. h_0 & c_0 are either set as 0 vectors or added to the learnable parms. If the initial cell state c_0 is set to 0, then the elements in c_t & h_t bounded within (-1, 1). The output y_t can be produced by a 1-layer neural network similar to the simple RNN case: y_t = ϕy($W_y h_t$ + b_y)

f_t :forget gate	$\boldsymbol{f}_t = \sigma(W_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f)$
i_t :input gate	$\boldsymbol{i}_t = \sigma(W_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i)$
o_t :output gate	$\boldsymbol{o}_t = \sigma(W_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o)$
x_t :input	$\tilde{\boldsymbol{c}}_t = \tanh(W_c \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c)$
c_t :memory cell state	$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$
h_t :hidden state	$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$

Gated Recurrent Unit (GRU): a simplified LSTM outperforming basic RNNs

- Network params are obvious like for LSTMs.
- No input/output gate or cell state necessary, but still has the ability to forget.
- z_t acts as the update gate, determining the incorporation of the current info to the hidden states; r_t is the reset gate which also impacts the maintenance of the historical information.

$$\boldsymbol{z}_t = \sigma(W_z \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_z)$$
$$\boldsymbol{r}_t = \sigma(W_r \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_r)$$
$$\tilde{\boldsymbol{h}}_t = \tanh(W_h \cdot [\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_h)$$
$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t$$

LSTMs are a good default choice to see how the model fits to data. To reduce overfitting or optimize compute, can switch to GRUs. No objective superior.
We can stack LSTMs to deepen net - output at time t = input of next LSTM at that same timestep t. So computation can be flexible - compute all ts for one LSTM and move to next, or do first on all and so on.
Bidirectional LSTMs: we stack LSTMs, but reverse the fwd pass for some ts (go from end to start of net); improves perf.
Sequence to Sequence Models: to do something like machine translation we need to learn the *conditional distribution* $p_\theta(y_{1:L}|x_{1:T})$ for x, y seq of any length.
Auto-Regressive Model: $p_\theta(y_{1:L}|x_{1:T}) = \Pi_{l=1 \text{ to } L}\ p_\theta(y_l|y_{<l}, v)$, v = enc($x_{1:T}$) dec
Encoder: 1) We first map words to word emb vectors (Word2Vec)
b. We then use a stacked LSTM to process each of the word emb vecs
c. We feed it LSTM word by word and end off our input with <eos> token
d. Instead of outputting at each time step, we wait for all the processing to be done. The final hidden state h_t and the output v is useful to us.
Decoder: 1) Taking the output we can predict the first translated word by sampling or beam search. We then feed into our decoder that word as well as h_t continuing our stacked LSTM, getting a translated word out
2) We continually feed in our next translated word and LSTM out state.
3) Eventually this constructs the entire sentence.

- WE do things differently during train and test!! In *training*: when we get our translated word out, we immediately compute loss and backprop based on the ground truth. Then pass the ground truth to the next step
- In testing, we purely run on the translated words we use

The encoder-decoder framework is very generalisable. We can make a model that can describe an image using a CNN enc (producing v) + LSTM decoder.
Word2Vec has small dims, and semantics are somewhat preserved (adding the embedding of king, female, -male roughly equals queen).

Attention and Transformers

Sentences fed into our Seq2Seq LSTMs can be very long. Our model performs poorly as they struggle to learn long-term dependencies. Needs to *focus*!
Attention in Transformers: *Single Head Attention*

- *Queries* are submitted to the system $q_i \in R^{(d_q)}$. The system checks matching similarity between queries and *keys* $k_j \in R^{(d_q)}$, and returns the retrieved *values*.
- Each key vector k_j is associated with a value vector $v_j \in R^{(d_v)}$ - i.e: such that when q_i and k_j are aligned, v_j is returned. Mathematically:
  - *Attention*(Q, K, V; a) = a (QK^T / √d_q)V (Attention Weights)
  - Q=(q_1,..., q_N)^T∈R^(Nd_q), K=(k_1,..., k_M)^T∈R^(Md_q), V=(v_1,..., v_M)^T∈R^(Nd_q)
  - a(.) is an activation function applied row wise. Hard atten is when *a* takes the onehot encoding of the argmax of input. Soft att uses soft-max as *a* instead, giving weighted sum of value vecs, instead of one vec
  - The dot product QK^T computes similarity between query q_i and key k_j
  - We divide by √d_q to retrieve values according to their weighting.
  - *Masking*: sometimes we mask some of the QK^T matrix so that the output value to this query is unaffected, useful for preventing future attention info from affecting current output (e.g: sequences)
- *Self-attention* is also in wide usage and sets K = Q.

