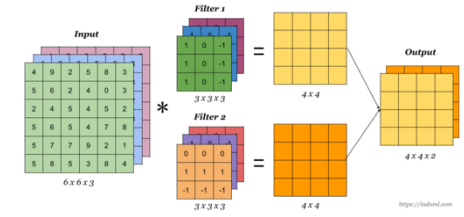
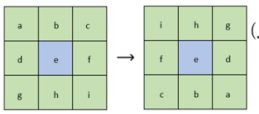
Convolutions & CNNs

Output shape (width or height): **floor((W – K + 2P) / S) + 1**

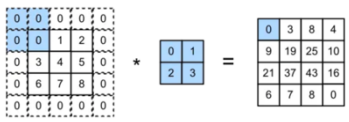
Output shape (depth/num of channels): num of filters applied





$$(f*g)(x) := \int_{\mathbb{R}^n} f(\tau)g(x-\tau)\mathrm{d}\tau$$

For correlation: it's g(x + tau)



#params of 1 filter: filter H x filter W x filter C + 1 (bias)

Fix mean and variance

$$\mu_B = \frac{1}{|B|}\sum_{i\in B} x_i \text{ and } \sigma_B^2 = \frac{1}{|B|}\sum_{i\in B}(x_i-\mu_B)^2 + \varepsilon$$

and adjust it separately

$$x_{i+1} = \gamma\frac{x_i-\mu_B}{\sigma_B} + \beta$$

(mean, variance)

Conv = only equivariant to translation; not equivariant to warp, flipping… hence data aug for better model general. too CNN approx. invariant if adding pooling layers.

Shift invariance: property that describes a system's unchanging response when the input is shifted; useful in CNNs (e.g. obj recog.) vs **shift equivariance** (e.g. obj detection, segmentation)

Assumptions: translation invariance (ashift in input should simply lead to shift in the hidden representation) + locality

Pooling layer ensures ***approximate* translation invariance**

Receptive field of a layer k

$$l_k = l_{k-1} + ((f_k-1)*\prod_{i=1}^{k-1} s_i)$$

where l_{k-1} is the receptive field of layer $k-1$, f_k is the filter size (height or width, but assuming they are the same here), and s_i is the stride of layer i.

Commutativity: $f*g = g*f$

Associativity: $f*(g*h) = (f*g)*h$

Distributivity: $f*(g+h) = (f*g)+(f*h)$

Associativity with scalar multiplication: $a(f*g) = (af)*g$

Derivative: $\mathrm{D}(f*g) = (\mathrm{D}f)*g = f*\mathrm{D}g$

For real-valued functions, of a continuous or discrete variable, conv = cross-correlation of f(x) & g(−x), or f(−x) & g(x)

CNN architecture

- convolutions implemented through weight sharing, interpreted as weights of a filter function
- more output channels => NN can learn more complex + high-level features. Padding to preserve input shape.
- final FC layers: can be comp. + memory expensive (bottleneck w/ increasing num of classes, e.g. 1000 for ImageNet)
- 1x1 conv acts like an MLP per pixel, which aggregates across channels of input; introduces complexity + nonlinearity
- **AlexNet** for ImageNet: deeper + bigger/wider LeNet. Add dropout, sigmoid => ReLU, maxpool, heavy data aug, model ensembling (model averaging across multiple well-performing CNN models to achieve SOTA results) vs LeNet for MNIST
- **VGG**: group layers into blocks (num of blocks varies). Blocks can be easily parameterized, creating a more organized, modular arch. Simplifies design process + easier to fine-tune model for specific learning tasks.
- More layers of *narrow* convolutions outperforms using fewer *wide* ones: lots of simple fns > few complex fns
- **Inception**: deep + parallel paths with blocks to capture different types of features more effectively. Combines benefits of various convolutions & pooling operations while optimising computational cost
- **Batchnorm** normalises features within each minibatch, stabilising training process & speeding up convergence. Regulatisation by noise injection + no dropout needed (both control NN capacity) + ideal minibatch 64-256). B = batch. Test: fix the gamma & beta learned in training + instead batch statistics, use running average for mean and variance
- **ResNet**: get increasingly powerful AND nested functions (might not be convex) with more layers. "Taylor expansion" style parametrization. Input to act fn becomes f(x) ***+ x*** (x sometimes * with 1x1 conv to change dim). Allows for deeper NNs w/ fn classes more likely to be nested. Better grad flow (solve vanishing). ResNet module: multiple ResNet blocks.
- **DenseNet** uses higher-order Taylor series expansion; feature maps reuse; may need to reduce res (transition layer)
- **curse of dim**: As the number of features or dimensions grows, the amount of data we need to generalise accurately grows exponentially

To approximate a (Lipschitz) continuous function $f\colon \mathbb{R}^d \to \mathbb{R}$ with ϵ accuracy one needs $O(\epsilon^{-d})$ samples

- **pooling**: permutation-invariant aggregation + downsampling; reduces res; hierarchical features; max-pooling breaks shift equivariance. **Separable filter**: filter can be written as conv of 2/more simple filters: e.g. 2D filter from 1D filters; and det(filter)=0.