Computational Complexity:

- Time: O(MNd_q + MNd_v): 1) Dot Product QK⊤ has O(MN dq) run-time cost, computation of A matrix given the dot product has O(MN) cost, and 2) dot product AV has O(MNd_v) cost.
- Space: O(MN+Nd_v) since back-prop requires storing intermediate results. Dot product QK⊤: O(MN), and the final output AV has O(N dv) cost.
- For self-attention, time: O(N^2d_q + N^2d_v), space: O(N^2 + Nd_v),

Multi-Head Attention: repeats the atten process multiple diff "views". i.e: input query "cat". Then depending on the definition of similarity (semantic, subject-verb, grammar), it can match to key "tiger" or key "pet" and so on.

- *Multihead*(Q, K, V, a) = *concat*($head_1$, ..., $head_h$)W^o
- $head_i$ = Attention(QW_i^Q, KW_i^K, VW_i^V, a)
- i.e: eadh head tells us a different aspect about the query, and so we feed each head a different portion of the <Q,K,V> triple. Then, concat heads
- Time: $\mathcal{O}(\underbrace{h(MN\tilde{d}_q + MN\tilde{d}_v)}_{\text{attention heads}} + \underbrace{h(\tilde{d}_q d_q(M + N) + \tilde{d}_v d_v M)}_{\text{input projections}} + \underbrace{Nh\tilde{d}_v d_{out}}_{\text{combined outputs}})$ Params we learn (apart from K, V)
- Space $\mathcal{O}(\underbrace{h(MN + N\tilde{d}_v)}_{\text{attention heads}} + \underbrace{h(\tilde{d}_q(M + N) + \tilde{d}_v M)}_{\text{input projections}} + \underbrace{Nd_{out}}_{\text{combined outputs}})$ W_i^Q, W_i^K, W_i^V, W^O

Attention Based Encoder+Decoder:

- $p_\theta(y_{1:L}|x_{1:T})$ = Πl=1 to L $p_\theta(y_l|y_{<l}, z_{1:T})$ (masked out in 1st dec layer) (encoder attention output is used in the decoder)
- *Position Encoding*: For an input seq of queries q_1, ..., q_N where the ordering info (subscript n) is useful for the task (time series regression), we need to add the ordering info to attention inputs. *Position Encoding* constructs the query input Q~ to the attention module as: $\tilde{Q} = (\tilde{q}_1, ..., \tilde{q}_N)^\top, \quad \tilde{q}_n = f(q_n, PE(n)).$
- PEs can be an NN & set of learnable params, or a function (sinusoidal PEs)
  - Predefined function is test time queries is unknown / longer than traintime. Else we might preferable NN learnable PEs.

Layer-Norm: helps stablise NNs; we compute the mean of all pre activations and stdev, and then do a ← (a - μ)/σ for each. Add&Norm(x) = LayerNorm(x + Sublayer(x)) - sublayer is MHead Att/FFN. We do the add with a resid conn.
Point Wise FFN: used as the "feed forward" layer in Transformers. The MHAtt (after "Add&Norm") returns a matrix of size N × d_{out}, of atten results for N queries. Can be processed "pointwise" (i.e. row-wise) with a FF NN, treating the rows in the atten outputs as "datapoint" inputs for next layer.
Attention Applications:

- Pre 2017: DRAW: VAE model combining RNNs and Gaussian MM Att. We treat the intermediate "drawing state" as latent variable: p(x|z) = $p_\theta(x|z_{1:T})$. Gaussian attention to select focused patch at time t
- *Image Captioning*: CNN low-level features are indexed by spatial location. Sort CNN features and then process with RNNs
- BERT: Pretrained on masses of data, w/ random masking & predicting.
- MHA can be applied to any Set of Input Data: graphs, point clouds even.
  - Cross-modality usages are then easy - just need to emb. diff. modalities into same space and then apply attention
- Image complete, super res, regression & supervis learning...

Attention is Expensive: We can improve their efficiency with techniques:

- Local attention by defining "neighbourhood"
- Sparse attention by defining attention weight matrix patterns:
  - We have rules such that only for certain pairs of (i, j) the attention entry is computed. Different att heads have diff sparsity patterns
- Low Rank Approximations:
  - if the activation a(.) is linear, (QK^T V) == Q(K^TV) - faster to compute
- Efficient Transformers: Learnable inducing points for set summary:
  - Each data point attends to these summary points, which attend back to the data which refines the representation & lowers computation

Privacy Issues: Transformer-based LLMs have billions of parameters. Can easily memorise input data. Currently they're pre-trained on open-web text without any privacy protection; attacks can steal sensitive input data

Diffusion Models

Gaussian joint distribution p(x, y): p(x) = N (x; μ_x, (σ_x)^2), p(y|x) = N (y; ax, (σ_y)^2)
Marginal distribution: p(y) = N (y; aμ_x, a^2(σ_x)^2 + (σ_y)^2)
Posterior distribution: p(x|y) = N (x; ~μ, σ~^2), ~σ^-2 = (σ_x)^-2 + a^2(σ_y)^-2, ~μ = ~σ^2 (ay(σ_y)^-1 + μ_x(σ_x)^-1)

$$p(\boldsymbol{x}_{t-1}, \boldsymbol{x}_{t+1}|\boldsymbol{x}_t) = \frac{p(\boldsymbol{x}_{t-1}, \boldsymbol{x}_{t+1}, \boldsymbol{x}_t)}{p(\boldsymbol{x}_t)} = \frac{p(\boldsymbol{x}_{t-1}, \boldsymbol{x}_t)p(\boldsymbol{x}_{t+1}|\boldsymbol{x}_{t-1}, \boldsymbol{x}_t)}{p(\boldsymbol{x}_t)} = p(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)p(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)$$

where the last equality is due to the chain structure, i.e., $p(\boldsymbol{x}_{t+1}|\boldsymbol{x}_{t-1}, \boldsymbol{x}_t) = p(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)$.

Expensive to train and simulate for higher dim data because xθ(xt, t) defines an R^dim(x) → R^dim(x) mapping
Generating diffusion model requires multiple steps of denoising, so evaluating xθ(xt, t) for many steps is very expensive
Latent diffusion models address this issue, which first learns a (stochastic) auto-encoder to map the high-resolution image x to a lower-dimensional embedding z with dim(z) << dim(x), then trains a diffusion model in z space
First stage training of auto-encoder aims to minimise:

$$J(\theta, \phi) := \mathbb{E}_{p_{\text{data}}(x)}\mathbb{E}_{q_\phi(z|x)}\left[\|x - G_\theta(z)\|_2^2\right] + \beta\mathrm{KL}[q_\phi(z|x) \parallel \mathcal{N}(0, I)]$$

(similar to training beta-VAE)
After first stage training, training data x is mapped to latent space using qϕ(z|x) for diffusion model training. For latent diff model, data distribution is effectively the aggregated posterior

$$q_\phi(z) := \int q_\phi(z|x)p_{\text{data}}(x)\, dx$$

Latent diff model is trained in almost the same way as training a diff model in x space, except that all the xt variables are replaced by zt variables with z0 := z and q(x0) := qϕ(z).
After training, in gen time, model first approx samples z ~ qϕ(z) by simulating the backward process using latent diff model, starting from zT ~ N (0, I). This z sample is then decoded to a generated image as x = Gθ(z).