VAE

The definition of divergence is weaker than distance: does not need to satisfy symmetry or triangle inequality

We can work with PDFs: $p(\boldsymbol{x}) = \frac{dP}{d\boldsymbol{x}}, \forall P \in \mathcal{P}$.

Probabilistic graph models (joint distribution): used to describe dependency structure

$$p(\boldsymbol{x}_1,...,\boldsymbol{x}_D) = \prod_{i=1}^{D} p(\boldsymbol{x}_i|pa(\boldsymbol{x}_i)),$$

$$p(x_1,x_2,z_1,z_2) = p(z_1)p(z_2)p(x_1|z_1,z_2)p(x_2|z_2).$$

$$q(z_1,z_2|x_1,x_2) = q(z_1|x_1,x_2)q(z_2|x_2).$$

KL Divergence: minimize this in order fit a distribution to a target one, asymmetric: $\mathrm{KL}[p||q] \neq \mathrm{KL}[q||p]$ in general.

$$\mathrm{KL}[p||q] = \mathbb{E}_{p(\boldsymbol{x})}[-\log g(\boldsymbol{x})] \geq -\log \mathbb{E}_{p(\boldsymbol{x})}[g(\boldsymbol{x})] \quad \text{(Jensen's inequality)}$$

$$= -\log \int p(\boldsymbol{x})\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})}d\boldsymbol{x} = -\log 1 = 0,$$

$$\mathrm{KL}[p||q] = \int p(\boldsymbol{x})\log\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}d\boldsymbol{x}, \quad p,q \in \mathcal{P},$$

MLE:

given a dataset $p_{\mathrm{data}}(\boldsymbol{x})$ we want to fit to a generative model $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ with parameter theta: $\boldsymbol{\theta}^* = \arg\min \mathrm{KL}[p_{\mathrm{data}}(\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{x})]$.

Expand the above objective:

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

Gradient wrt KL term is tractable, MC still required for first term

We can ignore the constant terms wrt theta and instead work with the *maximum likelihood* object:

$$\boldsymbol{\theta}^* = \arg\max \mathbb{E}_{p_{\mathrm{data}}(\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x})] \longrightarrow \boldsymbol{\theta}^* = \arg\max \frac{1}{N}\sum_{n=1}^{N}\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_n).$$

Variational Inference:

Latent variable model: $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$., often constructed as $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z};\mathbf{0},\mathbf{I})$, $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = \mathcal{N}(\boldsymbol{x};G_{\boldsymbol{\theta}}(\boldsymbol{z}),\sigma^2\mathbf{I})$,

G$\boldsymbol{\theta}$ () defined as a neural network transform parametrized by weights theta. We optimize MLE objective wrt $\boldsymbol{\theta}$, which involves computing integral → intractable as it computes non-linear transformation G$\boldsymbol{\theta}$(z) for every single configuration of z within Gaussian p(z)

Solution: use the variational lower bound as a tractable approximation to marginal log-likelihood

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \mathbb{E}_{q(\boldsymbol{z})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})] - \mathrm{KL}[q(\boldsymbol{z})||p(\boldsymbol{z})] := \mathcal{L}(\boldsymbol{x},q,\boldsymbol{\theta}).$$

maximising $\mathcal{L}(\boldsymbol{x},q,\boldsymbol{\theta})$ is equivalent to minimising $\mathrm{KL}[q(\boldsymbol{z})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})]$.

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

Variational auto-encoder approach defines *q* distribution as a neural network: $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z};\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}),diag(\boldsymbol{\sigma}_{\boldsymbol{\phi}}^2(\boldsymbol{x})))$, $\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}),\log\sigma_{\boldsymbol{\phi}}(\boldsymbol{x}) = \mathrm{NN}_{\boldsymbol{\phi}}(\boldsymbol{x})$.

VAE optimization objective: $\boldsymbol{\phi}^*,\boldsymbol{\theta}^* = \arg\max \mathcal{L}(\boldsymbol{\phi},\boldsymbol{\theta})$, $\mathcal{L}(\boldsymbol{\phi},\boldsymbol{\theta}) = \mathbb{E}_{p_{\mathrm{data}}(\boldsymbol{x})}[\underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})] - \mathrm{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})]}_{:=\mathcal{L}(\boldsymbol{x},\boldsymbol{\phi},\boldsymbol{\theta})}]$.

Monte Carlo estimation: $\mathcal{L}(\boldsymbol{\phi},\boldsymbol{\theta})$ is still intractable, so we replace expectation with MC approximation:

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})] \approx \log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}), \quad \boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}).$$

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{x},\boldsymbol{\phi},\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}}\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}), \quad \boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}).$$

$$\nabla_{\boldsymbol{\phi}}\mathcal{L}(\boldsymbol{x},\boldsymbol{\phi},\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})] - \nabla_{\boldsymbol{\phi}}\mathrm{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})].$$

Reparametrization trick: directly sampling *z* from *q* and passing samples through the model to compute gradients is not differentiable wrt parameters ϕ of dist. from which *z* is drawn → hard to backpropagate

$$\mathcal{L}(\boldsymbol{\phi},\boldsymbol{\theta}) \approx \frac{1}{M}\sum_{m=1}^{M}\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|T_{\boldsymbol{\phi}}(\boldsymbol{x}_m,\boldsymbol{\epsilon}_m)) - \mathrm{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}_m|\boldsymbol{x}_m)||p(\boldsymbol{z}_m)],$$

$$\boldsymbol{x}_1,...,\boldsymbol{x}_m \sim \{\boldsymbol{x}_n\}^M, \ \boldsymbol{\epsilon}_1,...,\boldsymbol{\epsilon}_M \sim \mathcal{N}(\mathbf{0},\mathbf{I}),$$

$$\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \quad \Leftrightarrow \quad \boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}} + \boldsymbol{\sigma}_{\boldsymbol{\phi}}\odot\boldsymbol{\epsilon}, \ \boldsymbol{\epsilon}\sim\mathcal{N}(\boldsymbol{\epsilon};\mathbf{0},\mathbf{I})$$

Conditional VAE: generate data conditioned on additional information (class labels, viewing angle). *y* is the additional info

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{y}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z},\boldsymbol{y})p(\boldsymbol{z})d\boldsymbol{z},$$

If ***x*** is cont., then $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x},\boldsymbol{y}) = \mathcal{N}(\boldsymbol{z};G_{\boldsymbol{\theta}}(\boldsymbol{z},\boldsymbol{y}),\sigma^2\mathrm{I})$, G(z,y) is a neural network. We maximize a variational lower bound:

$$\boldsymbol{\phi}^*,\boldsymbol{\theta}^* = \arg\max \mathcal{L}(\boldsymbol{\phi},\boldsymbol{\theta}),$$

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

or minimize KL divergence: $\mathrm{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x},\boldsymbol{y})||p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x},\boldsymbol{y})]$.

GAN.

Constructs a binary classification task to assist the learning of generative model dist. $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ to fit data dist. $p_{\mathrm{data}}(\boldsymbol{x})$ $\tilde{p}(\boldsymbol{x},y) = \tilde{p}(\boldsymbol{x}|y)\tilde{p}(y)$, $\tilde{p}(y) = \mathrm{Bern}(0.5)$, $\tilde{p}(\boldsymbol{x}|y) = \begin{cases} p_{\mathrm{data}}(\boldsymbol{x}), & y=1 \\ p_{\boldsymbol{\theta}}(\boldsymbol{x}), & y=0 \end{cases}$

Fit a binary classifier (discriminator) by maximizing the MLE objective:

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

Generator fools discriminator by **minimizing** log prob. of making the **right** decisions $\boldsymbol{\theta}^*(\boldsymbol{\phi}) = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x})}[\log(1-D_{\boldsymbol{\phi}}(\boldsymbol{x}))]$. Two-player game objective for GAN is: $\min_{\boldsymbol{\theta}}\max_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi})$.

MC approximation: the evaluation of the obj. can directly define sampling process of $p_{\boldsymbol{\theta}}(\boldsymbol{x})$, which also defines the distribution in an implicit way:

$$\mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x})}[\log(1-D_{\boldsymbol{\phi}}(\boldsymbol{x}))] \approx \log(1-D_{\boldsymbol{\phi}}(\boldsymbol{x})), \quad \boldsymbol{x}\sim p_{\boldsymbol{\theta}}(\boldsymbol{x}) \quad \Leftrightarrow \quad \boldsymbol{z}\sim p(\boldsymbol{z}), \quad \boldsymbol{x} = G_{\boldsymbol{\theta}}(\boldsymbol{z}),$$

Jensen-Shannon divergence minimization: For a fixed generator, setting the gradient of GAN objective = 0 results in:

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

where $\mathrm{JS}[p_{\mathrm{data}}(\boldsymbol{x})||p_{\boldsymbol{\theta}}(\boldsymbol{x})]$ is the Jensen-Shannon divergence between $p_{\mathrm{data}}(\boldsymbol{x})$ and $p_{\boldsymbol{\theta}}(\boldsymbol{x})$. Since Jensen-Shannon divergence is a valid divergence measure, this means with infinite capacity for the generator, $\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi}^*(\boldsymbol{\theta}))$ is minimised iff. $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = p_{\mathrm{data}}(\boldsymbol{x})$.