KV Caching: if we want to see e.g: 6 tokens ahead we need to do 6 fwd passes This is insanely expensive. When we generate a new word we have to reprocess the entire input seq. KV caching stores K, V from prev steps and reuses! (as past results do not influence future ones - we masked them out!) However KV cache is *huge*: |kv cache| = 2*batch_size*num*seq_length* hidden_dim*BF_16. To save on this rather than having KV cache per head, share it across the entire thing. [DeepSeek] instead projects down to low dim caches that and when retrieving projects back up! (multiheadlatent attention

Deep Learning at Scale for the GPU Poor

Floating Points are used in ML models: sign, mantissa, exponent. Float32 gives us a great value range for deep learning.
Llama 65B: Model Dim D: 8192, Sequence Length N: 2048, Batch Size B: 1, Number of Layers (L): 80, Number of Attention Heads (H): 64.
Example: GPU Memory Footprint for Llama 65B (Float 32)
inference time: 32 (bits) * 65x10^9 (params) = (2.08 x 10^12 bits) = 260GB

- big batch sizes and sequence lengths will complicate things

training time: 260GB (model params) + 260GB (gradients) + 520GB (Adam optimiser, which has two coefficients per param: momentum & variance) 250GB + (activations for backward; ignoring bias) = 1.2TB even with B=1...
Calculating the backward activations for a transformer model:

1. Input → Norm: Stores activations of shape BND.
2. QKV Projection: Single input (BND) is used to compute Q, K, V (but only need to store the shared input).
3. Attention (Single Head): QK^T requires storing both Q and K → 2BND.
   - Attention matrix (before softmax) → BHN².
4. Softmax Output: Stored as BHN².
5. Skip Connection (Addition): No storage needed (gradient = 1).
6. Norm: Stores activations of shape BND.
7. MLP Block: Input stored as BND.
   - Hidden layer (4x larger) → 4BND Non-lin → 4BND. out → 4BND.

B=1, N=2048, D=8192; 4BND = 250GB.
Floating Point Operations (FLOPs): Basic unit of computation
Given w, x ∈ R^n, w+x takes n FLOPs, w*x takes 2n FLOPs (n mul, n-1 adds)
For A ∈ Rmp, B ∈ Rpn, AB costs 2nmp FLOPs.
Calculating the FLOPs to pass through a transformer:

1. Input -> Norm: ~20ND (we drop this as it's small compared to rest)
2. QKV Projection: 2*3ND² (NxD matrix * DxD; we do it 3 times for Q,K,V)
3. Attention: QK^T: 2*N²D; Attention Matrix for Softmax: 3*H*N²
4. Softmax for Value Network: 2N²D 5. Projection: 2ND²
6. Skip Connection: No FLOPs of note 7. MLP Block: 2ND*4D; output = 8ND²

Total FLOPs for fwd pass: BL * all FLOPs=BL(24ND²+4N²D+3HN²)= 2.7*10^14
Backwards Pass: 5.4*10^14 (twice as much as for fwd we compute Z, in backwards we compute dL/dZ and dZ/dW)
Total Training FLOPs (Approx): 6*dataset size in tokens*num params
For Deep Learning we need a Model, Data and Compute. One of the three will bottleneck our performance. e.g: Grok-3 has 5e26 FLOPs!! Required 100,000 H100s to run. On log scales, we see increasing either dataset size (tokens), params, or compute (PF days) improves our model loss linearly. One PF-day = Peta FLOP day = 10^15*24*3,600 A100 @ FP32 = 19*10^12 FLOPs (/s)
It was found in 2022 that we should be training for longer and don't need as large models - the optimization line was retuned thereafter. Optimal param-compute for any model - language or not - follows the same trend. Training smaller models for longer to have the same performance is ideal - as we can deploy those models more easily!

- TLDR; Extra time spent at train time linearly increases performance
- But extra time spent at test time *also* linearly increases performance (ensemble methods, reinforcement learning, etc...)
- We should scan over the dataset and model size and compute. It's important to scan over the diff compute values - half each time and see how the model performs to see our trend performance line (only takes 2N compute in total...)
  - Also experiment with bottlenecks in model and data size
  - Do scaling analysis to see the max compute you could have to improve meaningfully with
- We're running out of internet data, but synthetic data / RL is v good

How to maximize resources:

- *Gradient Accumulation:* if we only have batch size of one our grad gets perturbed a lot as it updates every single time we process 1 item. This causes terrible results. We can instead iteratively compute forward and backward passes and sum up gradients. Once we've got enough to make a good batch size, we take the mean and then update params.
- *Gradient Checkpointing*: instead of storing the activations, we can redo the forward pass for each step of the backprop. It is very expensive tho. *Gradient Checkpointing* improves this - we store the activations for *checkpoint* nodes - this can massively decrease the sizes necessary for our forward pass, as we work forward from checkpoints, not start.
- *Deepseek*: does a form of PCA on Queries (K&V), it t-forms to a compressed latent vector that's much smaller. Saves 1000xs of mem.

Mixture of Experts:

- 4 learnable proj mats - query, key, value, output. Each R^D²->we've 4D² params in the atten layer. But 8D² in MLP (Up proj mat goes from D to 4D - D4D; and down proj - 4DD; sums to 8D²). i.e: focus optim on MLP!
- In an MLP, we process each token n the same context. With a mixture of experts, instead have many MLP masks specialised for a different domain.
  - 100 experts = 100x more computation... but each token only sees a few experts - hence, fwd/back pass complexity only linearly increases
    - Routing networks decide which tokens go to which experts.
      - Similarity score computed by normalized dot product b/w token and learnable centroids then we sigmoid for the top k experts.
    - Each expert can be on a different node / gpu and this is efficient
  - DeepSeek has one "shared expert", 256 experts and k=8
  - *Routing Collapse*: uneven allocation of tokens to experts. Solutions:
    - Token Dropping: allow token to pass w/o expert if node overloaded
    - Auxiliary Loss: Penalise uneven expert utilisation in loss function
      - Not great: relies on hyperparam, high: ruins training. low: collapse
    - Bias Term: If there's an overloaded expert at the end of a forward/ back pass then add gamma to underused, and subtract from OU.
    - *DeepSeek* uses bias term and also sends token to max 4 nodes.
- i.e: we end up having much higher param count for much less calculation1

Finetuning: allows adopting model to specific downstream task

- Pretrained weights Φ, are updated Φ + ΔΦ: $\max_{\Phi} \sum_{(x,y)\sim Z} \sum_{t=1}^{|y|} \log(P_\Phi(y_t|x, y_{<t}))$ (Context-target tokens; Autoregressive foundation mod)
- Have to store gradient optims of each weight as all param in Φ is updated

LoRA: the model-less overparameterized space has a low intrinsic dim

- if ΔΦ has a low intrinsic rank, we can approximate ΔΦ with Θ where |Θ|<<|Φ|
- For a weight matrix in pretrained model, W_0∈R^(D×D); we're learning ΔW
- y = XW_0 + XΔW ≈ XW_0 + XL_1L_2; L_1 ∈ R^(D×r), L_2 ∈ R^(r×D), r≪D!!
  - We also freeze updates to the DxD matrix - only learn L_1 and L_2
  - Done across all layers, L_2 is initialised with 0s, L_1 with standard normal
  - r=1 and r=2 does well for GPT3 which has a full rank of 12,288!
  - No extra flops at inference, we just combine XW_0 +L_1L_2 and subt at end.
  - No longer need to store 99.99% of params if r=1; so no need to store most gradients/optimizer params. Still store model & activations.
  - Assuming batch size of 1, we divided our GPU requirements by 3!

QLoRA: k-bit quantisation: allows converting float32 → int8 for example:
x_{int8} = round(127 / absmax(x^FP32) X^FP32) = round(c^FP32 . X^FP32)
dequant(C^FP32, X^Int8) = X^Int8 / C^FP32 = X^FP32. C^x is quantisa const.

- If we have an outlier, most nums will get compressed into tiny range. Fixed by having quantized bins which have their own quant conts.
- Essentially QLora quantizes W_0 from bfloat16 to 4 bit! 4 bit repres:
  a. 4-bit NormalFloat: you give values near normal peak most accuracy
  b. Double Quantisation: For each block, store quant const (32 bits of mem). For a chunk size of 64 - we store an extra 32 bits = 0.5 bit/param. This is bad! We quantize the quantized constant to fix this: and have a chunk size 256 = 8/64 + 32/(64*256) = 0.127 bits per bit!
  c. Dequantise X at computation step.