Alternative "non-saturated" objective given fixed discriminator: Saturation problem (near-perfect classf. at start of training → vanishing gradient); so, **max**. log prob. of making **wrong** predictions. $\max_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x})]$.

Conditional GAN $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{y}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z},\boldsymbol{y})p(\boldsymbol{z})d\boldsymbol{z}$,

Distribution form of p is defined implicitly by the sampling process:

$$\boldsymbol{x}\sim p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z},\boldsymbol{y}) \quad \Leftrightarrow \quad \boldsymbol{z}\sim p(\boldsymbol{z}), \boldsymbol{x} = G_{\boldsymbol{\theta}}(\boldsymbol{z},\boldsymbol{y}),$$

Learning is done by optimizing an adversarial objective (similar to GAN, min max L(***x,y***))

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

In practice the component related to the generator parameters $\boldsymbol{\theta}$ is computed by

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

Using similar techniques, one can derive the optimal discriminator for a fixed generative model with parameter $\boldsymbol{\theta}$:

$$D_{\boldsymbol{\phi}^*(\boldsymbol{\theta})}(\boldsymbol{x},\boldsymbol{y}) = \frac{p_{\mathrm{data}}(\boldsymbol{x},\boldsymbol{y})}{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{y})p_{\mathrm{data}}(\boldsymbol{y}) + p_{\mathrm{data}}(\boldsymbol{x},\boldsymbol{y})}, \tag{20}$$

and with the optimal discriminator, minimising $\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi})$ w.r.t. $\boldsymbol{\theta}$ is equivalent to minimising the Jensen-Shannon divergence $\mathrm{JS}[p_{\mathrm{data}}(\boldsymbol{x},\boldsymbol{y})||p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{y})p_{\mathrm{data}}(\boldsymbol{y})]$.

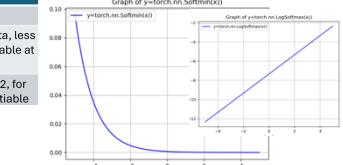
L1vs L2vs Smooth

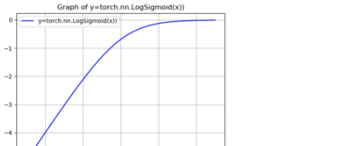
Loss	Formula	Notes
L2(MSE)	Σ|x_n −y_n|^2	Affected by outliers
L1	Σ|x_n −y_n|	Robust when dealing with noisy data, less sensitive to outliers.Non- differentiable at 0
Smooth L1	PIPPES	For values close to 0 its similar to L2, for others similar to L1. This is differentiable

Activations

Activation	Equation	Notes
Sigmoid	**σ(x) = 1/(1 + e^-x)**	Binary Classification, Cause vanishing problem
Tanh	**tanh(x) = (e^x - e^-x) / (e^x + e^-x)**	Scaled sigmoid-cause faster convergence than sigmoid-average close to 0. Okay for regression and continuous reconstruction
ReLU	**Max(0,x)**	Dying ReLU problem
Leaky ReLU	**Max(x,0.01x)**	Mitigates dying ReLU problem
Softplus	1/β* log(1 + e^βx) β is a hyperparameter	Smooth approximation of ReLU(>0),differentiable
ReLU6	**Min(max(0,x),6)**	Like ReLU but upper bound on 6
LogSigmoid	**Log(1/(1 + e^-x))**	Used within loss functions
Softmin	**(e^-x)/Σ(e^-x)**	Convert into inverse probabilities
Softmax	**(e^x)/Σ(e^x)**	Convert into probabilities
LogSoftmax	**Log((e^x)/Σ(e^x))**	Handling Numerical Stability-used as loss

Logsigmoid: output is always negative





Metrics:

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$$

$$\mathrm{Precision} = \frac{TP}{TP+FP}$$

$$\mathrm{Recall} = \frac{TP}{TP+FN}$$

$$F_1 = 2\times\frac{precision\times recall}{precision+recall}$$

$$F1 = \frac{TP}{TP+\frac{1}{2}(FP+FN)}$$

Derivatives (g is the fn)

Sigmoid: $g'(z) = g(z)(1-g(z))$

Tanh: $g'(z) = 1-g(z)^2$

ReLU: 1 if z > 0, 0 if z <= 0

Softmax (w/ CE loss):

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}log(\hat{y}_{i,c})$$

CE loss & grad w/ softmax

$$\frac{\partial L}{\partial z} = \frac{1}{N}(\hat{y}-y)$$

Losses. L1/2/CE: "reduce" to scalar using mean or

Smooth L1:

$$loss(x,y) = \frac{1}{n}\sum_i z_i$$

$$z_i = \begin{cases} 0.5(x_i-y_i)^2, & \text{if } |x_i-y_i|<1 \\ |x_i-y_i|-0.5, & \text{otherwise} \end{cases}$$

NLLL:

Assumption: Network output represents log liklihoods.

Make the desired output as large as possible and all others as small as possible

$$\ell(x,y) = \boldsymbol{L} = \{l_1,..,l_N\}^T, \quad l_n = -w_{y_n}x_{n,y_n},$$

$$w_c = weight[c]\cdot 1\{c\neq ignore_{index}\}$$

$$\ell(x,y) = \begin{cases} \sum_{n=1}^{N}\frac{1}{\sum_{n=1}^{N}w_{y_n}}l_n, & \text{if reduction = mean} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = sum} \end{cases}$$

CE loss: combines logsoftmax(output) & NLLL. Use for classif. Average across obsv. within minibatch

$$loss(x,class) = -\log\left(\frac{e^{x[class]}}{\sum_j e^{x[j]}}\right) = -x[class] + \log\left(\sum_j e^{x[j]}\right)$$

$$l_n = y_n\cdot(logy_n - logx_n) = y_n\left(log\left(\frac{y_n}{x_n}\right)\right)$$

(above) KL div. loss: dist. btw proba distributions

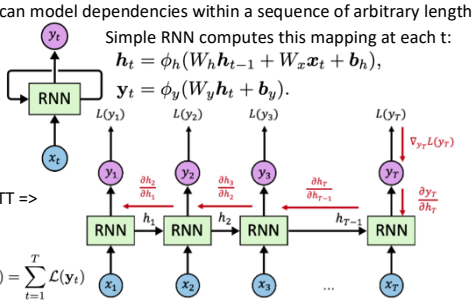
Binary CE loss: CE loss for only 2 classes. Inputs as probas or logits

$$\ell(x,y) = \boldsymbol{L} = \{l_1,...,l_N\}^T,$$

$$l_n = -w_n[y_n\cdot\log x_n + (1-y_n)\cdot\log(1-x_n)$$

RNNs

- can model dependencies within a sequence of arbitrary length



Simple RNN computes this mapping at each t:

$$\boldsymbol{h}_t = \phi_h(W_h\boldsymbol{h}_{t-1} + W_x\boldsymbol{x}_t + \boldsymbol{b}_h),$$
$$\boldsymbol{y}_t = \phi_y(W_y\boldsymbol{h}_t + \boldsymbol{b}_y).$$

$$\mathcal{L}(\theta) = \sum_{t=1}^{T}\mathcal{L}(\mathbf{y}_t)$$

$$\frac{d\boldsymbol{h}_t}{dW_h} = \frac{\partial \boldsymbol{h}_t}{\partial W_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{\partial \boldsymbol{h}_{t-1}}{\partial W_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{d\boldsymbol{h}_{t-2}}\frac{d\boldsymbol{h}_{t-2}}{dW_h} = \ldots$$
$$= \sum_{\tau=1}^{t}\left(\prod_{l=\tau}^{t-1}\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}\right)\frac{\partial \boldsymbol{h}_\tau}{\partial W_h},$$

Can truncate: $\text{truncate}[\frac{d\boldsymbol{h}_t}{dW_h}] = \sum_{\tau=\max(1,t-L)}^{t}\left(\prod_{l=\tau}^{t-1}\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}\right)\frac{\partial \boldsymbol{h}_\tau}{\partial W_h}$

- weights/biases = shared across cells, where each cell = single-layer NN with 1 input and 1 output at t. no matter #cells/timesteps, no increase in #learnable params! BUT **issue w/ gradients**: vanishing or exploding => diff to train.

$$\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}^\top = \phi_h'(W_h\boldsymbol{h}_l + W_x\boldsymbol{x}_{l+1} + \boldsymbol{b}_h)\odot W_h$$

Some **tricks**: grad clipping; good W_h init; unitary/orthog. W_h

$$\boldsymbol{g} \leftarrow \frac{\gamma}{||\boldsymbol{g}||}\boldsymbol{g}$$ Hyperparam lambda; clip $\boldsymbol{g}$ if $||\boldsymbol{g}|| > \eta$

LSTMs

- Weights/biases shared across cells; sigmoid and tanh activation fns; final output/prediction = short-term memory outputted from last cell
- different paths for long-term & short-term memories to avoid gradient vanishing/expl problem from RNNs
- memory cell states and gates to control error flows

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

h0, c0 set to 0 or learnable. If 0, then we have the elems in c_t and h_t

GRUs

- improves simple RNN with gates like LSTM
- vs. LSTM: GRU removes the input/output gates and the cell state, but maintains the forgetting mechanism in some form

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

Seq2seq models

Given dataset of input-output sequence pairs $(\boldsymbol{x}_{1:T}, \mathbf{y}_{1:L})$, the goal of sequence prediction is to build a model $p_\theta(\mathbf{y}_{1:L}|\boldsymbol{x}_{1:T})$ to fit the data.

Note here that the $\boldsymbol{x}$, $\mathbf{y}$ sequences might have different lengths, and the input/output length T and L can vary across input-output pairs. So to handle sequence outputs of arbitrary length, we define an *auto-regressive model*

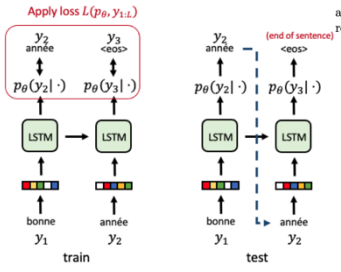
$$p_\theta(\mathbf{y}_{1:L}|\boldsymbol{x}_{1:T}) = \prod_{l=1}^{L} p_\theta(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v}), \quad \boldsymbol{v} = enc(\boldsymbol{x}_{1:T})$$

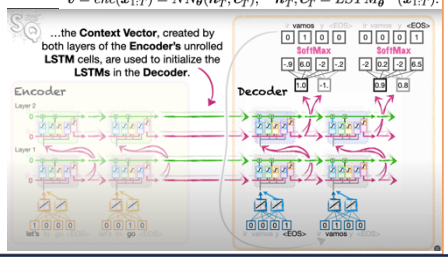
Here $p_\theta(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v})$ is defined by a sequence decoder, e.g. an LSTM:

$$p_\theta(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v}) = p_\theta(\mathbf{y}_l|\boldsymbol{h}_l^d, \boldsymbol{c}_l^d), \quad \boldsymbol{h}_l^d, \boldsymbol{c}_l^d = LSTM_\theta^{dec}(\mathbf{y}_{<l}), \tag{28}$$

and the decoder LSTM has its internal recurrent states $\boldsymbol{h}_0^d$, $\boldsymbol{c}_0^d$ initialised using the input sequence representation $\boldsymbol{v} = enc(\boldsymbol{x}_{1:T})$. The encoder also uses an LSTM, meaning that

$$\boldsymbol{v} = enc(\boldsymbol{x}_{1:T}) = NN_\theta(\boldsymbol{h}_T^e, \boldsymbol{c}_T^e), \quad \boldsymbol{h}_T^e, \boldsymbol{c}_T^e = LSTM_\theta^{enc}(\boldsymbol{x}_{1:T}).$$





Attention

Single-head attention: *scaled dot product attention*

$$Attention(Q, K, V; a) = a\left(\frac{QK^\top}{\sqrt{d_q}}\right)V$$

$$a(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

$$Q = (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_N)^\top \in \mathbb{R}^{N\times d_q}$$
$$K = (\boldsymbol{k}_1, \ldots, \boldsymbol{k}_M)^\top \in \mathbb{R}^{M\times d_q}$$
$$V = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_M)^\top \in \mathbb{R}^{M\times d_v}$$

Time and Space Complexity for 1 head

$$\mathcal{O}(N^2 d_q + N^2 d_v) \text{ and } \mathcal{O}(N^2 + N d_v)$$

It is clear that the time and space complexity figures of multi-head attention are h times of those for a single head plus the extra costs for linear projections. Assume the projection matrices have sizes $W_i^Q \in \mathbb{R}^{d_q\times \tilde{d}_q}$, $W_i^K \in \mathbb{R}^{d_q\times \tilde{d}_q}$, $W_i^V \in \mathbb{R}^{d_v\times \tilde{d}_v}$ and $W^O \in \mathbb{R}^{h\tilde{d}_v\times d_{out}}$. This means

time complexity: $\mathcal{O}(\underbrace{h(MN\tilde{d}_q + MN\tilde{d}_v)}_{\text{attention heads}} + \underbrace{h(\tilde{d}_q d_q(M+N) + \tilde{d}_v d_v M)}_{\text{input projections}} + \underbrace{N h\tilde{d}_v d_{out}}_{\text{combined outputs}})$,

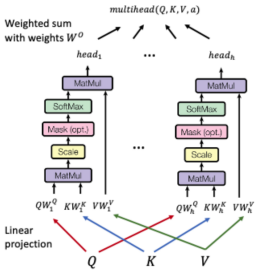
space complexity: $\mathcal{O}(\underbrace{h(MN + N\tilde{d}_v)}_{\text{attention heads}} + \underbrace{h(\tilde{d}_q(M+N) + \tilde{d}_v M)}_{\text{input projections}} + \underbrace{N d_{out}}_{\text{combined outputs}})$.

soft attention: $a(.)$ is the softmax function, $A_{ij} = softmax(\langle \boldsymbol{q}_i, \boldsymbol{k}_1\rangle, \ldots, \langle \boldsymbol{q}_i, \boldsymbol{k}_M\rangle)/\sqrt{d_q})$

hard attention: one-hot vector for each row with $j^* = \arg\max_j \langle \boldsymbol{q}_i, \boldsymbol{k}_j\rangle$ equal to 1

In transformers, layer normalisation is applied together with a residual link: $Add\&Norm(x) = LayerNorm(x + Sublayer(x))$, where $Sublayer(\cdot)$ can either be a multi-head attention block or a point-wise feed-forward network.

Multi-head attention: multiple alignment processes by projecting inputs into diff. subspaces, then performing dot product attention in subspace. Outputs are concatenated and projected

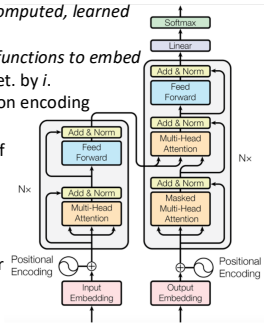


Transformers

Position encoding: attention is equivariant to row permutations bc non-linearity a(.) is applied row-wise. Ordering info needs to be added to attention inputs, can either be learned or computed, learned is good if we know max value of index

Sinusoid embedding: use multiple periodic functions to embed input index, frequency of sin/cos wave is det. by *i*. Allows network to learn very flexible position encoding function

Point wise feed forward network: used as ff layer, multi-head attention (after *Add&Norm*) returns a matrix of size N x d_out, which can be processed "point-wise", treating rows in the attention outputs as "datapoint" inputs for next layer



Margin Ranking/Ranking/Contrastive Losses

$$loss(x, y) = \max(0, -y \cdot (x_1 - x_2) + margin)$$

Useful to push classes as far away as possible and for metric learning

Practical: take category that scores is closest or higher than correct one change until difference is at least the margin

This class of loss functions has a unique purpose—instead of directly predicting labels or values, they focus on predicting relative distances between inputs.

- $\ell(x, y) = \mathcal{L} = \{l_1, \ldots, l_N\}^T$,

Triplet Margin Loss

$$l_n(x_a, x_p, x_n) = \max(0, m + |f(x_a) - f(x_p)| - |f(x_a) - f(x_n)|$$

actual, positive sample, negative sample

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

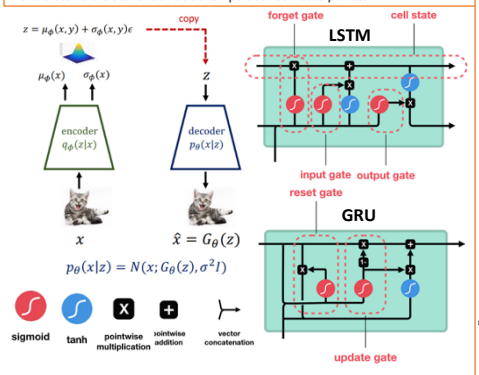
Cosine Embedding Loss

$$loss(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - margin), & \text{if } y = -1 \end{cases}$$

Measure weather two inputs are similar or dissimilar

Basically a normalised Euclidian distance

The Cosine Embedding Loss is especially useful for learning nonlinear embeddings and for semi-supervised learning tasks, where the objective is to understand the semantic relationships between data points.



Power rule:	$\frac{d}{dx}(x^n) = nx^{n-1}$
Exponential (e):	$\frac{d}{dx}e^x = e^x$
Exponential (general):	$\frac{d}{dx}a^x = a^x \ln(a)$
Natural log:	$\frac{d}{dx}(ln(x)) = \frac{1}{x}$
General log:	$\frac{d}{dx}(log_a(x)) = \frac{1}{x ln(a)}$
Product rule:	$\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$
Quotient rule:	$\frac{d}{dx}[\frac{f(x)}{g(x)}] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Chain rule:	$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$
Chain rule:	$\frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx}$

Total for the backward pass: 2*forward pass

Rule Name	Property
Log of 1	$\log_a 1 = 0$
Log of the same number as base	$\log_b b = 1$
Product Rule	$\log_b(mn) = \log_b m + \log_b n$
Quotient Rule	$\log_b(\frac{m}{n}) = \log_b m - \log_b n$
Power Rule	$\log_b m^n = n \log_b m$
Change of Base Rule	$\log_b = \frac{\log_b}{\log_a a}$ (OR) $\log_b \cdot \log_a a = \log_b b$
Equality Rule	$\log_a a = \log_b c \Rightarrow a = c$
Number Raised to Log	$b^{\log_b x} = x$
Other Rules	$\log_b a^m = \frac{m}{n}\log_b a$; $-\log_b a = \log_b \frac{1}{a}$ (OR) $= \log_{\frac{1}{b}} a$

FLOPs Total training flops ≈ 6*dataset size in tokens*number of parameters

Efficient inference: grad accumul., checkpointing

Gradient Accumulation enables training with larger effective batch sizes on limited hardware by computing and summing gradients across multiple mini-batches before performing a single update step. This method delays the optimizer step, thus accumulating gradients from several mini-batches, allowing for larger batch processing without increasing GPU memory usage.

Gradient Checkpointing reduces the memory footprint during training by selectively storing only a subset of the forward pass activations. Non-stored activations are recomputed during the backward pass as needed. This technique trades additional computation for lower memory usage, facilitating the training of larger models within memory constraints.

Finetuning The pretrained weights Φ, are updated Φ + ΔΦ according to

$$\max_\Phi \sum_{(x,y)\sim Z}\sum_{t=1}^{y}\log(P_\Phi(y_t|x, y_{<t}))$$

Context-target tokens, Autoregressive foundation model

Here, every parameter in Φ is updated therefore need to store gradients + optimisers of every weight. For every downstream task a new model of size Φ is required + we need to relearn ΔΦ parameters.

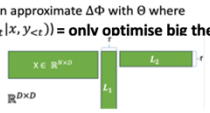
Low rank Adaptation (LoRA)

Li et al. demonstrated that the model-loss overparametrised space has a low intrinsic dimension/rank.

What if ΔΦ also has low intrinsic rank we can approximate ΔΦ with Θ where $|\Theta| << |\Phi|$ $\max_\Theta \sum_{(x,y)\sim Z}\sum_{t=1}^{y}\log(P_{\Phi+\Theta}(y_t|x, y_{<t}))$ = **only optimise bie theta**

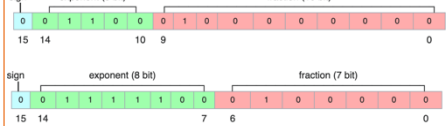
- For a pretrained set up of $\mathbf{y} = \boldsymbol{W_0}\mathbf{x}$
- $\mathbf{y} = \boldsymbol{W_0}\mathbf{x} + \mathbf{X}\Delta\boldsymbol{W} \approx \boldsymbol{W_0}\mathbf{x} + \boldsymbol{X}\boldsymbol{L_1}\boldsymbol{L_2}$

Where, $\boldsymbol{L_1} \in \mathbb{R}^{D\times r}$ and $\boldsymbol{L_2} \in \mathbb{R}^{r\times D}$ and $r << D$



At inference time we combine: $\boldsymbol{W_0} + \boldsymbol{L_1}\boldsymbol{L_2}$

- No additional inference time (no extra steps to compute both $\boldsymbol{XW_0}$ and $\boldsymbol{XL_1L_2}$)
- We can always subtract $\boldsymbol{L_1L_2}$ when we are done and combine with a different set of parameters $\boldsymbol{L_1'L_2'}$



LoRA (Low Rank Adaptation) streamlines fine-tuning of pre-trained deep learning models by approximating updates to the model's weights with two smaller, low-rank matrices, A and B, where A*B approximates the weight update ΔW. This approach significantly reduces the computational resources needed for fine-tuning by decreasing the number of parameters to adjust, enabling efficient updates on modest hardware without substantially sacrificing performance.

QLoRA (Quantized Low Rank Adaptation) enhances LoRA by incorporating quantization, which further compresses the model updates by reducing the numerical precision of the A and B matrices. This method not only lowers memory usage but also accelerates computation, particularly on hardware optimized for lower precision arithmetic, making fine-tuning even larger models feasible on constrained hardware environments.

Deep learning models are more sensitive to underflow + overflow than to precision

Hardware requirements scale quadradically with fraction but not exponent

LoRA recap: $\boldsymbol{y} = \boldsymbol{XW_0} + \boldsymbol{XL_1L_2}$ QLoRA looks to quantise $\boldsymbol{W_0}$ from bfloat16 to 4 bit!

HOW? Double Quantisation When required for computation we dequantise weights from NormalFloat4 to BFLoat16

$\boldsymbol{y} = \boldsymbol{X}dequant(\boldsymbol{W_0}) + \boldsymbol{XL_1L_2}$

Conv

- Don't assume that in for a single kernel, the same kernel is applied to each channel of the input image
- Max pool and average pool are applied PER channel (not aggregating across channels of input image): only operate on width and height of input

- Mean and var learned gamma beta: to give new dist

- Valid padding = no padding at all
- Same padding = as much padding as needed to preserve the dims of input
- 0 padding needed for same padding: (K – 1) / 2

- Num of params for a conv layer with (more than 1) F filters of the same shape: F * (width of filter x height of filter x input channels + 1) where 1 is bc we have 1 bias per filter
- Batchnorm output: same shape as input. Num learnable params: 2 x num of channels in input. It's 2 bc you have gamma and beta
- Pooling layers = no learnable params

Backprop / gradients

- Add the expression for dJ/dy_hat when using J = cross entropy loss with softmax being used: 1/m (y_hat – y)
- If largest singular value of a weight matrix > 1, then gradient explosion happens

DAG

- Conditional = joint / priors
    - Hence for q(z1,z2 | x1, x2): the priors of x1 and x2 cancel out and you don't write them as factors